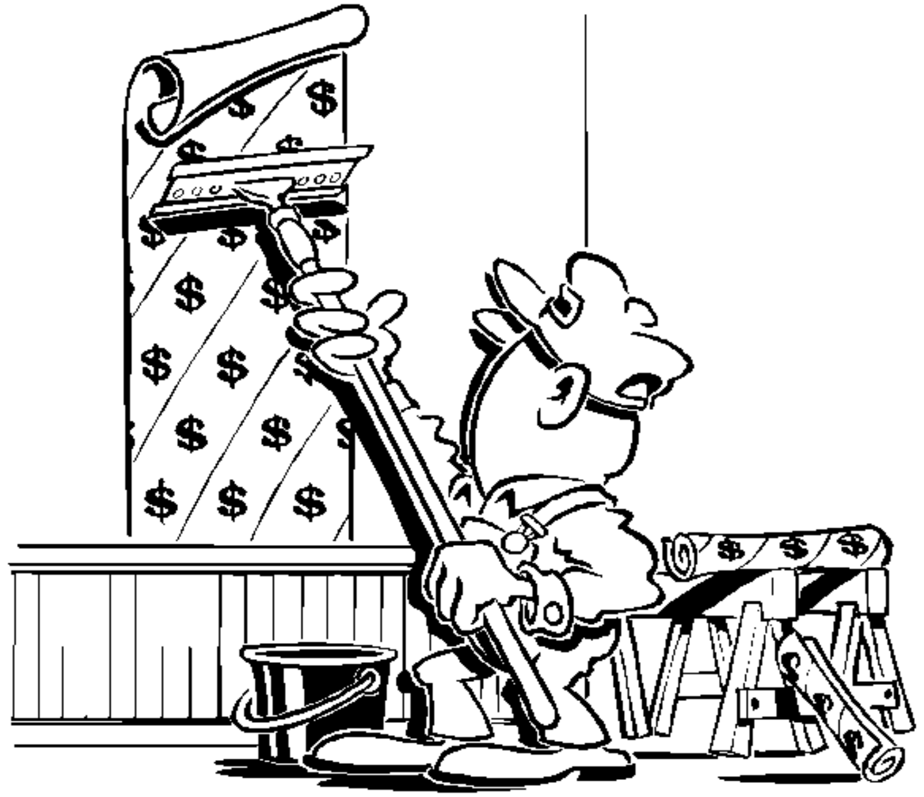
Advanced Computer Graphics
Advanced Texturing Methods



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

Parameterization

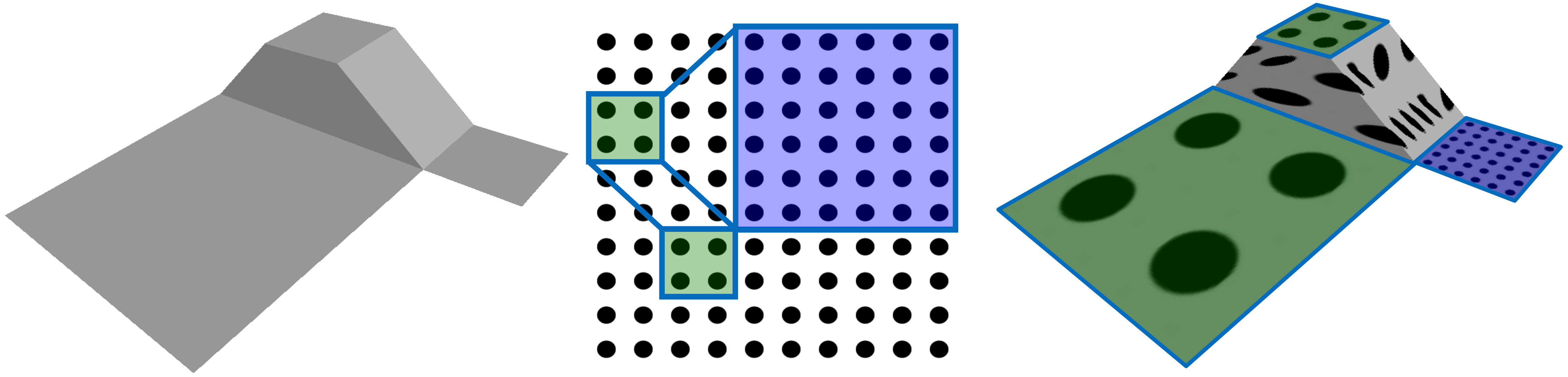
- Informal definition: finding a function $f : (u, v) \in \mathbb{R}^2 \longrightarrow \mathbb{R}^3$ that "describes" a surface (= 2-manifold) in 3D space
  - The region of "useful" (u,v) values is called parameter domain (mostly $[0,1]^2$)
- Example: a possible parameterization of the sphere (with the well-known problems)

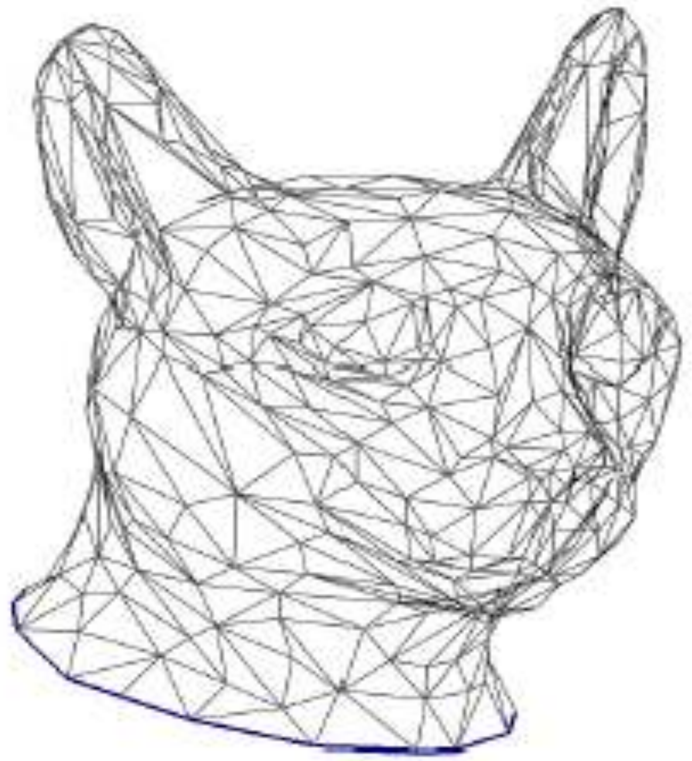
$$f(u, v) = \begin{pmatrix} \cos(2\pi u) \sin(\pi v) \\ \sin(2\pi u) \sin(\pi v) \\ \cos(\pi v) \end{pmatrix} \; , \; (u, v) \in [0, 1]^2$$

- In computer graphics:
  - Surface = mesh,
  - Function = (u,v) coordinates for vertices, linear interpolation in-between
  - "Texturing" , "uv mapping"

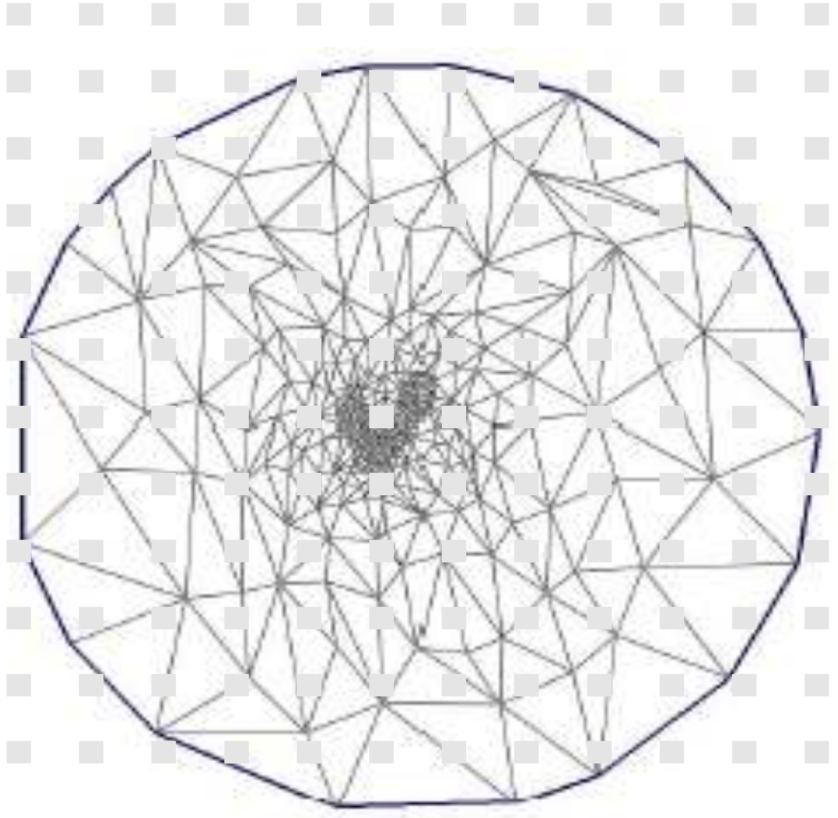
Problems with (Simple) Parameterizations

- Distortions in size & form
- Consequence: relative over- or under-sampling
- Example:





Mesh

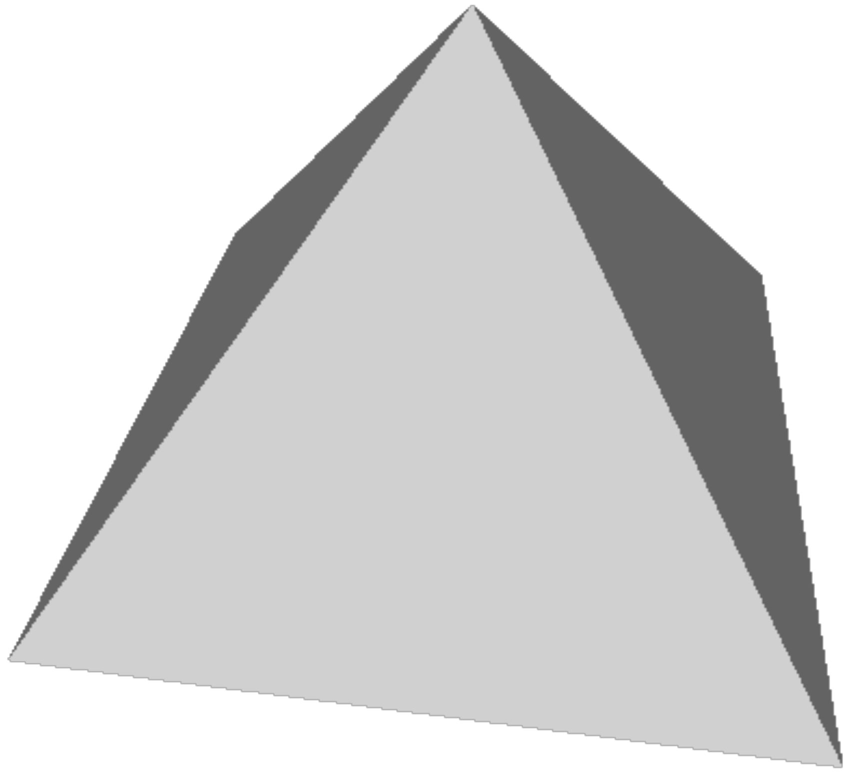


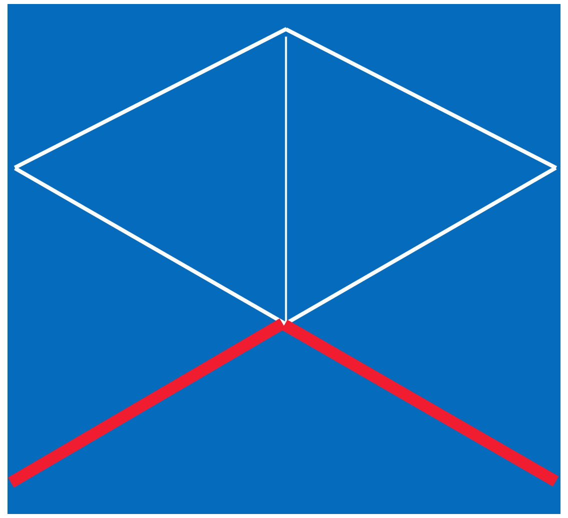
Embedding

Distortion

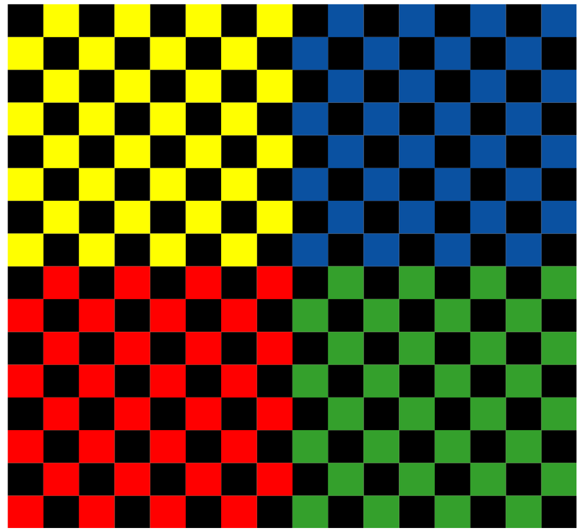
One Technique to Remedy: Seams ("Nähte")

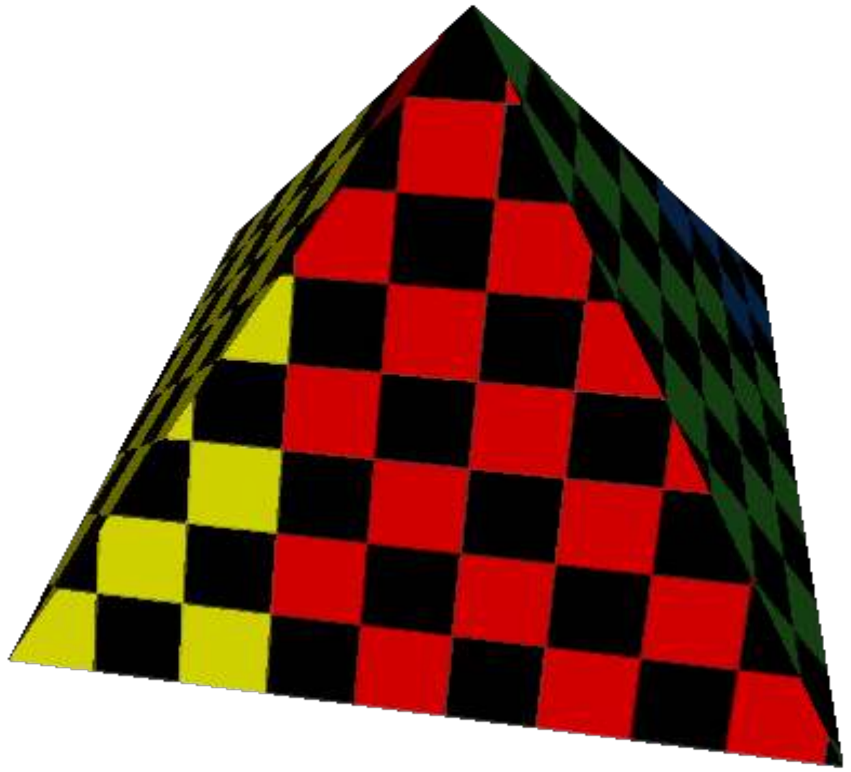
- Idea: cut up the mesh along certain edges and unfold it into a plane (aka. *surface development* or *unwrapping*)
- Results in *seams*, *i.e.* "double edges" in the parameter domain (aka. *uv space*)
- Unavoidable with non-planar topology, e.g., closed 2-manifolds



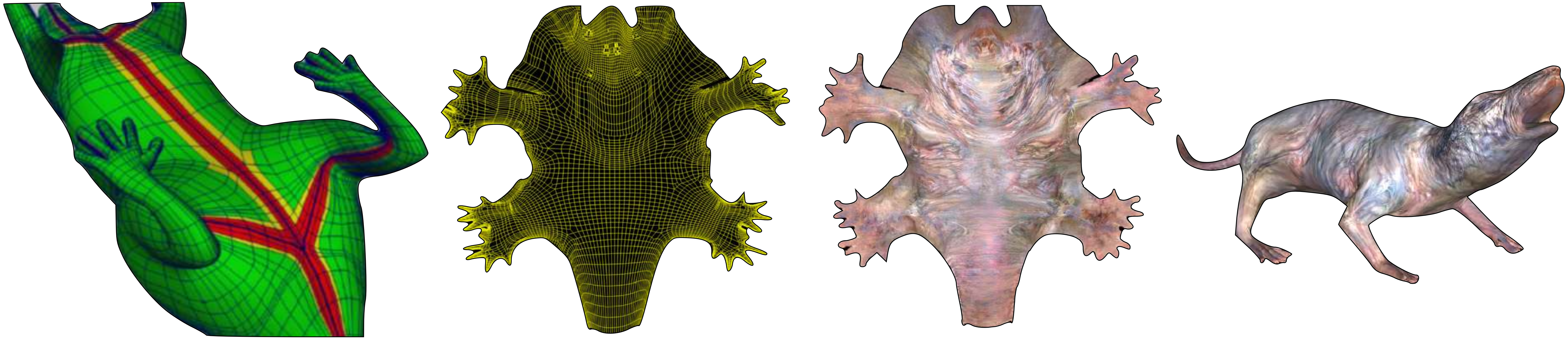


Seams

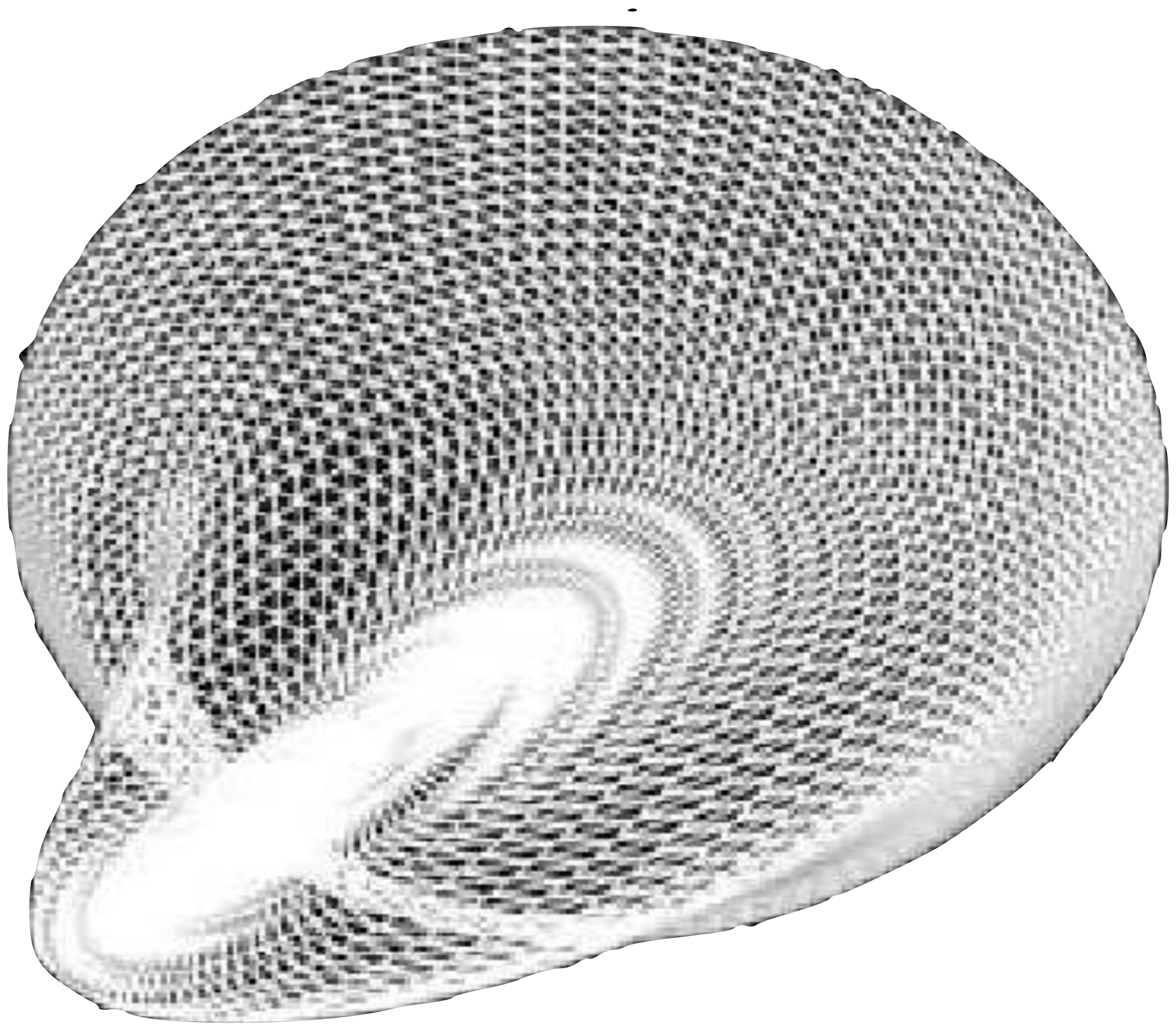


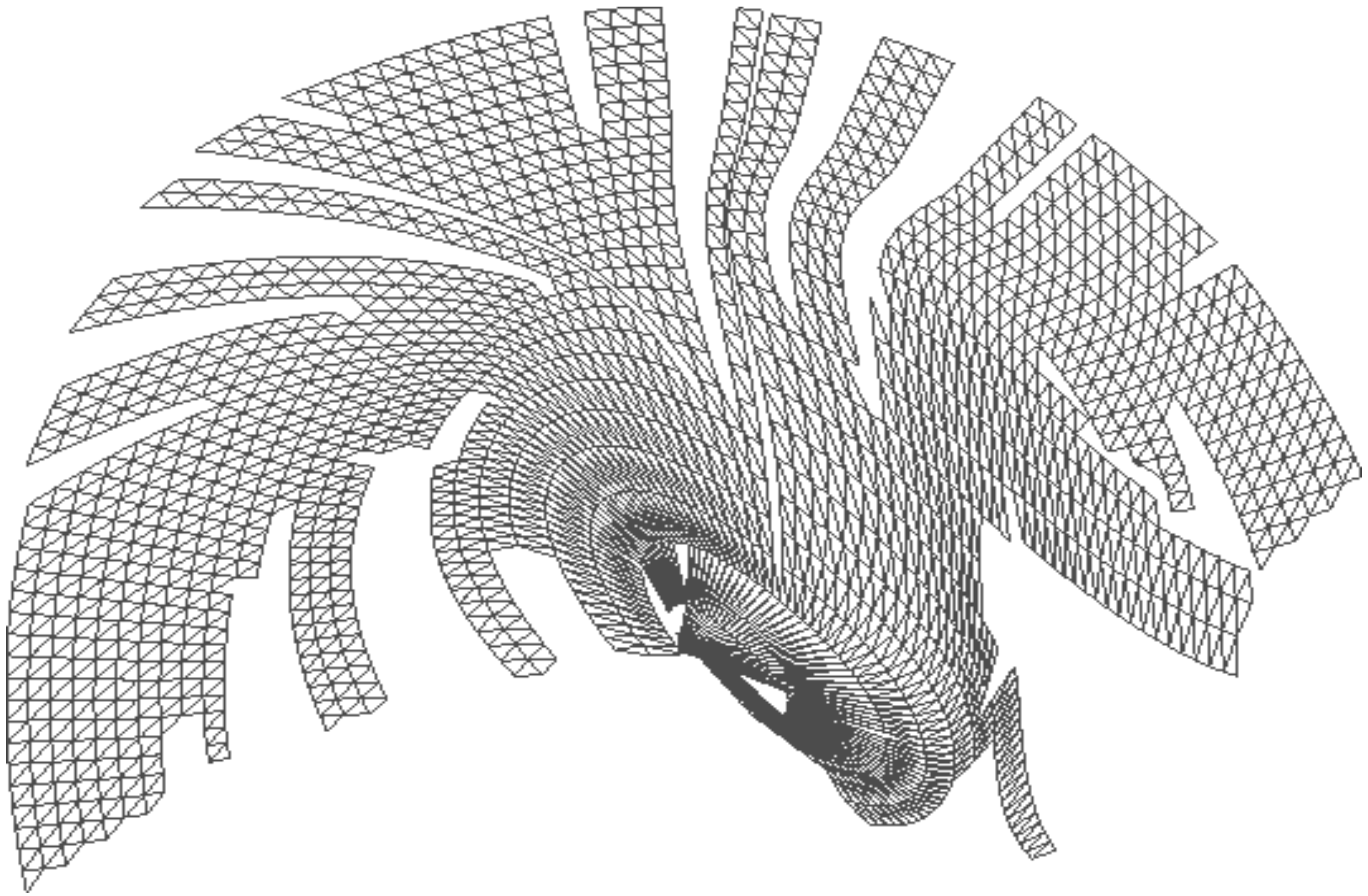


- Cut the object along only one contiguous sequence of edges (preferably at inconspicuous places)
- Effect: the resulting mesh is now topologically equivalent to a disc
- Then embed this open mesh into the 2D plane



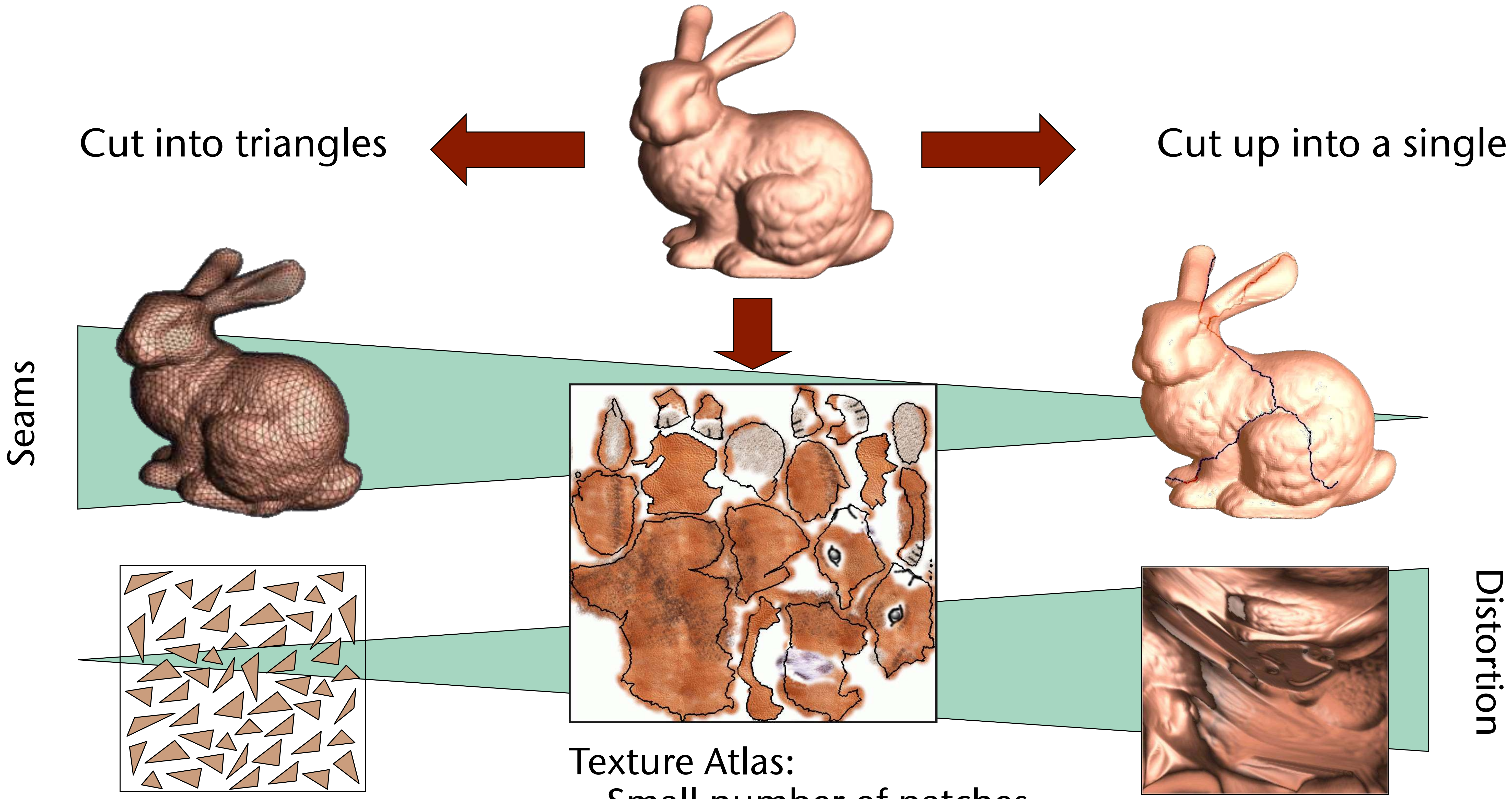
- Goal: minimize the distortion
- Straight-forward remedy: multiple seams
  - Problem: produces a severely fragmented embedded grid





- Another problem with seams: vertices on the seam must have multiple, different (u,v) coordinates
- Remedy: create multiple copies of those vertices
- New problem in case of deformations of the mesh

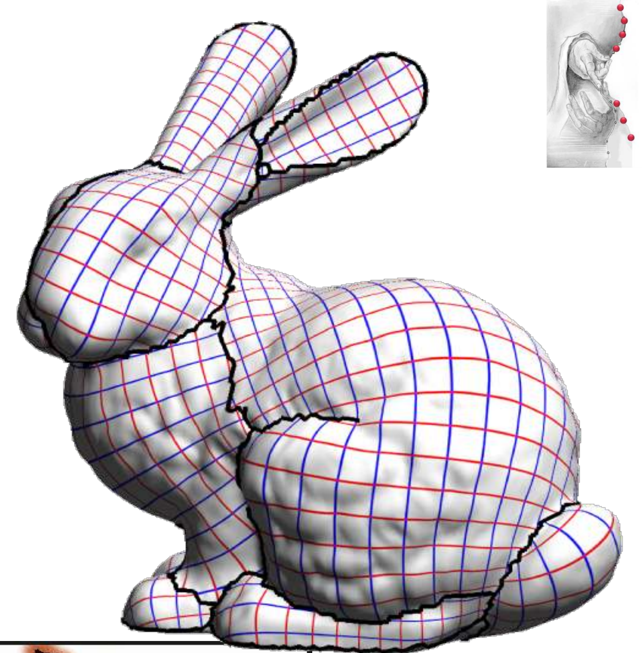
Dichotomy: Distortion or Seams

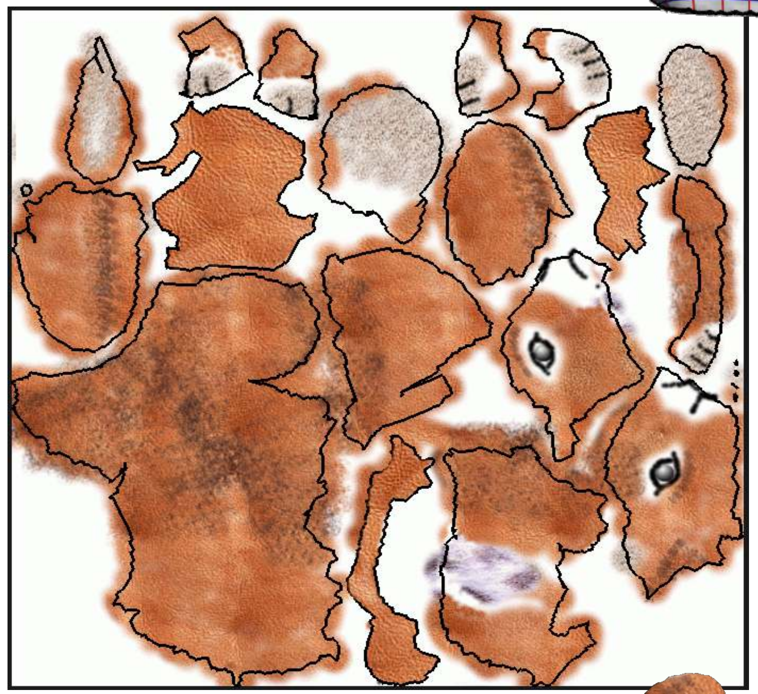


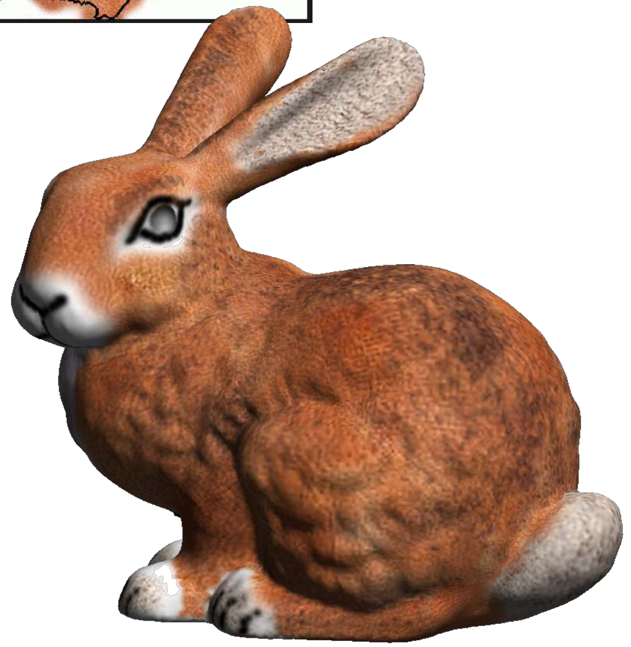
Texture Atlas:
- Small number of patches
- Short and hidden seams

The Texture Atlas

- Idea:
  - Cut the 3D surface into individual patches
  - Map = individual parameter domain in texture space for a single patch
  - Texture Atlas = set of these patches with their respective maps (= parameter domains)
- Statement of the optimization problem:
  - Choose a compromise between seams and distortion
  - Hide the cuts in less visible areas
    - How do you do that automatically?
  - Determine a compact arrangement of texture patches (a so-called *packing problem*)



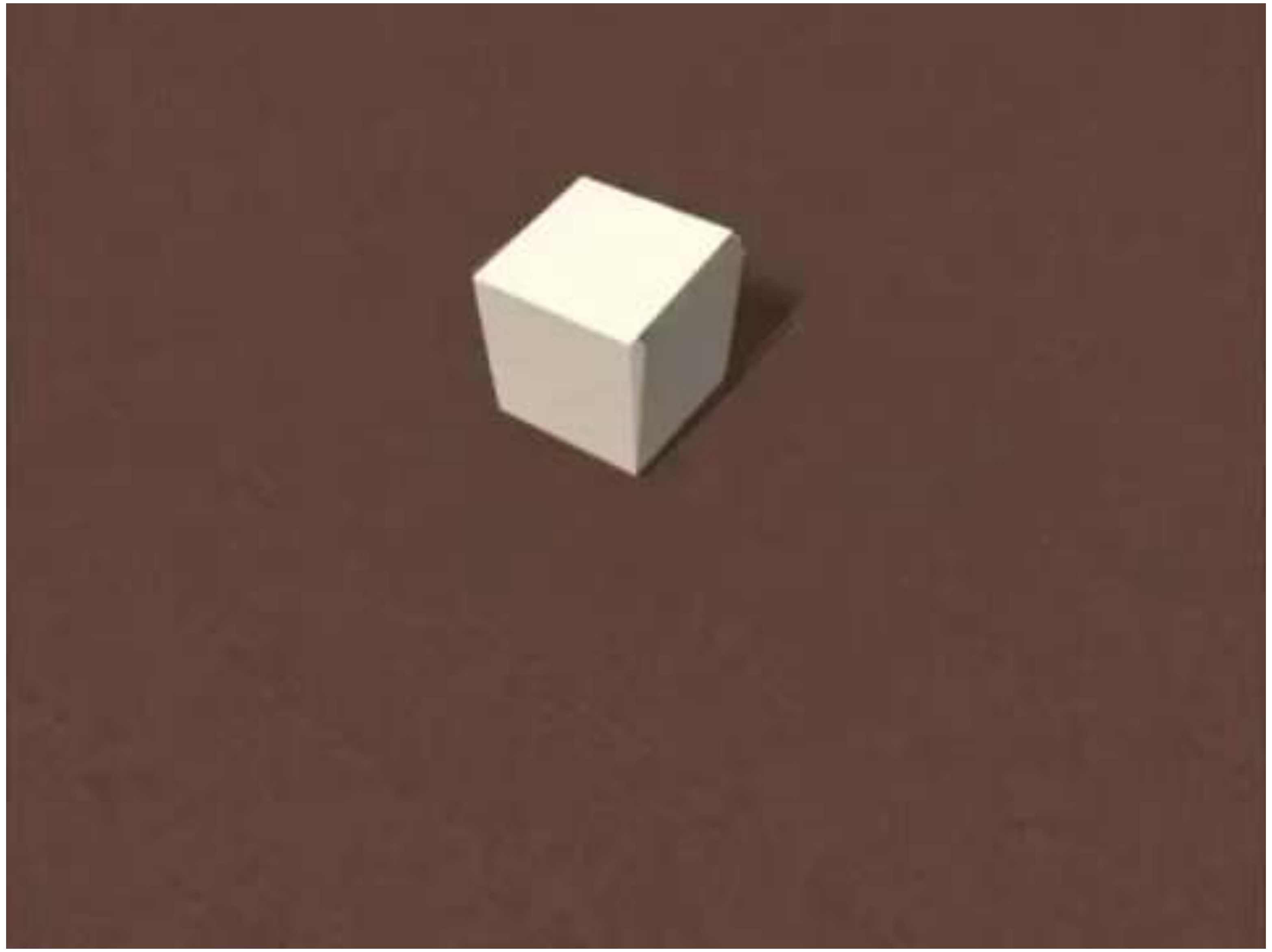


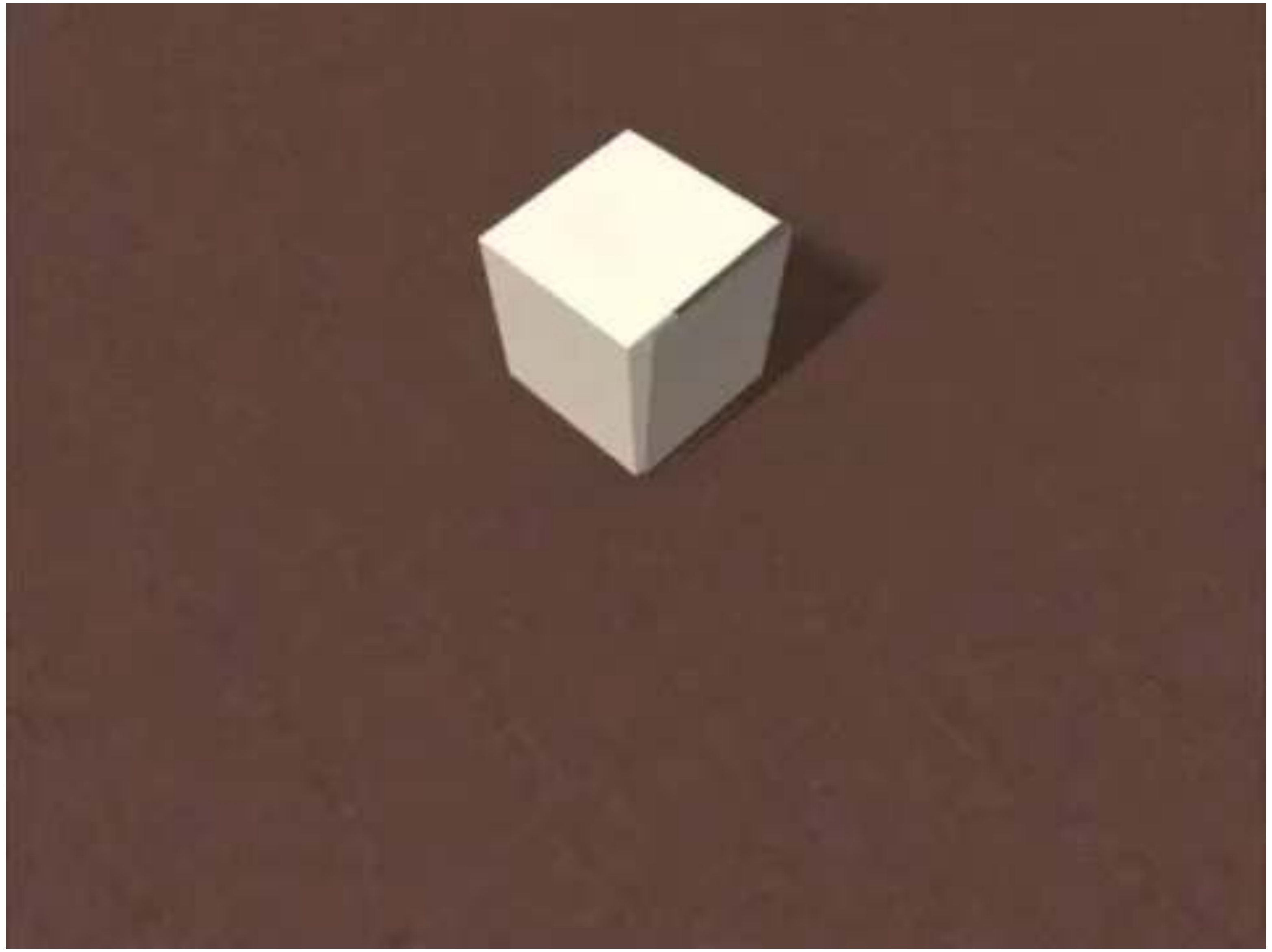


Examples

Digression: A Geometric Brain-Teaser

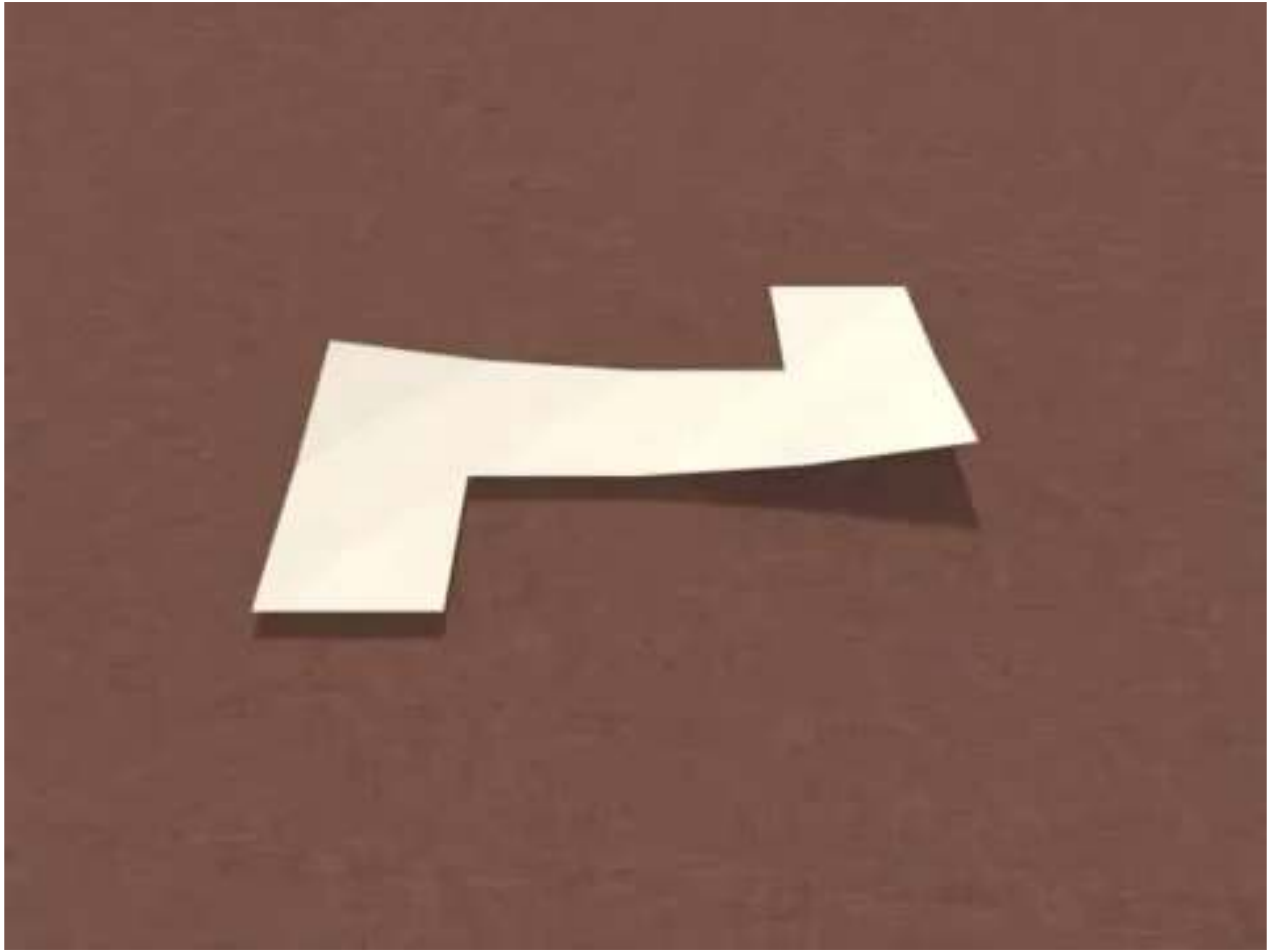
- A cube can be unfolded into a cross
- Into what other forms can a cube be unfolded, too?





Katie Park / unfoldit.org

- Side note: the (unfolded) cube can be folded into a parallelogram

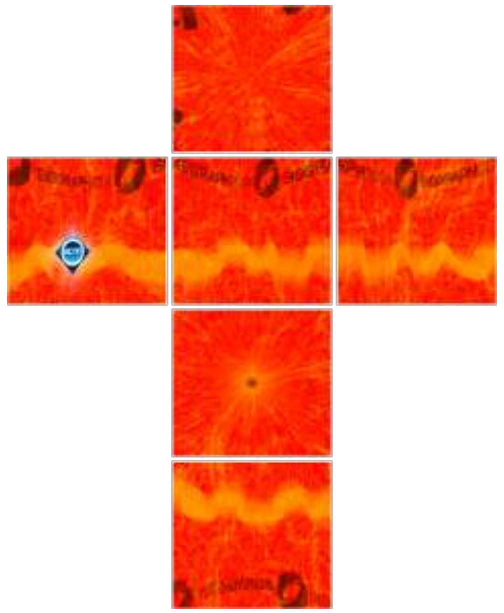
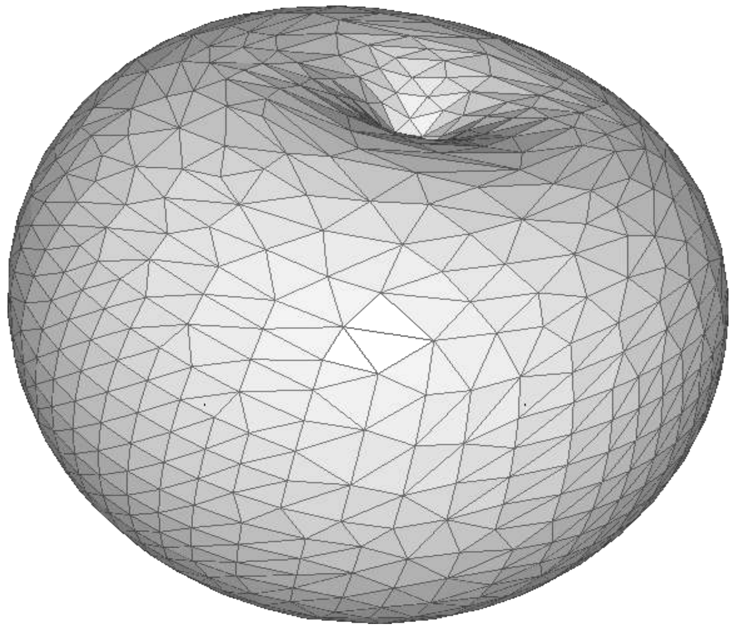


Cube Maps

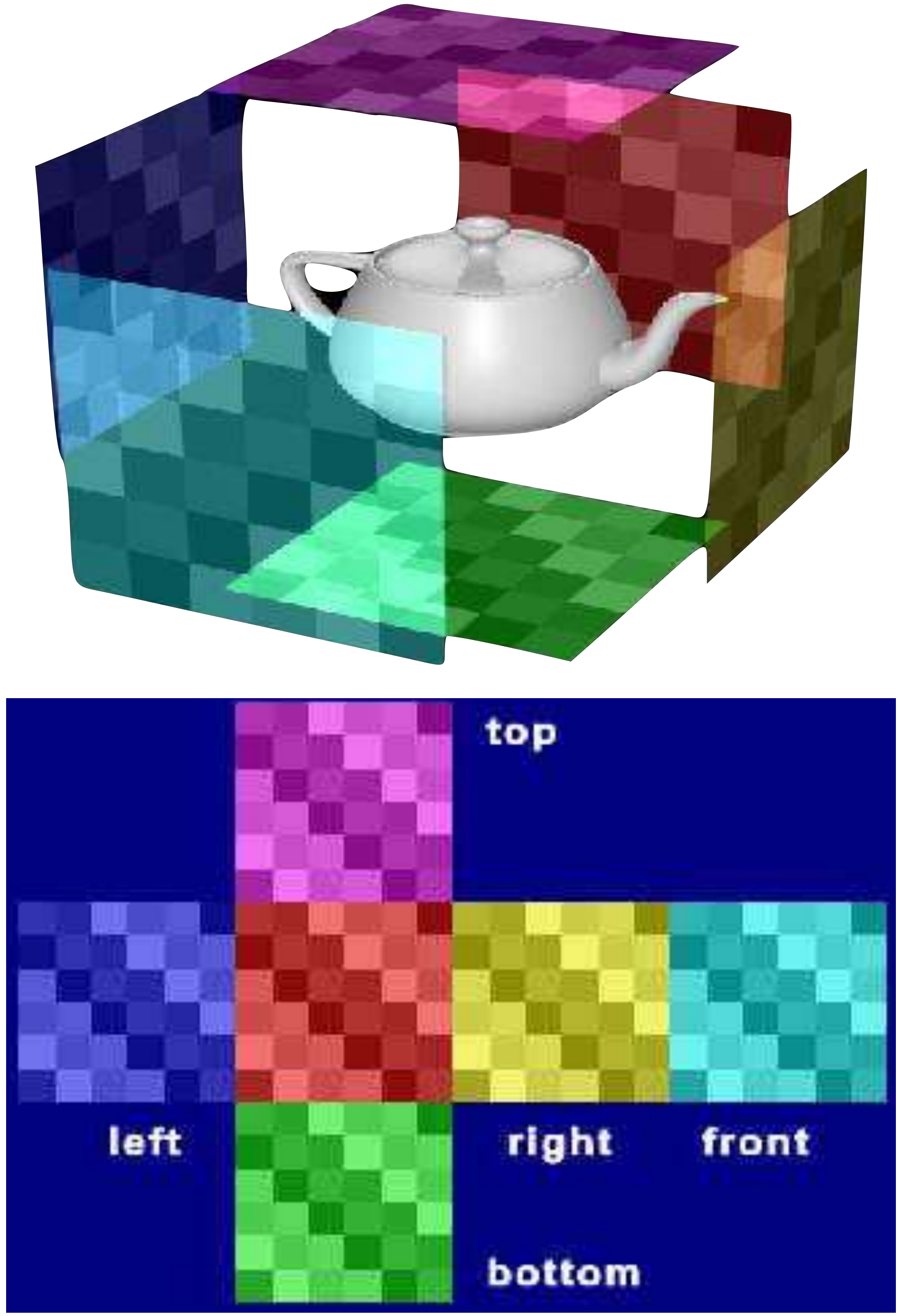
- Parameter domain = surface of unit cube
  - Six quadratic texture bitmaps
  - 3D texture coordinates in (old) OpenGL:

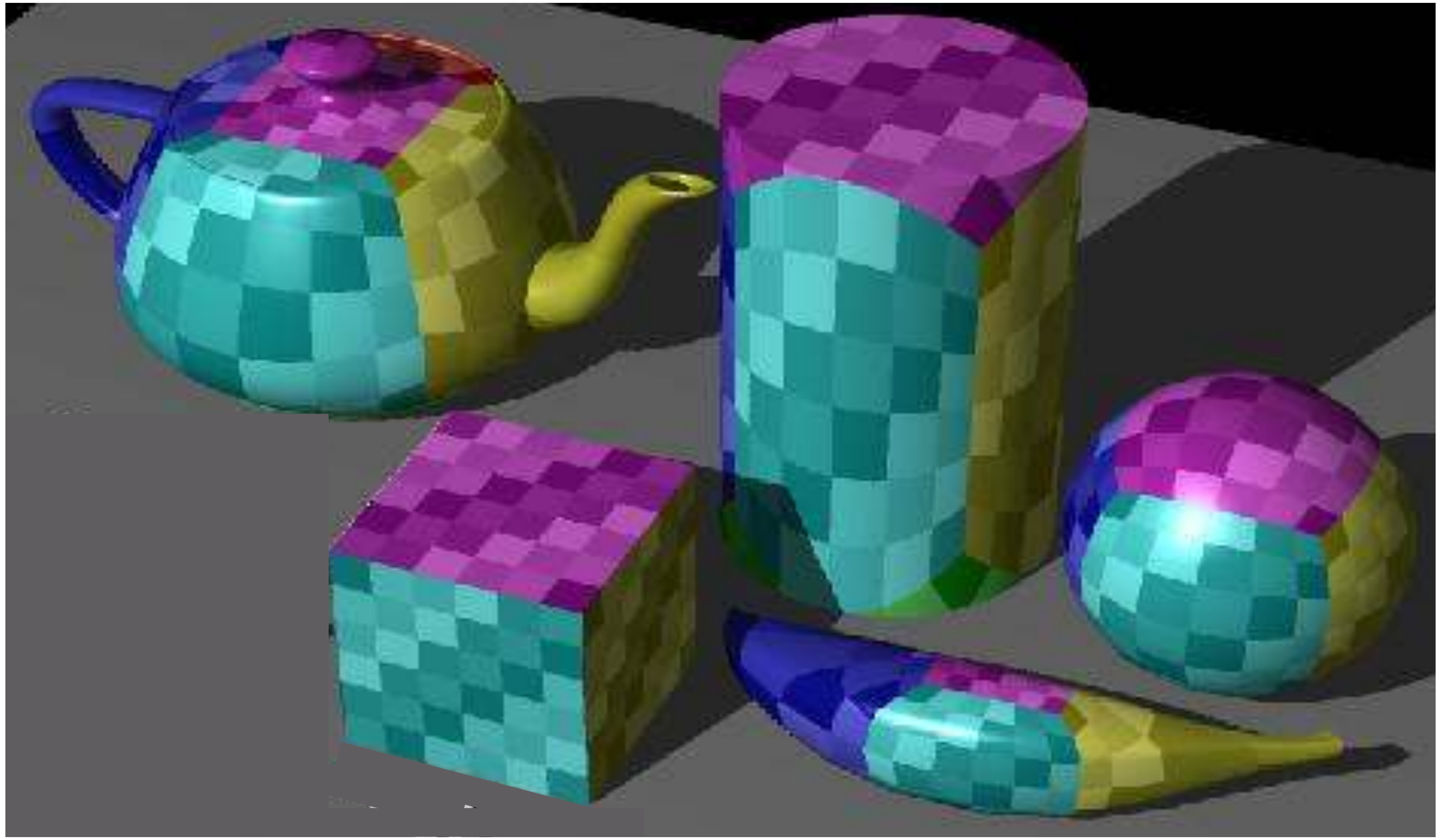
```
glTexCoord3f( s, t, r );
glVertex3f( x, y, z );
```

  - Largest component of (*s*,*t*,*r*) determines the cube side = bitmap, intersection point determines (*u*,*v*) within the bitmap
- Rasterization of cube maps:
  1. Interpolation of (*s*,*t*,*r*) in 3D
  2. Projection onto the cube ⟶ (*u*,*v*)
  3. Texture look-up in 2D
- Pro: relatively uniform, OpenGL support
- Slight con: needs 6 images


Examples





Cube Maps in OpenGL

Just FYI (not relevant for exam)

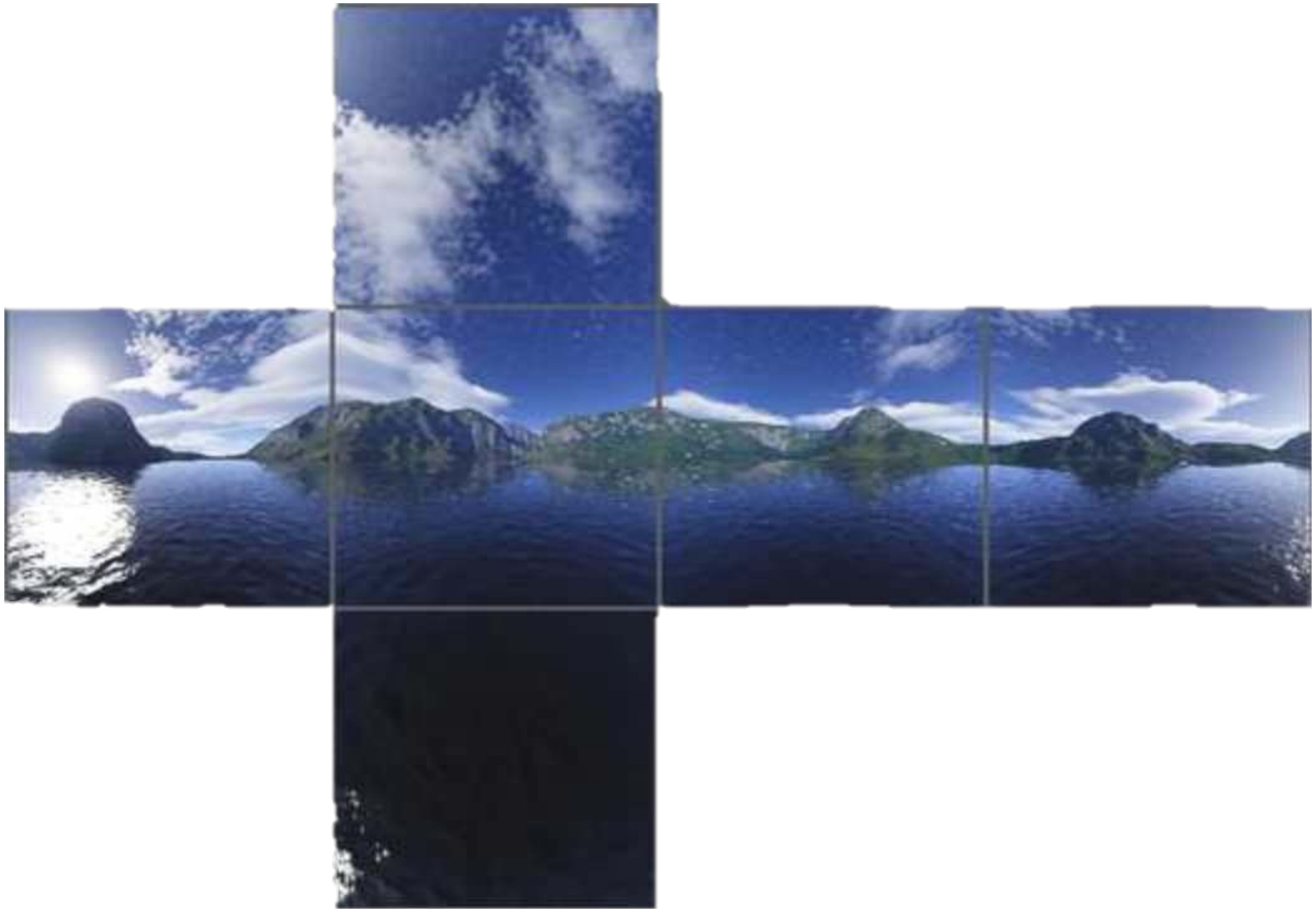
```
glGenTextures( 1, &textureID );
glBindTexture( GL_TEXTURE_CUBE_MAP, textureID );
glTexImage2D( GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGBA8, width, height,
              0, GL_RGB, GL_UNSIGNED_BYTE, pixels_face0 );
... Load the texture of the other cube faces
glTexParameteri( GL_TEXTURE_CUBE_MAP,
                 GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
... Set more texture parameters, like filtering

glEnable( GL_TEXTURE_CUBE_MAP );
glBindTexture( GL_TEXTURE_CUBE_MAP, textureID );
glBegin( GL_... );
glTexCoord3f( s, t, r );
glVertex3f( ... );
...
```

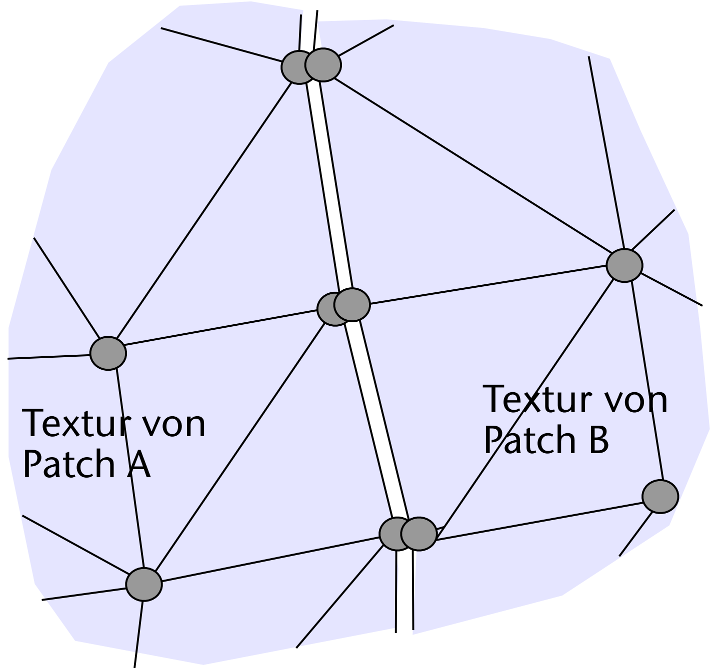
Analog:
`GL_TEXTURE_MAG_FILTER`,
`GL_TEXTURE_WRAP_T`,
`etc.` …

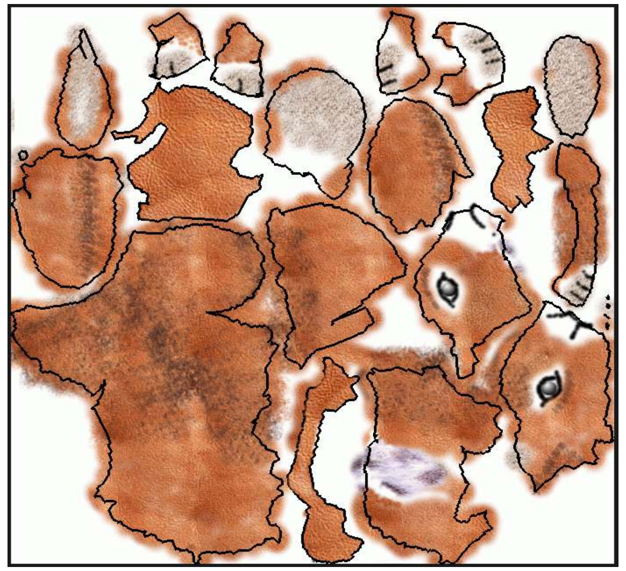
Just like with all other vertex attributes in OpenGL: first send all attributes, then the coordinates

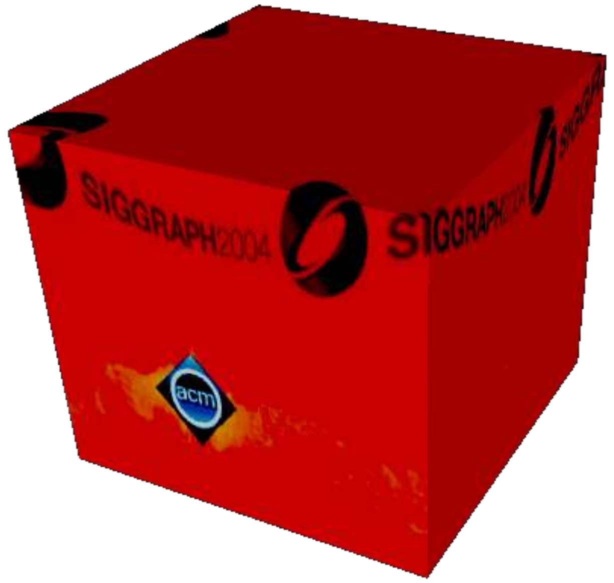
Example Cube Map for a Sky Box



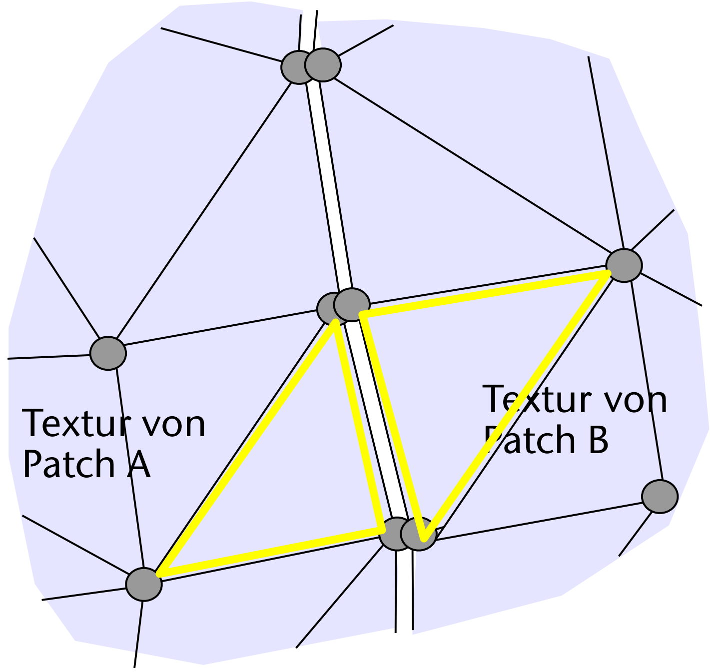
Texture Atlas vs. Cube Map

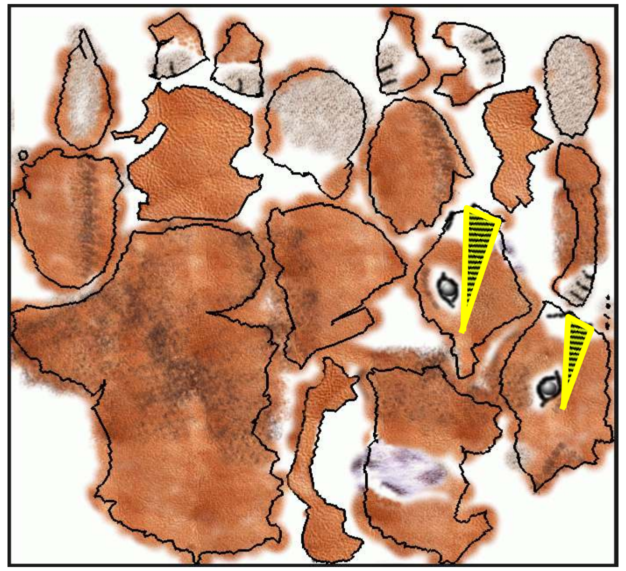


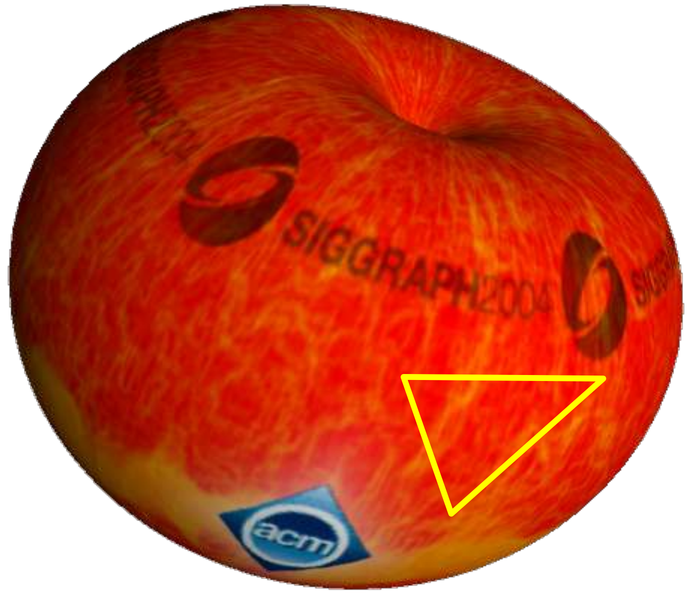


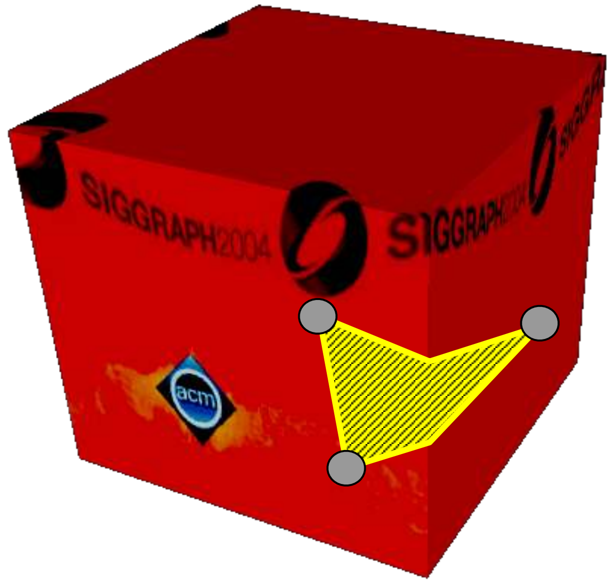


Texture Atlas vs. Cube Map



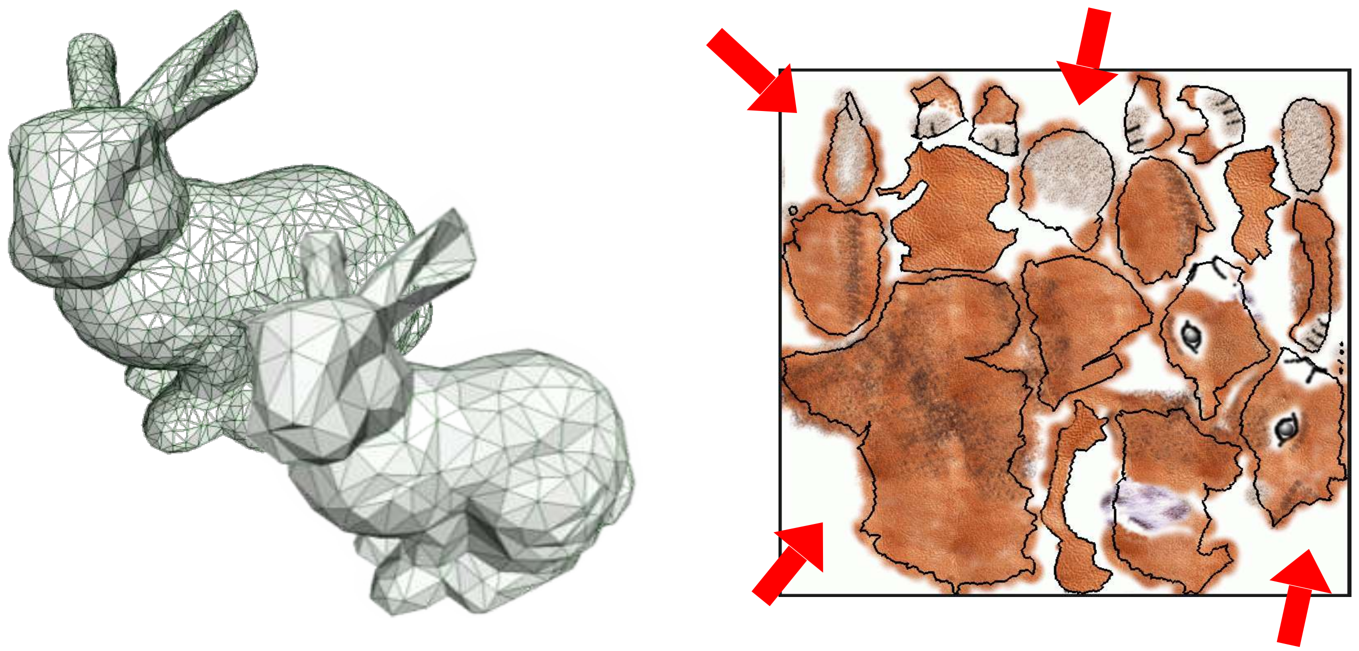


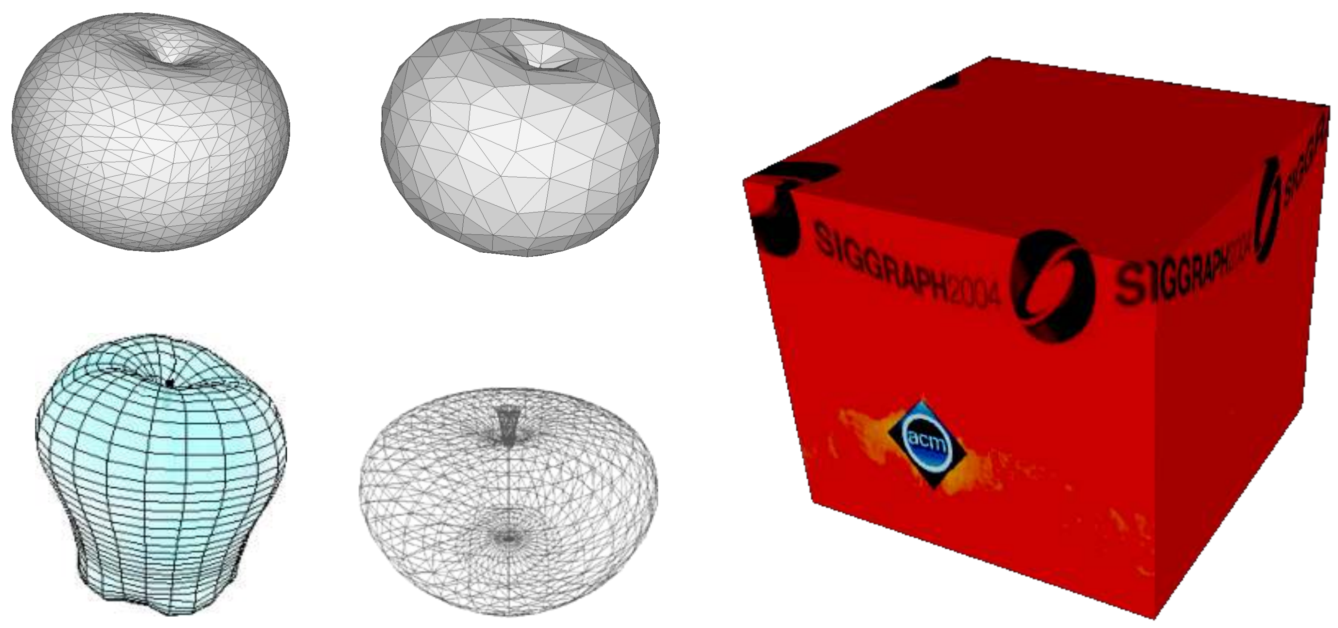




- Must prevent seams manually
  - E.g., by making colors match across seams
- MIP-mapping is difficult

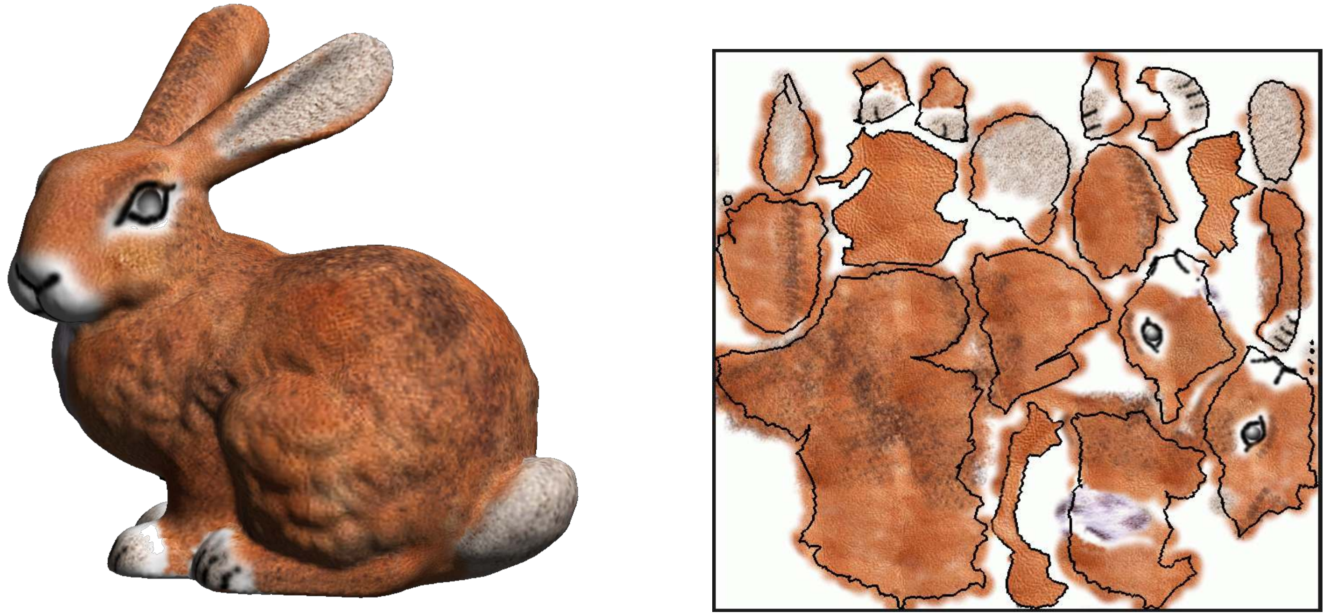
- No seams, automatically
  - Because there are no gaps in the parameter domain
- MIP-mapping is okay

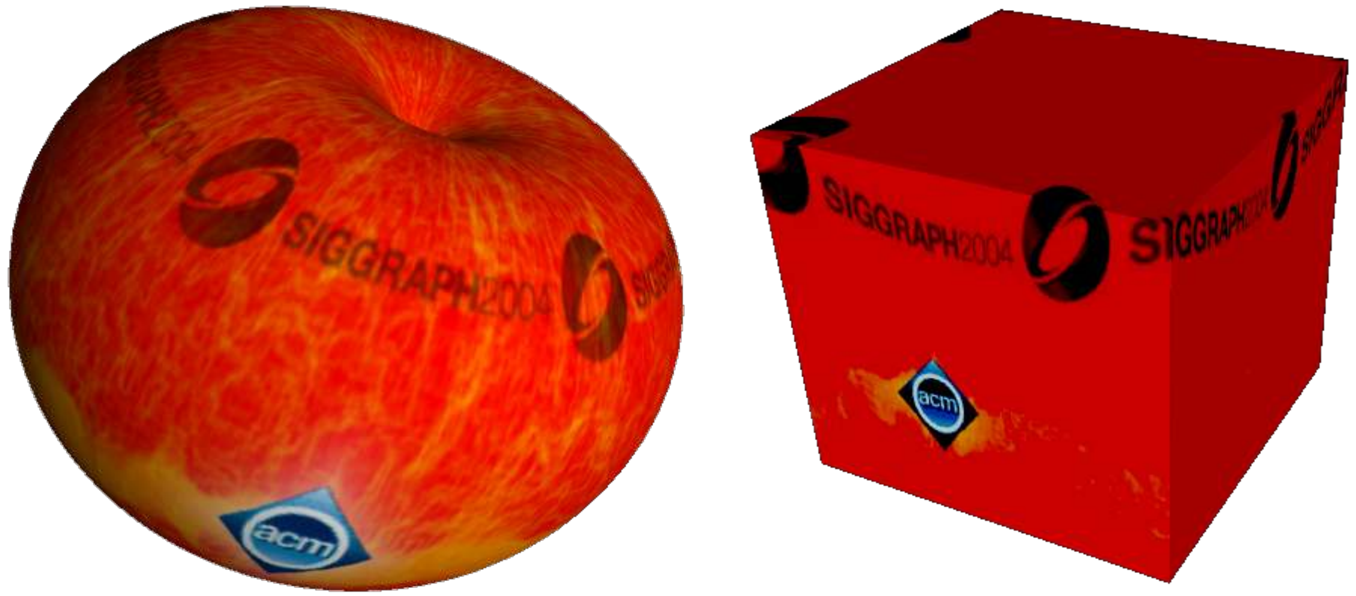




- Must prevent seams manually
- Triangles must lie inside patches
- MIP-mapping is difficult
- Only valid for a specific mesh
- Texels are wasted

- No seams, automatically
- Triangles can cross multiple patches
- MIP-mapping is okay
- Valid for many meshes
- All texels are used





- Must prevent seams m… …lly
- Triangles may lie w… …the patches
- MIP-mapping i… …cult
- Only valid … …pecific mesh
- Texels … …asted

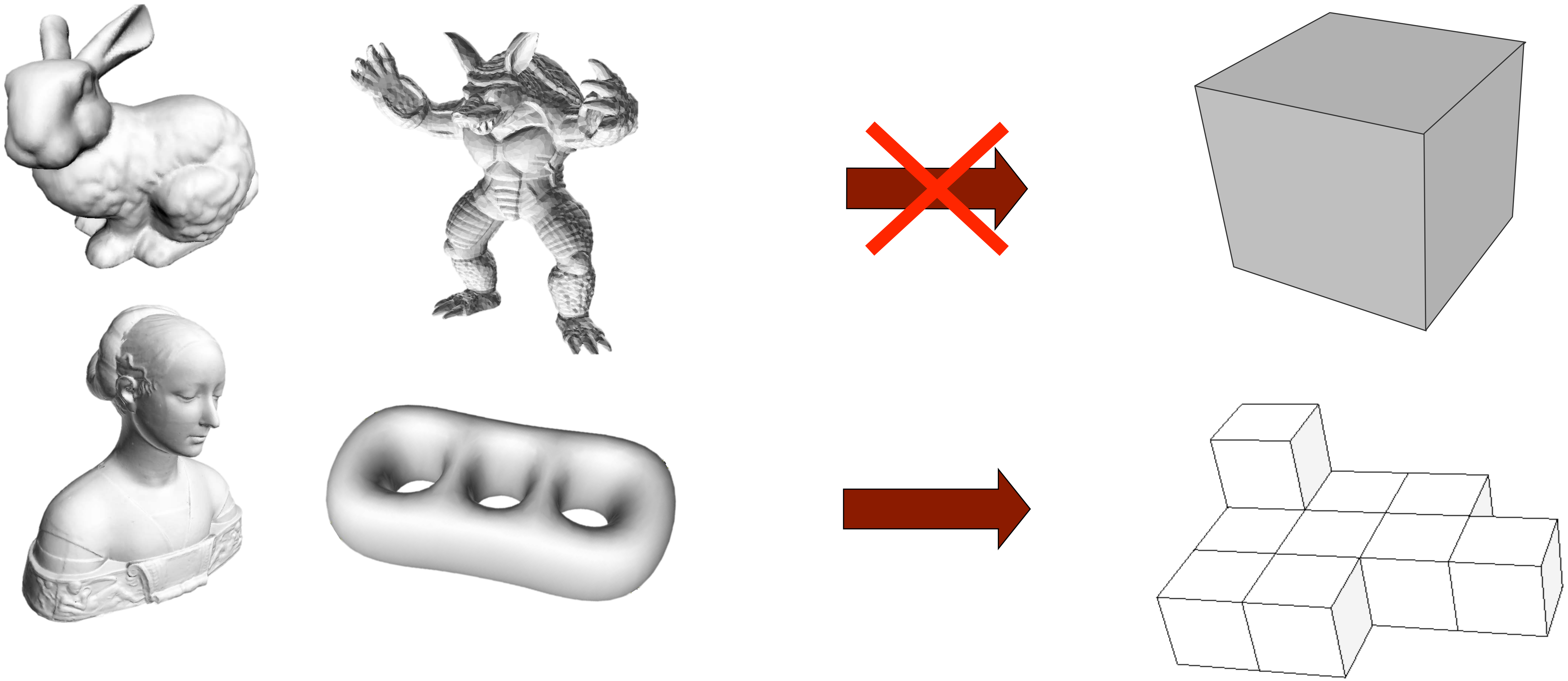
- No seams automati…
- Triangles can lie… …e patches
- MIP-mappi…
- Valid fo… …hes
- All texels… …ed

Works for any shape

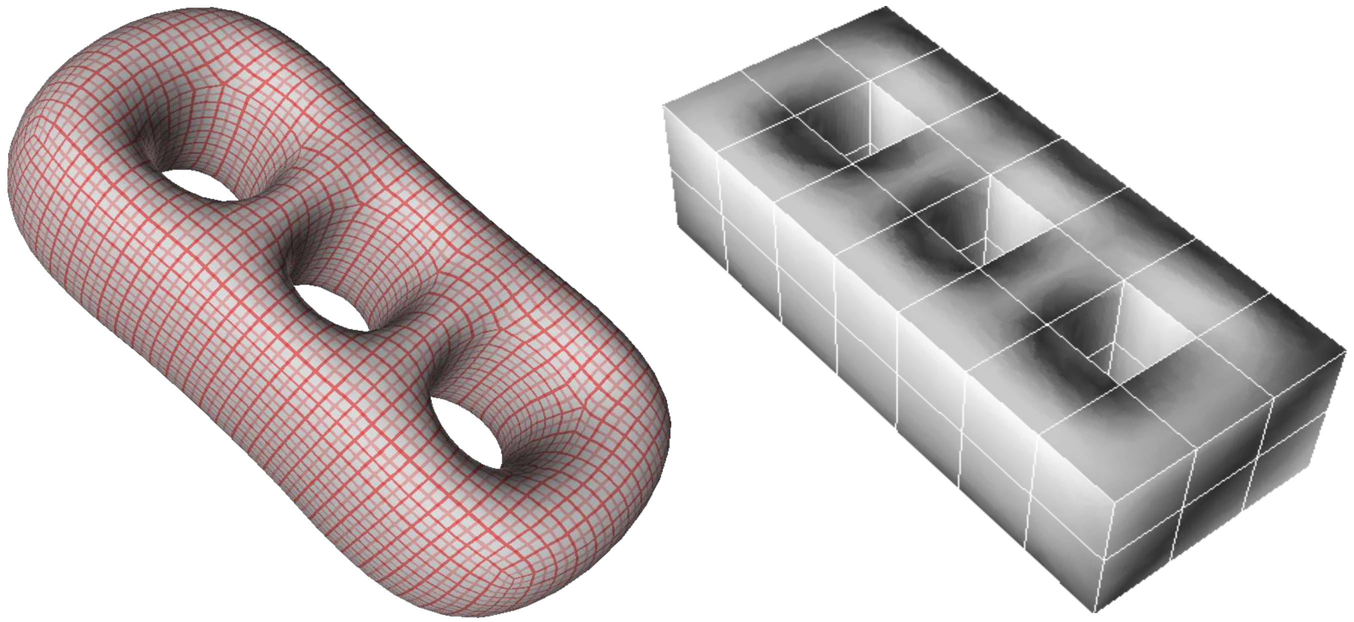
Only for "sphere-like" objects

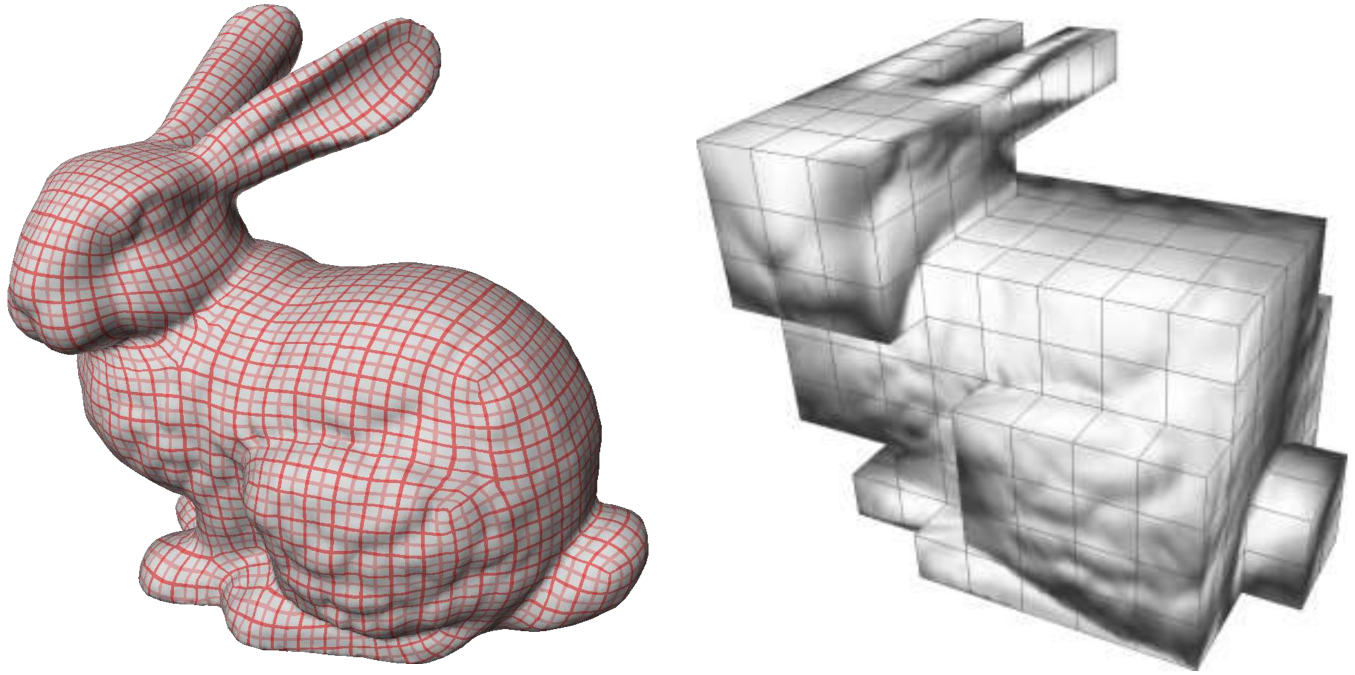
Polycube Maps

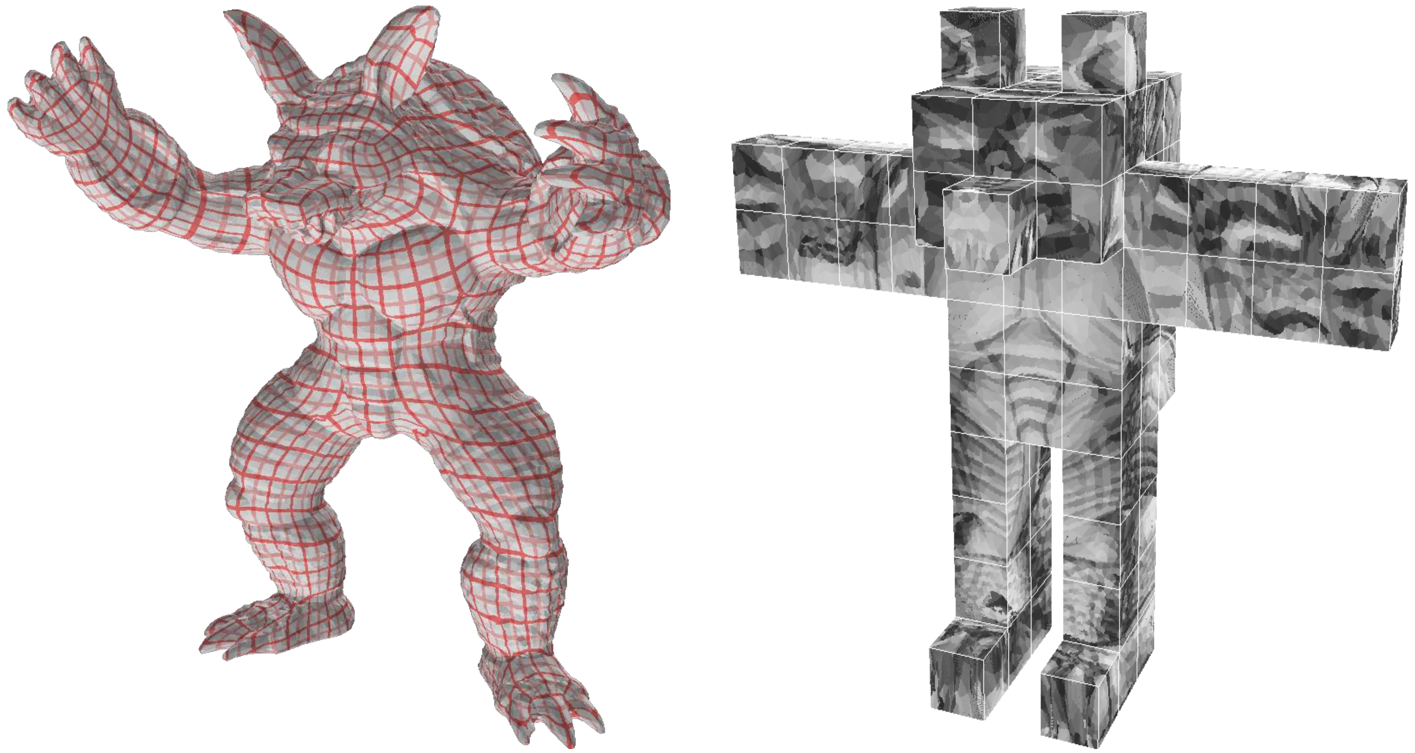
- Use many cube maps instead of a single cube ⟶ polycube map
- Adapted to geometry and topology

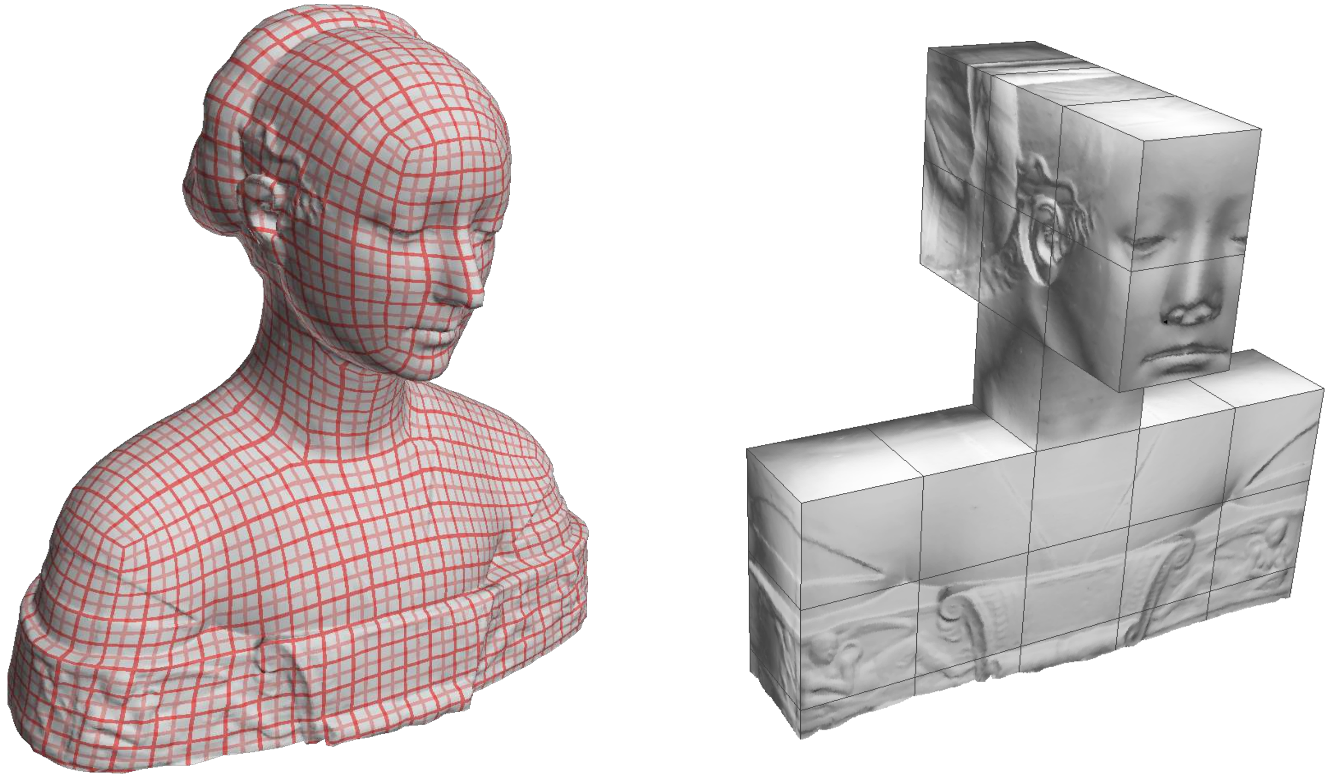


Examples



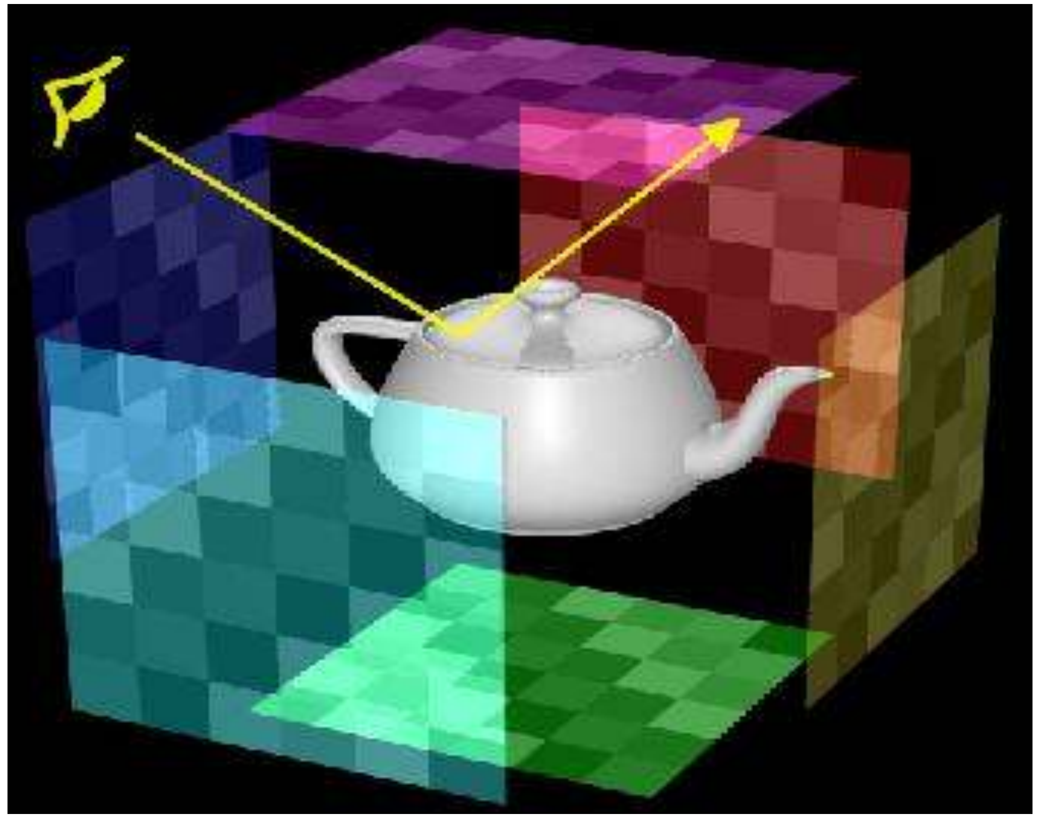




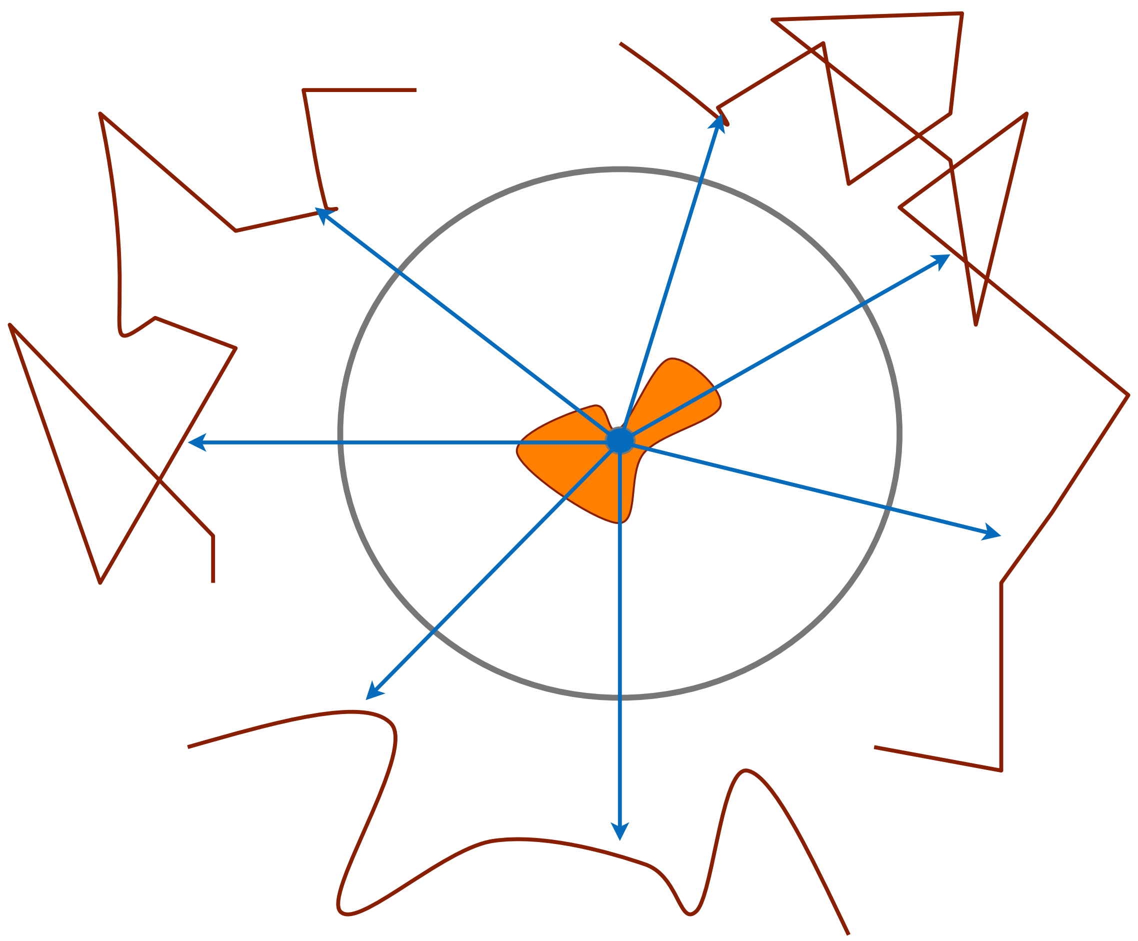


Environment Mapping

- With very reflective objects, one would like to see the surrounding environment reflected in the object
- Trivial in ray-tracing, but not for polygonal rendering by rasterization
- The idea of environment mapping:
  - "Photograph" the environment in a texture, and store as a cube map (aka. environment map)
  - Use the reflection vector (of the eye ray) as an index into that texture (a.k.a. reflection mapping)



- For every spatial direction, the environment map saves the color of the light that reaches a specific point
- Only correct for *one* position
- No longer correct if the environment changes



Historical Examples of Applications

Lance Williams, Siggraph 1985

Flight of the Navigator (1986)
First feature film to use the technique

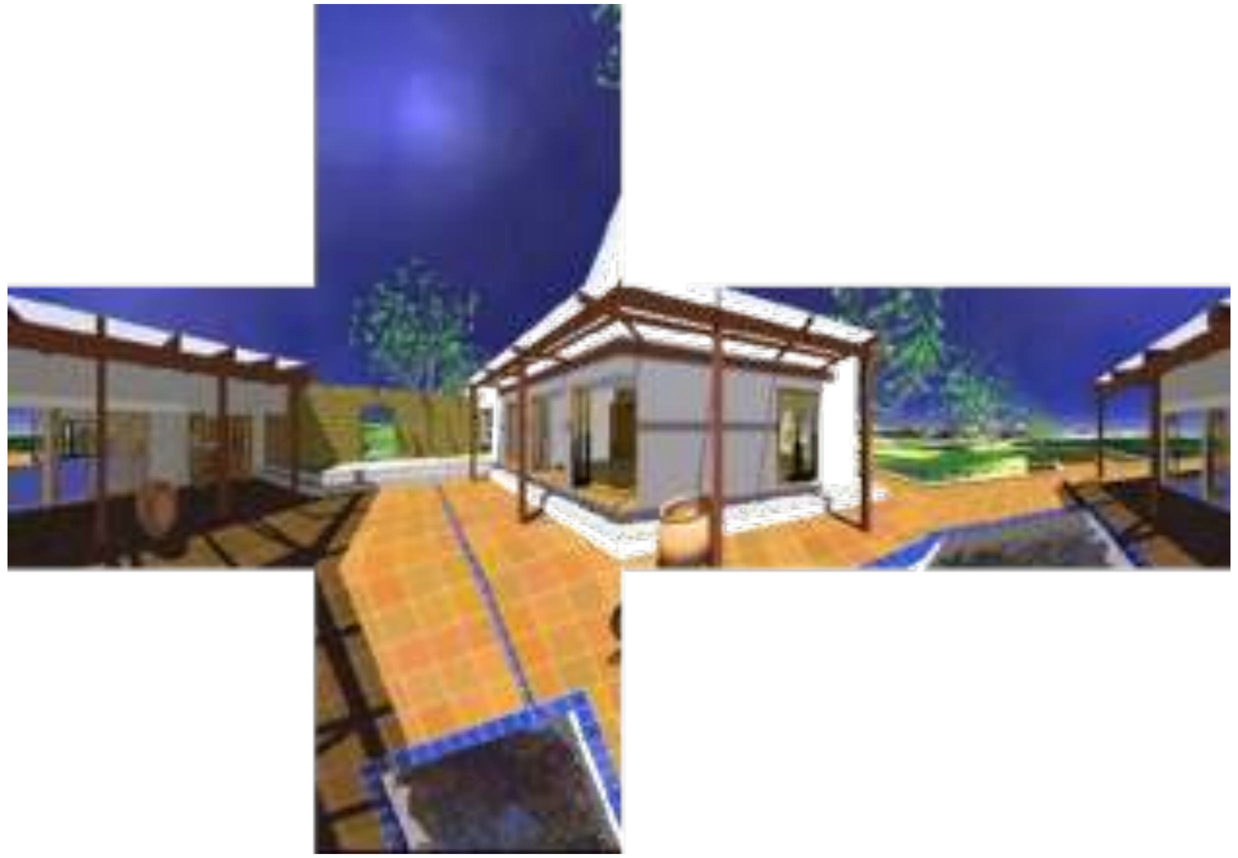
Terminator 2: Judgment Day
(1991, Industrial Light + Magic)
Most visible appearance

Environment Mapping Steps

- Generate or load a 2D texture that depicts the environment
- During rasterization, for every pixel on the reflected object:
  1. Calculate the normal **n** and the view vector **v**
  2. Calculate a reflection vector **r** from **n** and **v**
  3. Calculate texture coordinates (*u*,*v*) from **r**
  4. Color the pixel with the texture value (texel)
- The problem: how does one parameterize the space of the reflection vectors?
  - I.e.: how do you map spatial directions (= 3D unit vectors) onto [0,1]x[0,1]?
- Desired characteristics:
  - Uniform sampling (number of texels per solid angle should be "as constant as possible" in all directions)
  - View-independent ⟶ only one texture for all viewpoint positions
  - Hardware support (texture coordinates should be easy to generate)

Cube Environment Mapping

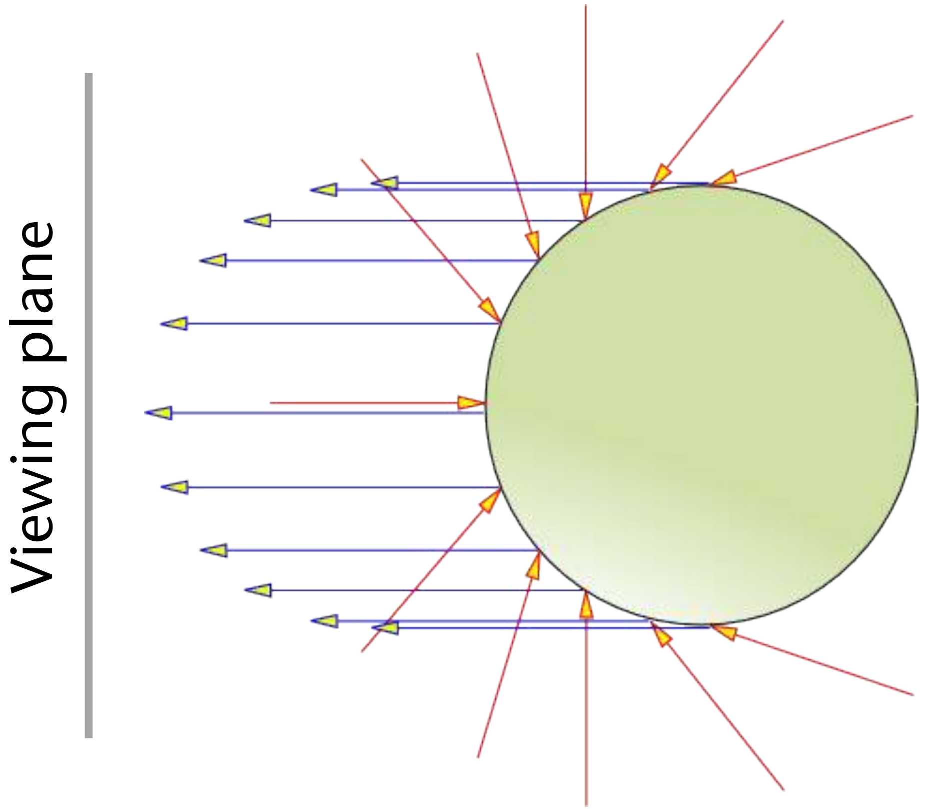
- Just like "normal" cube maps, except use the reflected vector **r** = (r_x, r_y, r_z) = (s, t, r)
- This reflected vector **r** could be automatically calculated by fixed-function OpenGL for each vertex (**`GL_REFLECTION_MAP`**)



FYI

Older Technique: Spherical Environment Mapping

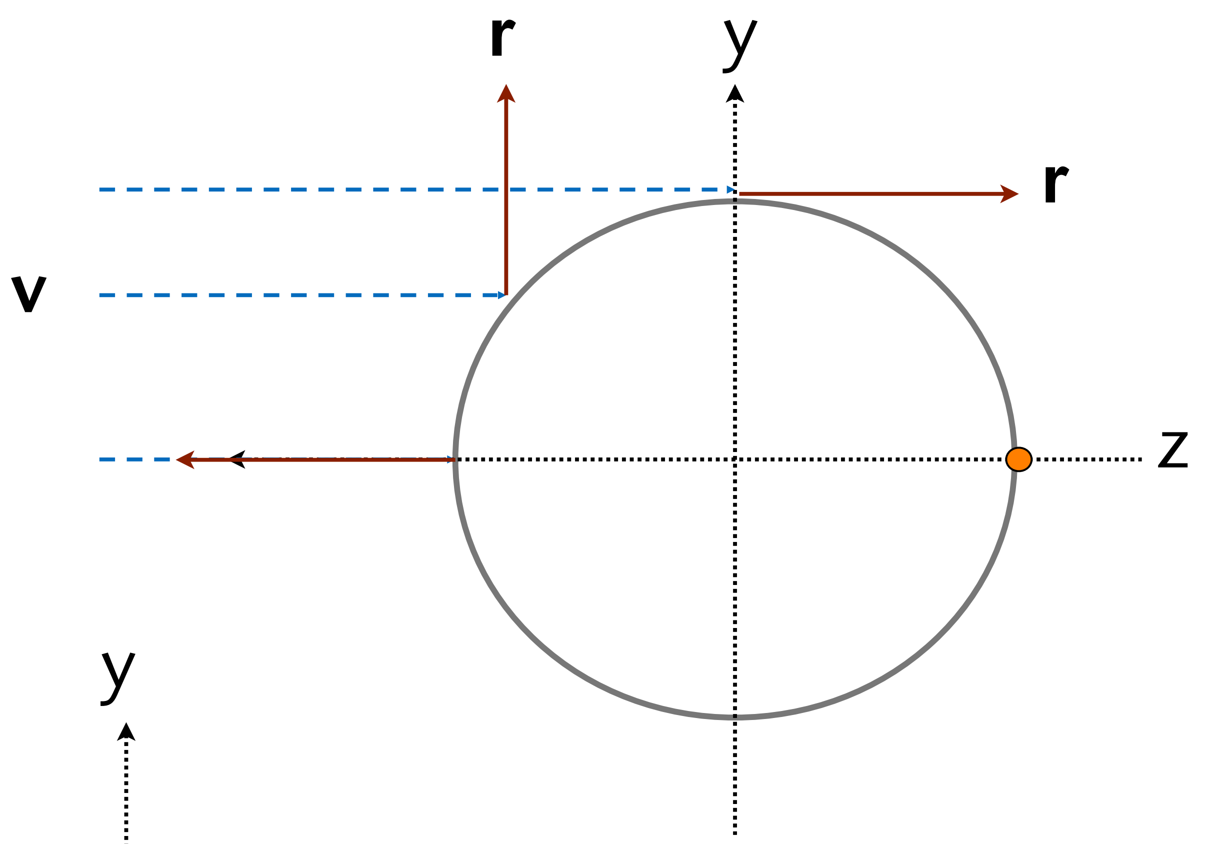
- Sometimes, a cube map cannot be use, depending on the way the environment map is generated
- Generating the environment map with a sphere:
  - Photography of a reflective sphere; or
  - Ray-tracing of the scene with all primary rays being reflected at a perfectly reflective sphere

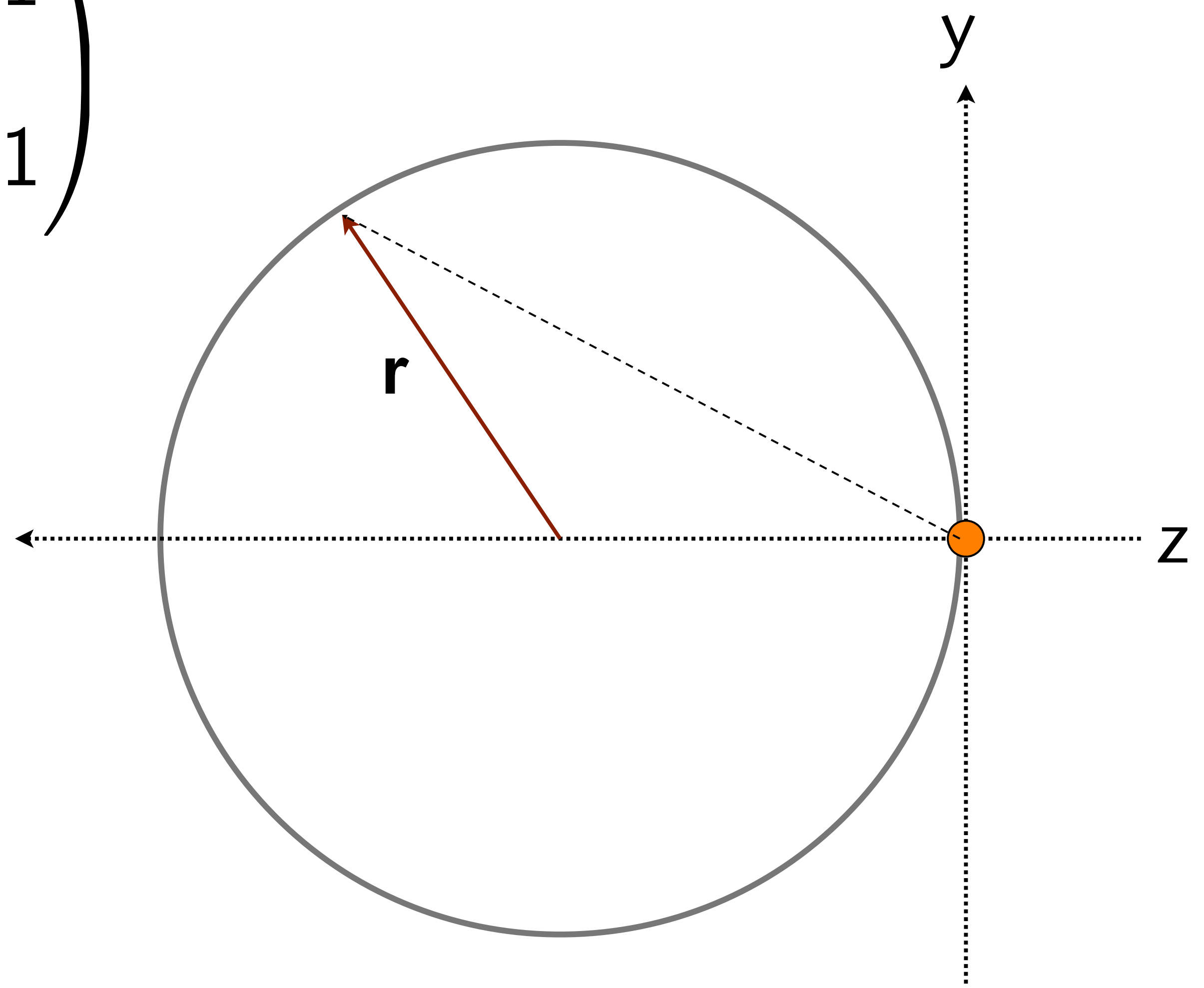


Mapping of the directional vector **r** onto (*u*,*v*)

- The sphere map contains (theoretically) a texel for every direction, except **r** = (0, 0, -1)
- Mapping:

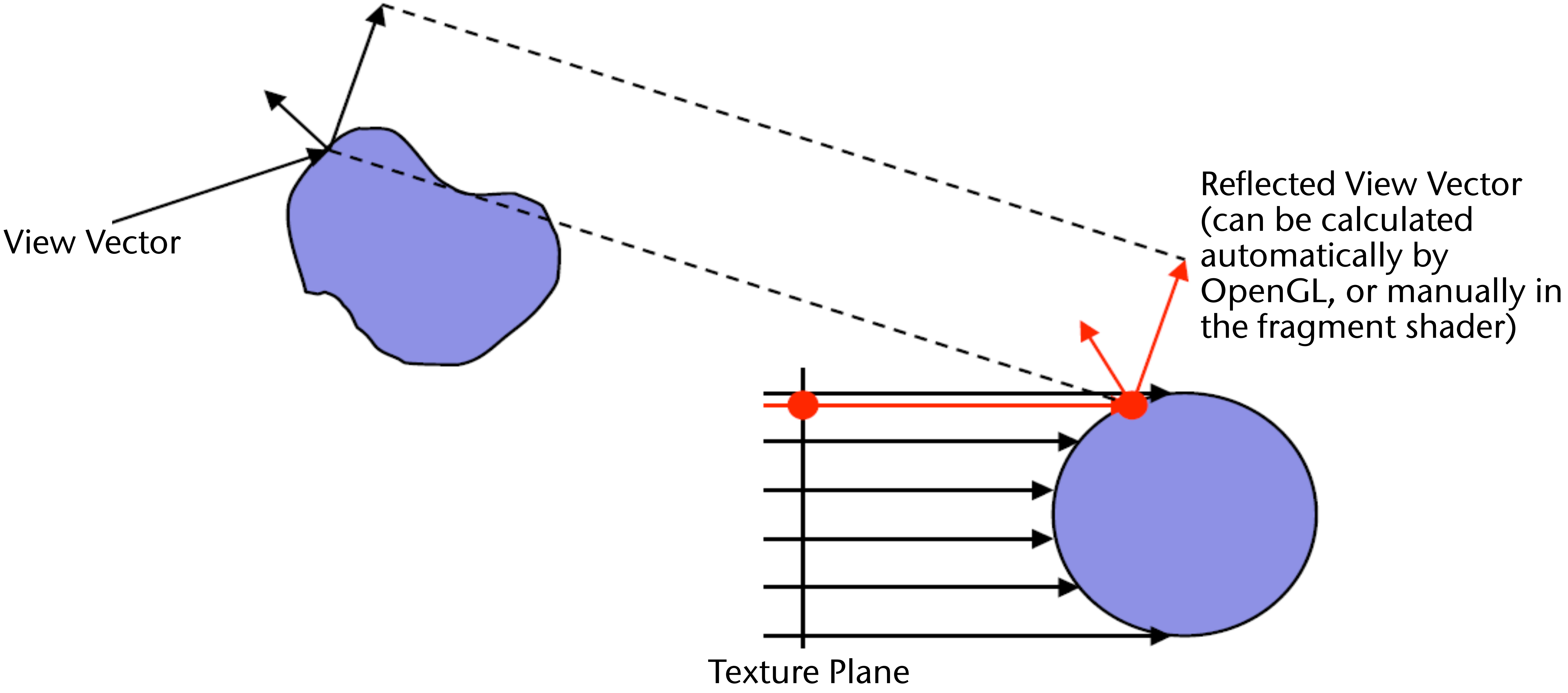
$$\begin{pmatrix} u \\ v \end{pmatrix} = \frac{1}{2} \begin{pmatrix} \frac{r_x}{\|(r_x, r_y, r_z) + (0,0,1)\|} + 1 \\ \frac{r_y}{\|(r_x, r_y, r_z) + (0,0,1)\|} + 1 \end{pmatrix}$$





FYI

- Application of the sphere mapping to texturing:



Bremen

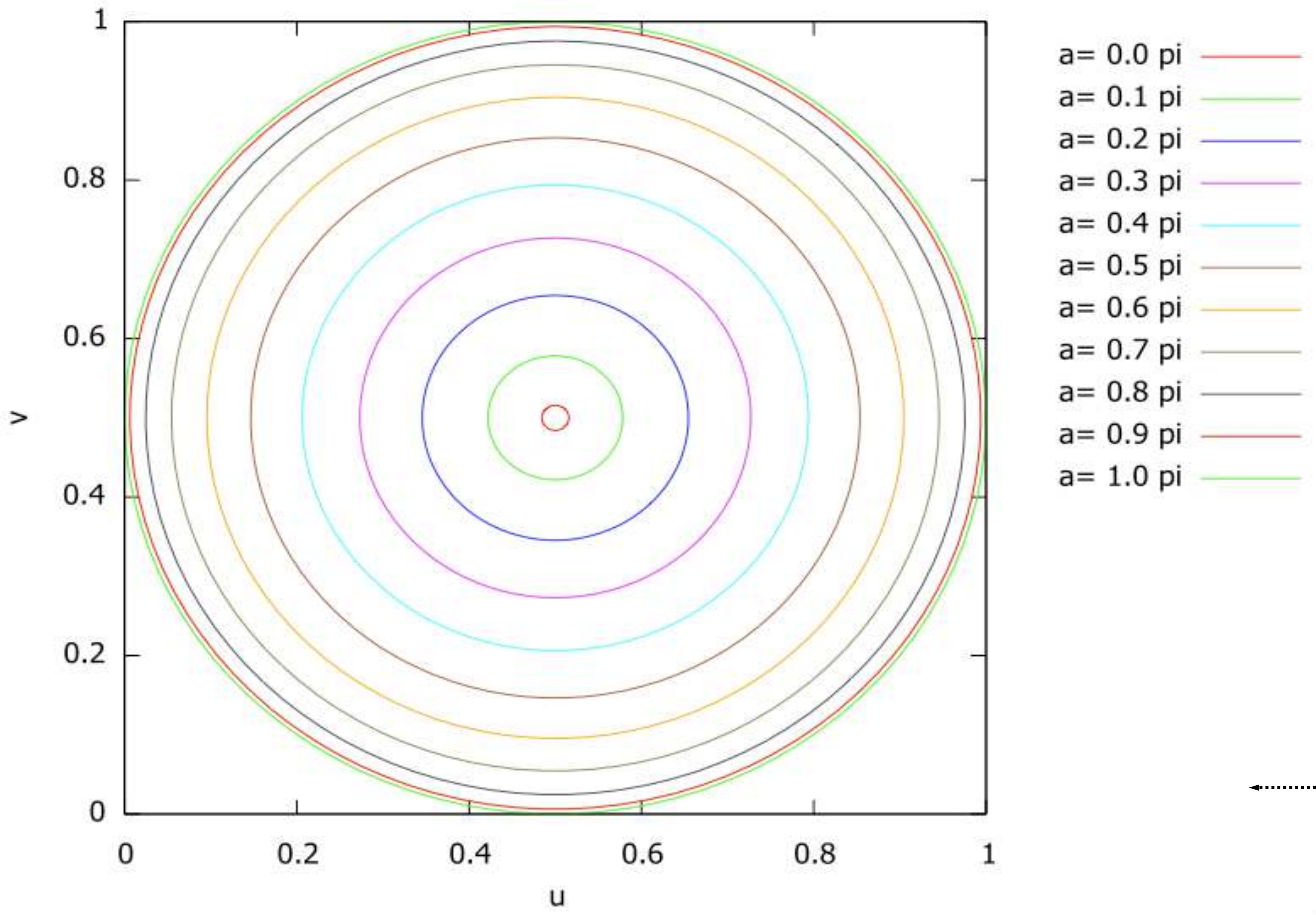
Simple Example

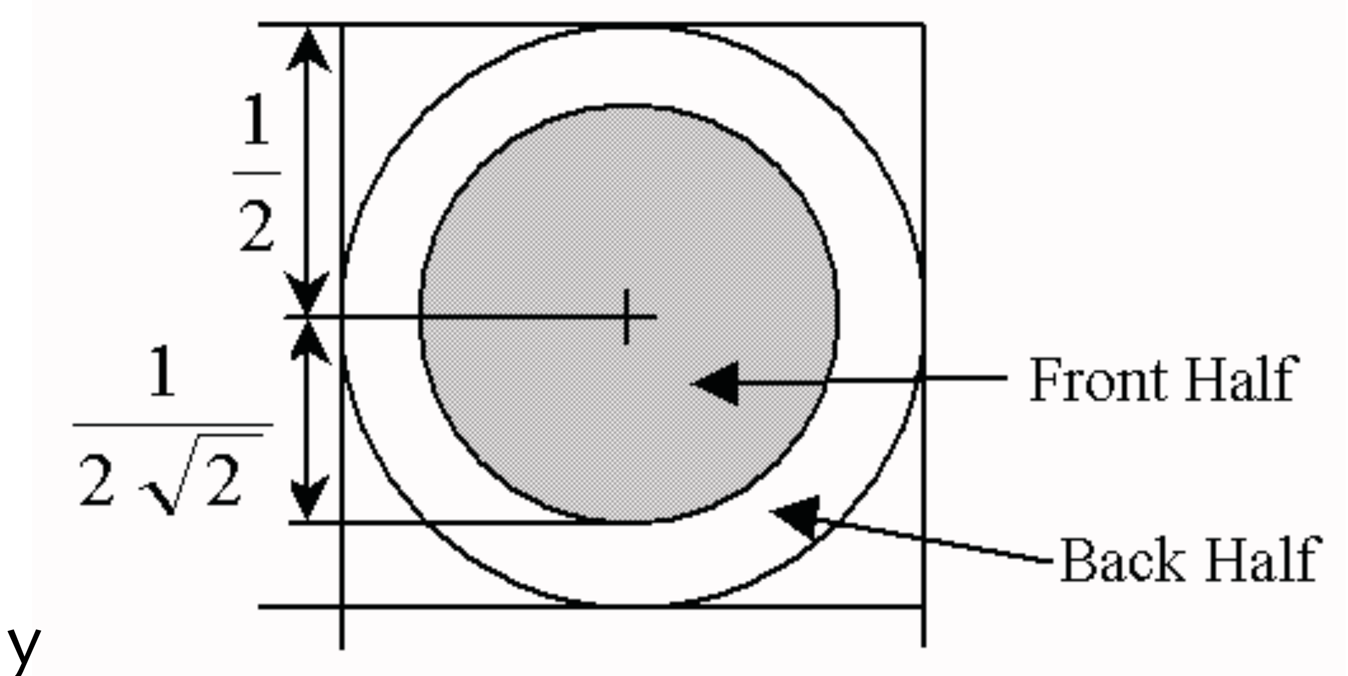
FYI

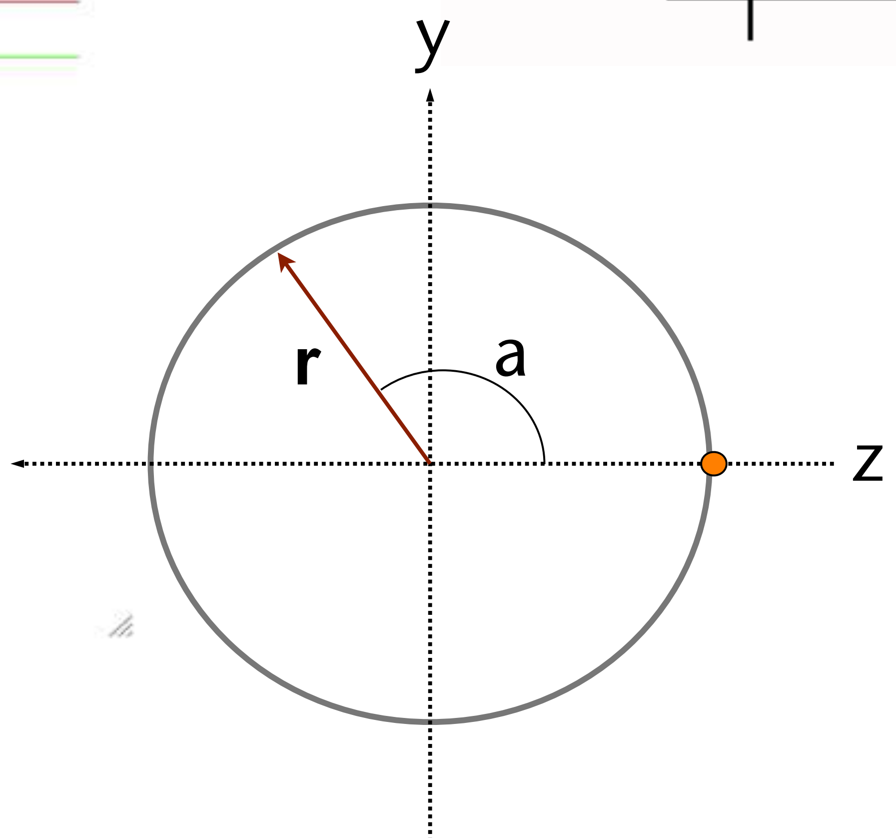
Problems

FYI

- Unfortunately, the mapping/sampling is not very uniform:

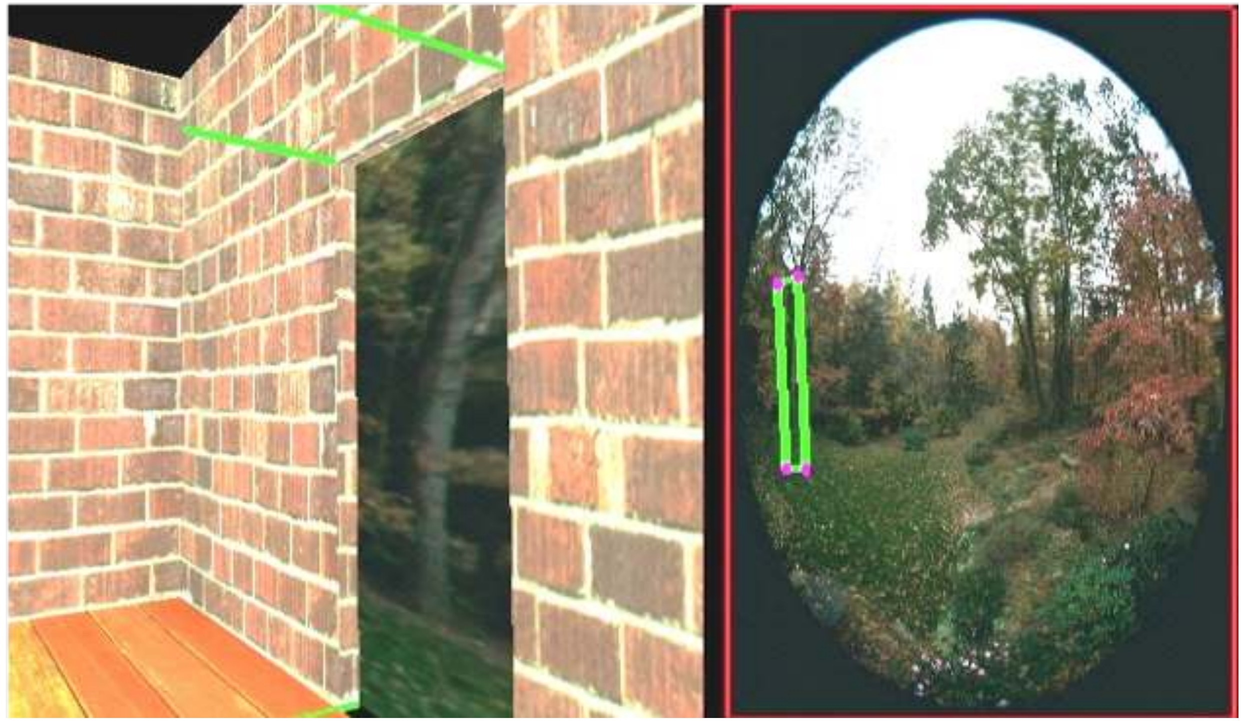


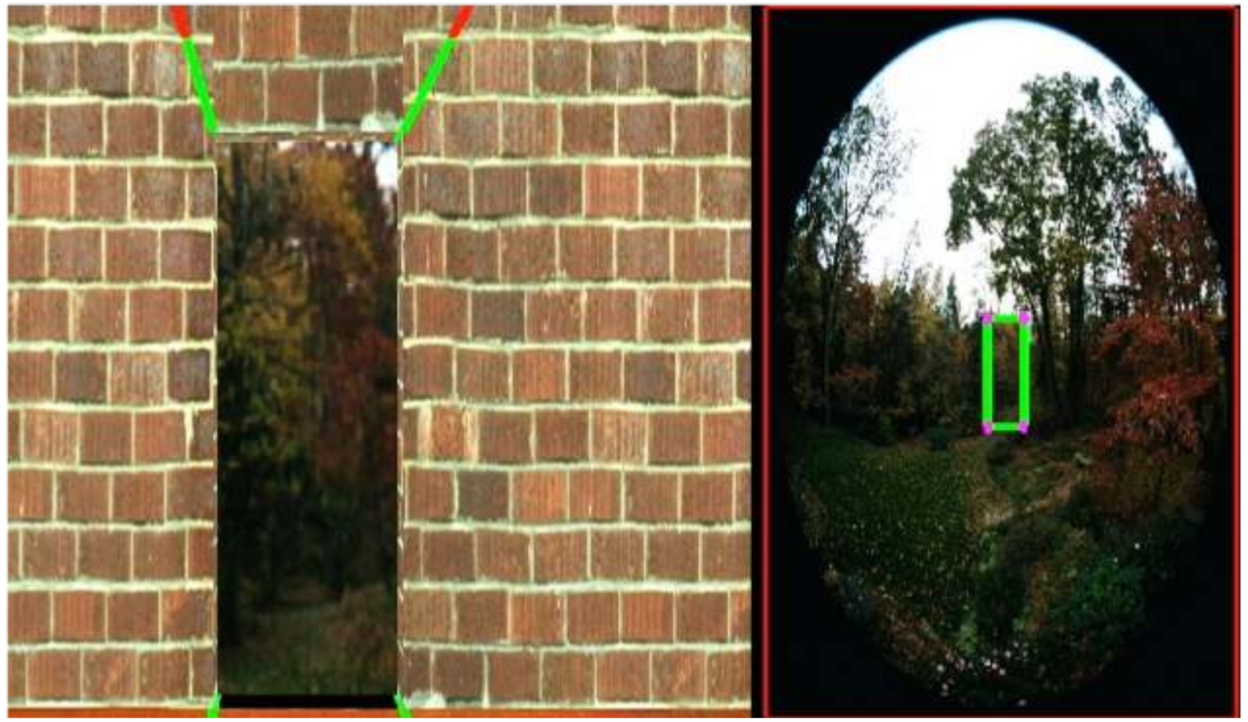




FYI

- Speckles if the reflecting vector comes close to the edge of the texture (through aliasing and "wrap-around")
- Texture coords are interpolated linearly (by the rasterizer), but the sphere map is non-linear
  - 2D rasterization hardware doesn't know about sphere maps, it just linearly interpolates texture coords
- Long polygons can cause serious "bends" in the texture





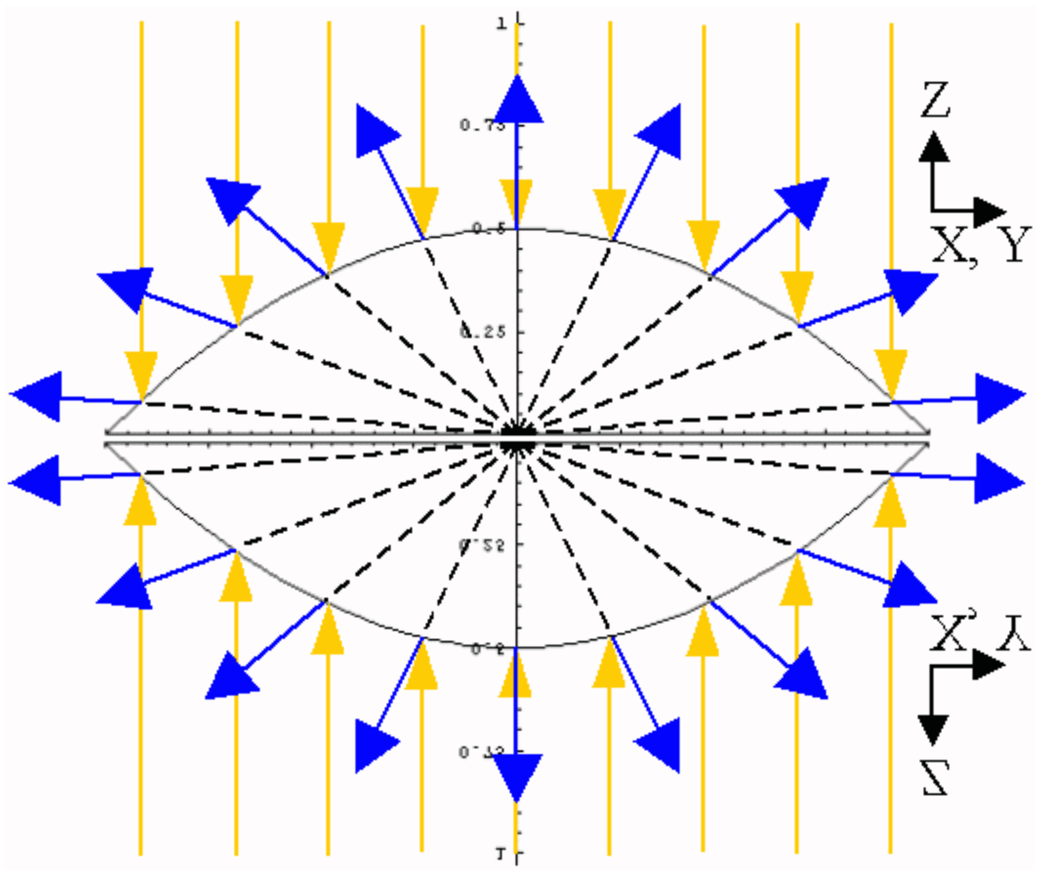
FYI

- Other cons:
  - Textures are difficult to generate by program (other than ray-tracing)
  - Viewpoint dependent: the center of the spherical texture map represents the vector that goes directly back to the viewer!
    - Can be made *view independent* with some OpenGL extensions
- Pros:
  - Easy to generate texture coordinates
  - Supported in OpenGL

Dual Parabolic Environment Mapping

FYI

- Idea:
  - Map the environment onto two textures via a reflective double paraboloid
  - Pros:
    - Relatively uniform sampling
    - *View independent*
    - Relatively simple computation of texture coordinates
    - Also works in OpenGL
    - Also works in a single rendering pass (just needs multi-texturing)
  - Cons:
    - Produces artifacts when interpolating across the edge



Dynamic Environment Maps

FYI

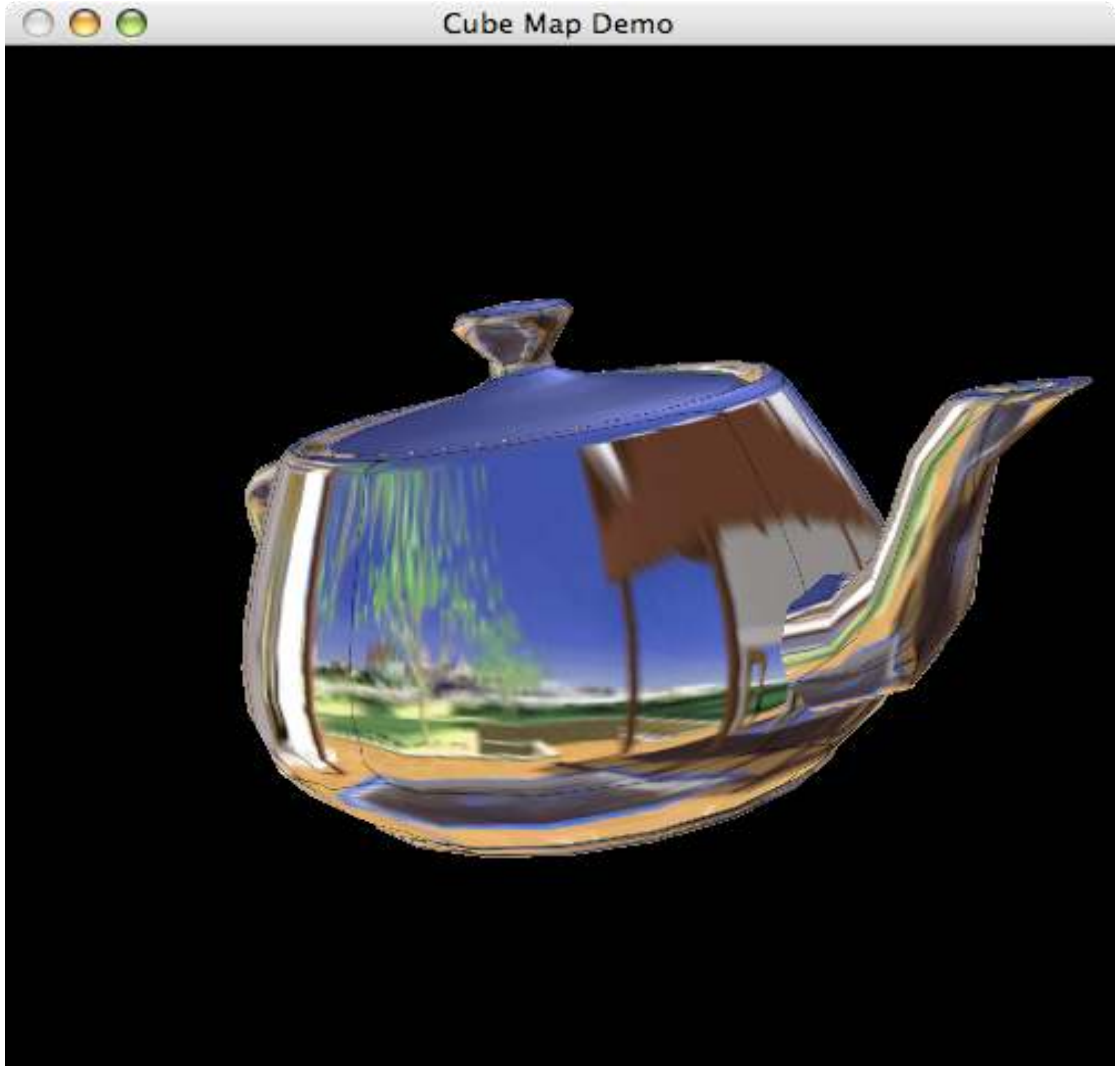
- Until now: environment map was invalid as soon as something in the environmental scene had changed!
- Idea:
  - Render the scene from the "midpoint" outward (typically 6x for a cube map)
  - Transfer framebuffer to texture (using the appropriate mapping)
  - Render the scene again from the viewpoint, this time with environment mapping

➢ Multi-pass rendering

- Typically used with cube maps ⟶ dynamic cube maps

Demo with Static Environment

FYI



Only for Your Information - Not Part of the Exam

Dynamic Environment Mapping in OpenGL Using 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)
...
```

Only for Your Information - Not Part of the Exam

```
// 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 that should appear later as reflection
  ...
  // read-back into corresponding texture map
  glCopyTexImage2D( cm_face[i], 0, GL_RGB, 0, 0, cm_size, cm_size, 0 );
}
```

Only for Your Information - Not Part of the Exam

```
// 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 reflective object in 7th pass
...

// 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.

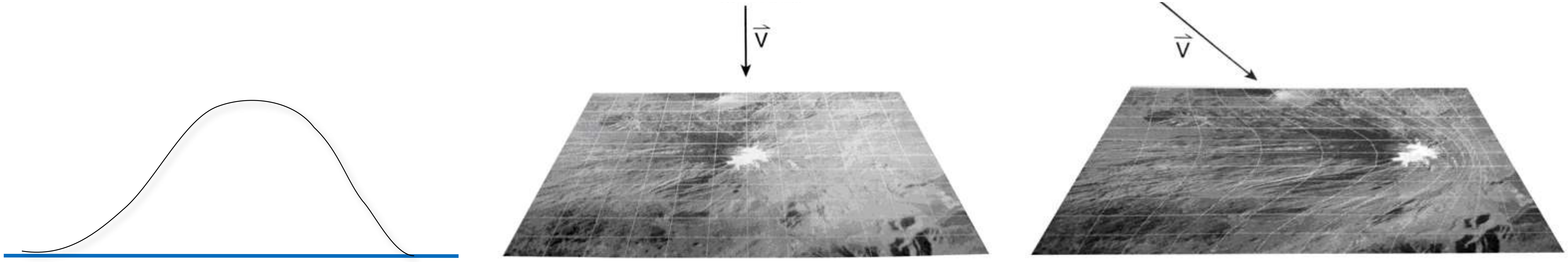
For Further Reading (On the course's homepage)

- "OpenGL Cube Map Texturing" (Nvidia, 1999)
  - With example code
  - Here several details are explained (e.g. the orientation)
- "Lighting and Shading Techniques for Interactive Applications" (Tom McReynolds & David Blythe, Siggraph 1999);
- SIGGRAPH '99 Course: "Advanced Graphics Programming Techniques Using OpenGL" (is part of the above document)

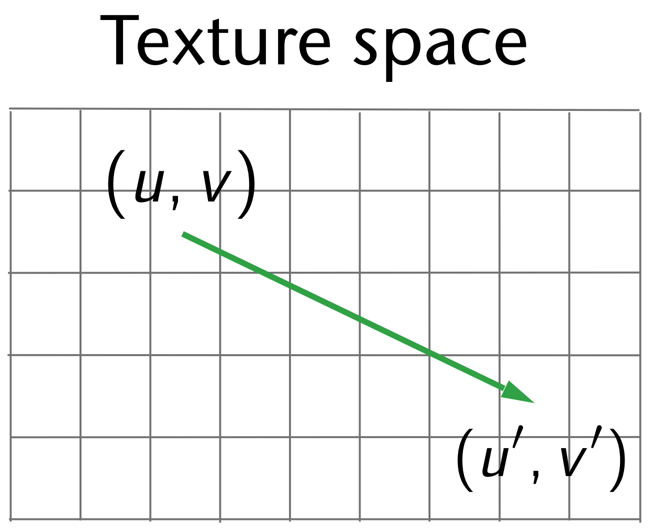
An Optical Illusion

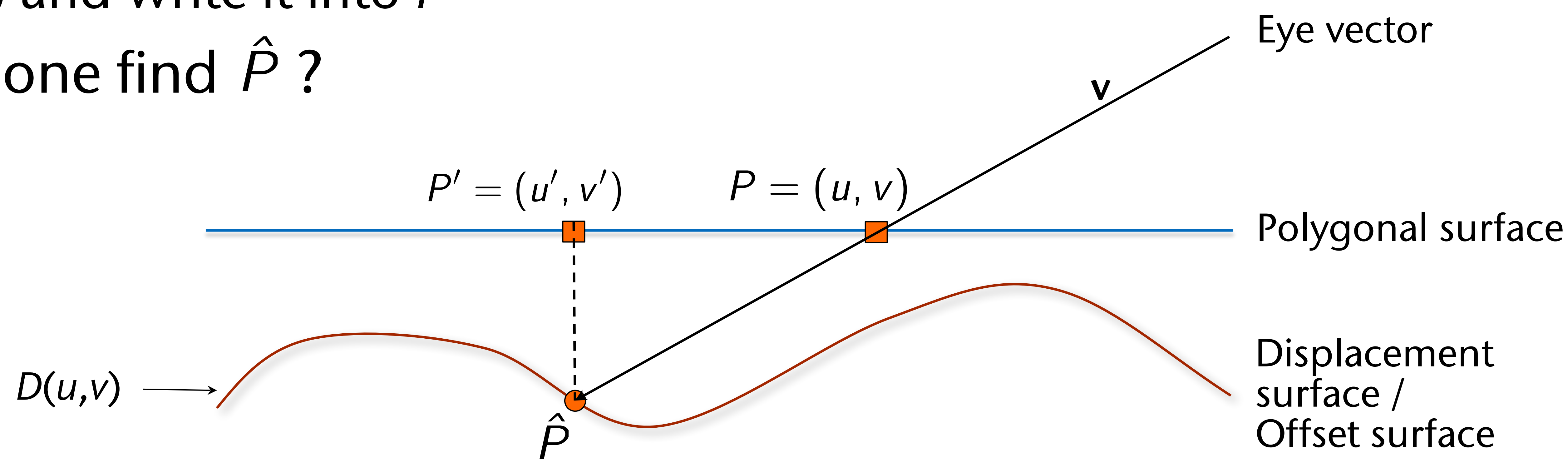
Parallax Mapping

- Motion parallax: near/distant objects shift differently relative to one another
- Problem with bump/normal mapping:
  - Given: coarse 3D geometry + 2D texture + detailed height map
  - Only the lighting is affected – the image of the texture on the surface remains *unchanged*, regardless of the viewing direction
- Example of effect of motion parallax:



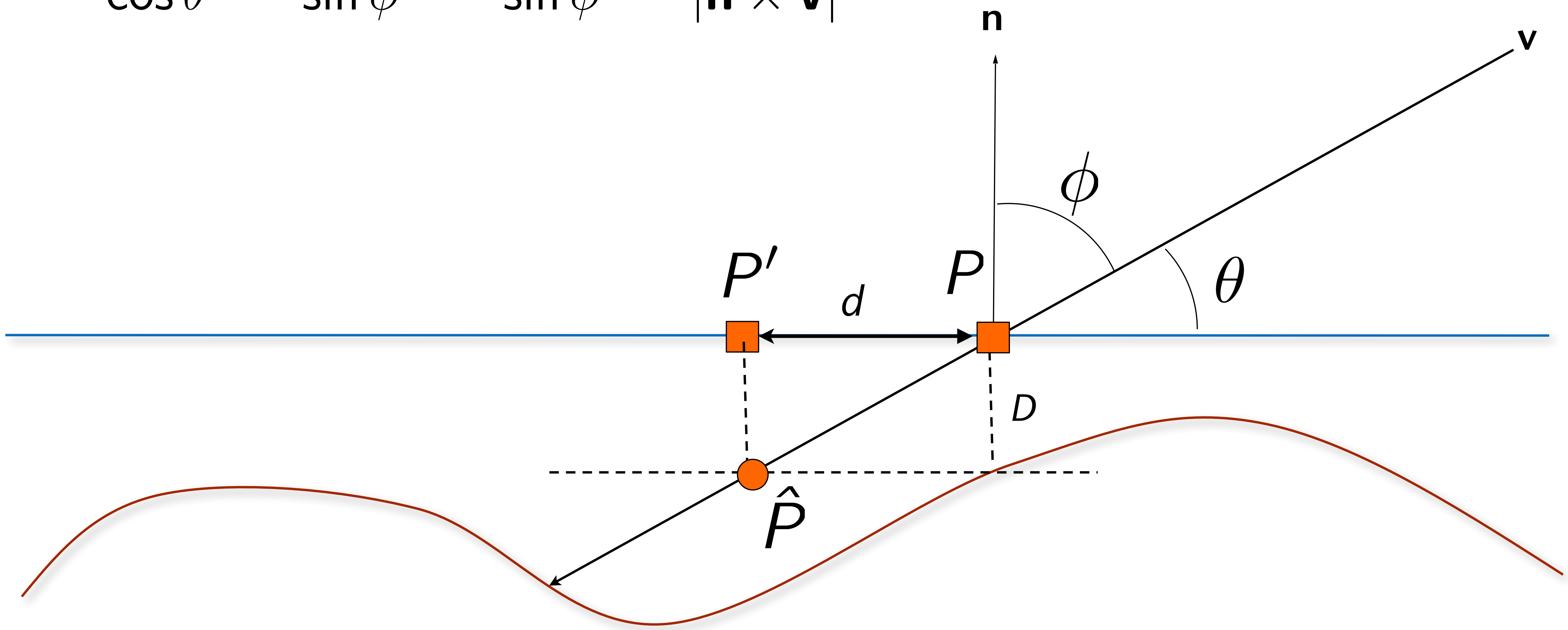
- Goal: "fake" motion parallax of a *detailed* offset surface, although we only render a *coarse* polygonal geometry
- The general task in parallax mapping:
  - Assume that scan line conversion is at pixel P
  - Determine point $\hat{P}$ that *would* be seen along **v**
  - Project $\hat{P}$ onto polygonal surface → P'
  - Read texel at (u', v') and write it into P
- Problem: how does one find $\hat{P}$?





Simplest Idea

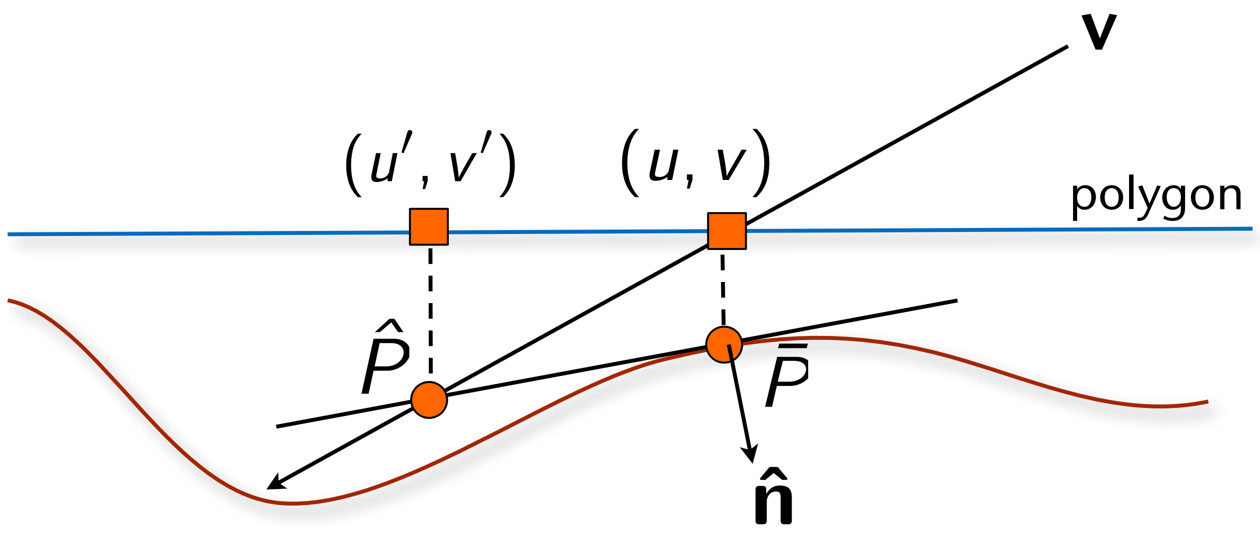
- We know the height $D = D(u,v)$ at point $P = P(u,v)$
- Use this as an approximation of $D(u',v')$ in point $P' = P'(u',v')$
- $\frac{D}{d} = \tan\theta = \frac{\sin\theta}{\cos\theta} = \frac{\cos\phi}{\sin\phi} = \frac{\cos\phi}{\sin\phi} = \frac{|\mathbf{nv}|}{|\mathbf{n}\times\mathbf{v}|}$



Improvement

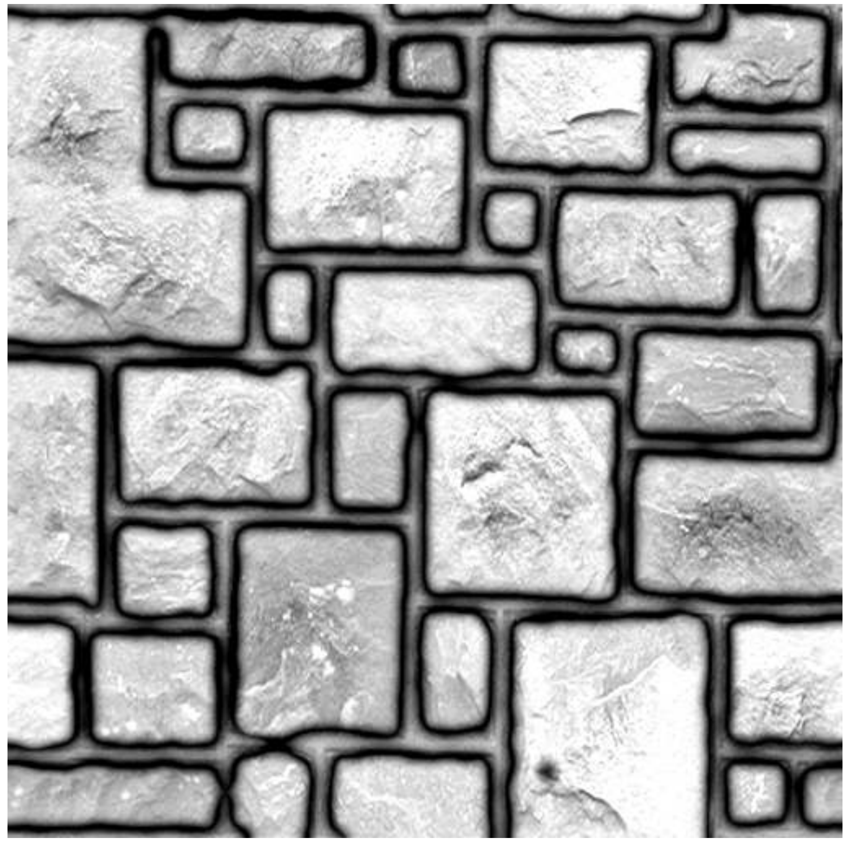
- Let $\bar{P} = (u, v, D)$ with $D = D(u,v)$
- Approximate the heightmap in $\bar{P}$ by a tangent plane (similar to bump mapping)
- Calculate $\hat{P}$ = point of intersection between that plane and the view vector:

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



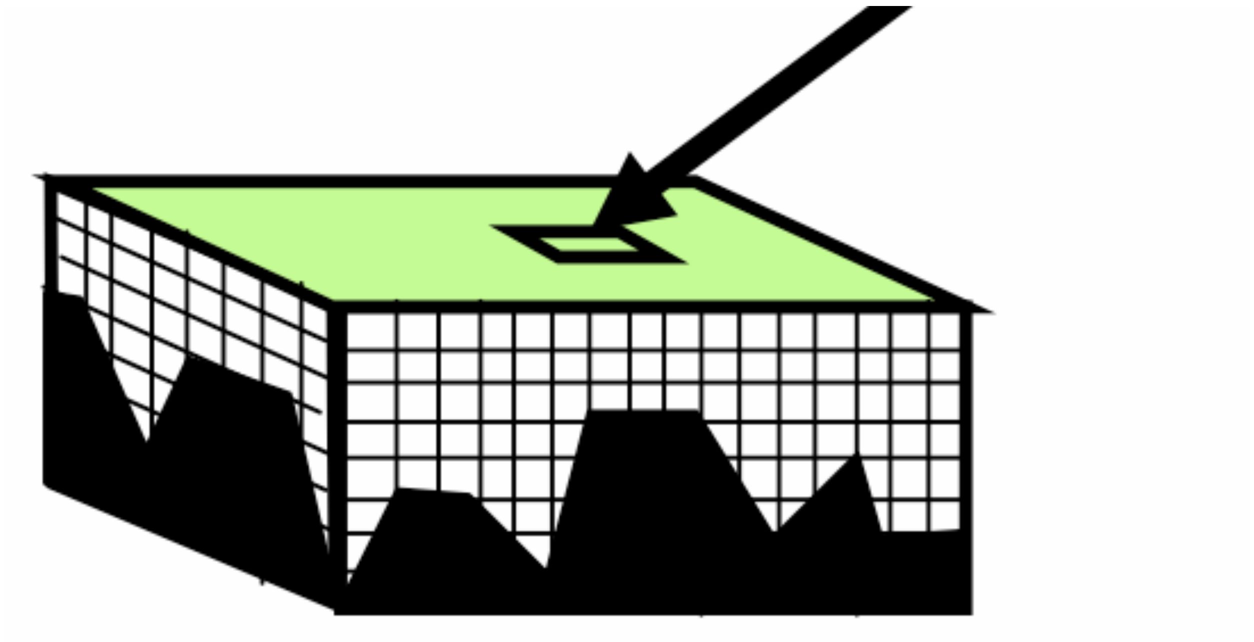
- Solve for t
- Then compute $\begin{pmatrix} u' \\ v' \end{pmatrix} = \begin{pmatrix} u \\ v \end{pmatrix} + t\mathbf{v}'$, with $\mathbf{v}' = (v_x, v_y, 0)$ (i.e., proj. in pgon's plane)
- Additional ideas: iterate; approximate heightmap with higher order

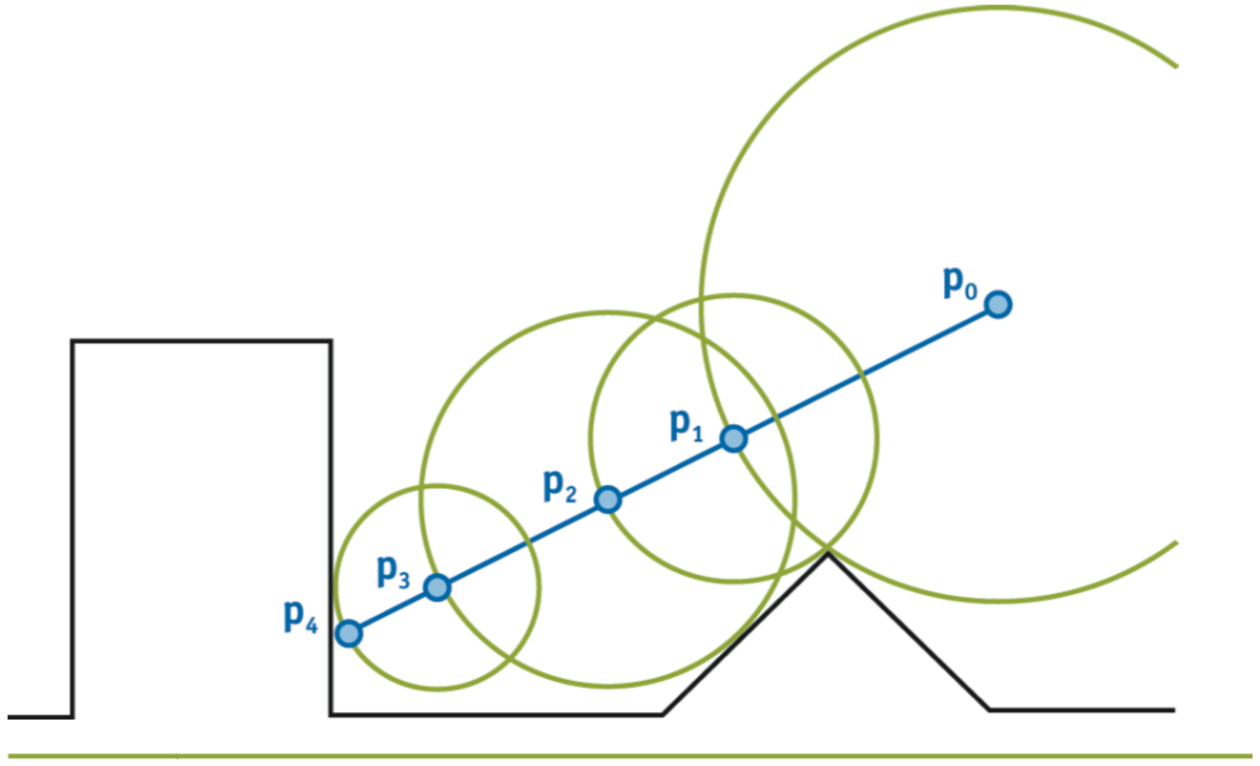
- Storage:
  - Put the image in the RGB channels of the texture
  - Put the height map in the alpha channel
- Process at rendering time:
  - Compute P' (see previous slide)
  - Calculate (u',v') of P' and lookup texel
  - Perturb normal by bump mapping (see CG1)
    - Note: today one can calculate directional derivatives for D_u and D_v "on the fly" (needed in bump mapping)
  - Evaluate lighting model with texel color and perturbed normal



Alternative

- Do sphere tracing along the view vector, until you hit the offset surface
  - If the heightmap contains heights that are not too large, it is sufficient to begin relatively close underneath/above the plane of reference
  - If the angle of the view vector is not too acute, then a few steps are sufficient
- For a number of voxel layers underneath the plane of reference, save the smallest distance to the offset surface for every cell





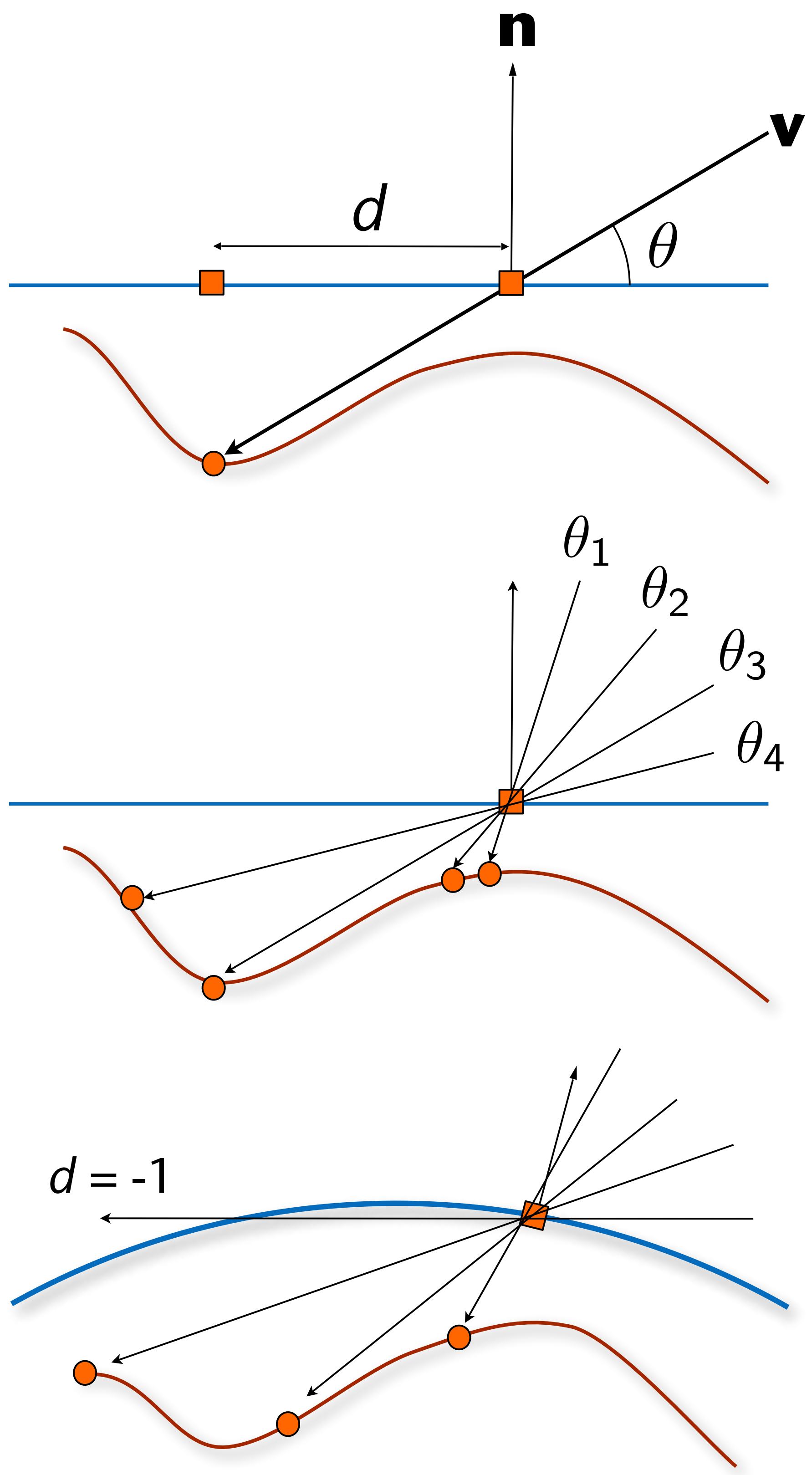
Example: Parallax Mapping vs Simple Texture Mapping

Left cube:
Phong lighting

Right cube:
Phong lighting,
with normal and
parallax mapping

View-Dependent Displacement Mapping (VDM)

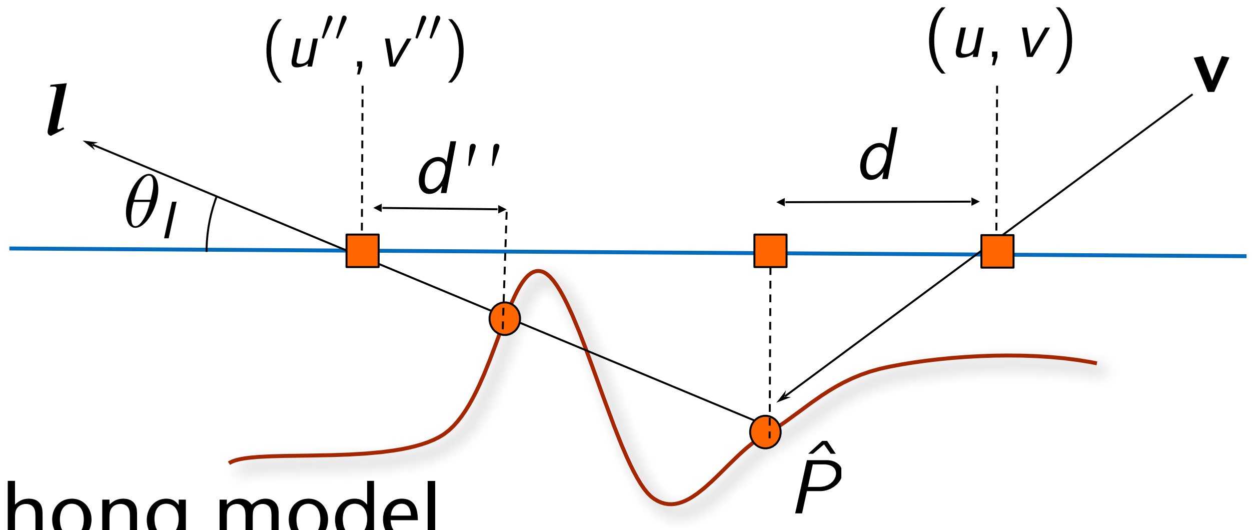
- Idea: precompute all possible texture coordinate displacements for all possible situations
- In practice:
  - Parameterize the viewing vector by (θ, ϕ) in the local coordinate system of the polygon
  - Precompute the texture displacement for all (*u*,*v*) and all possible (θ, ϕ)
    - E.g., by ray-casting of an explicit, temporarily generated mesh of the offset surface
  - Carry out the whole procedure for a set of *possible* curvatures *c* of the base surface
- Results in a 5-dim. "texture" (LUT): $d(u, v, \theta, \phi, c)$



- Advantage: results in a correct silhouette
  - Reason: $d(u, v, \theta, \phi, c) = -1$ for many parameters near the silhouette
  - These are the pixels that lie outside of the (true) silhouette!
- Further enhancement: self shadowing
  - Idea is similar to ray tracing: use "shadow rays"

1. Determine $\hat{P}$ from D and θ,ϕ (just like before) $\rightarrow$ (u,v) displacement d
2. Determine vector $\boldsymbol{l}$ from $\hat{P}$ to the light source and calc θ_l , ϕ_l from that
3. Determine $P'' = (u'', v'')$ from $\hat{P}$ and θ_l and ϕ_l
4. Make lookup in our "texture" $D \longrightarrow d''$
5. Test: $d'' + d < \|(u'', v'') - (u, v)\|$

$\rightarrow$ pixel (u,v) is in shadow, i.e., don't add light source in Phong model



Result

Bump Mapping

VDM

- Names:
  - Steep parallax mapping, parallax occlusion mapping, horizon mapping, view-dependent displacement mapping, ...
  - There are still many other variants ...
  - "Name ist Schall und Rauch!" ("A name is but noise and smoke!")

More Results

Bump mapping

Standard VDM

VDM with self-shadowing

All Examples Were Rendered with VDM