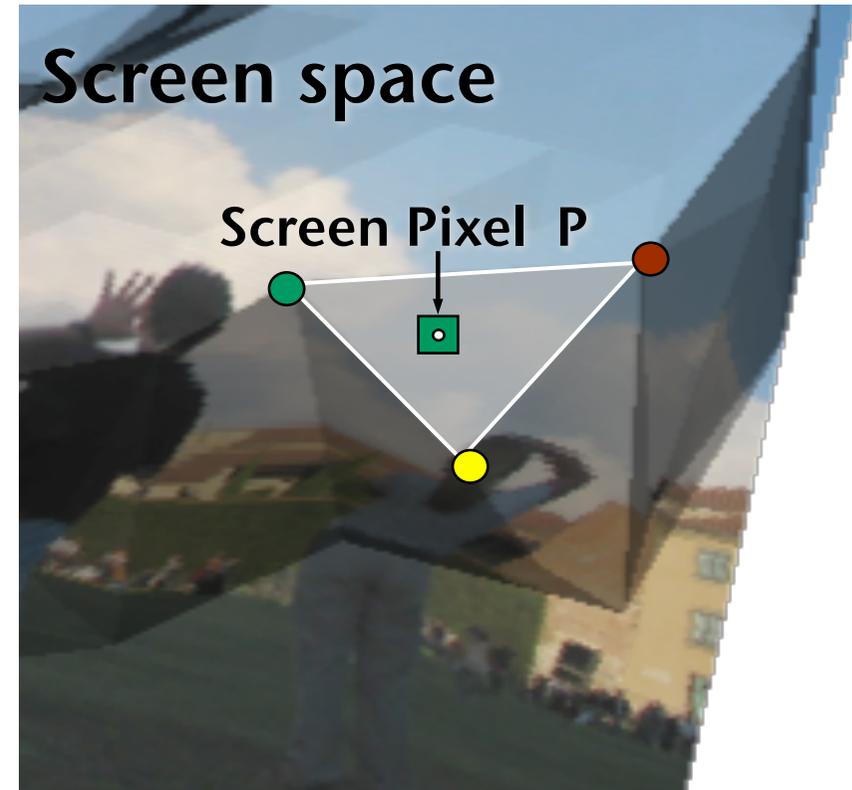
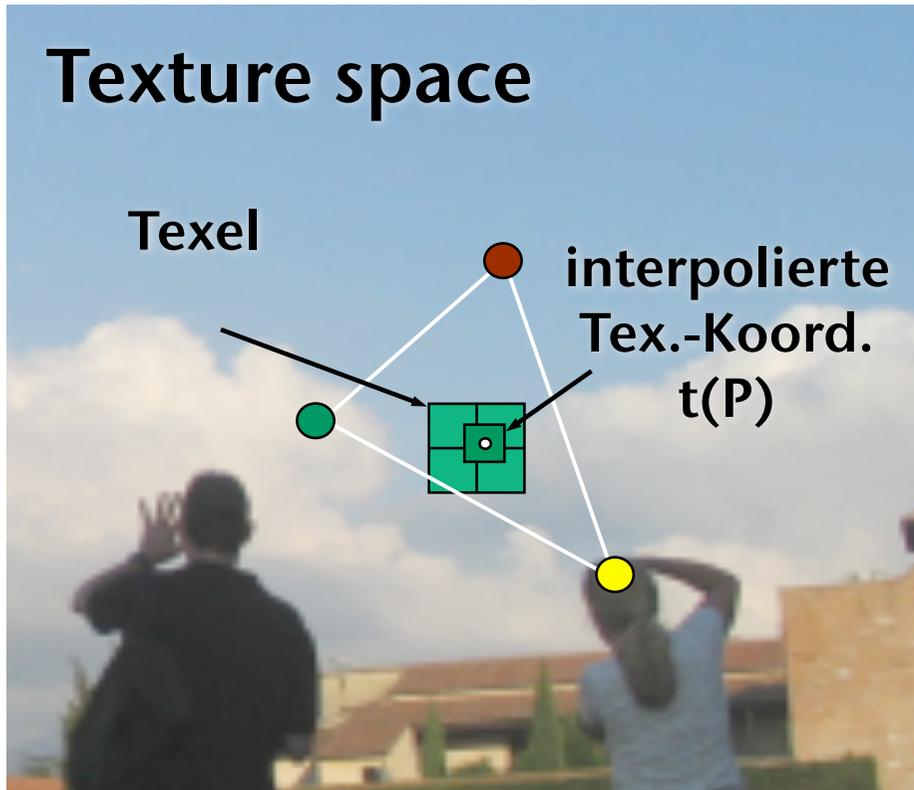




Texturinterpolation



- Nearest neighbour, oder
- Bilineare Interpolation der Texel

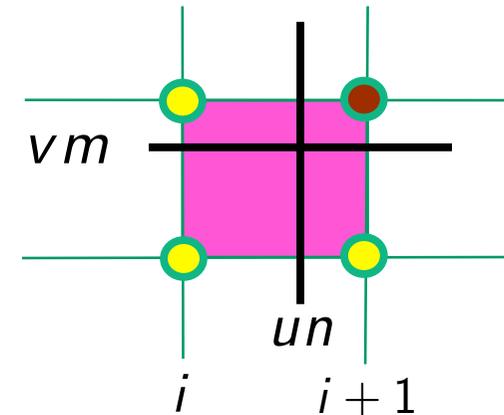


Rekonstruktionsmethoden

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

1. Nearest neighbour (Punktfilter):

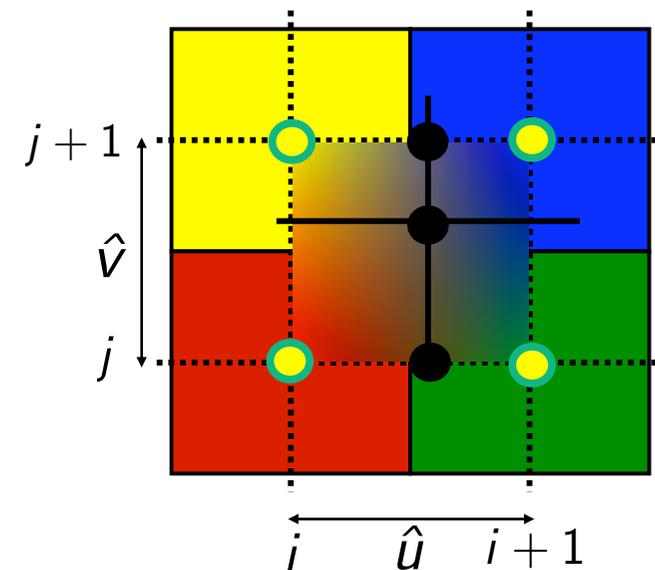
$$C_{\text{tex}} = C[\lfloor un \rfloor, \lfloor vm \rfloor]$$



2. Bilineare Interpolation:

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

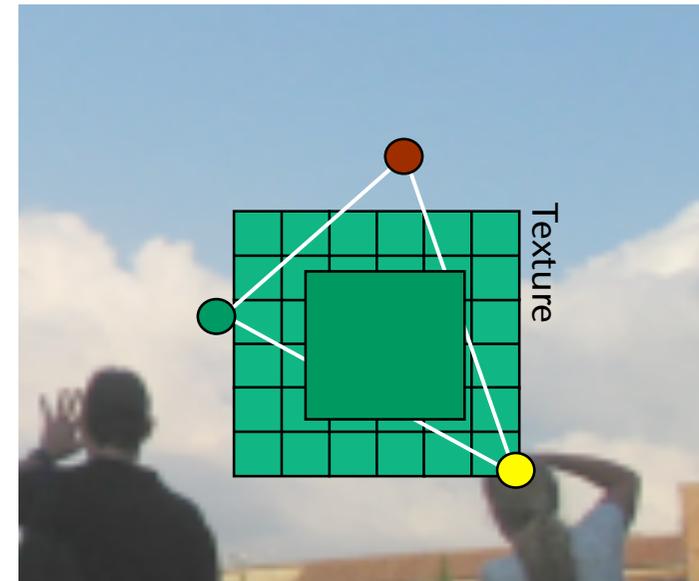
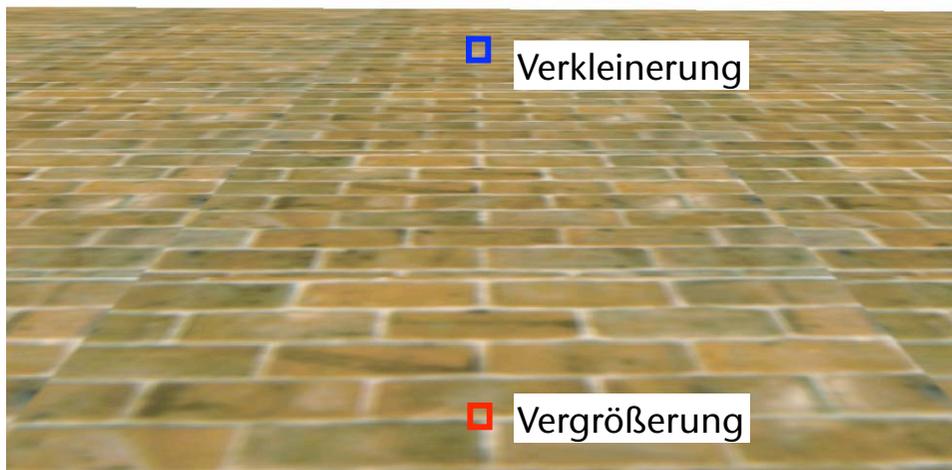
$$c = (1 - \hat{u})((1 - \hat{v}) \text{red} + \hat{v} \text{yellow}) + \hat{u}((1 - \hat{v}) \text{green} + \hat{v} \text{blue})$$





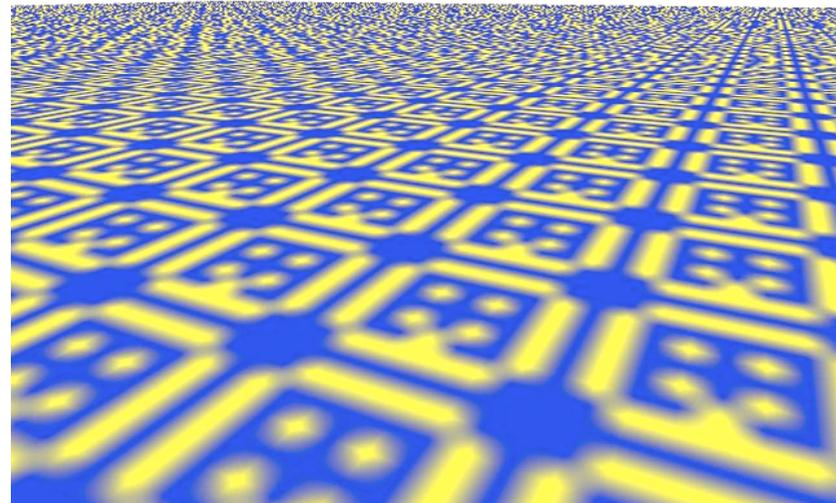
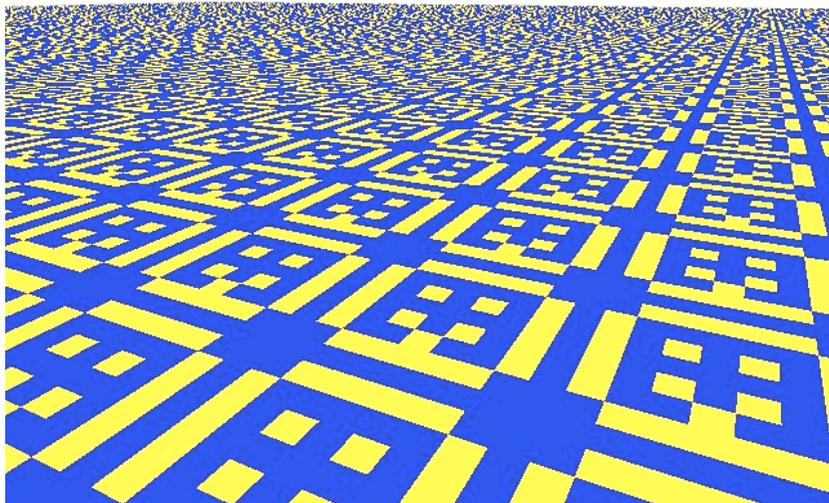
Texturverkleinerung

- Bilineare Interpolation OK, wenn Pixelgröße \leq 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 "heies" Problem
- Es gibt viele Mglichkeiten
- 1. Auch hier den einfachen Punktfiler → Aliasing
- 2. Lineare Interpolation hilft nur wenig

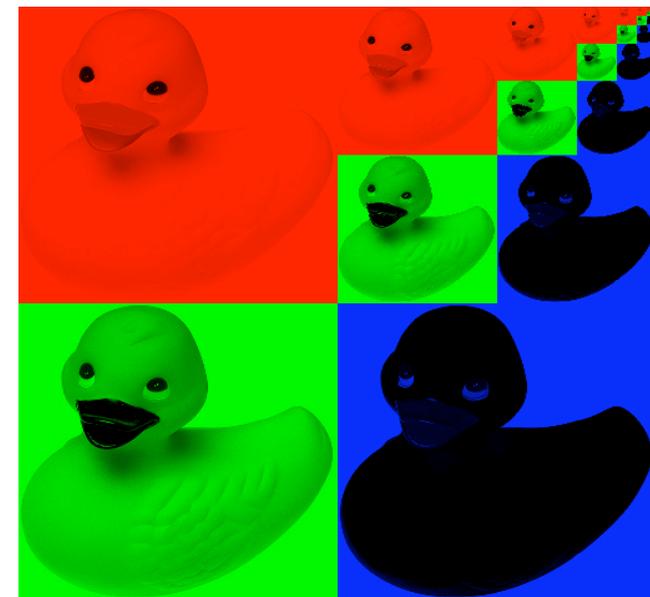
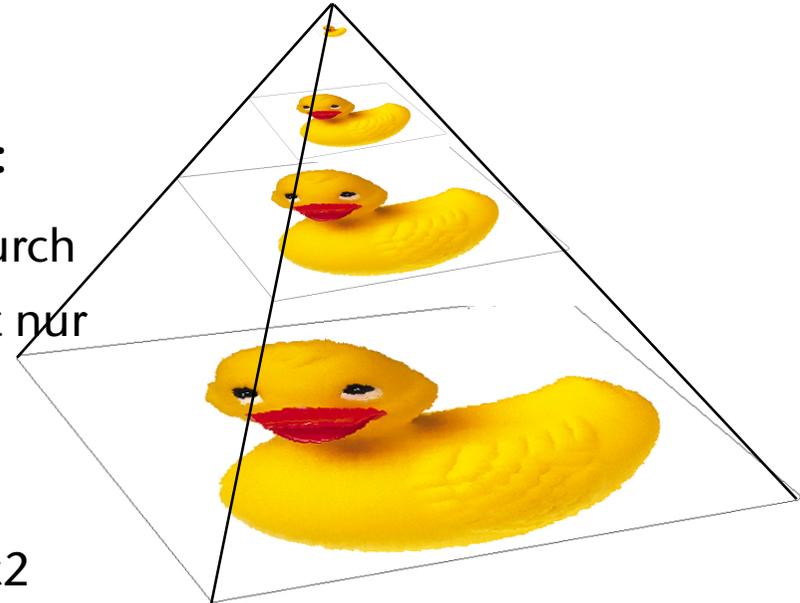




- Bei starker Verkleinerung müsste eigentlich eine Mittelung von vielen Texeln durchgeführt werden, da sie alle auf 1 Pixel auf dem Bildschirm abgebildet werden
 - Für Echtzeitanwendungen und Hardwarerealisierungen ist das zu aufwendig
 - Lösung: Preprocessing
 - Vor dem Start verkleinerte Versionen 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")

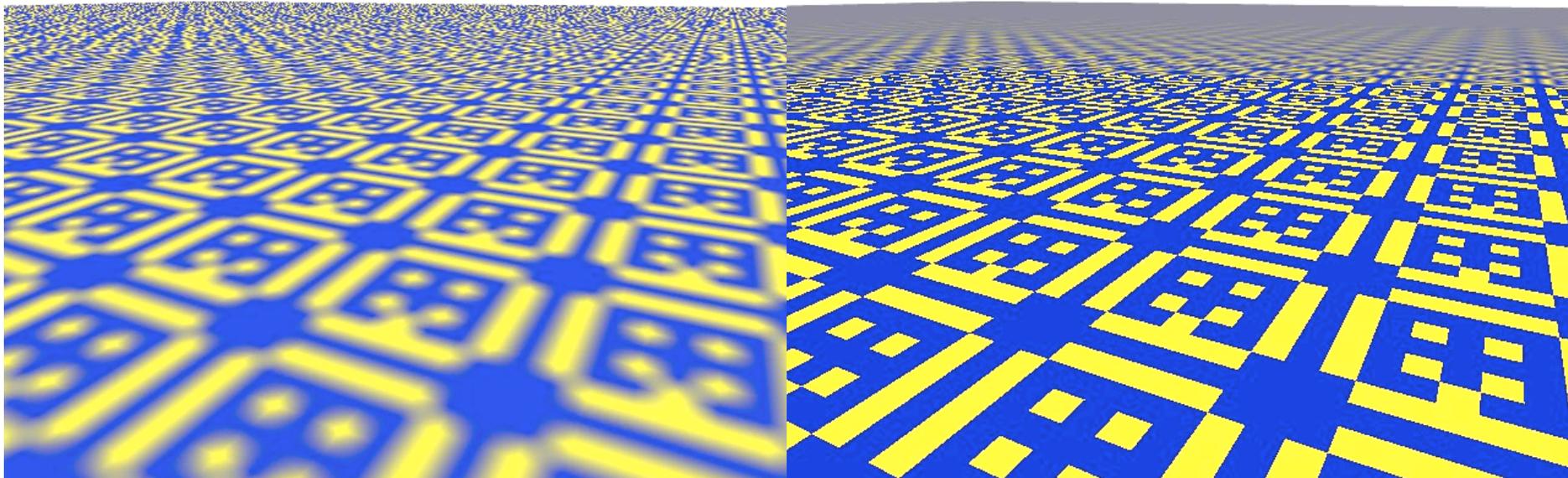


- Eine MIP-Map ist eine "Bild-Pyramide":
 - Jeder Level entsteht aus dem darunter durch Zusammenfassen mehrerer Pixel und hat nur die Größe $1/4$
 - Daher: orig. Bild muß $2^n \times 2^n$ groß sein!
 - Einfachste Art der Zusammenfassung: 2×2 Pixel mitteln
 - Oder: irgend einen anderen Bild-Filter anwenden
- Intern wird aus einem 2^n -Bild ein 2^{n+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



bilinear gefiltert

MIP-Map



Filterspezifikation in OpenGL



- Magnification:

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

- *param* = GL_NEAREST: Punktfiler
= 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 nächsten Mipmap-Levels,
dazwischen trilineare Interpolation



Mipmaps in OpenGL

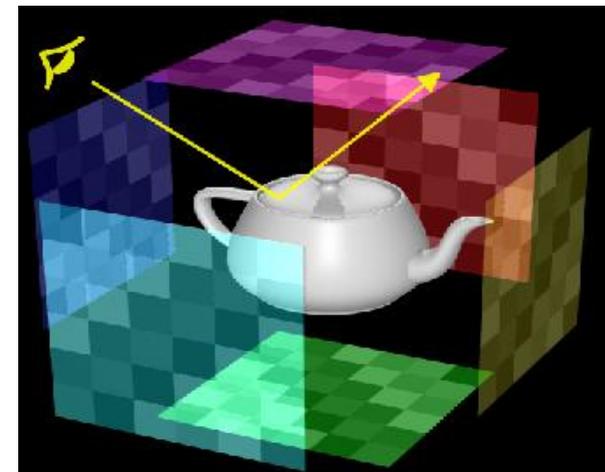
- Der `level` Parameter von `glTexImage2D` bestimmt, welcher Level der Mipmap gesetzt wird
- 0 ist die größte Map, jede weitere hat dann halbe Größe, bis hin zu 1x1
- Alle Größen müssen vorhanden sein
- Hilfsfunktion:

```
gluBuild{12}DMipmaps( target, components,  
                    width, [height,] format, type, data )  
mit Parametern wie glTexImage{12}D()
```



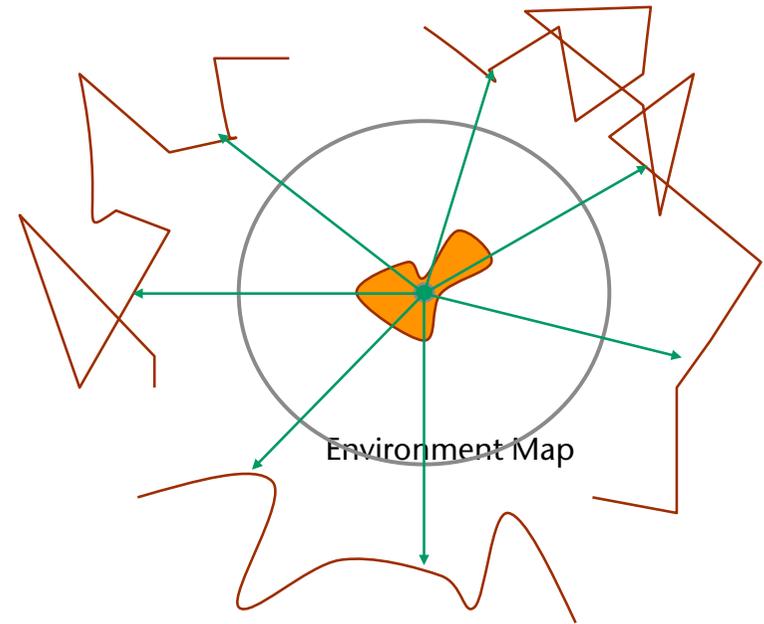
Environment Mapping

- 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**



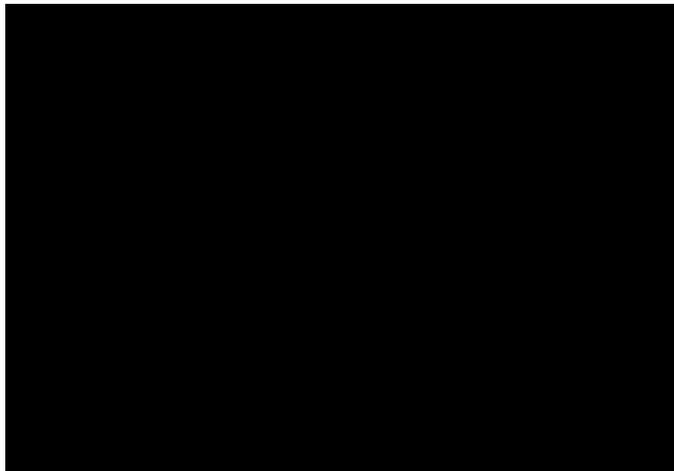


- Die Environment-Map speichert also für jede Raumrichtung die Lichtfarbe, die aus dieser Richtung in einem bestimmten Punkt eintrifft
- Stimmt natürlich nur für eine Position
- Stimmt nicht mehr, falls das Environment sich ändert





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

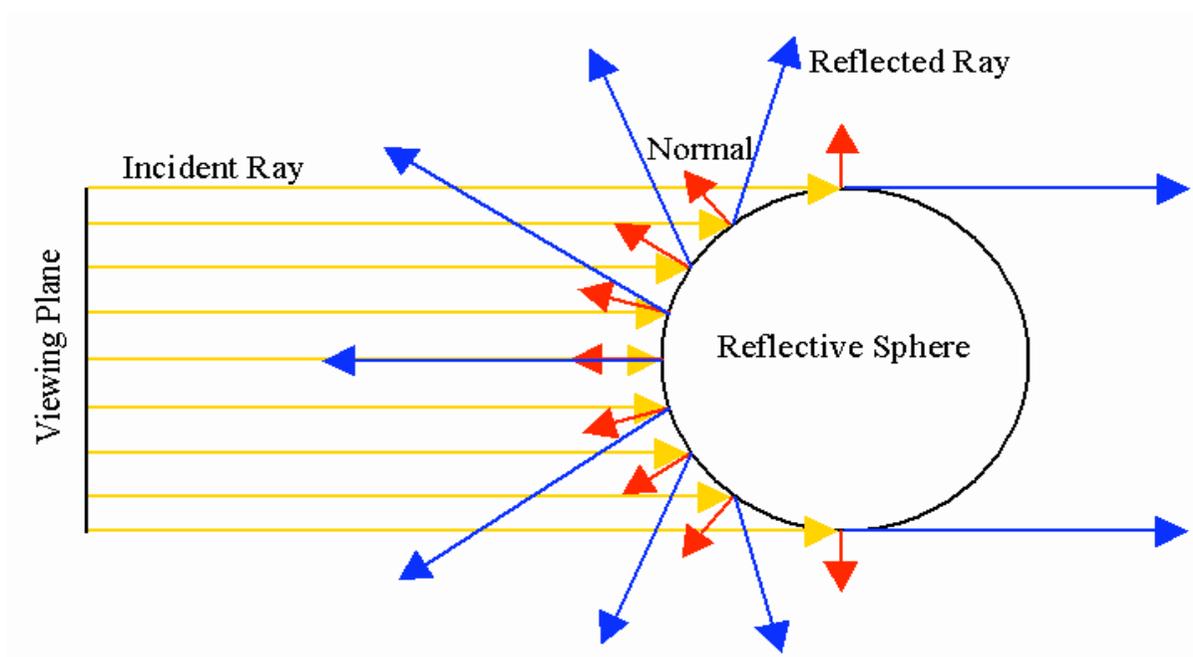
- Generiere oder lade eine 2D-Textur, die das Environment darstellt
- Für jedes Pixel des reflektierenden Objektes
 - Berechne die Normale \mathbf{n}
 - Berechne einen Reflexionsvektor \mathbf{r} aus \mathbf{n} und dem View-Vektor \mathbf{v}
 - Berechne Texturkoordinaten (u,v) aus \mathbf{r}
 - Färbe mit dem Texturwert das Pixel
- Das alte Problem: welche Parametrisierung?
 - Wie bildet man Raumrichtungen auf $[0,1] \times [0,1]$ ab?
- Gewünschte Eigenschaften:
 - 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

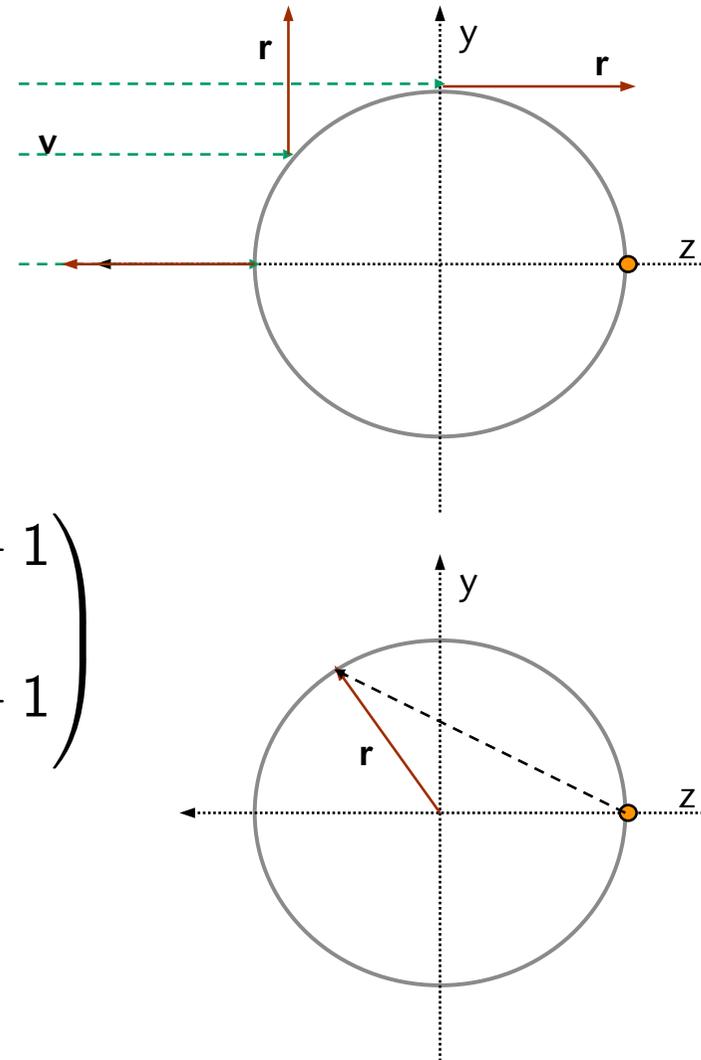
- Erzeugung der Environment-Map (= Textur):
 - Photographie einer spiegelenden Kugel; oder
 - Ray-Tracing der Szene mit spezieller "rotierender Kamera" und anschließendem Mapping





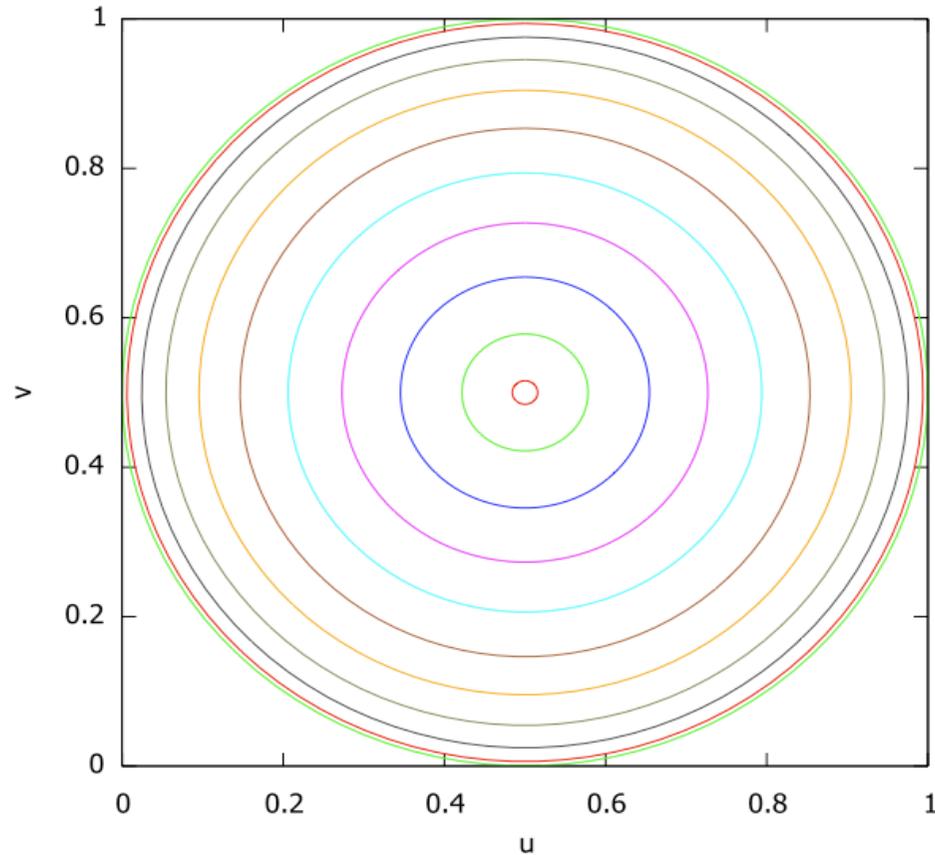
- Abbildung des Richtungsvektors \mathbf{r} auf (u, v) :
 - Die Sphere-Map enthält (theoretisch) einen Farbwert für **jede** Richtung, außer $\mathbf{r} = (0, 0, -1)$
 - Mapping:

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

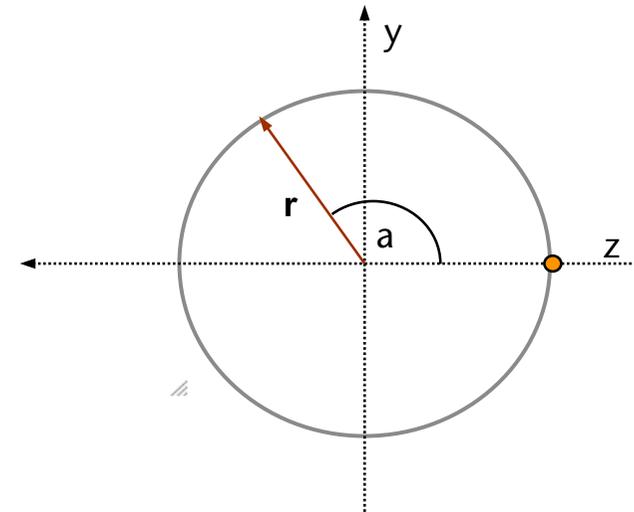




- Das Mapping ist leider sehr nicht-uniform:

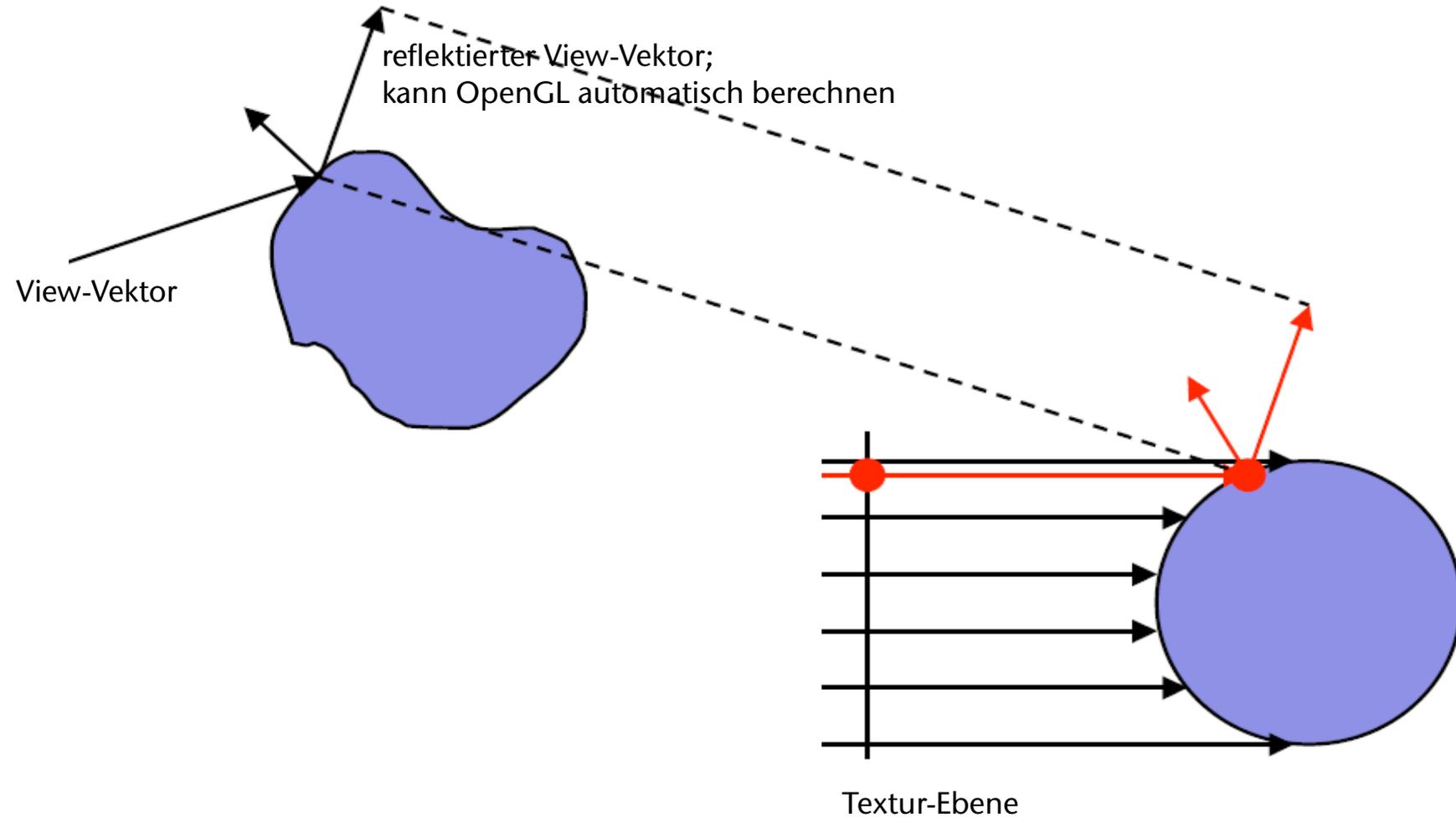


- a= 0.0 pi
- a= 0.1 pi
- a= 0.2 pi
- a= 0.3 pi
- a= 0.4 pi
- a= 0.5 pi
- a= 0.6 pi
- a= 0.7 pi
- a= 0.8 pi
- a= 0.9 pi
- a= 1.0 pi





- Anwendung der Sphere Map zur Texturierung:



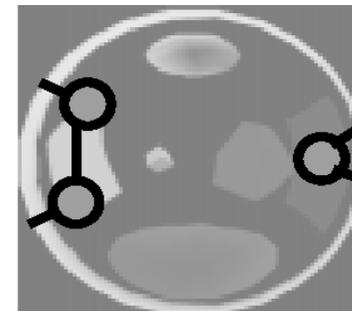


Einfaches Beispiel

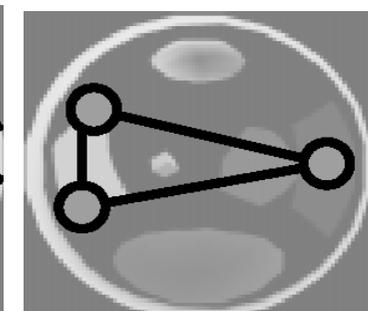




- Nachteile:
 - Maps (Texturen) sind schwierig per Computer zu erzeugen
 - Sehr nicht-uniformes Sampling
 - Nur halbwegs korrekt, wenn sich das reflektierende Objekt nahe am Ursprung (in View Space) befindet
 - Sparkles / speckles wenn der reflektierte Vektor in die Nähe des Randes der Textur kommt (durch 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)



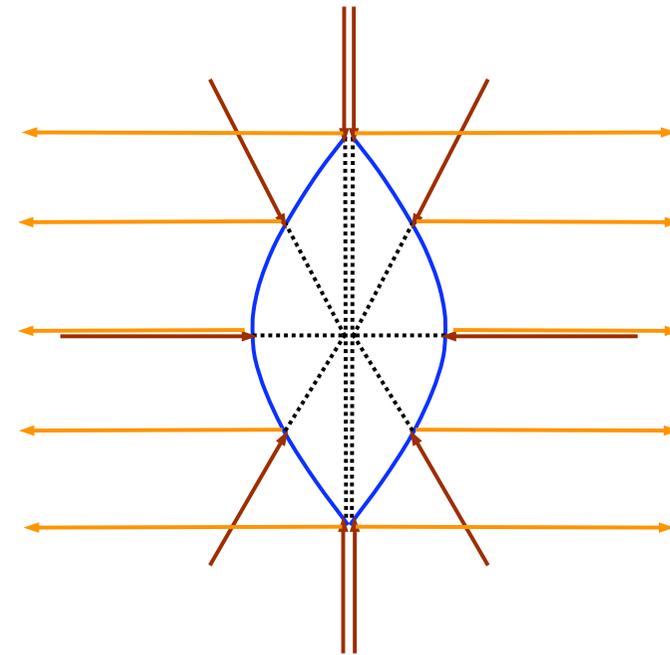
Parabolic Environment Mapping

[Heidrich'98]



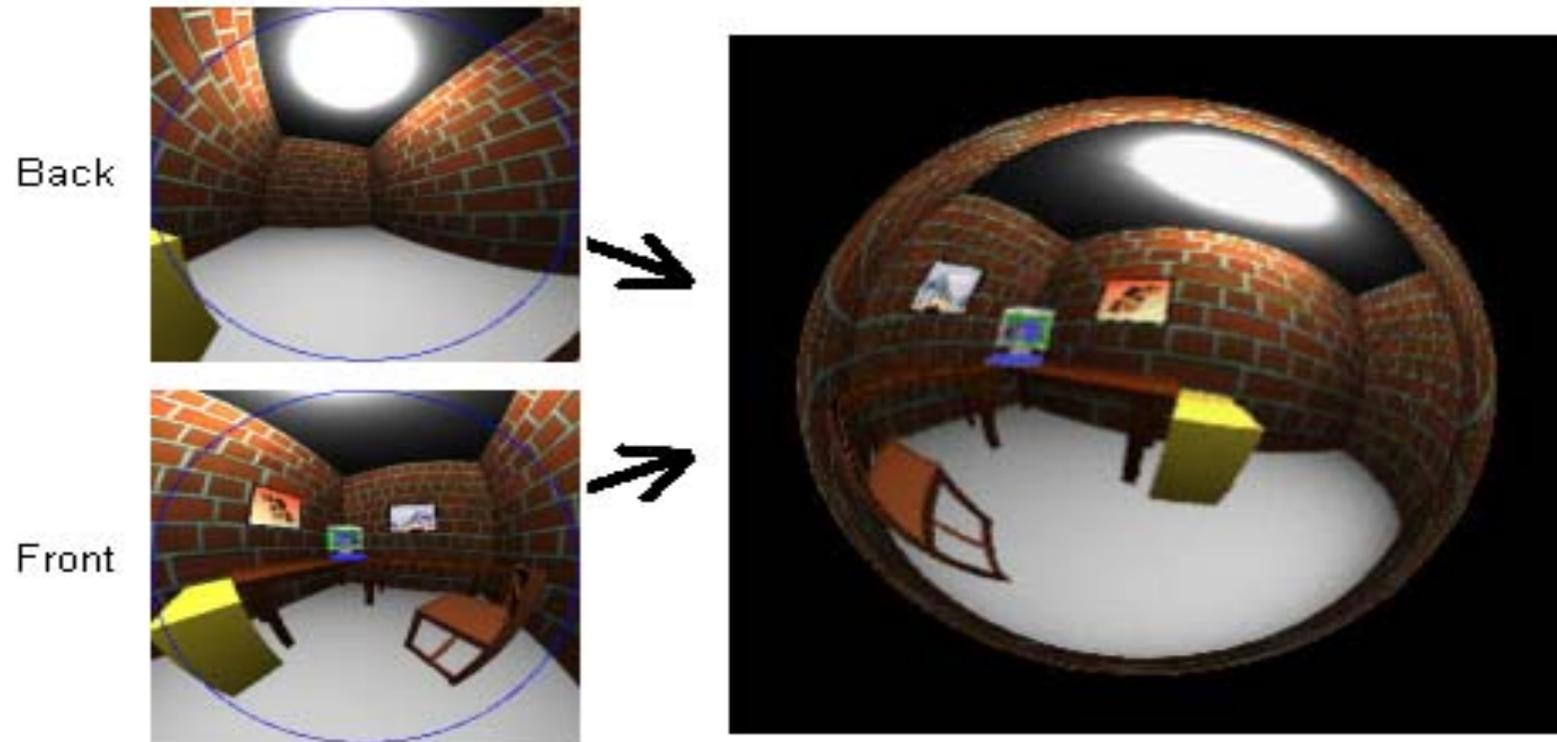
■ Idee:

- 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 Interpolation über die "Kante" hinweg





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





- Cube Map:

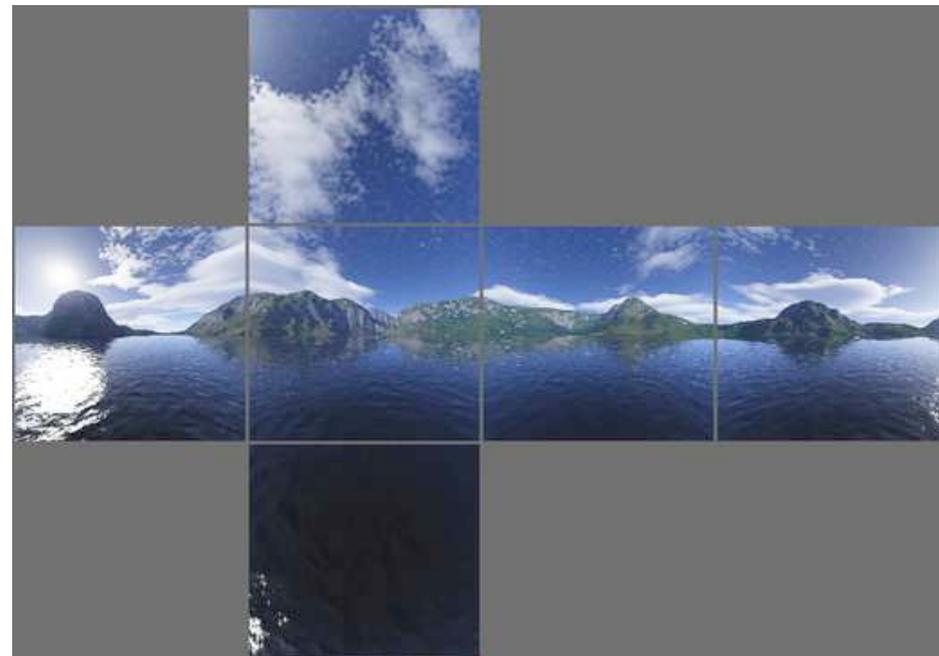
- Sechs Bilder von der Mitte eines Würfels durch seine Stirnflächen [Greene '86, Voorhies '94]

- 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



Tasten:

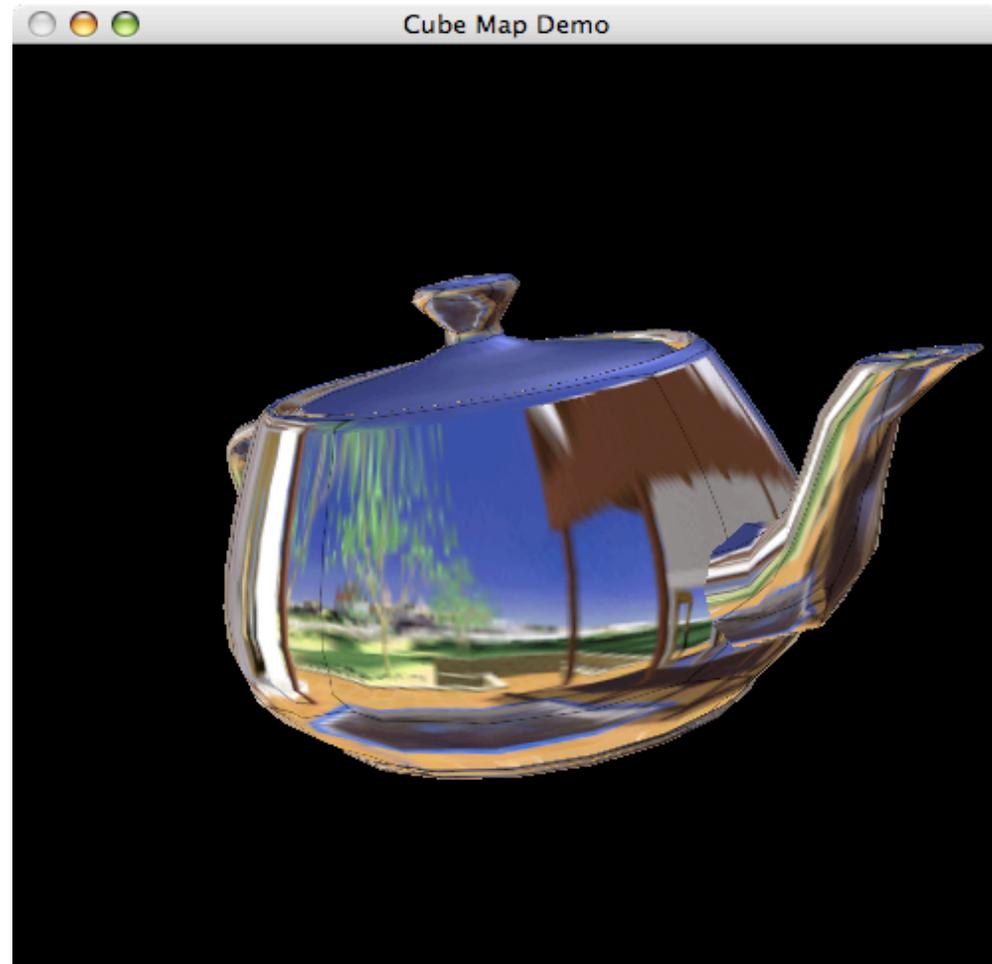
s = "shape"

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 );
glTexParameterf( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP );
glTexParameterf( 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 coordinate generation
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.



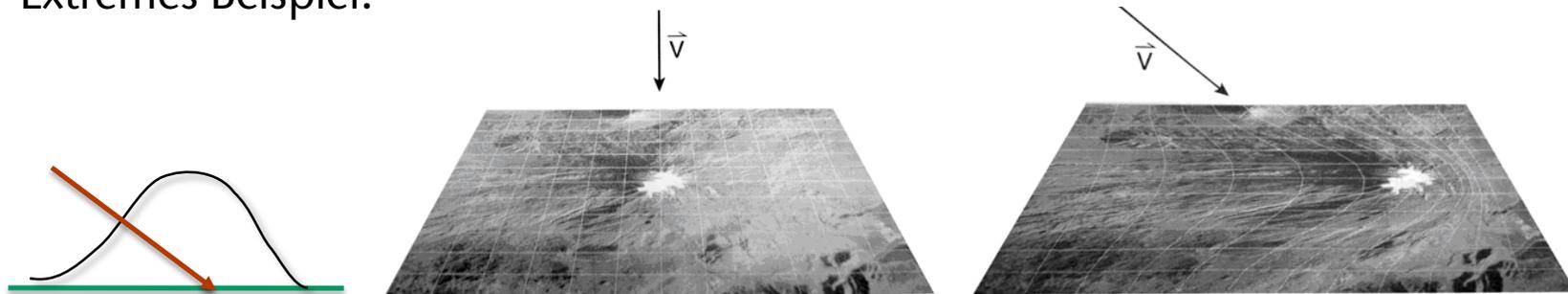
Zum Nachlesen

- 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)



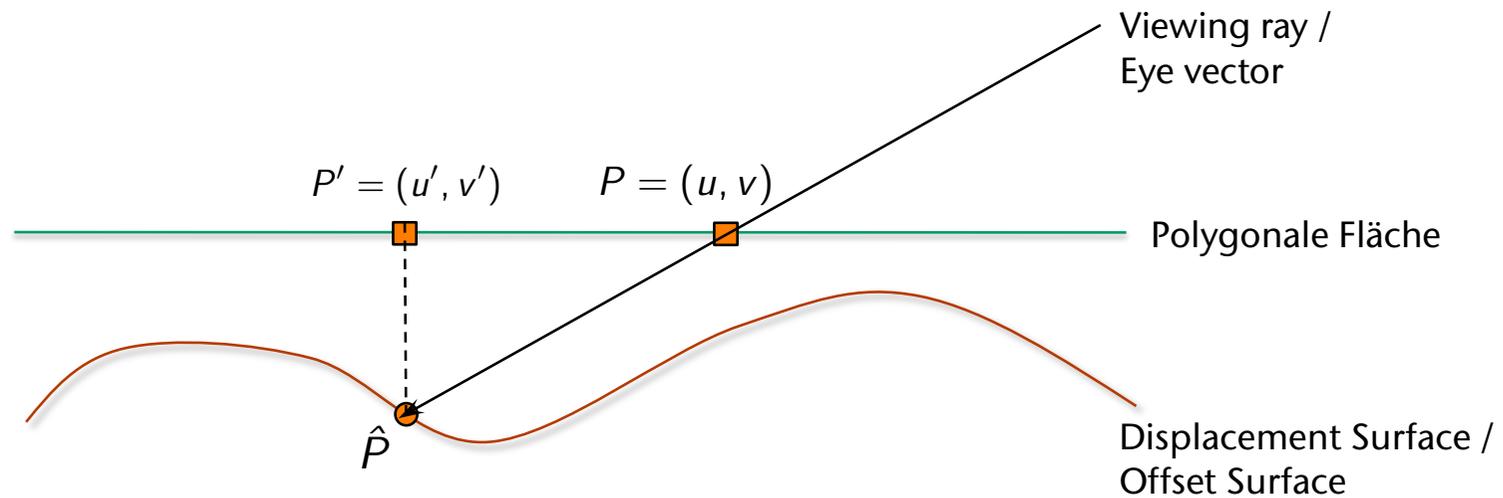
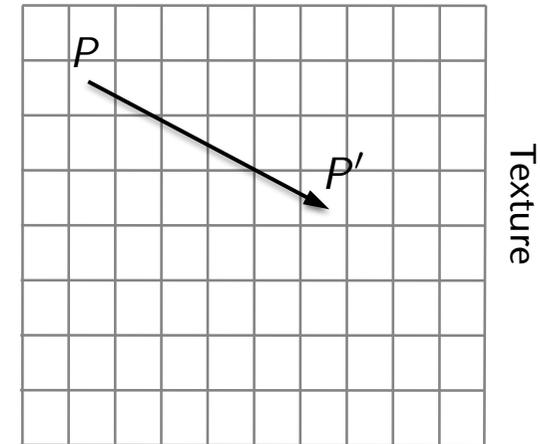
Parallax Mapping

- Problem des Bump- / Normal-Mapping:
 - Nur das Lighting wird beeinflusst – das Bild der Textur bleibt unverändert, egal aus welcher Richtung man schaut
 - Bewegungsparallaxe: nahe / entfernte Objekte verschieben sich verschieden stark relativ zueinander (oder sogar in verschiedene Richtung! je nach Fokussierungspunkt)
 - Extremes Beispiel:





- Idee des Parallax Mapping:
 - Scan-Line-Conversion steht bei P
 - Bestimme \hat{P}
 - Projiziere \hat{P} auf P'
 - Schreibe das zugehörige Texel als Farbe
- Problem: wie findet man P' ?



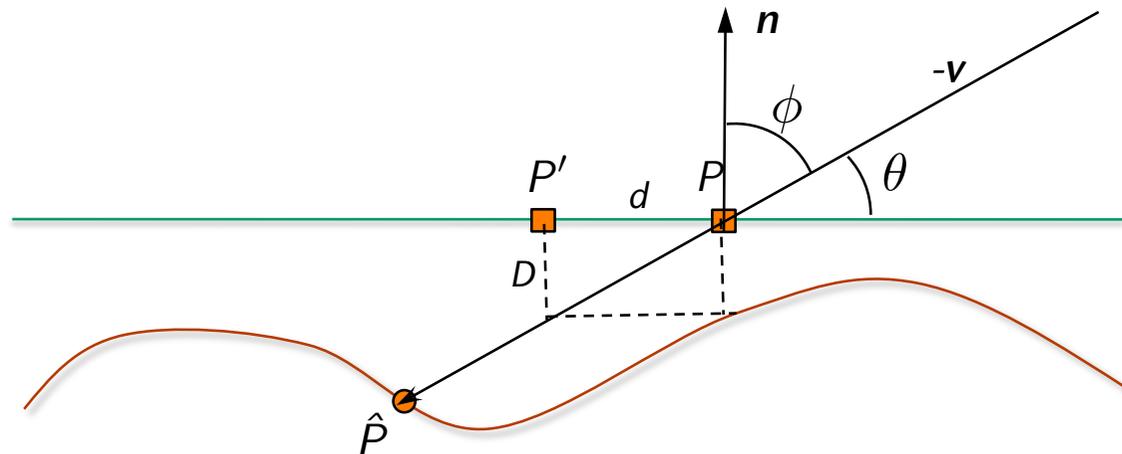


■ Einfachste Idee:

[Kaneko et al., 2001]

- Man kennt die Höhe in $P = D(u, v)$
- Verwende diese als Näherung für $D(u', v')$

$$\frac{D}{d} = \tan \theta = \frac{\sin \theta}{\cos \theta} = \frac{\cos \phi}{\sin \phi} = \frac{|\mathbf{n}\mathbf{v}|}{|\mathbf{n} \times \mathbf{v}|} .$$



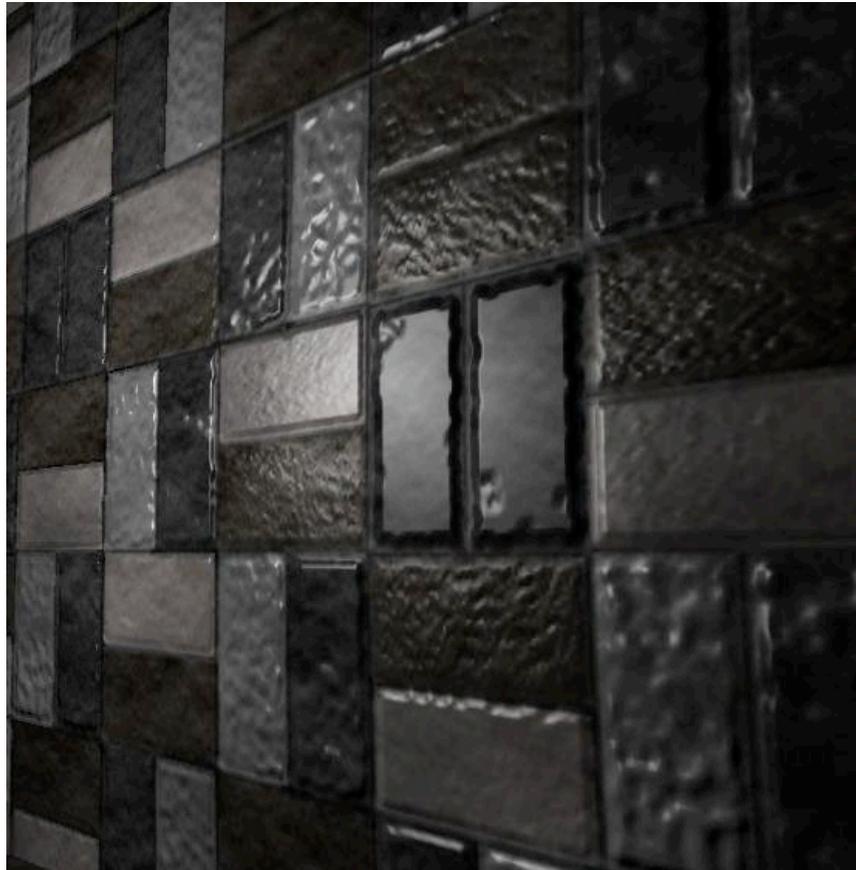


- Speicherung:
 - Das eigtl Bild in RGB der Textur
 - Höhenfeld im Alpha-Kanal
- NB: Richtungsableitungen für D_U und D_V (zur Perturbation der Normale) kann man heute "on the fly" ausrechnen

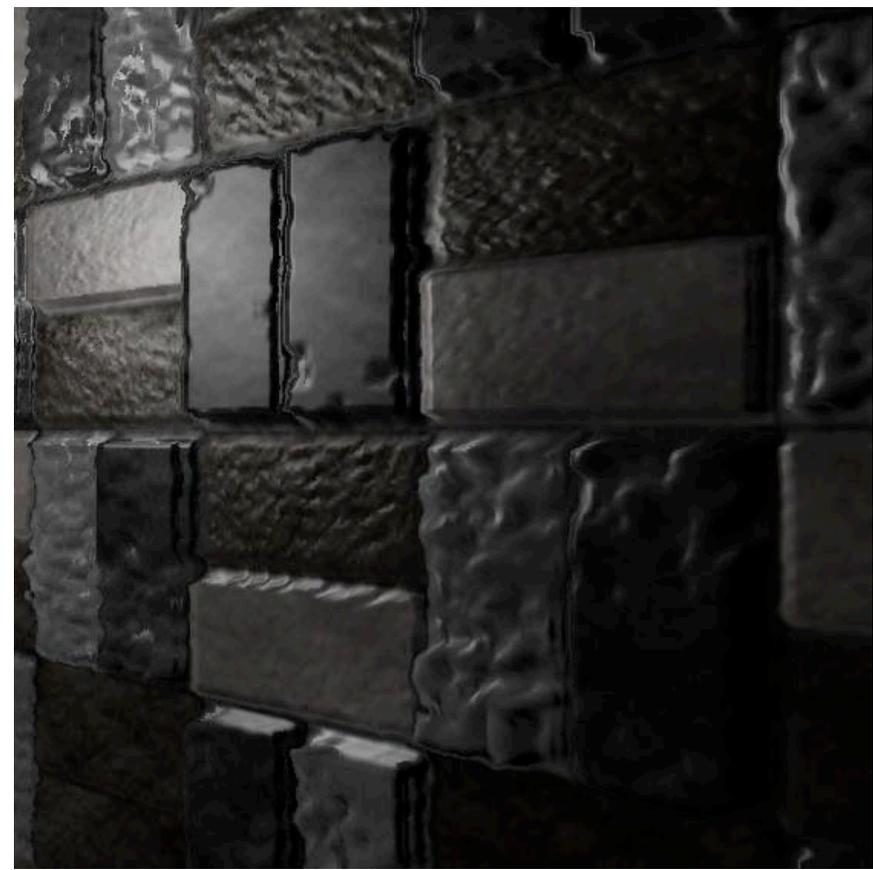




Beispiel



Normales Bump Mapping



Parallax Mapping
(Parallaxe wurde hier zur
Demonstration stark übertrieben)



■ Verbesserung:

[Premecz, 2006]

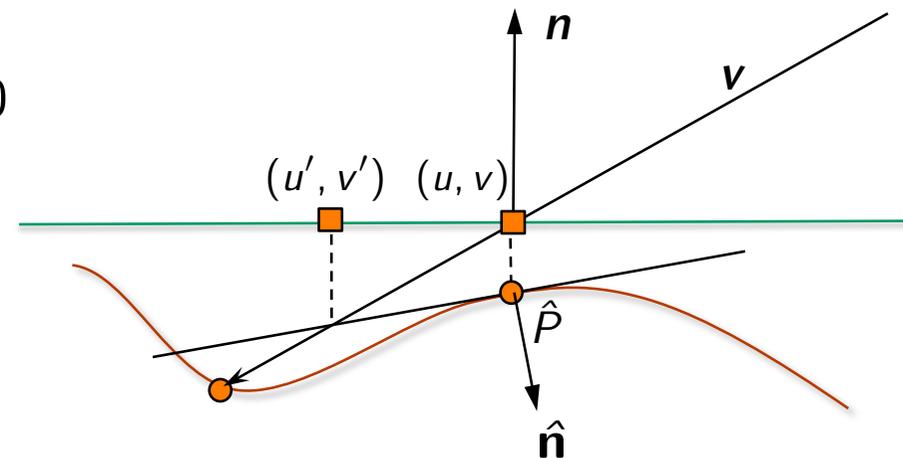
- Approximiere das Höhenfeld in $\hat{P} = (u, v, h)$ durch eine Ebene
- Berechne Schnittpunkt zwischen Ebene und View-Vektor

■ $h = D(u, v)$,

$$\mathbf{n} \left(\begin{pmatrix} u \\ v \\ 0 \end{pmatrix} + t\mathbf{v} - \begin{pmatrix} u \\ v \\ h \end{pmatrix} \right) = 0$$

■ Weiterführende
(naheliegende) Ideen:

- Iterieren
- Höhere Approximation des Höhenfeldes



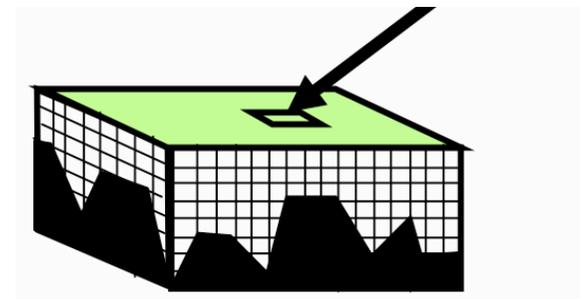
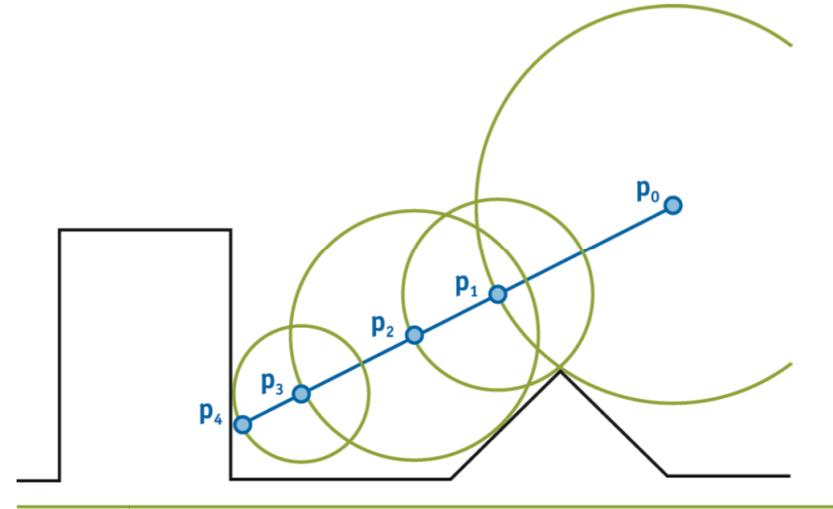
Diplomarbeit ...



Alternative

[Donnelly, 2005]

- Mache Sphere-Tracing entlang des View-Vektors, bis man die Offset-Fläche trifft
 - Falls die Height-Map nicht zu große Höhen enthält, genügt es, relativ dicht unterhalb/oberhalb der Referenzfläche zu beginnen
 - Falls der View-Vektor nicht zu "flach" liegt, genügen wenige Schritte
- Speichere für eine Schicht unterhalb der Referenzfläche für jede Zelle den kleinsten Abstand zur Offset-Fläche



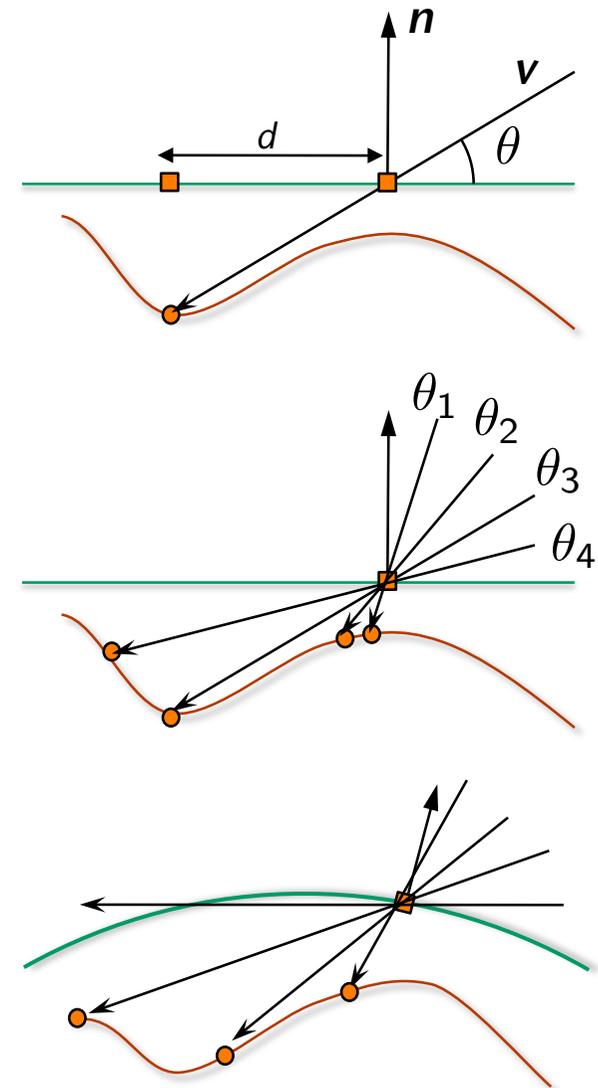


View-Dependent Displacement Mapping

[2003]

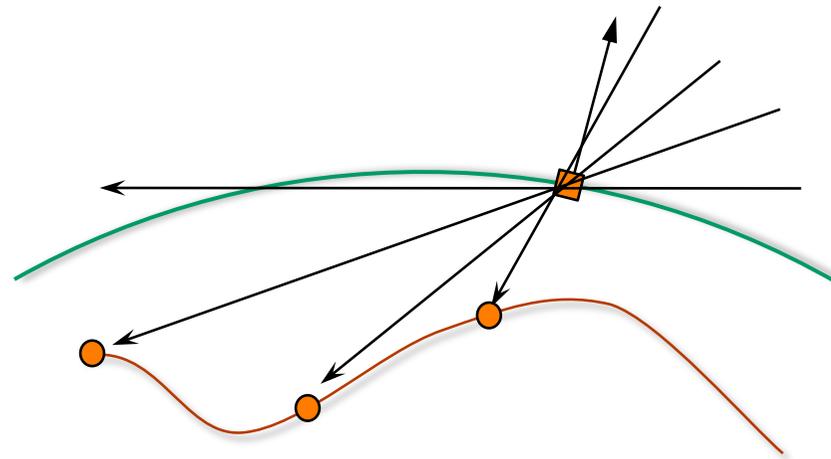
- Idee: berechne alle möglichen Texture-Koordinaten-Displacements für alle möglichen Situationen vor
- Konkret:
 - Parametrisiere den Viewing-Vektor durch θ und ϕ im lokalen Koord.system des Pgons
 - Berechne für alle (u,v) und ein bestimmtes (θ, ϕ) das Textur-Displacement vor
 - Ray-Casting eines explizit temporär generierten Meshes
 - Führe dies für alle möglichen (θ, ϕ) durch
 - Führe das Ganze für eine Reihe von möglichen Krümmungen c der (groben) Oberfläche durch
 - Ergibt eine 5-dim. Textur (LUT):

$$d(u, v, \theta, \phi, c)$$





- Vorteil: ergibt korrekte Silhouette
 - Denn: bei manchen Parametern liefert $d(u, v, \theta, \phi, c) = -1$
 - Das sind genau die Pixel, die außerhalb der aktuellen Silhouette liegen!





- Weitere Erweiterung: Self Shadowing (Selbst-Abschattung)

- Idee wie beim Ray-Tracing: "Schatten-Strahl"

1. Bestimme \hat{P} aus d und θ, ϕ

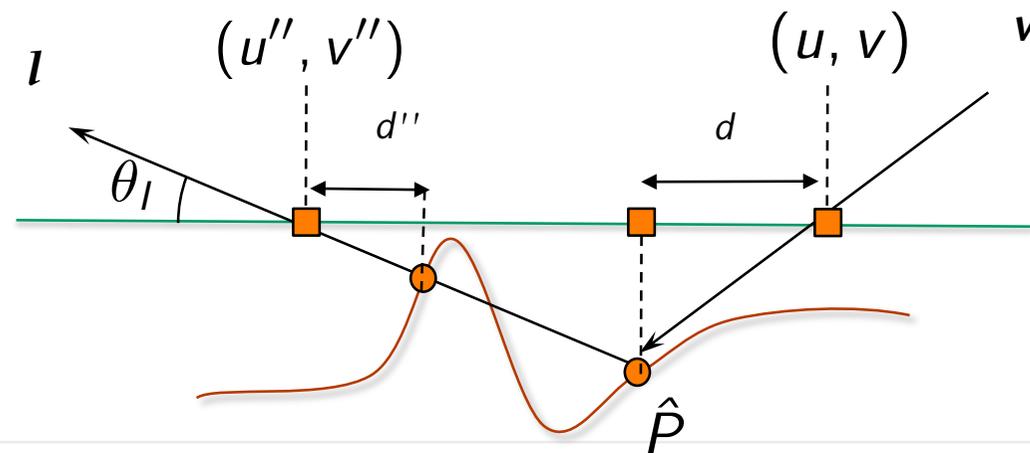
2. Bestimme Vektor \mathbf{l} von \hat{P} zur Lichtquelle; und daraus θ_l und ϕ_l

3. Bestimme $P'' = (u'', v'')$ aus \hat{P} und θ_l und ϕ_l

4. Indiziere damit d

5. Test: $d(u'', v'', \theta_l, \phi_l, c) < d(u, v, \theta, \phi, c)$

→ Pixel ist im Schatten





■ **Resultat:**



Bump Mapping



Displacement Mapping

■ **Namen ("... sind Schall und Rauch!"):**

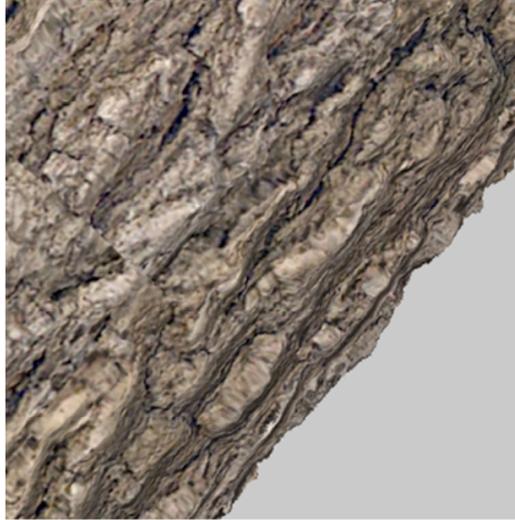
- Steep parallax mapping, parallax occlusion mapping, horizon mapping, view-dependent displacement mapping, ...
- Es gibt noch viele weitere Varianten ...



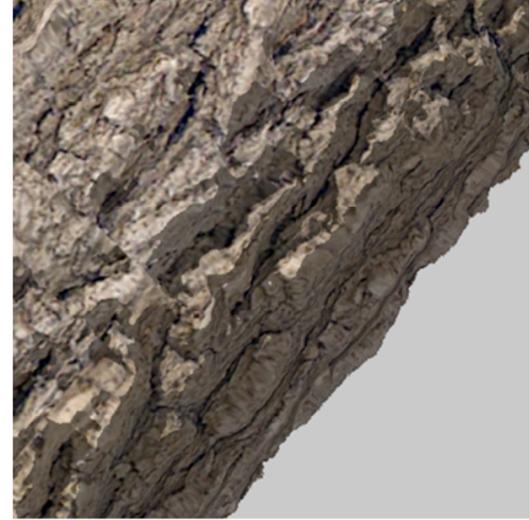
Resultate



Bump mapping



Einfaches Displacement Mapping



View-dependent displacement mapping mit self-shadowing



Alle Beispiele sind mit VDM gerendert

