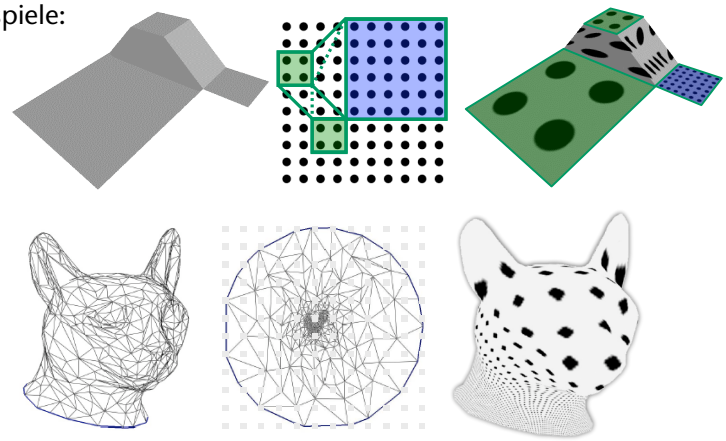
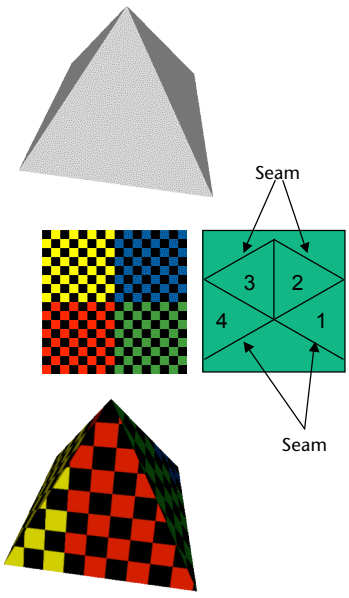
Probleme bei der Parametrisierung

- Verzerrungen: Größe & Form
- Folge: Relatives over- bzw. under-sampling
- Beispiele:

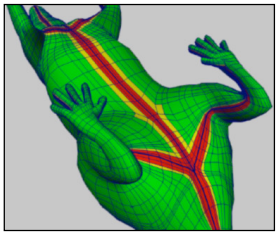


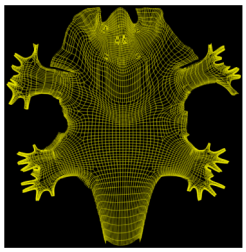
Seams (Textursprünge)

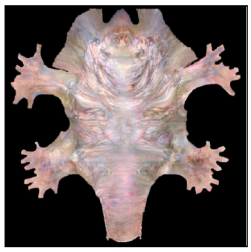
- Ziel: Verringern der Verzerrung
- Idee: Aufschneiden des Netzes entlang gewisser Kanten
- Ergibt „doppelte“ Kanten, auch „*seams*“ genannt
- Unvermeidlich bei nicht-planarer Topologie



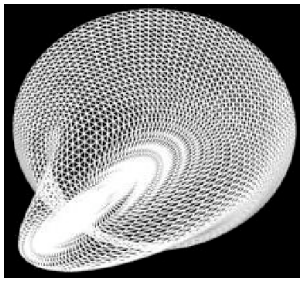
- Idee 1 [Piponi2000]:
  - Das Objekt entlang nur einer zusammenhängenden Kante so aufschneiden, daß eine topologische Scheibe entsteht
  - Dieses aufgeschnittene Netz dann in die Ebene einbetten

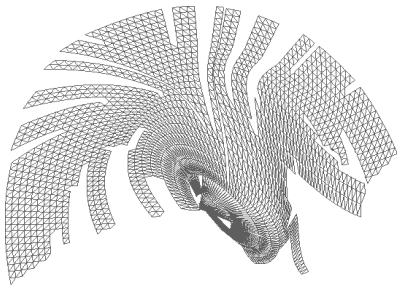






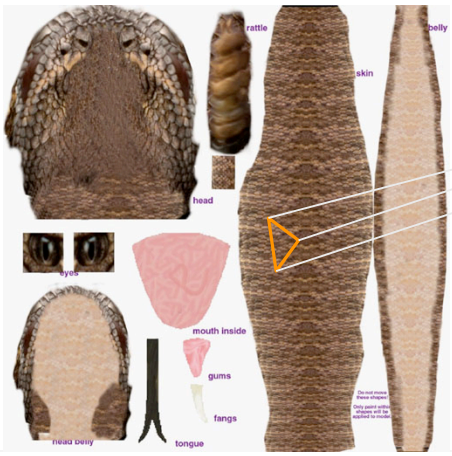
- Probleme:
  - Es gibt immer noch Verzerrungen
  - Mehrfaches "Einschneiden" ergibt stark "zerfranstes" eingebettetes Gitter

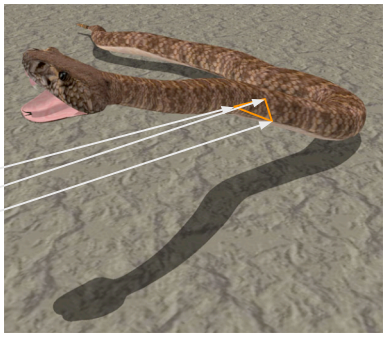


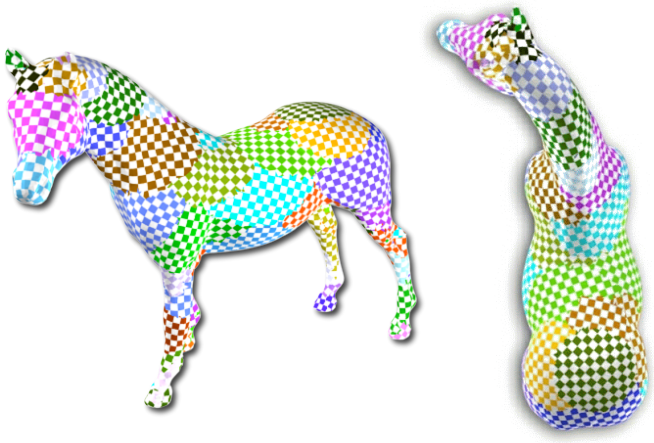


- Idee 2:
  - Zerschneide Fläche in einzelne Patches
  - Wähle Kompromiß zwischen Seams und Verzerrung
  - „Verstecke“ Schnitte in wenig sichtbaren Regionen
- Textur-Atlas = Vereinigung dieser Patches mit ihren jeweiligen Parametrisierungen
- Karte (map) = einzelnes Parametergebiet in Texture-Space
- Möglichst kompakte Anordnung der Texturpatches

- Beispiele



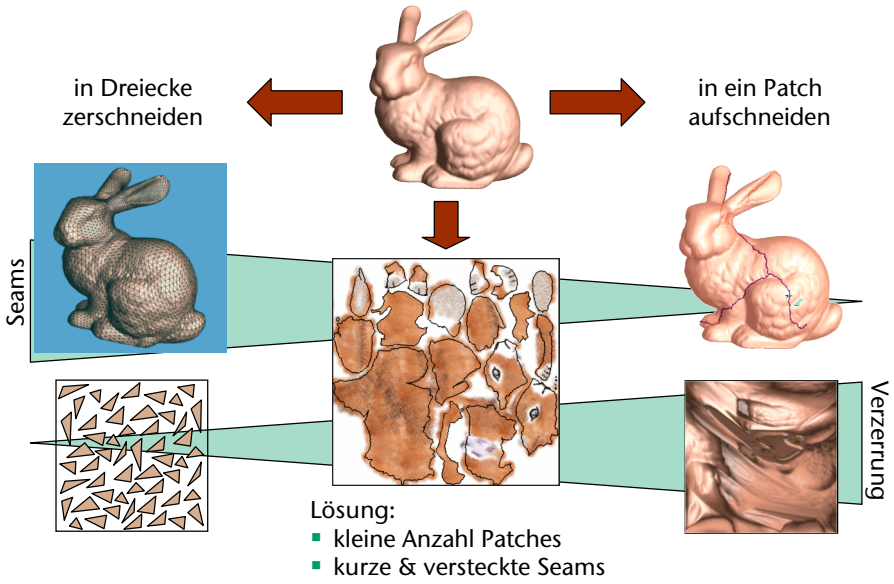




Patches - Seams

- problems
  - discontinuity of parameterization
  - visible artifacts in texture mapping
  - require special treatment
  - vertices along seams have several (u, v) coordinates
- solution
  - small number of patches
  - short and hidden seams

Verzerrung oder Seams?



Lösung:
- kleine Anzahl Patches
- kurze & versteckte Seams

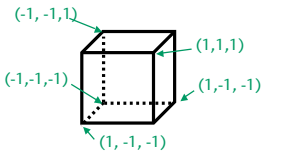
Cube Maps

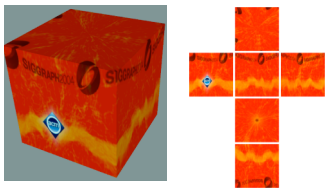
[Greene '86, Voorhies '94]

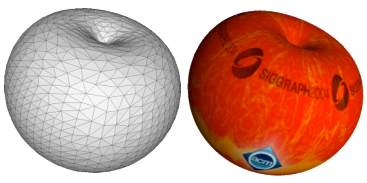
- Ω = Einheits-Würfel:
  - Sechs quadratische Textur-Bitmaps
  - 3D Texturkoordinaten:

```
glTexCoord3f( s, t, r );
glVertex3f( x, y, z );
```

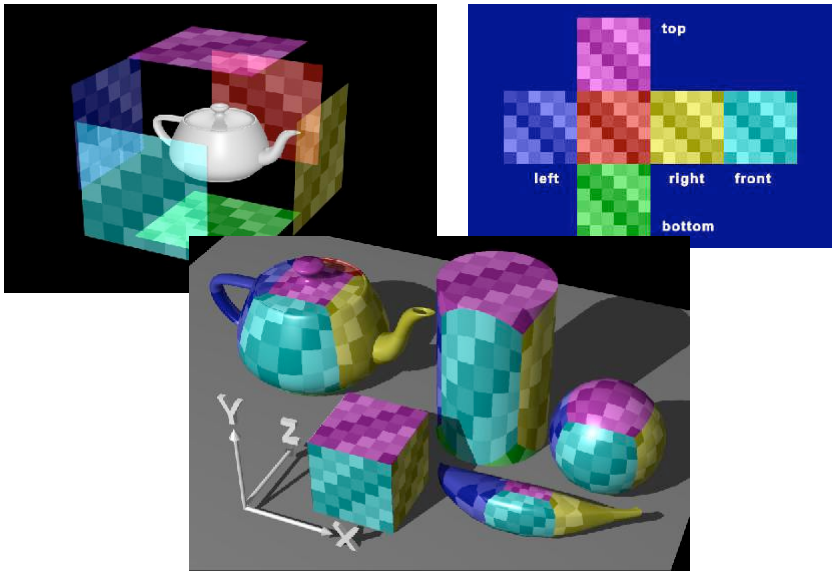
  - Größte Komponente von (*s*,*t*,*r*) wählt die Karte aus, Schnittpunkt liefert (*u*,*v*) innerhalb der Karte
- Rasterisierung
  - Interpolation von (*s*,*t*,*r*) in 3D
  - Projektion auf Würfel
  - Textur look-up in 2D
- Vorteile: rel. uniform, OpenGL
- Nachteil: man benötigt 6 Texturen



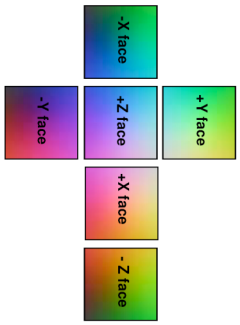




Beispiele



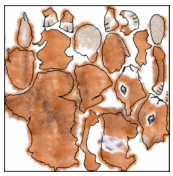
- Weitere Anwendung: man kann eine Cube-Map auch sehr gut verwenden, um irgendeine Funktion der Richtung zu speichern! (vorberechnet als LUT)
- Beispiel: Normierung eines Vektors
  - Jedes Cube-Map-Texel (s,t,r) speichert in RGB den Vektor $\frac{(s,t,r)}{\|(s,t,r)\|}$
  - Jetzt kann man beliebige Texturkoordinaten mittels `glTexCoord3f()` angeben, und bekommt den normierten Vektor
- Achtung: bei dieser Technik sollte man (meistens) jegliche Filterung ausschalten!

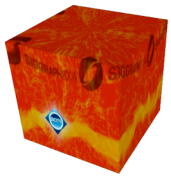


Cube-Maps in OpenGL

```
glBindTexture( GL_TEXTURE_CUBE_MAP );
glTexImage( GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGB, w, h,
            0, GL_RGB, GL_UNSIGNED_BYTE, image );
glTexParameter( GL_TEXTURE_CUBE_MAP, ... );
...
glEnable( GL_TEXTURE_CUBE_MAP );
glBindTexture( GL_TEXTURE_CUBE_MAP );
glBegin( GL_... );
glTexCoord3f( s, t, r );
glVertex3f( ... );
...
```


Textur-Atlas vs. Cube-Map

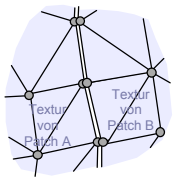


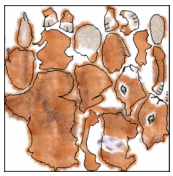


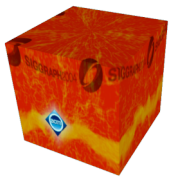
- Seams
- Dreiecke innerhalb der Patches
- nur für ein Dreiecksnetz
- Mip-Mapping schwierig
- verschwendete Texel
- Sampling-Artefakte an den Rändern der Patches

- Keine seams
- Dreiecke in mehreren Patches
- für viele Dreiecksnetze
- Mip-Mapping okay
- alle Texel benutzt
- Keine Ränder, keine Sampling-Artefakte

Textur-Atlas vs. Cube-Map



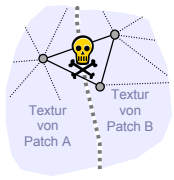




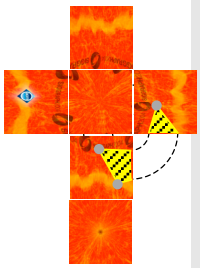
- seams
- Dreiecke innerhalb der Patches
- nur für ein Dreiecksnetz
- Mip-Mapping schwierig
- verschwendete Texel
- Sampling-Artefakte an den Rändern der Patches

- keine seams
- Dreiecke in mehreren Patches
- für viele Dreiecksnetze
- Mip-Mapping okay
- alle Texel benutzt
- keine Ränder, keine Sampling-Artefakte

Textur-Atlas vs. Cube-Map



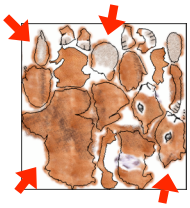
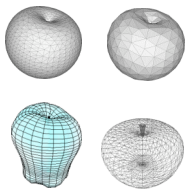
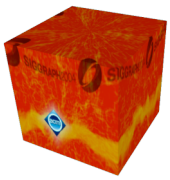




- seams
- Dreiecke innerhalb der Patches
- nur für ein Dreiecksnetz
- Mip-Mapping schwierig
- verschwendete Texel
- Sampling-Artefakte an den Rändern der Patches

- keine seams
- Dreiecke in mehreren Patches
- für viele Dreiecksnetze
- Mip-Mapping okay
- alle Texel benutzt
- keine Ränder, keine Sampling-Artefakte

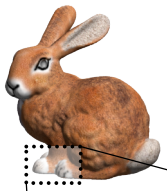
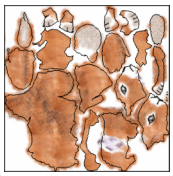
Textur-Atlas vs. Cube-Map

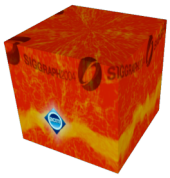
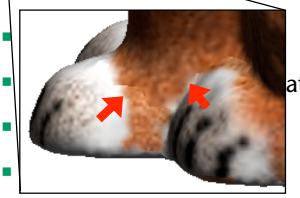





- seams
- Dreiecke innerhalb der Patches
- nur für ein Dreiecksnetz
- Mip-Mapping schwierig
- verschwendete Texel
- Sampling-Artefakte an den Rändern der Patches

- keine seams
- Dreiecke in mehreren Patches
- für viele Dreiecksnetze
- Mip-Mapping okay
- alle Texel benutzt
- keine Ränder, keine Sampling-Artefakte

Textur-Atlas vs. Cube-Map

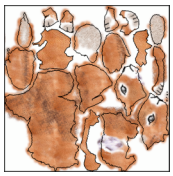



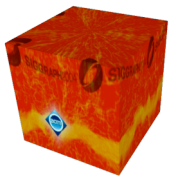



- atches
- verschwendete Texel
- Sampling-Artefakte an den Rändern der Patches

- keine seams
- Dreiecke in mehreren Patches
- für viele Dreiecksnetze
- Mip-Mapping okay
- alle Texel benutzt
- keine Ränder, keine Sampling-Artefakte

Textur-Atlas vs. Cube-Map



- seams
- Dreiecke innerhalb der Patches
- nur für ein Dreiecksnetz
- Mip-Mapping schwierig
- verschwendete Texel
- Sampling-Artefakte an Rändern der Patches

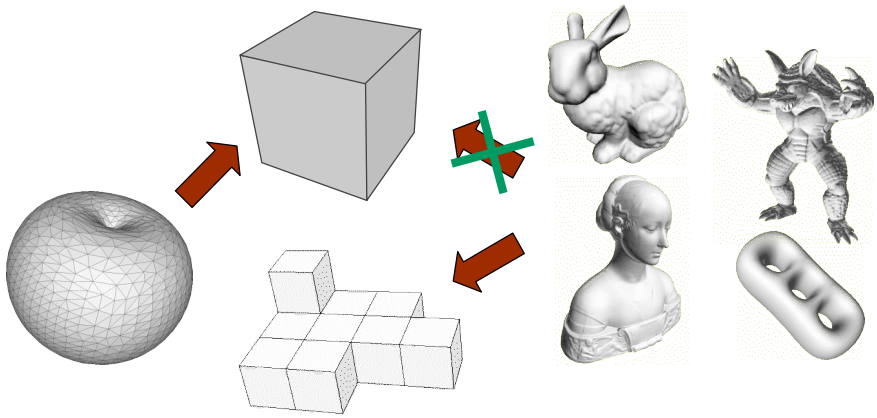
für beliebige Meshes

- keine seams
- Dreiecke in mehreren Patches
- für viele Dreiecksnetze
- Mip-Mapping okay
- alle Texel benutzt
- keine Ränder, keine Sampling-Artefakte

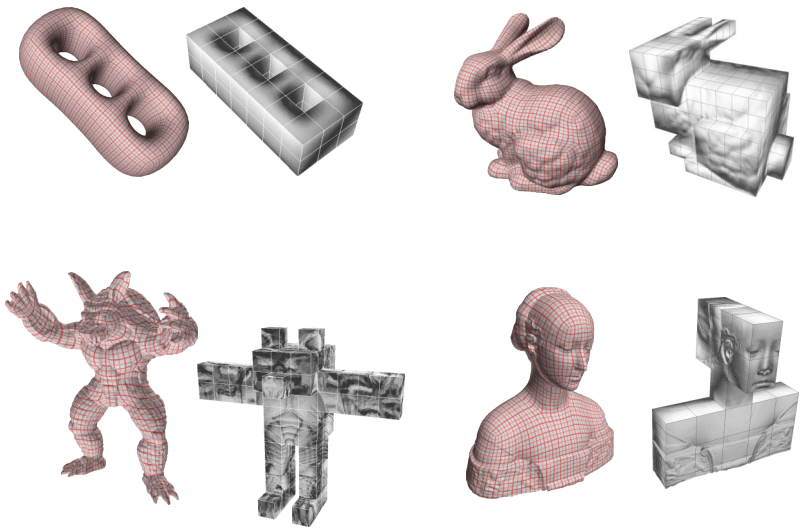
nur für „Kugeln“

Polycube-Maps

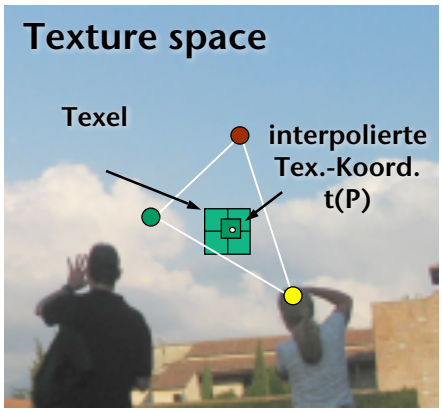
- Polycube statt eines einzelnen Würfels
- An Geometrie und Topologie angepaßt

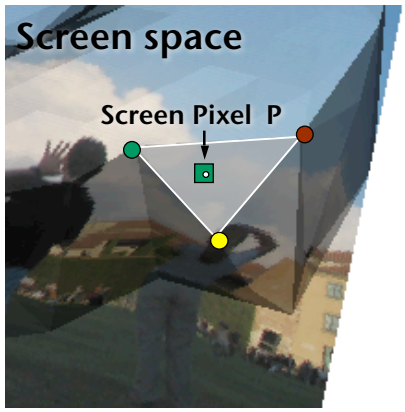


Beispiele



Texturinterpolation

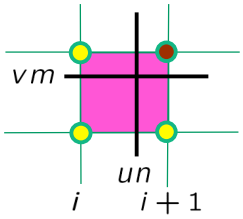




- Nearest neighbour, oder
- Bilineare Interpolation der Texel

Rekonstruktionsmethoden

- Textur = $m \times n$ Array C von Texeln,
  $t(P) = (u, v) \in [0, 1] \times [0, 1]$



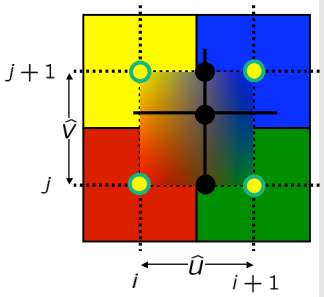
1. Nearest neighbour (Punktfilter):

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

2. Bilineare Interpolation:

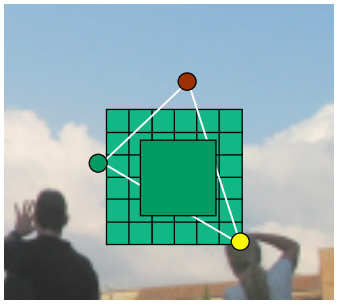
$$\hat{u} = un - \lfloor un \rfloor \ , \ \hat{v} = vm - \lfloor vm \rfloor$$

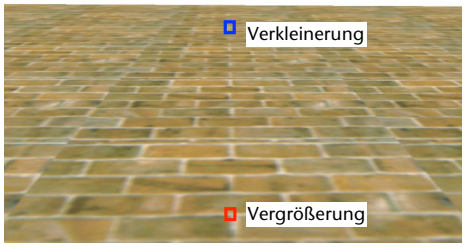
$$c = (1-u)[(1-v)\blacksquare + v\blacksquare] + u[(1-v)\blacksquare + v\blacksquare]$$

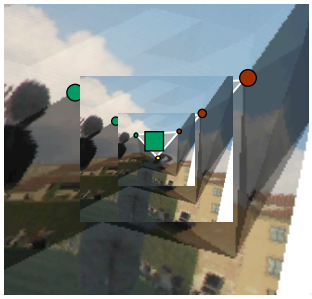


Texturverkleinerung

- Bilineare Interpolation gut, 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"?

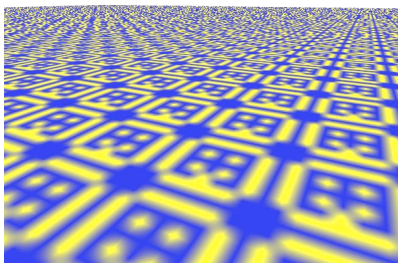


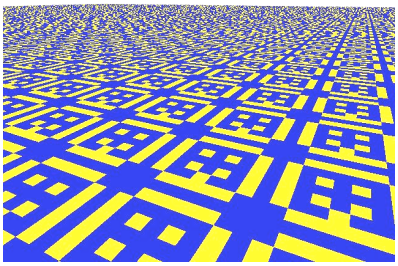




- Schwierigeres und "heißes" Problem
- Es gibt viele Möglichkeiten

1. Auch hier den einfachen Punktfilter → Aliasing
2. Lineare Interpolation hilft nur wenig

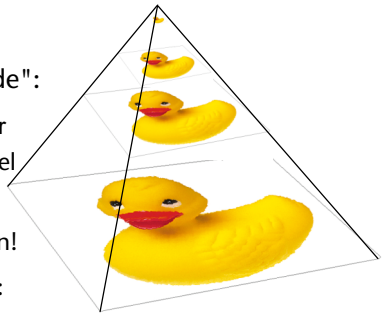


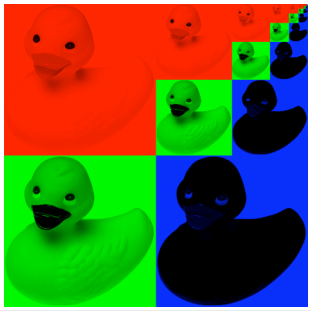


- Bei starker Verkleinerung müsste eigentlich eine Mittelung von vielen Texeln durchgeführt werden, da sie alle auf ein Pixel auf dem Bildschirm abgebildet werden
- Für Echtzeitanwendungen und Hardwarerealisierungen ist das zu aufwendig
- Lösung: Preprocessing
  - Vor dem Start verkleinerte Versionen der Textur anlegen, die schon gemittelt sind
  - Wenn jetzt viele Texel auf einen Bildschirmpixel abgebildet werden, wird die beste passende Verkleinerung verwendet anstatt der Originaltextur

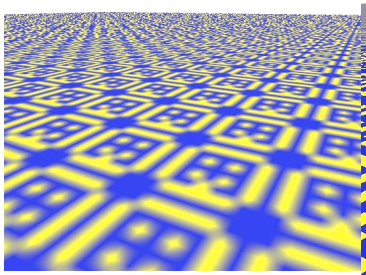
→MIP-Maps (lat. "multum in parvo" = vieles im Kleinen")

- 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 aus einem 2n-Bild ein 2n+1-Bild gemacht
- MIP-Map hat Speicherbedarf 1.3x

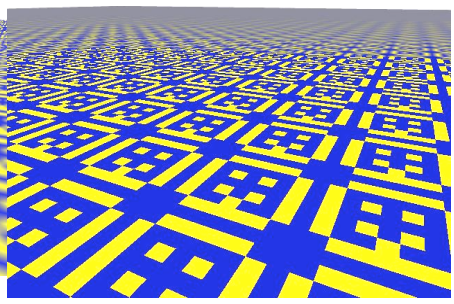




- Abhängig von der Distanz des Betrachters zum Pixel wird von OpenGL entschieden, welcher Texturlevel sinnvoll ist (pro Pixel)
  - Der ideale Level ist der, bei dem 1 Texel auf 1 Pixel abgebildet wird



binlinear gefiltert



MIP-Map

Filterspezifikation in OpenGL

- 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

- 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()`

Automatische Erzeugung von Textur-Koordinaten in OpenGL

- `glEnable( GL_TEXTURE_GEN_S );` // S, T, R, Q
- `glTexGeni( GL_S, GL_TEXTURE_GEN_MODE,` *mode*);
- mode =

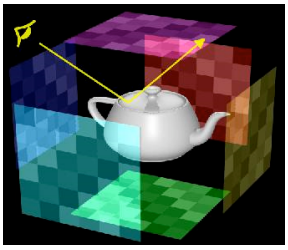
`GL_OBJECT_LINEAR`	: Texturkoord. = Distanz des Vertex von einer Ebene; die Ebene wird spezifiziert mit `glGenTexfv( GL_S, GL_OBJECT_PLANE, v )`
`GL_EYE_LINEAR`	: benutze Vertex-Koord. **nach** `MODEL_VIEW`
`GL_SPHERE_MAP`	: für Environment-Mapping (gleich)
`GL_NORMAL_MAP`	
`GL_REFLECTION_MAP`	

Beispiel

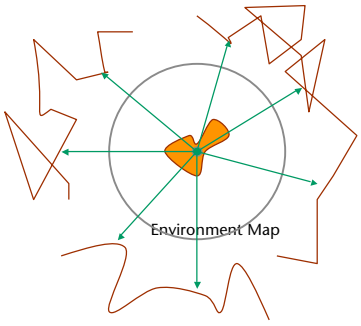
```
glEnable( GL_TEXTURE_GEN_S );
glTexGeni( GL_S,
           GL_TEXTURE_GEN_MODE,
           GL_OBJECT_LINEAR );
glTexGenfv( GL_S,
            GL_OBJECT_PLANE,
            xPlane );
```

Environment Mapping

- Bei stark spiegelnden Objekten würde man gerne die Umgebung im Objekt gespiegelt sehen
- Raytracing kann das, nicht aber das Phong-Modell
- Die Idee des Environment-Mapping:
  - "Photographiere" die Umgebung in einer Textur
  - Speichere diese in einer sog. Environment Map
  - Verwende den Reflexionsvektor (vom Sehstrahl) als Index in die Textur
  - Daher a.k.a. reflection mapping



- Die Environment-Map speichert also für jede Raumrichtung die Lichtfarbe, die aus dieser Richtung in einem bestimmten Punkt eintrifft
- Stimmt natürlich nur für eine Position
- Stimmt nicht mehr, falls das Environment sich ändert



Historische Anwendungsbeispiele

Lance Williams, Siggraph 1985

Flight of the Navigator in 1986;
first feature film to use the technique

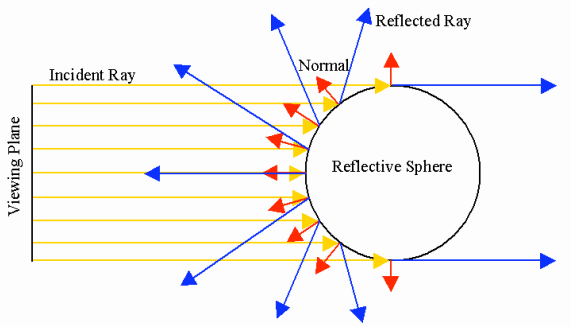
Terminator 2: Judgment Day - 1991
most visible appearance — Industrial Light + Magic

Die Einzelschritte

- Generiere oder lade eine 2D-Textur, die das Environment darstellt
- Für jedes Pixel des reflektierenden Objektes
  - Berechne die Normale **n**
  - Berechne einen Reflexionsvektor **r** aus **n** und dem View-Vektor **v**
  - Berechne Texturkoordinaten (u,v) aus **r**
  - Färbe mit dem Texturwert das Pixel
- Das alte Problem: welche Parametrisierung?
  - Wie bildet man Raumrichtungen auf [0,1]x[0,1] ab?
- Gewünschte Eigenschften:
  - Uniformes Sampling (mögl. konstant viele Texel pro Raumwinkel in allen Richtungen)
  - View-unabhängig (mögl. nur eine Textur für alle Kamera-Pos.)
  - Hardware-Support (Textur-Koordinaten sollten einfach zu erzeugen sein)

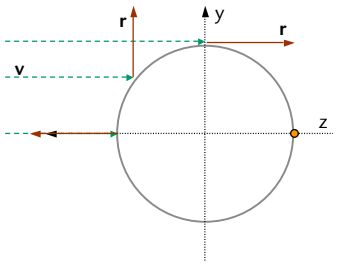
Spherical Environment Mapping

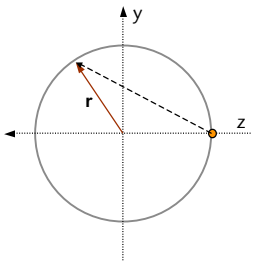
- Erzeugung der Environment-Map (= Textur):
  - Photographie einer speigelenden Kugel
  - Ray-Tracing der Szene mit spezieller "rotierender Kamera" und anschließendem Mapping



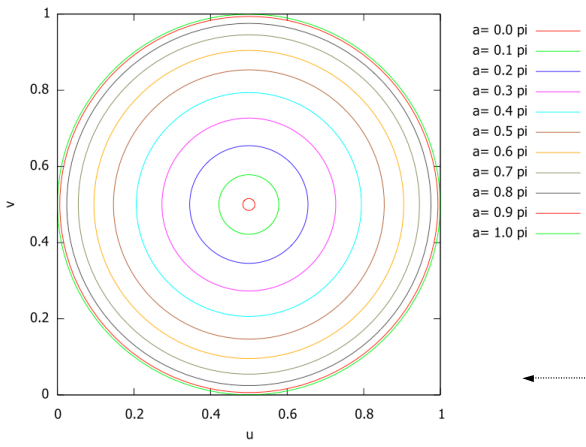
- Abbildung des Richtungsvektors **r** auf (u,v):
  - Die Sphere-Map enthält (theoretisch) einen Farbwert für jede Richtung, außer **r** = (0, 0, -1)
  - Mapping:

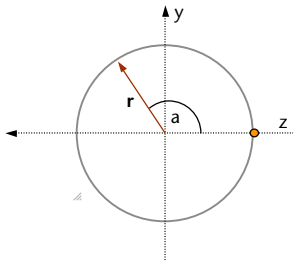
$$\begin{pmatrix} u \\ v \end{pmatrix} = \frac{1}{2}\left(\begin{array}{c} \frac{r_x}{\sqrt{r_x^2+r_y^2+(r_z+1)^2}}+1 \\ \frac{r_y}{\sqrt{r_x^2+r_y^2+(r_z+1)^2}}+1 \end{array}\right)$$



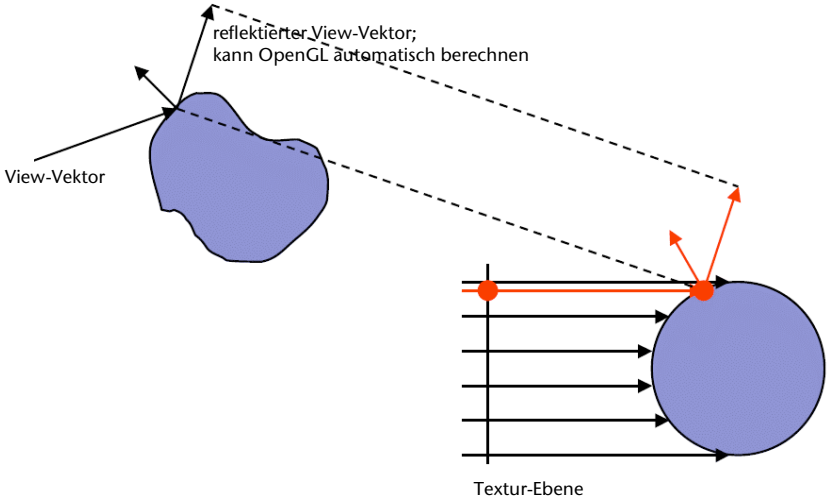


- Das Mapping ist leider sehr nicht-uniform:

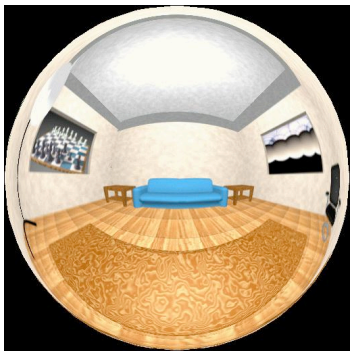




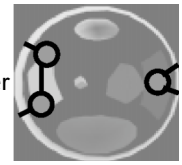
- Anwendung der Sphere Map zur Texturierung:



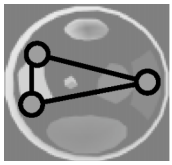
Einfaches Beispiel



- Nachteile:
  - Maps (Texturen) sind schwierig per Computer zu erzeugen
  - Sehr nicht-uniformes Sampling
  - Nur halbwegs korrekt, wenn sich das reflektierende Objekt nahe am Ursprung (in View Space) befindet
  - Sparkles / speckles wenn der reflektierte Vektor in die Nähe des Randes der Textur kommt (durch Aliasing und "wrap-around")
  - View-point dependent: das Zentrum der Sphere-Map repräsentiert den Vektor, der direkt zum Viewer zurück geht!
- Vorteile:
  - einfach, Textur-Koordinaten zu erzeugen
  - unterstützt in OpenGL



beabsichtigte Interpolation

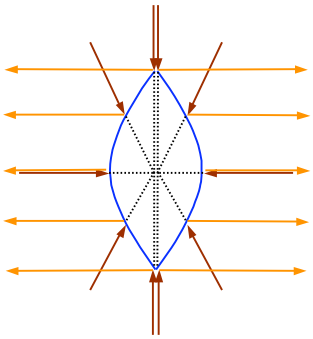


tatsächliche Interpolation (Wrapping)

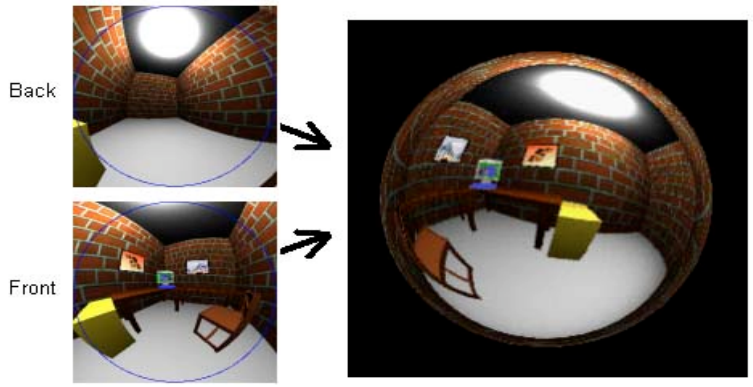
Parabolic Environment Mapping

[Heidrich'98]

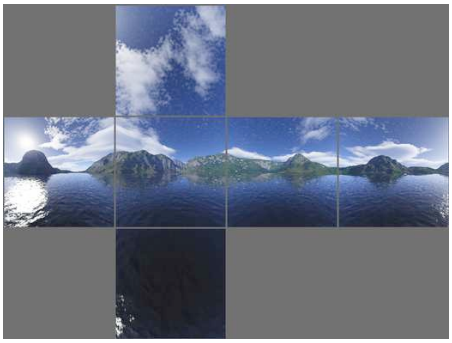
- Idee:
  - Bilde das Environment durch eine reflektierendes Doppel-Paraboloid auf zwei Texturen ab
  - Vorteile:
    - rel. uniformes Sampling
    - wenig Verzerrung
    - rel. einfache Textur-Koordinaten
    - geht auch in OpenGL
    - geht auch in einem Rendering-Pass (benötigt nur Multi-Texturing)
  - Nachteile:
    - Artefakte bei Interplation über die "Kante" hinweg



- Die Bilder der Umgebung (= Richtungsvektoren) sind immer noch Kreisscheiben (wie bei sphere map)
- Vergleich:



- Cube Map:
  - Sechs Bilder von der Mitte eines Würfels durch seine Stirnflächen [Greene '86, Voorhies '94]
  - Vorteile:
    - relativ uniform
    - unterstützt in OpenGL
    - belege Abbildung zu Linien
  - Nachteile:
    - Bearbeitung von 6 Texturen
    - Spalten



Cubic Environment Mapping

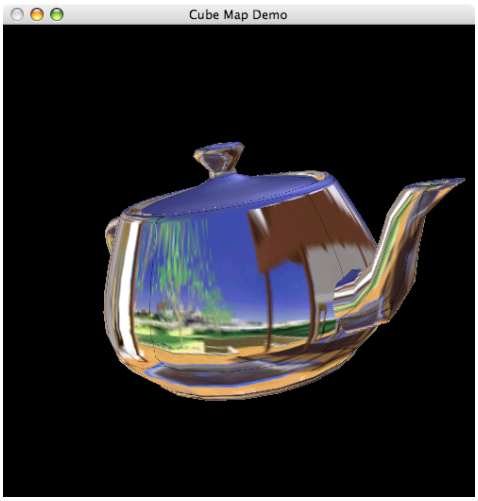
- Wie früher bei den "normalen" Cube Maps
- Einziger Unterschied: verwende den reflektierten Vektor zur Berechnung der Texturkoordinaten
- Dieser reflektierte Vektor kann von OpenGL automatisch pro Vertex berechnet werden (**`GL_REFLECTION_MAP`**)

Demo mit statischem Environment

```
cd demos/cubemap;
./cm_demo
```

Tasten:

s =	"shape"
space	= reflection / normal map
c =	clamp / repeat
m =	texture matrix * (-1,-1,-1)
a/z =	increase / decrease texture LOD bias on / off



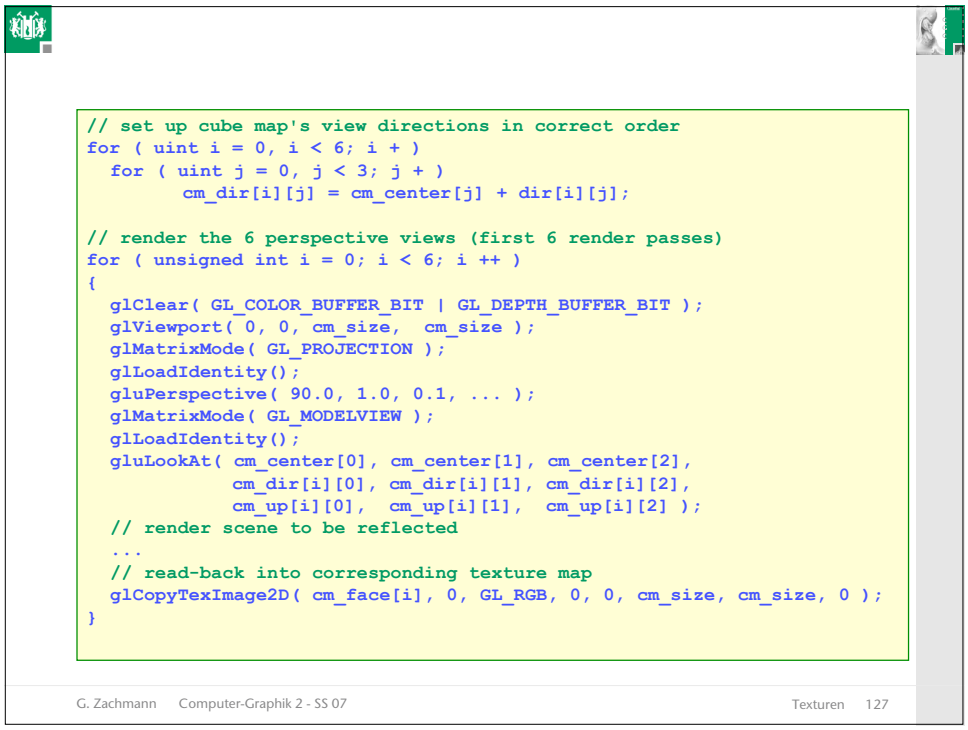
Dynamische Environment Maps

- Bisher: Environment Map wurde ungültig, sobald in der umgebenden Szene sich etwas geändert hat!
- Idee:
  - Rendere die Szene (typischerweise) 6x vom "Mittelpunkt" aus
  - Übertrage Framebuffer in Textur (unter Verwendung des passenden Mappings)
  - Render Szene nochmal vom Viewpoint aus, diesmal mit Environment-Mapping

→Multi-pass-Rendering

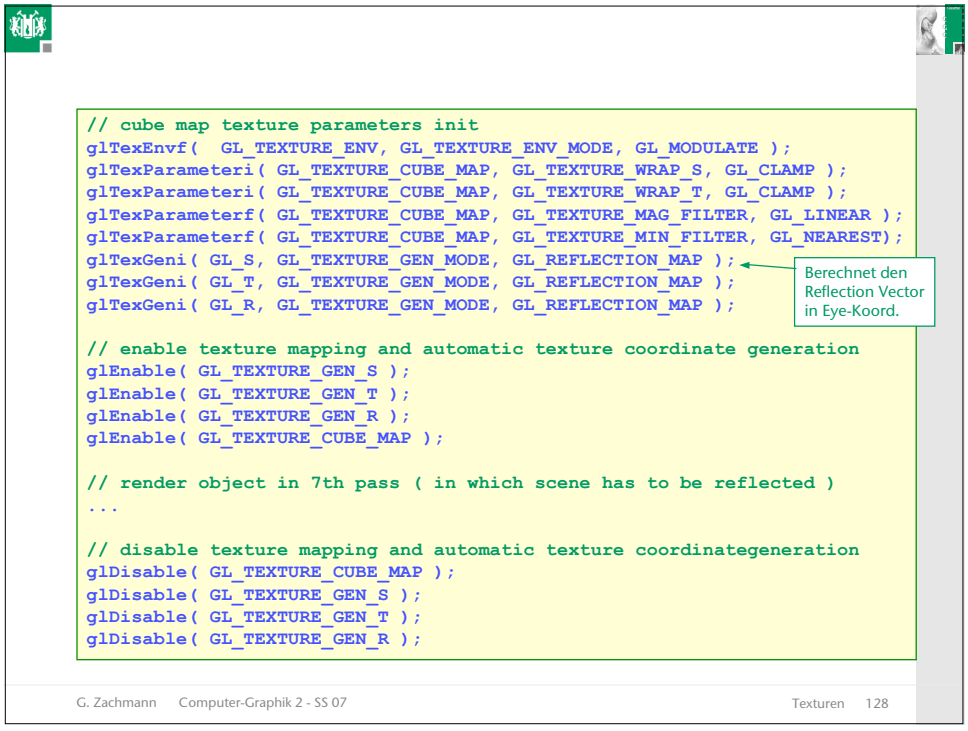
Dynamisches Environment Mapping in OpenGL mittels Cube Maps

```
GLuint cm_size = 512;   // texture resolution of each face
GLfloat cm_dir[6][3];   // direction vectors
float dir[6][3] = {
    1.0, 0.0, 0.0,      // right
   -1.0, 0.0, 0.0,      // left
    0.0, 0.0, -1.0,     // bottom
    0.0, 0.0, 1.0,      // top
    0.0, 1.0, 0.0,      // back
    0.0, -1.0, 0.0      // front
};
GLfloat cm_up[6][3] =   // up vectors
{   0.0, -1.0,  0.0,    // +x
    0.0, -1.0,  0.0,    // -x
    0.0, -1.0,  0.0,    // +y
    0.0, -1.0,  0.0,    // -y
    0.0,  0.0,  1.0,    // +z
    0.0,  0.0, -1.0     // -z
};
GLfloat cm_center[3];   // viewpoint / center of gravity
GLenum cm_face[6] = {
    GL_TEXTURE_CUBE_MAP_POSITIVE_X,
    GL_TEXTURE_CUBE_MAP_NEGATIVE_X,
    GL_TEXTURE_CUBE_MAP_NEGATIVE_Z,
    GL_TEXTURE_CUBE_MAP_POSITIVE_Z,
    GL_TEXTURE_CUBE_MAP_POSITIVE_Y,
    GL_TEXTURE_CUBE_MAP_NEGATIVE_Y
};
// define cube map's center cm_center[] = center of object
// (in which scene has to be reflected)
...
```



```
// set up cube map's view directions in correct order
for ( uint i = 0, i < 6; i + )
  for ( uint j = 0, j < 3; j + )
        cm_dir[i][j] = cm_center[j] + dir[i][j];

// render the 6 perspective views (first 6 render passes)
for ( unsigned int i = 0; i < 6; i ++ )
{
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glViewport( 0, 0, cm_size,  cm_size );
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  gluPerspective( 90.0, 1.0, 0.1, ... );
  glMatrixMode( GL_MODELVIEW );
  glLoadIdentity();
  gluLookAt( cm_center[0], cm_center[1], cm_center[2],
             cm_dir[i][0], cm_dir[i][1], cm_dir[i][2],
             cm_up[i][0],  cm_up[i][1],  cm_up[i][2] );
  // render scene to be reflected
  ...
  // read-back into corresponding texture map
  glCopyTexImage2D( cm_face[i], 0, GL_RGB, 0, 0, cm_size, cm_size, 0 );
}
```

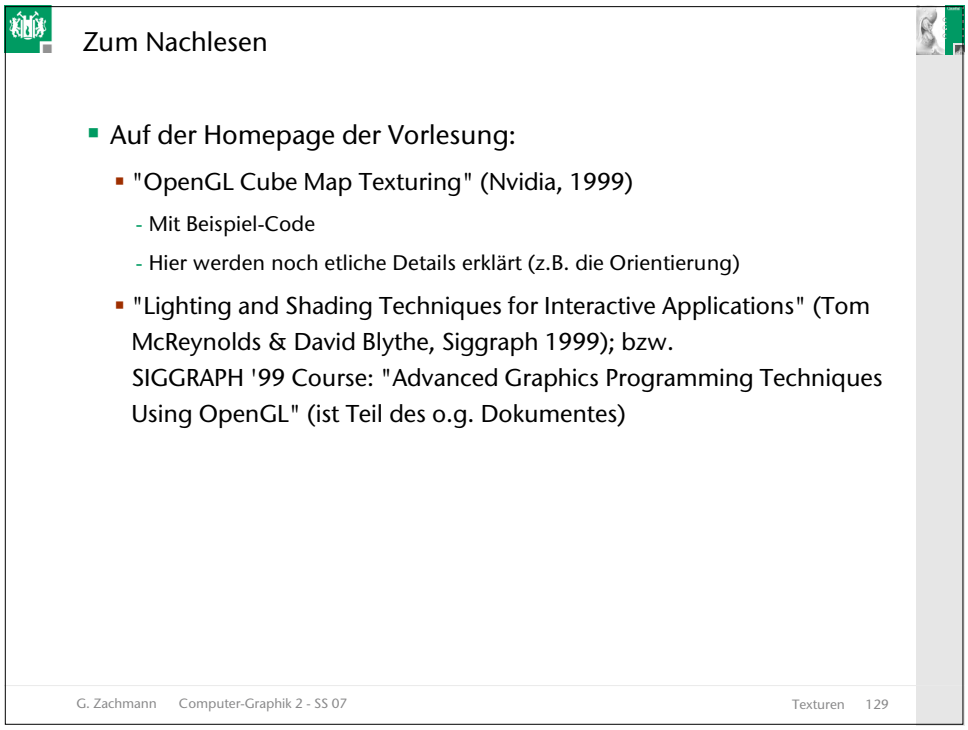
```
// cube map texture parameters init
glTexEnvf(  GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
glTexParameteri( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP );
glTexParameteri( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP );
glTexParameterf( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameterf( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP );
glTexGeni( GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP );
glTexGeni( GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP );

// enable texture mapping and automatic texture coordinate generation
glEnable( GL_TEXTURE_GEN_S );
glEnable( GL_TEXTURE_GEN_T );
glEnable( GL_TEXTURE_GEN_R );
glEnable( GL_TEXTURE_CUBE_MAP );

// render object in 7th pass ( in which scene has to be reflected )
...

// disable texture mapping and automatic texture coordinategeneration
glDisable( GL_TEXTURE_CUBE_MAP );
glDisable( GL_TEXTURE_GEN_S );
glDisable( GL_TEXTURE_GEN_T );
glDisable( GL_TEXTURE_GEN_R );
```

Berechnet den Reflection Vector in Eye-Koord.


Zum Nachlesen
Auf der Homepage der Vorlesung:
"OpenGL Cube Map Texturing" (Nvidia, 1999)
- Mit Beispiel-Code
- Hier werden noch etliche Details erklärt (z.B. die Orientierung)
"Lighting and Shading Techniques for Interactive Applications" (Tom McReynolds & David Blythe, Siggraph 1999); bzw.
SIGGRAPH '99 Course: "Advanced Graphics Programming Techniques Using OpenGL" (ist Teil des o.g. Dokumentes)
G. Zachmann Computer-Graphik 2 - SS 07 Texturen 129