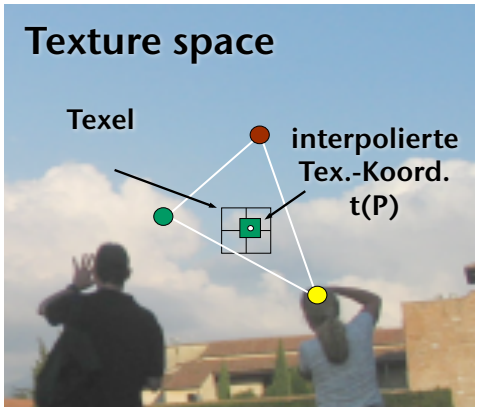
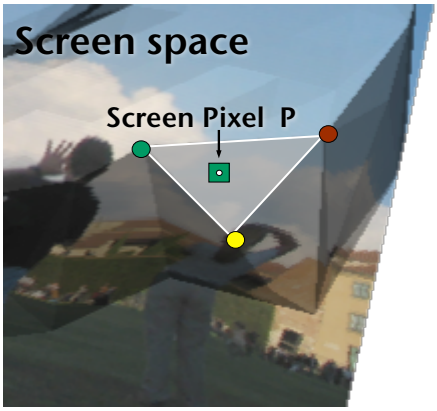
Textur-Interpolation



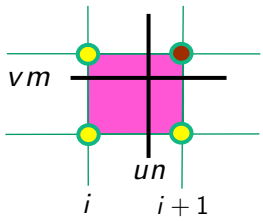


- Nearest neighbour, oder
- Bilineare Interpolation der Texel

Rekonstruktionsmethoden

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

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



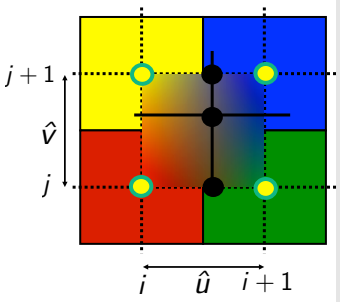
1. Nearest neighbour (Punktfilter):

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

2. Bilineare Interpolation:

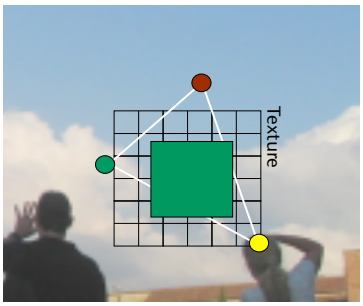
$$\hat{u} = un - \lfloor un \rfloor \ , \ \hat{v} = vm - \lfloor vm \rfloor$$

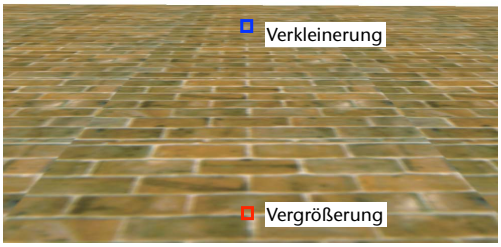
$$c = (1-\hat{u})\big((1-\hat{v})\blacksquare + \hat{v}\ \blacksquare\big) + \hat{u}\big((1-\hat{v})\blacksquare + \hat{v}\ \blacksquare\big)$$

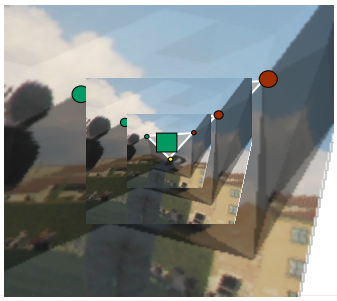


Texturverkleinerung

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

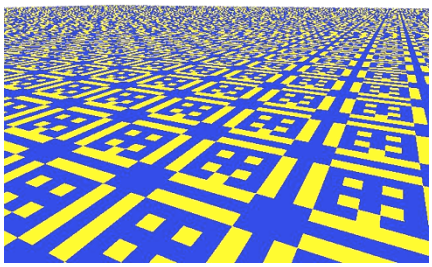


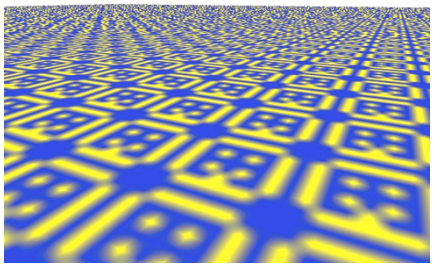




- Schwierigeres und "heißes" Problem
- Es gibt viele Möglichkeiten zur Lösung

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

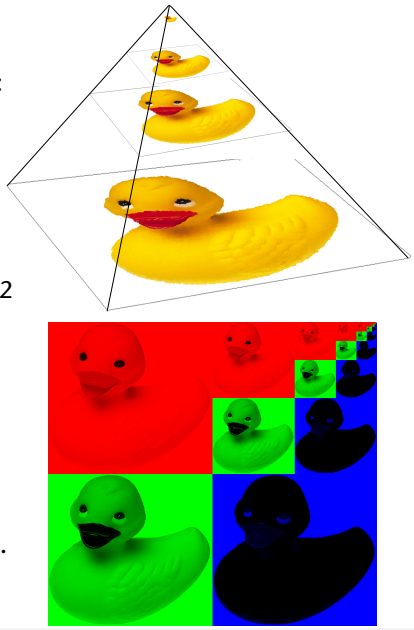




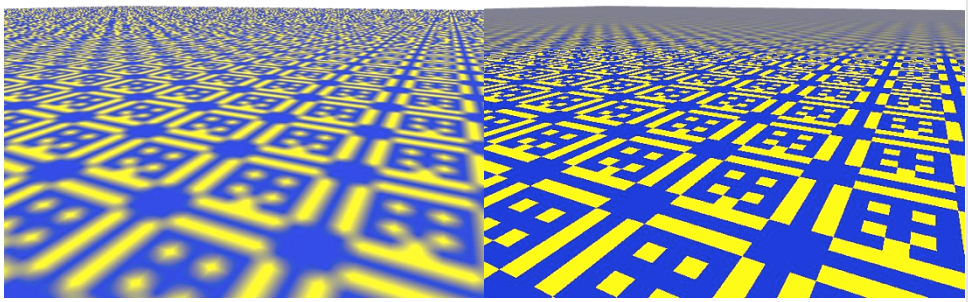
- Bei starker Verkleinerung müsste eigentlich eine Mittelung von vielen Texeln durchgeführt werden, da sie alle auf 1 Pixel auf dem Bildschirm abgebildet werden
- Für Echtzeitanwendungen und Hardwarerealisierungen ist das zu aufwendig
- Lösung: Preprocessing
  - Vor dem Start verkleinerte Versionen der 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^nx2^n groß sein!
  - Einfachste Art der Zusammenfassung: 2x2 Pixel mitteln
  - Oder: irgend einen anderen Bild-Filter anwenden
- Intern wird ein 2^n-Bild in einem 2^{n+1}-Bild gespeichert
- MIP-Map hat Speicherbedarf 1.3x Orig.



- 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

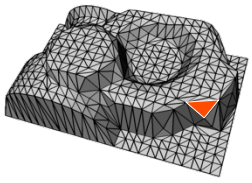
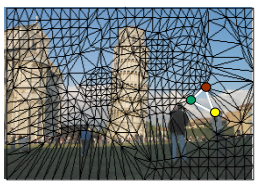
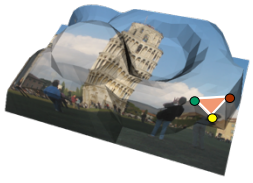
- Der **`level`** Parameter von **`glTexImage2D`** bestimmt, welcher Level der Mipmap gesetzt wird
- 0 ist die größte Map, jede weitere hat dann halbe Größe, bis hin zu 1x1
- Alle Größen müssen vorhanden sein
- Hilfsfunktion:

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

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

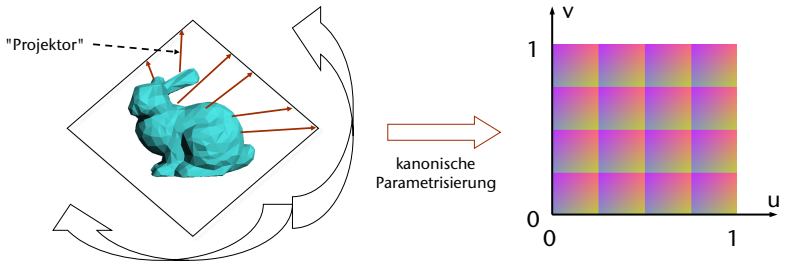
Einfache Parametrisierung

- Wie kommt man zu den Texturkoordinaten an jedem Vertex?
- Triviale Texturierung eines Terrains:
  - 3D-Koordinaten nach unten projzieren

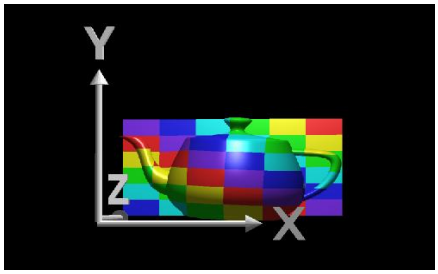
  - Achtung: dies ist nicht notwendig eine "gute" Texturierung!

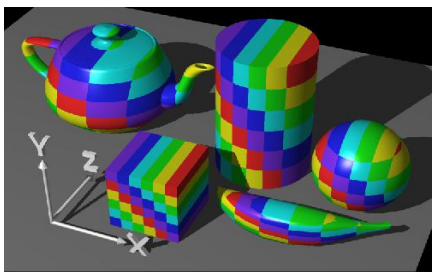
- Idee: ein 2-stufiger Prozess [Bier & Sloan, 1986]
  - Lege (konzeptionell) einen "kanonisch" parametrisierbaren Hüllkörper um das ganze Objekt
  1. Projiziere Vertices (nicht notw. dessen Vertex-Koord.!) auf diesen Hüllkörper
  2. Verwende die Texturkoordinaten des projizierten Punktes auf dem Hüllkörper



Einige Hüllkörper und deren Parametrisierung

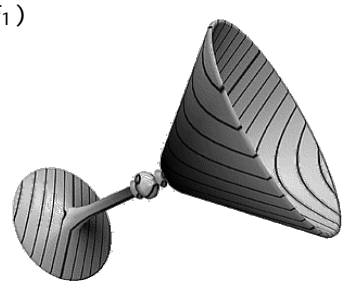
- Ebene:
  - Projiziere Punkt (x,y,z) auf Ebene $\rightarrow (x,y)$
  - $(u,v) = (s_x x + t_x, s_y y + t_y)$
- Verallgemeinerung:
  - Definiere 2 beliebige Ebenen E_1 und E_2
  - u := dist(P, E_1)
    v := dist(P, E_2)
  - Dieses Feature bietet OpenGL





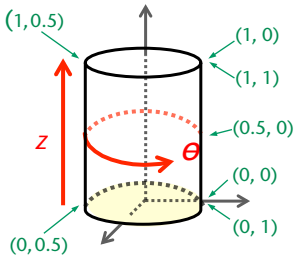
Beispiel

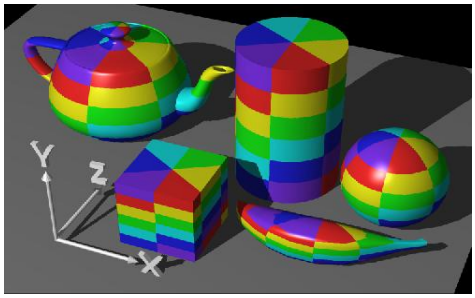
- Erzeuge Höhenlinien mittels dieser Technik:
  - 1D-Textur
  - u := dist(P, E_1)



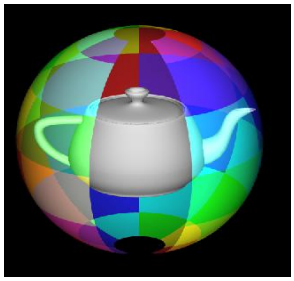
- Viele weitere ungewöhnliche Anwendungen von Texture-Mapping auf http://www.graficaobscura.com/texmap/index.html

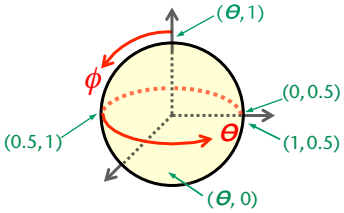
- Zylinder-Parametrisierung:
  - Konvertiere kartesische Koord. (x,y,z) in zylindrische Koord. $\rightarrow (r \sin \Theta, r \cos \Theta, z)$
  - $(u, v) = (\Theta/2\pi, z)$
- Beachte "Naht" bei (θ=0 & θ=2π)

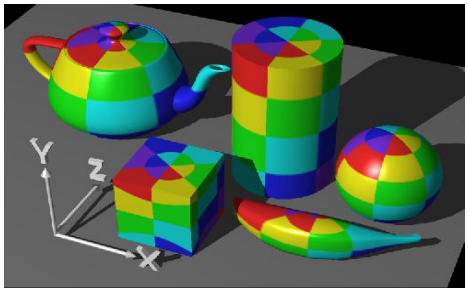




- Kugel-Parametrisierung:
  - Stelle Punkt in sphärischen Koord. dar
    → $r \cdot (\sin\theta \sin\phi, \cos\theta \sin\phi, \cos\phi)$
  - $(u, v) = (\theta/2\pi, \phi/\pi + 1)$
- Beachte: Singularität bei Nord- und Südpol!

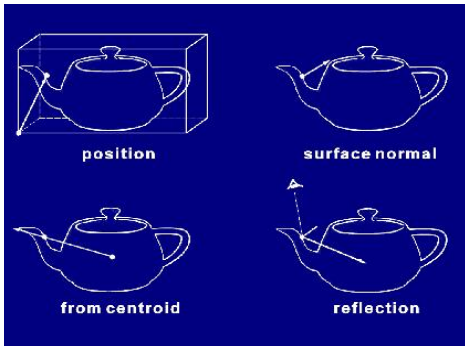






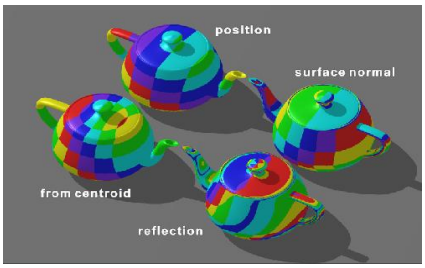
Was soll man projizieren?

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

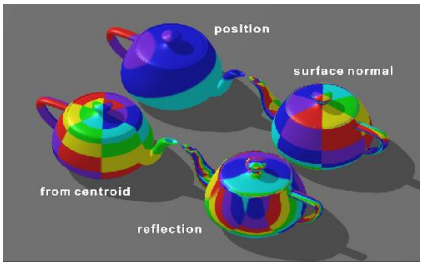


- Beispiele:

planar

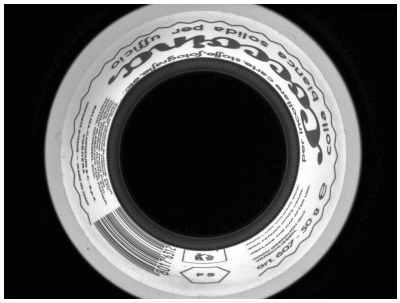


cylindrical



Weitere Beispiele für die Verwendung von Texturen

- Interpolation von Vertex-Farben im HSV-Raum:
  - Erzeuge eine 3D-Textur, wobei (u,v,w) als (H,S,V) interpretiert werden
  - Jedes Texel enthält den RGB-Wert, der die gleiche Farbe wie der HSV-Wert hat
  - Spezifiziere an den Vertices des Dreiecks 3(!) Texturkoordinaten, die die Farbe des Vertex im HSV-Raum angeben
- Image-Warping:



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 (später)

`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 );
```