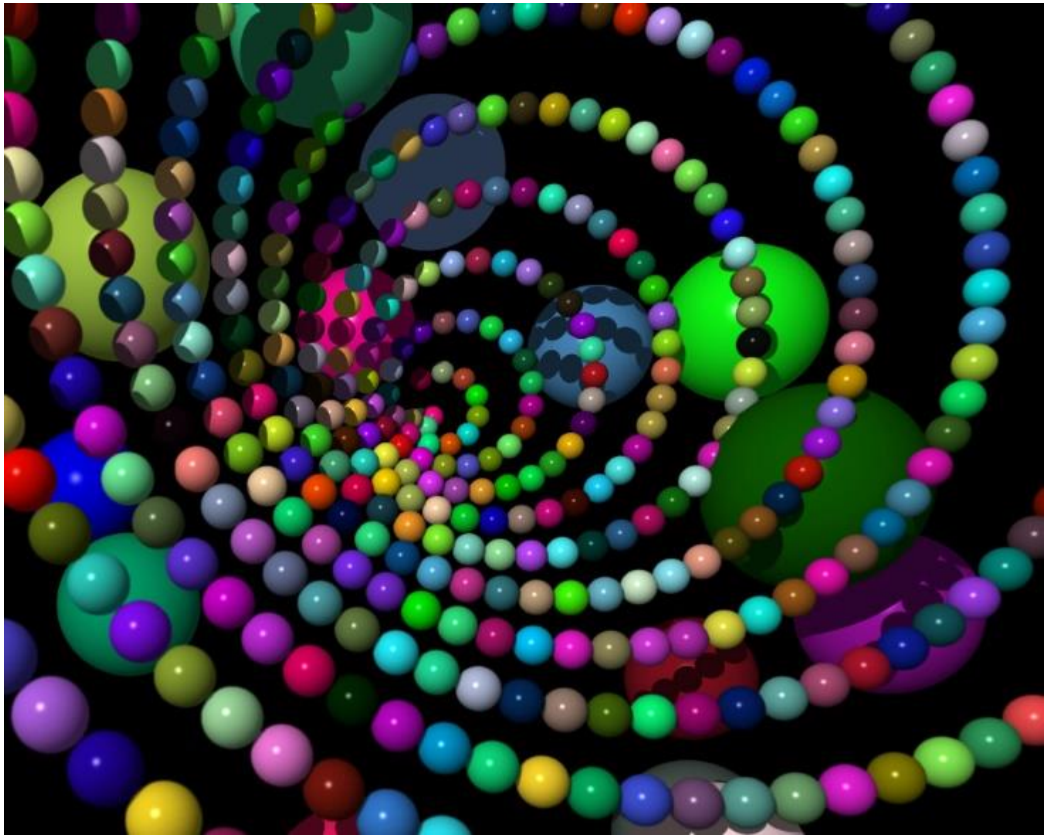
Computer Graphics

Texture Mapping

Introduction and Motivation

- so far: detail through polygons & materials
- images look very “plasticy”



[image: Marijn Stollenga]

Introduction and Motivation

- example: brick wall
- problem: extremely many polygons & materials needed for detailed structures
  $\rightarrow$ inefficient for memory and processing
- new approach necessary: texture mapping
- introduced by Ed Catmull (1974), extended by Jim Blinn (1976)

Introduction and Motivation

- several properties can be modified
  - color: diffuse component of surface
  - reflection: specular component of surface to simulate reflection (environment mapping)
  - normal vector: simulate 3D surface structure (bump mapping)
  - actual surface: raise/lower points to actually modify surface (displacement mapping)
  - transparency: make parts of a surface entirely or to a certain degree transparent

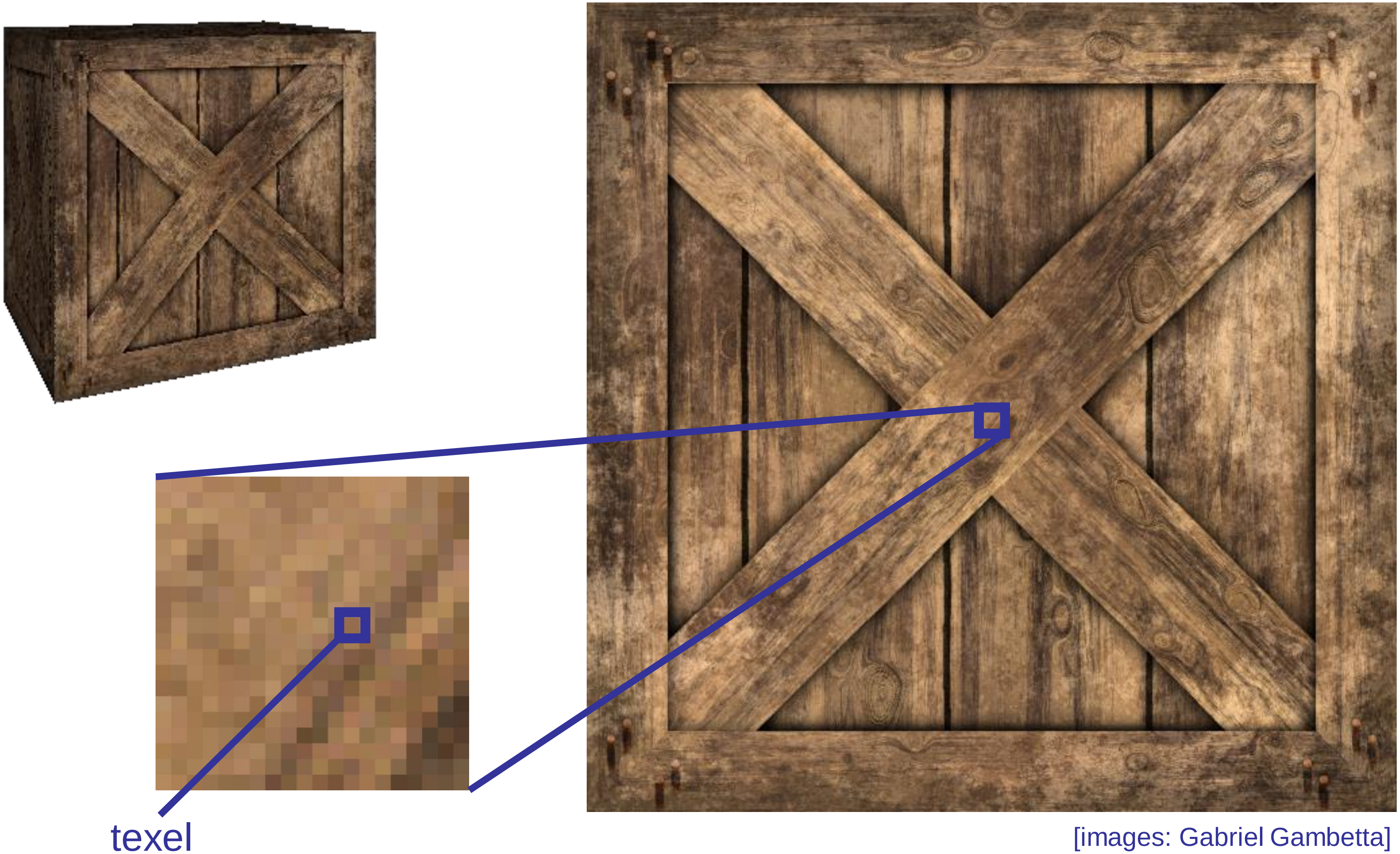
Texture Mapping

General Approach

Texture Mapping

- **texture**: typically 2D pixel image
- **texel**: pixel in a texture
- determines the appearance of a surface
- procedure to map the texture onto the surface needed
  - easy for single triangle
  - complex for arbitrary 3D surface
- goal: find easy way to do this mapping

Texture Mapping



[images: Gabriel Gambetta]

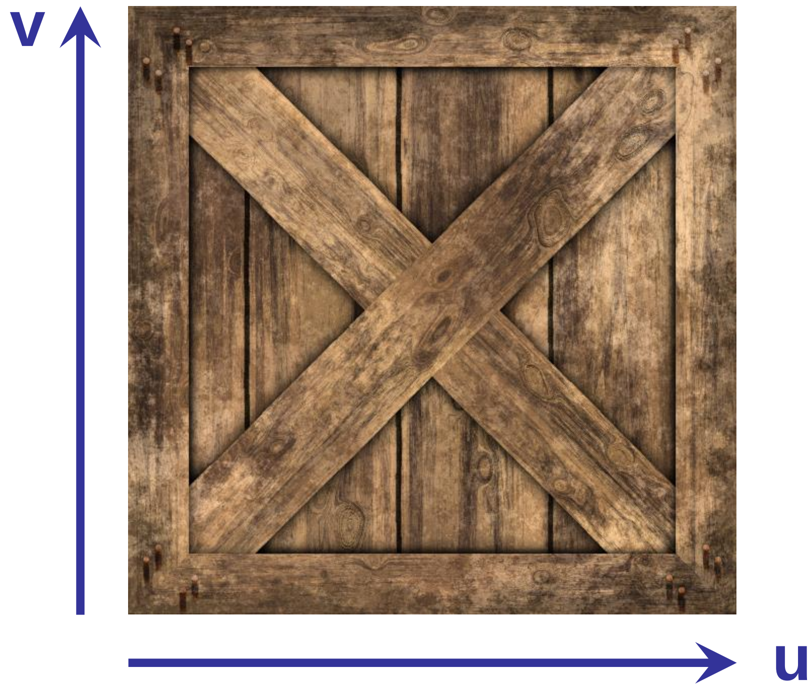
Texture Mapping

- rendering pipeline slightly modified to use new texture mapping function

- algorithm: for each pixel to be rendered
  - find depicted surface point
  - find point in texture (texel) that corresponds to surface point
  - use texel color to modify the pixel’s shading

Texture Mapping: Definitions

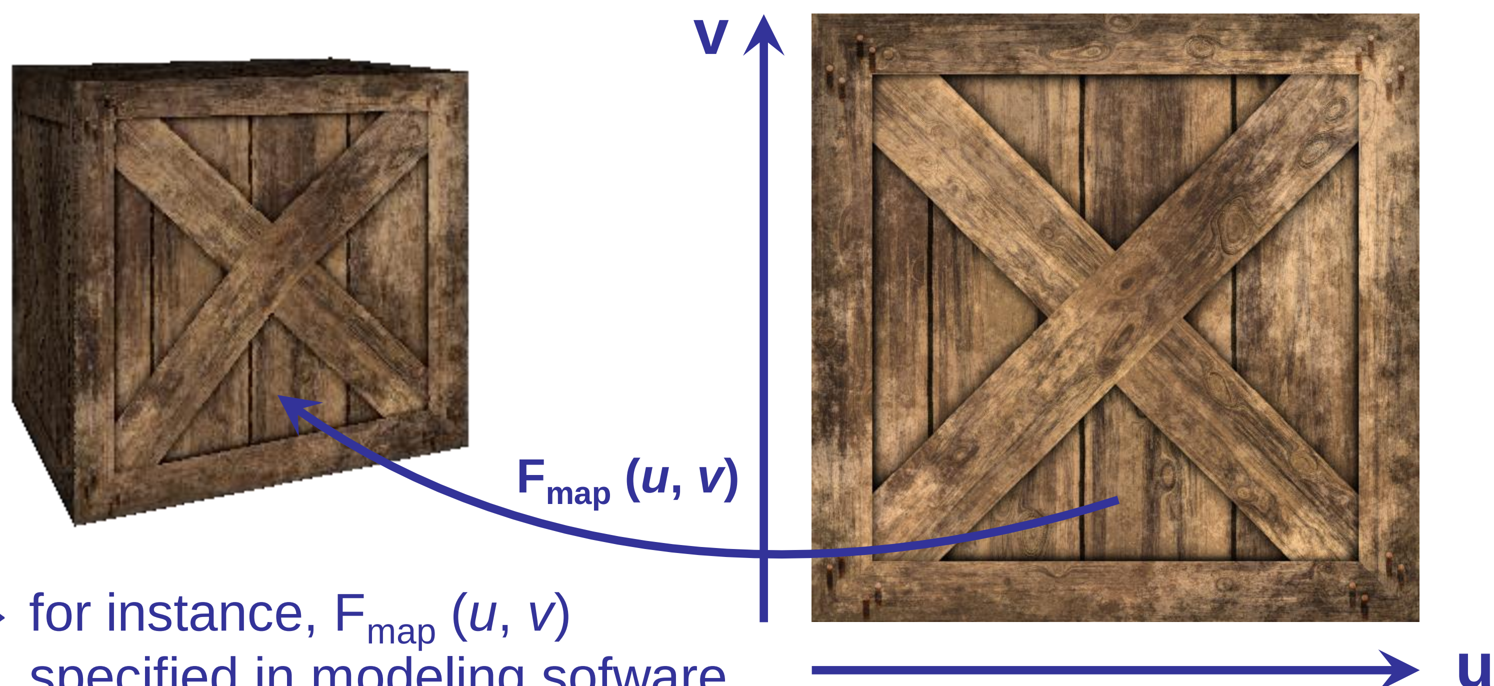
- 2D texture: function that maps points on the (u, v) plane to (r, g, b) values: $(r, g, b) = c_{tex}(u, v)$



[image: Gabriel Gambetta]

Texture Mapping: Definitions

- texture mapping function maps (u, v) values to (x, y, z) positions on objects: (x, y, z) = F_{map} (u, v)

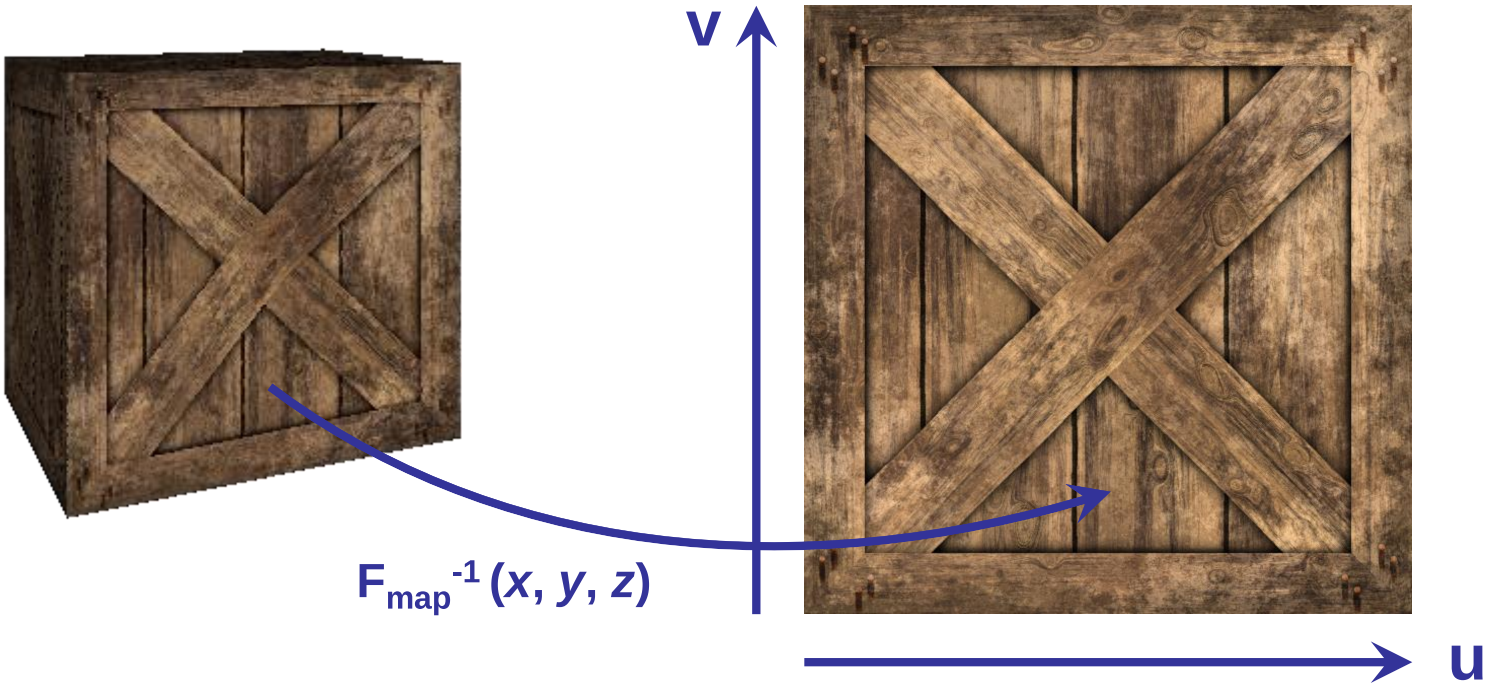


→ for instance, F_{map} (u, v) specified in modeling sofware

[images: Gabriel Gambetta]

Texture Mapping: Definitions

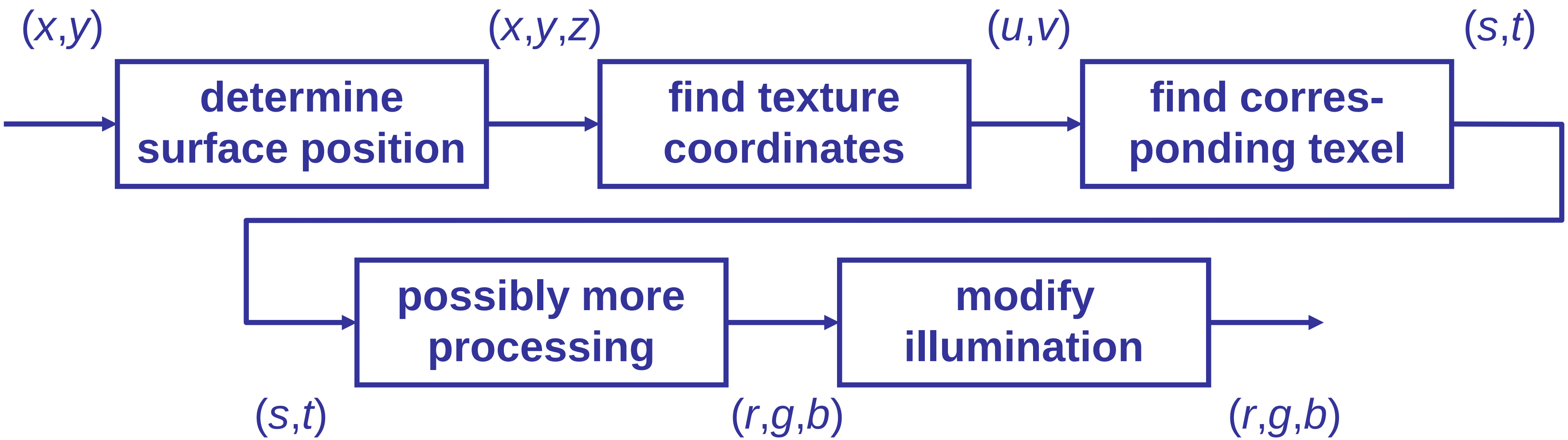
- for rendering, we need to solve the inverse function: find (u, v) for a (x, y, z) position: $(u, v) = F_{map}^{-1}(x, y, z)$



[images: Gabriel Gambetta]

Texture Mapping: General Procedure

- general texture mapping pipeline:



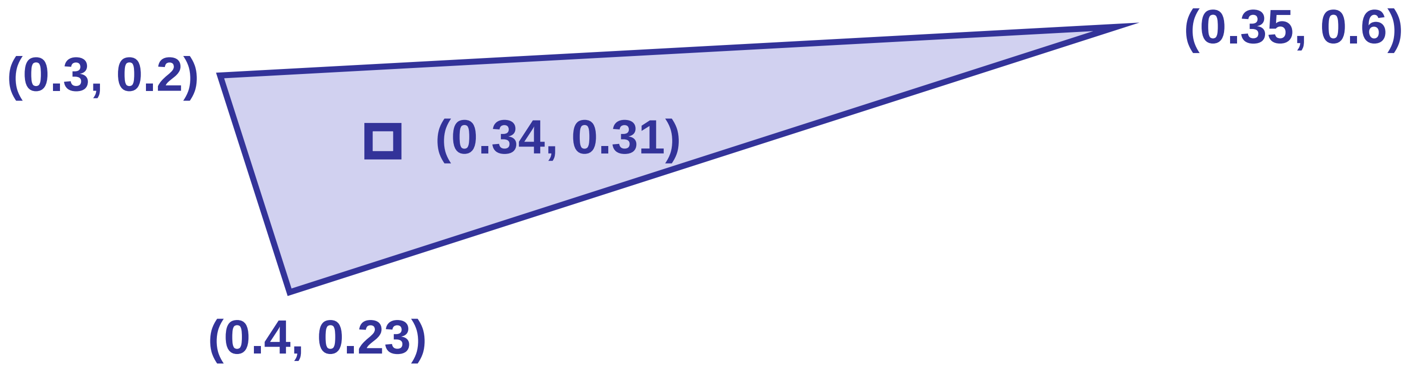
1. compute texture color for surface point
2. use to modify parameters in Phong illumination

u,v-Coordinates: Projector Functions

- goal: derive u, v texture coordinates from a given 3D point that is being rendered
- $F_{map}^{-1}: \Re^3 \rightarrow \Re^2$, so $F_{map}^{-1}(x, y, z) = (u, v)$
- several typical possibilities
  - (manual) parameterization of the surface
  - use of inherent (u, v) coordinates (e.g., freeform surfaces or primitive shapes)
  - two step technique

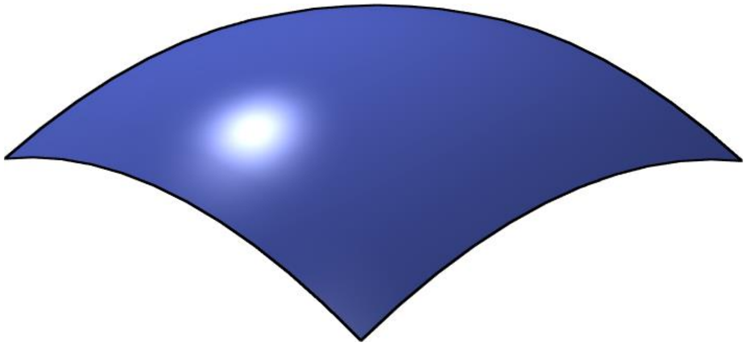
(Manual) Surface Parameterization

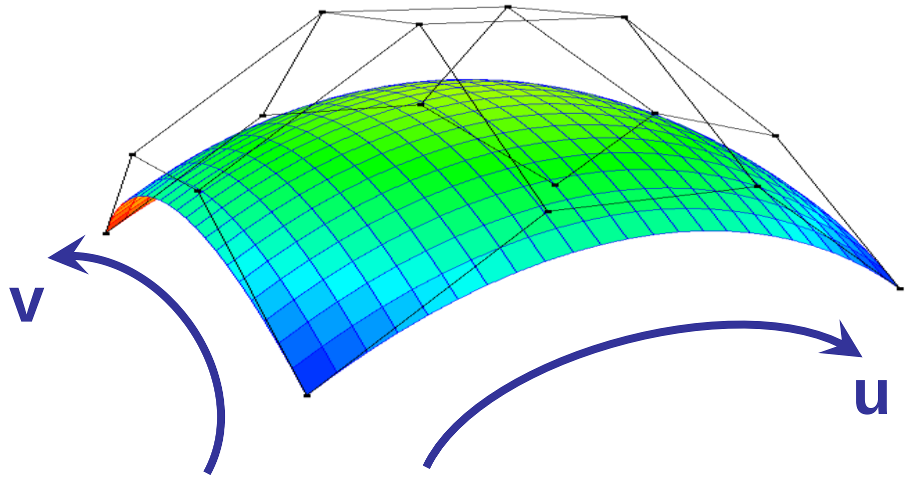
- simplest technique: specification of (u, v) texture coordinates during modeling for all vertices of a polygon
- interpolation between these values for points inside the polygon (e.g., barycentric interpolation for triangles)



Inherent (*u*, *v*) Coordinates

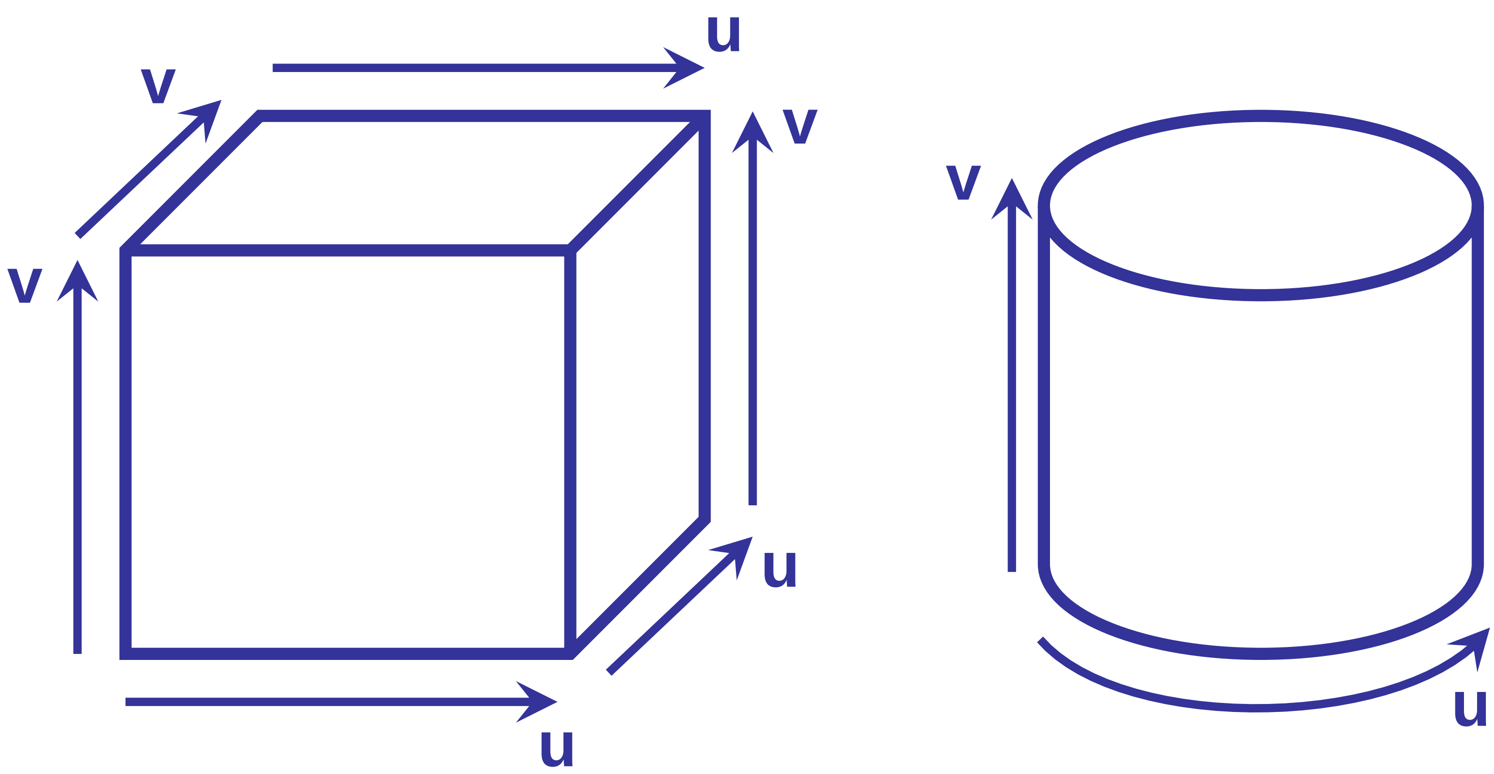
- (*u*, *v*) coordinates derived from parameter directions of surface patches
(e.g., Bézier and spline patches)





Inherent (*u*, *v*) Coordinates

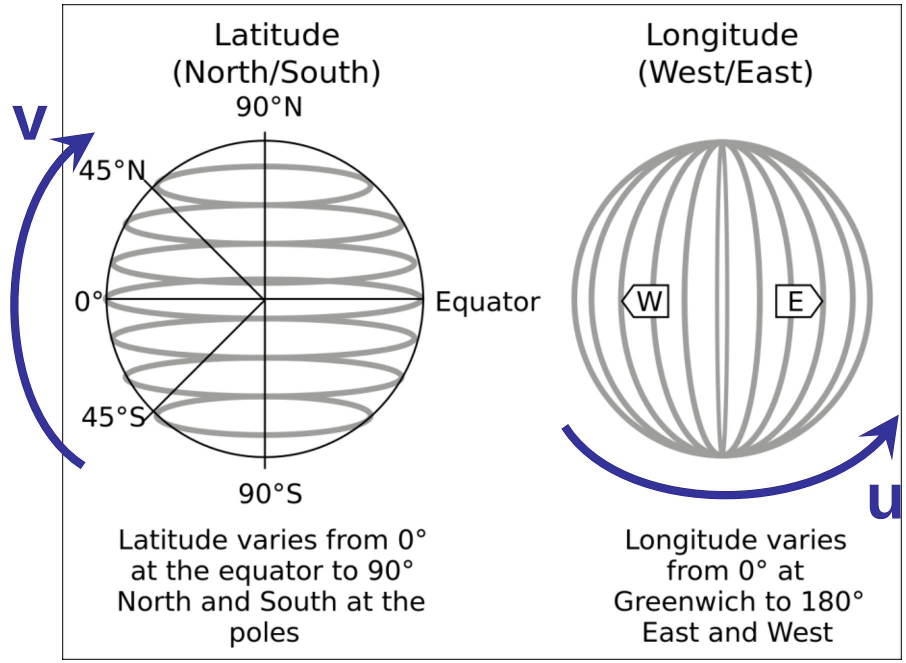
- obvious (*u*, *v*) coordinates derived for primitive shapes (e.g., boxes, spheres, cones, cylinders, etc.)



Inherent (*u*, *v*) Coordinates

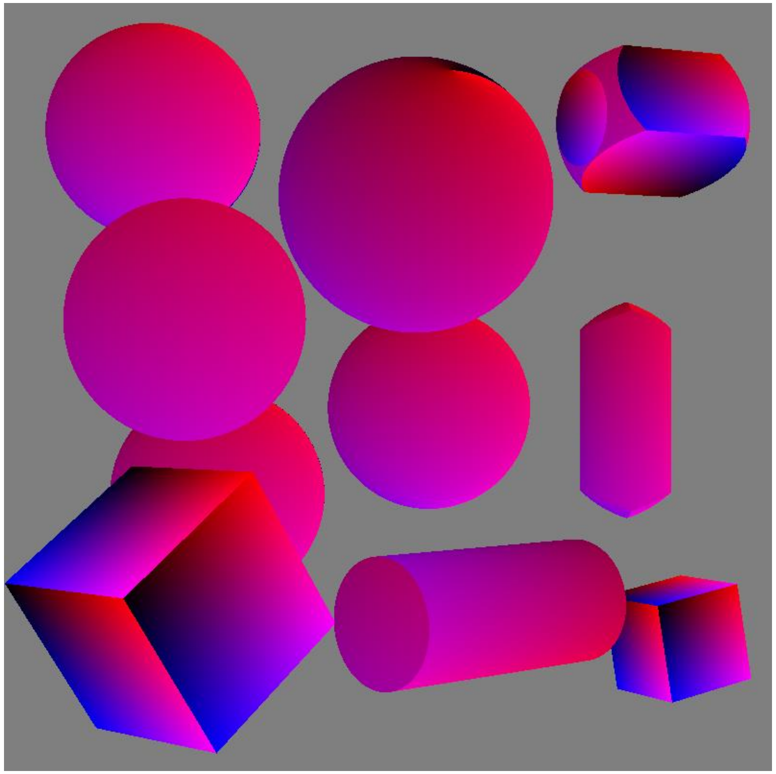
- obvious (*u*, *v*) coordinates derived for primitive shapes (e.g., boxes, spheres, cones, cylinders, etc.)

[image: Flickr Kevin Gill]



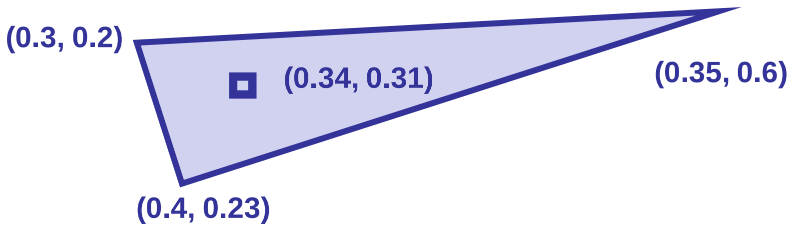
Inherent (*u*, *v*) Coordinates

- examples for simple shapes, with (u, v) coordinates mapped to red-blue color

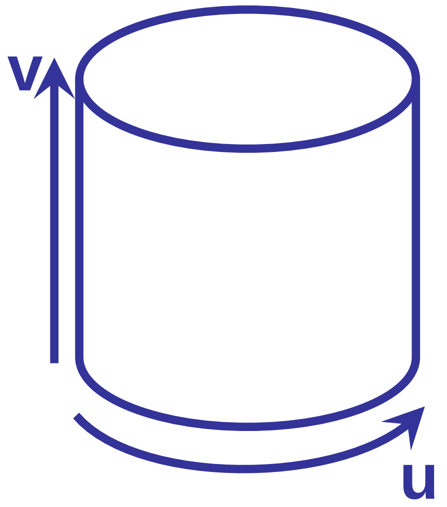


u,v-Coordinates: Projector Functions

- manual specification:
  → flexible, but tedious
  and inconvenient



- inherent (*u*, *v*) coordinates:
  → inflexible (relies on
  a few simple shapes)
  but easy to compute



- combination of both that is flexible and easy to compute? → two-step approach

Texture Mapping

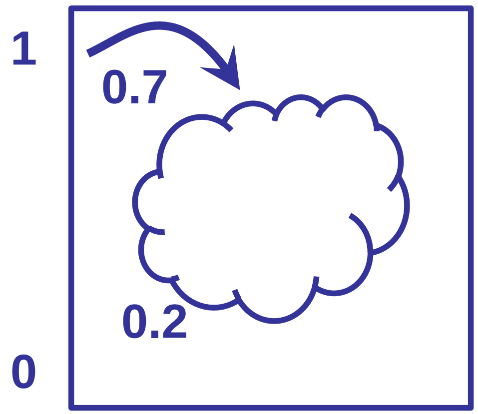
Two-Step Approach

Two Step Approach

- problem with previous techniques:
  - not flexible enough (inherent coordinates)
  - too tedious (manual parameterization)
- new idea:
  - texture mapped on simple intermediate surface that has inherent coordinates
  - then transfer onto complex objects
- common intermediate surfaces: cylinder, sphere, plane, box

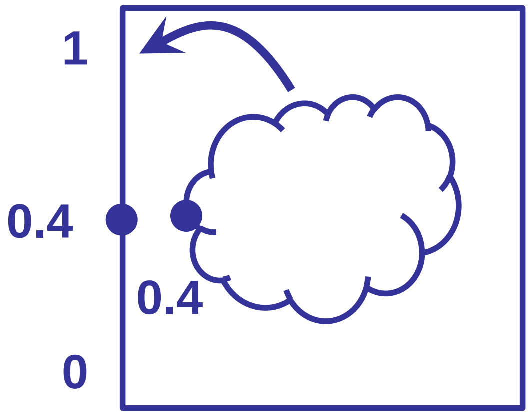
Two Step Approach

- two steps:
  – mapping of 2D texture coordinates onto simple 3D surface (s-mapping)
  – mapping of the now 3D texture pattern onto complex object (o-mapping)



Two Step Approach

- in practice – inverse approach:
  - mapping of object point onto simple surface
    O: $f(x_o, y_o, z_o) = (x_i, y_i, z_i)$
  - mapping of surface point onto texture
    S: $f(x_i, y_i, z_i) = (u, v)$

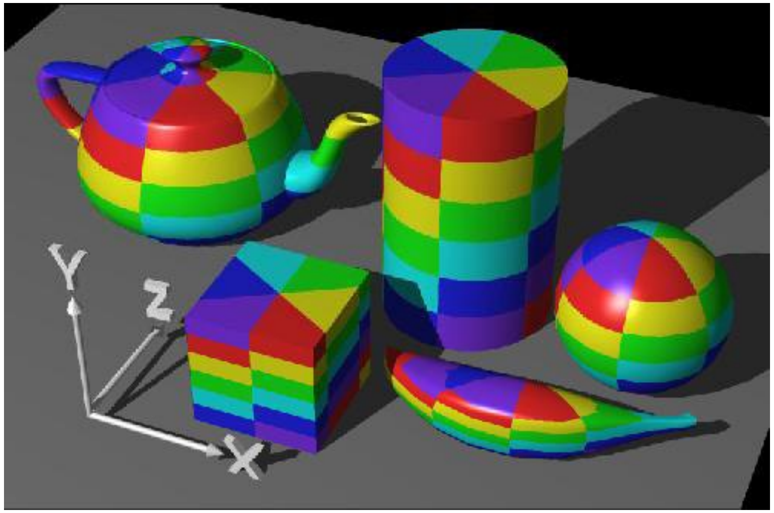


Cylindrical Mapping

- mapping onto cylinder surface given by height h_0 and angle θ_0

$$S : (\theta, h) \to (u, v) = \left(\frac{r}{c}(\theta - \theta_0), \frac{1}{d}(h - h_0) \right)$$

  using scaling factors c, d, and the radius r
- discontinuity along one line parallel to center axis



from R. Wolfe: *Teaching Texture Mapping*

Examples of Cylindrical Maps

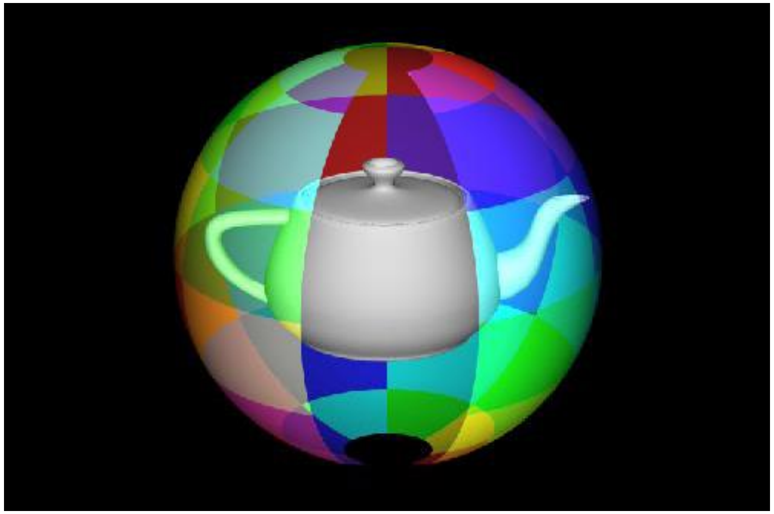
Examples of Cylindrical Maps

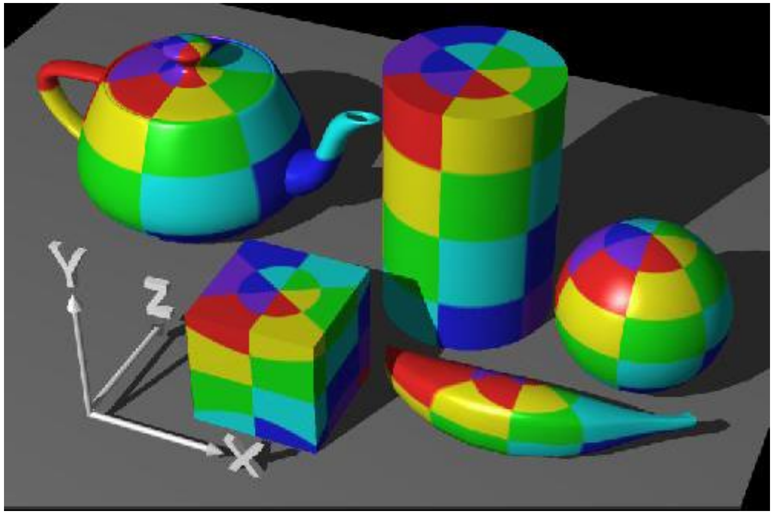
Spherical Mapping

- mapping onto surface of a sphere given by spherical coordinates

$$S : (r, \phi, \theta) \rightarrow (u, v) = \left(\frac{\theta}{2\pi}, \frac{(\pi / 2) + \phi}{\pi} \right)$$

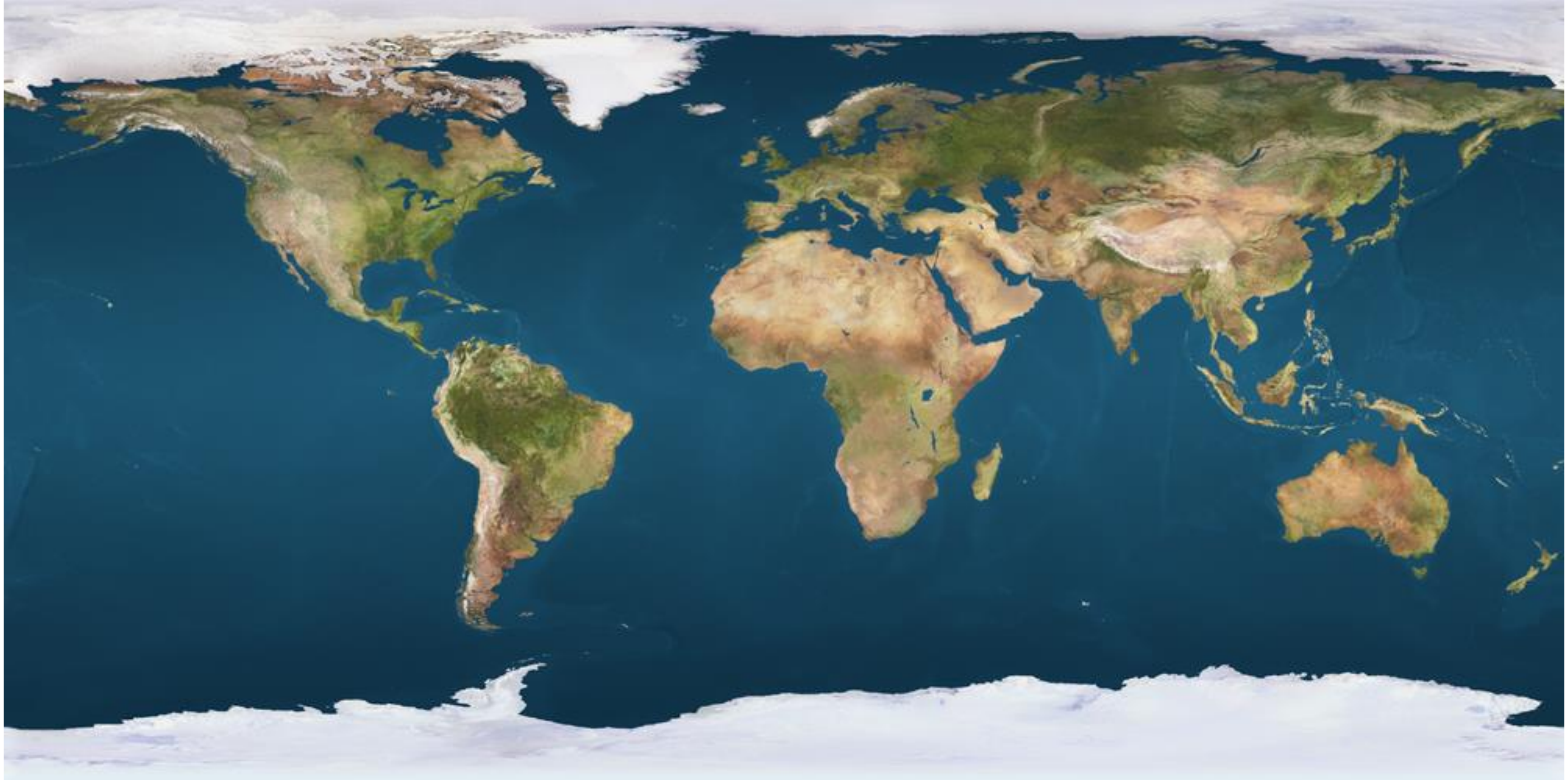
- no non-distorting mapping possible between plane and sphere surface





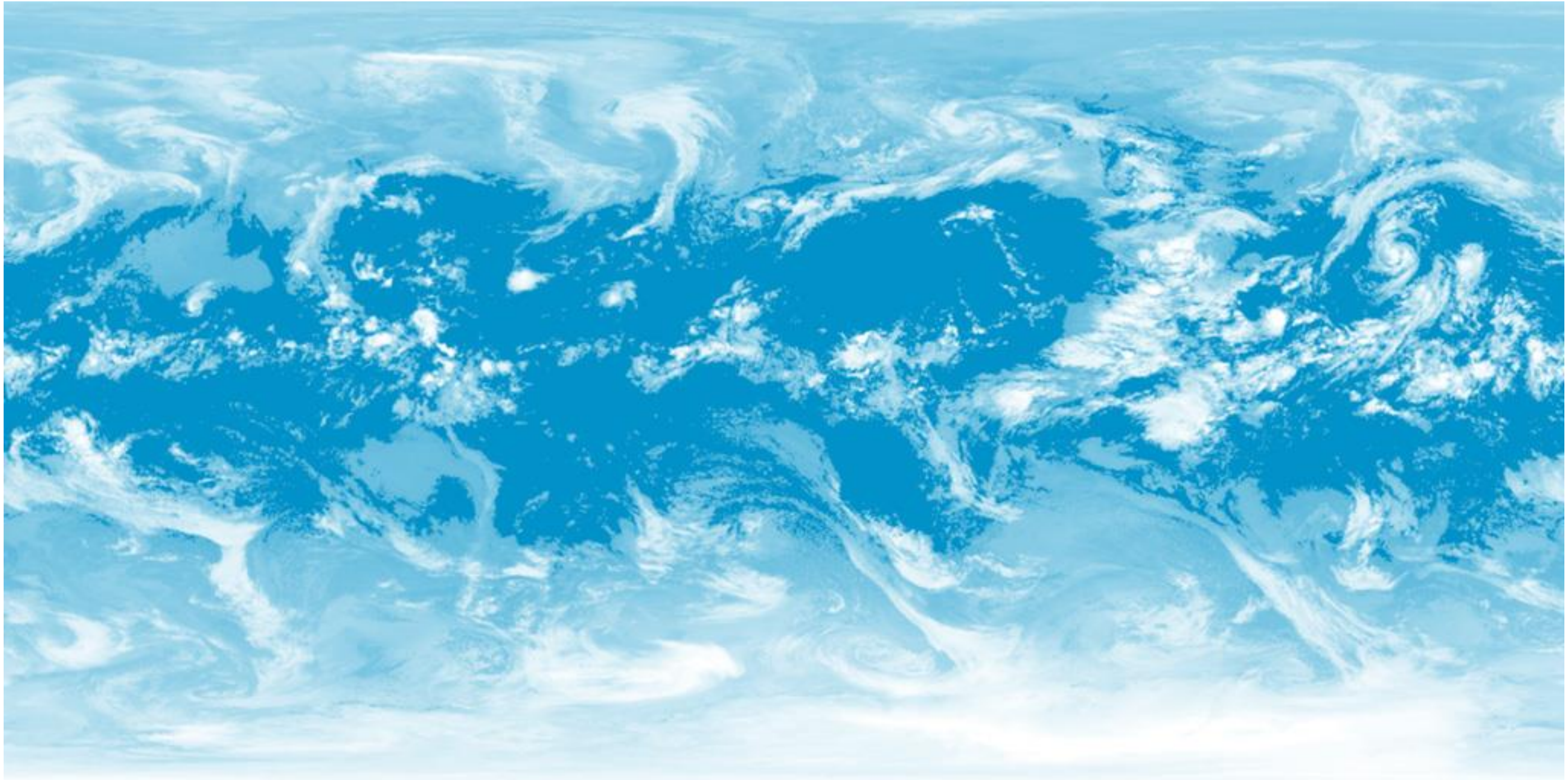
from R. Wolfe: *Teaching Texture Mapping*

Examples for Spherical Maps



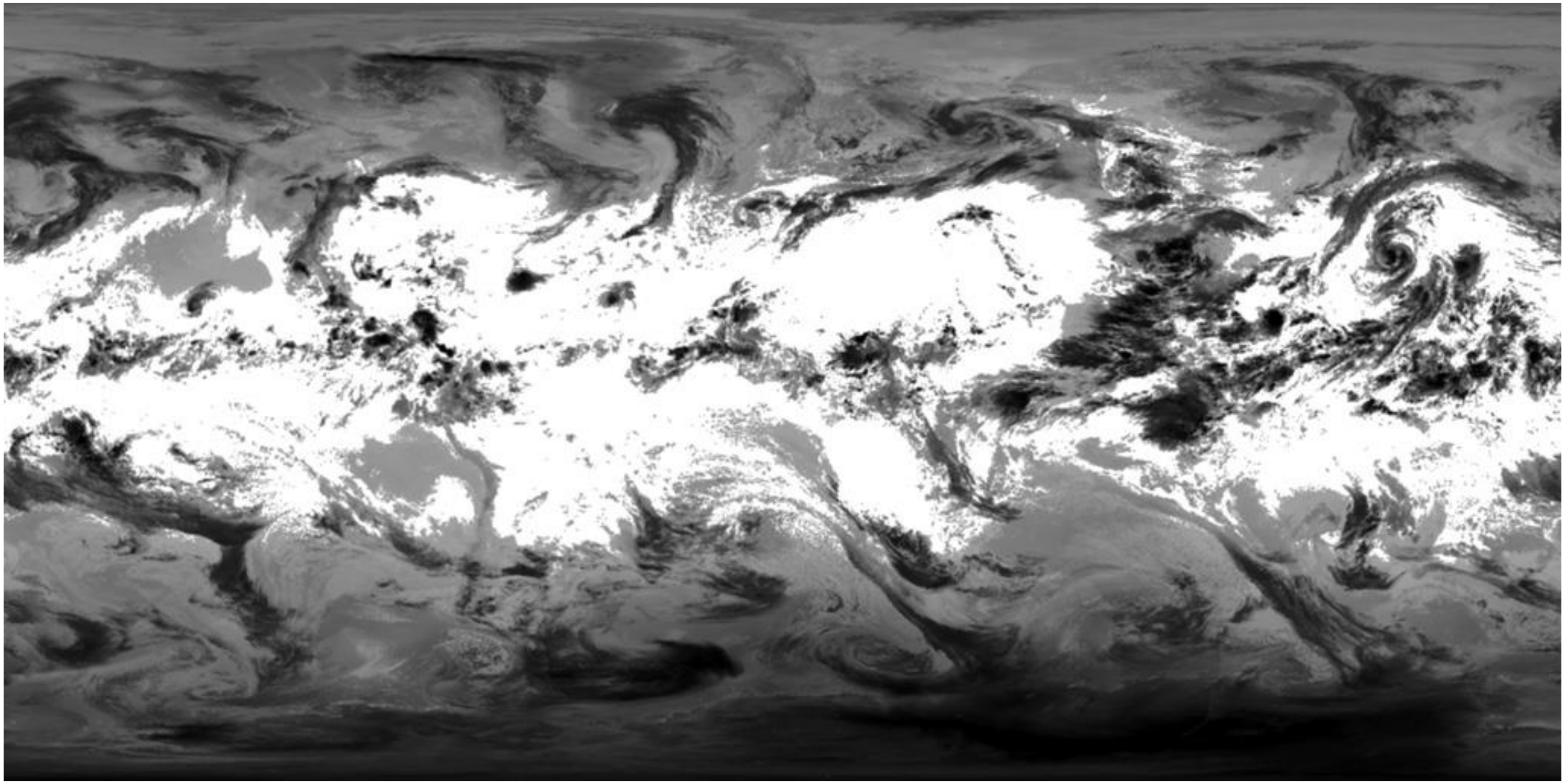
- **spherical** texture map = **cylindrical** map projection!
  the latter does not use cylindrical but **spherical textures**
- notice the distortion at the polar regions

Examples for Spherical Maps



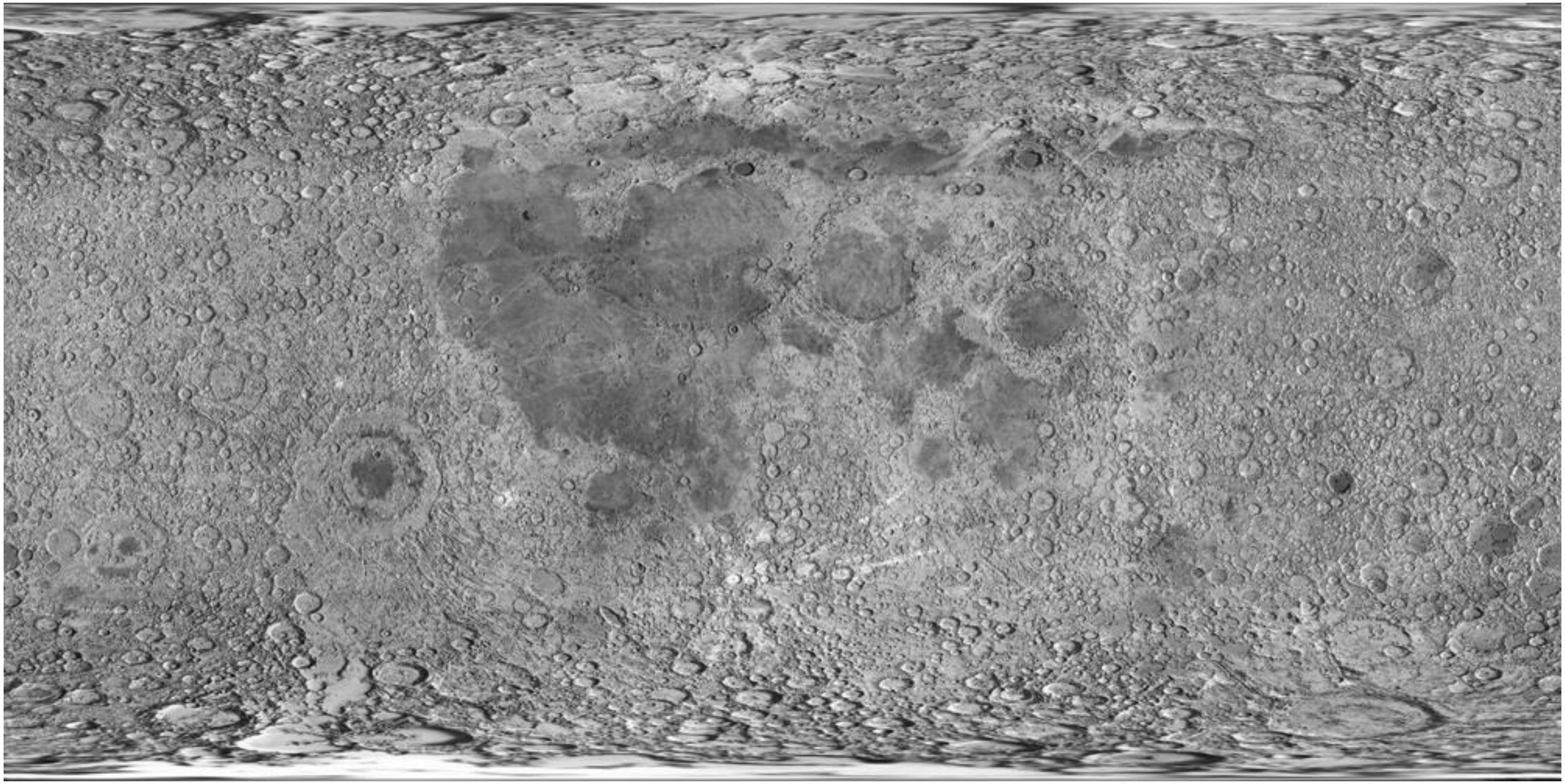
- **spherical** texture map = **cylindrical** map projection! the latter does not use cylindrical but **spherical textures**
- notice the distortion at the polar regions

Examples for Spherical Maps



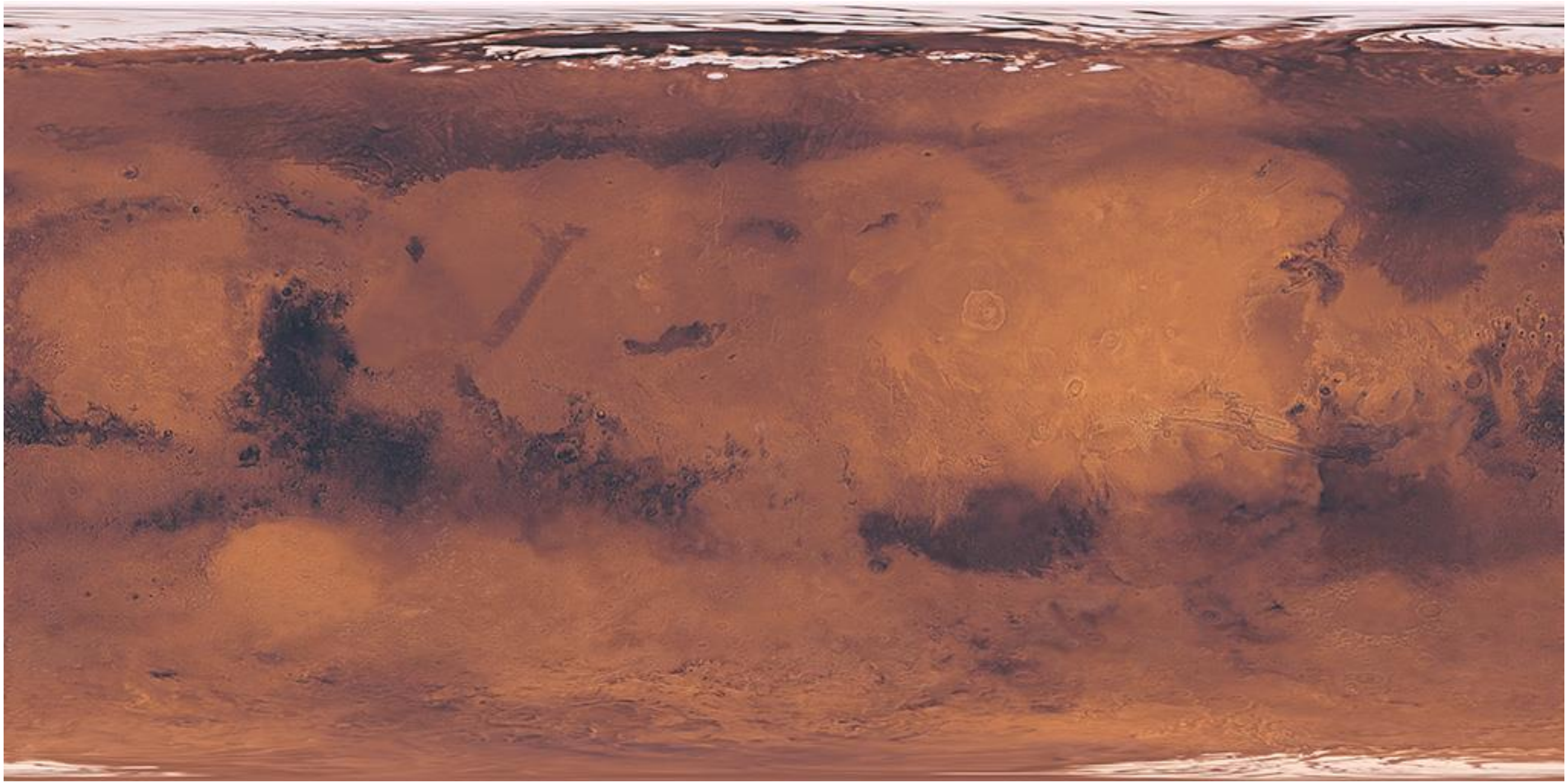
- **spherical** texture map = **cylindrical** map projection!
  the latter does not use cylindrical but **spherical textures**
- notice the distortion at the polar regions

Examples for Spherical Maps



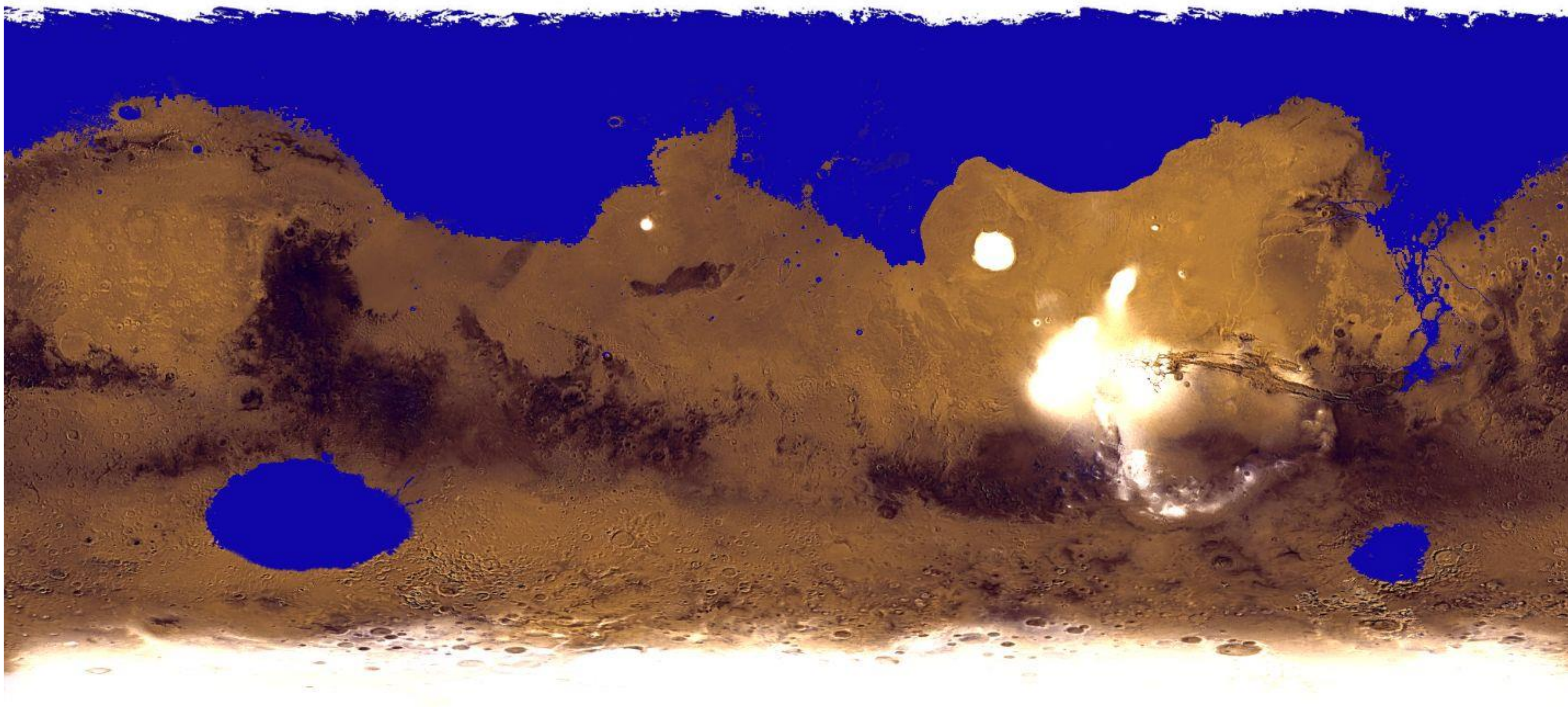
- **spherical** texture map = **cylindrical** map projection!
  the latter does not use cylindrical but **spherical textures**
- notice the distortion at the polar regions

Examples for Spherical Maps



- **spherical** texture map = **cylindrical** map projection!
  the latter does not use cylindrical but **spherical textures**
- notice the distortion at the polar regions

Examples for Spherical Maps



- **spherical** texture map = **cylindrical** map projection!
  the latter does not use cylindrical but **spherical textures**
- notice the distortion at the polar regions

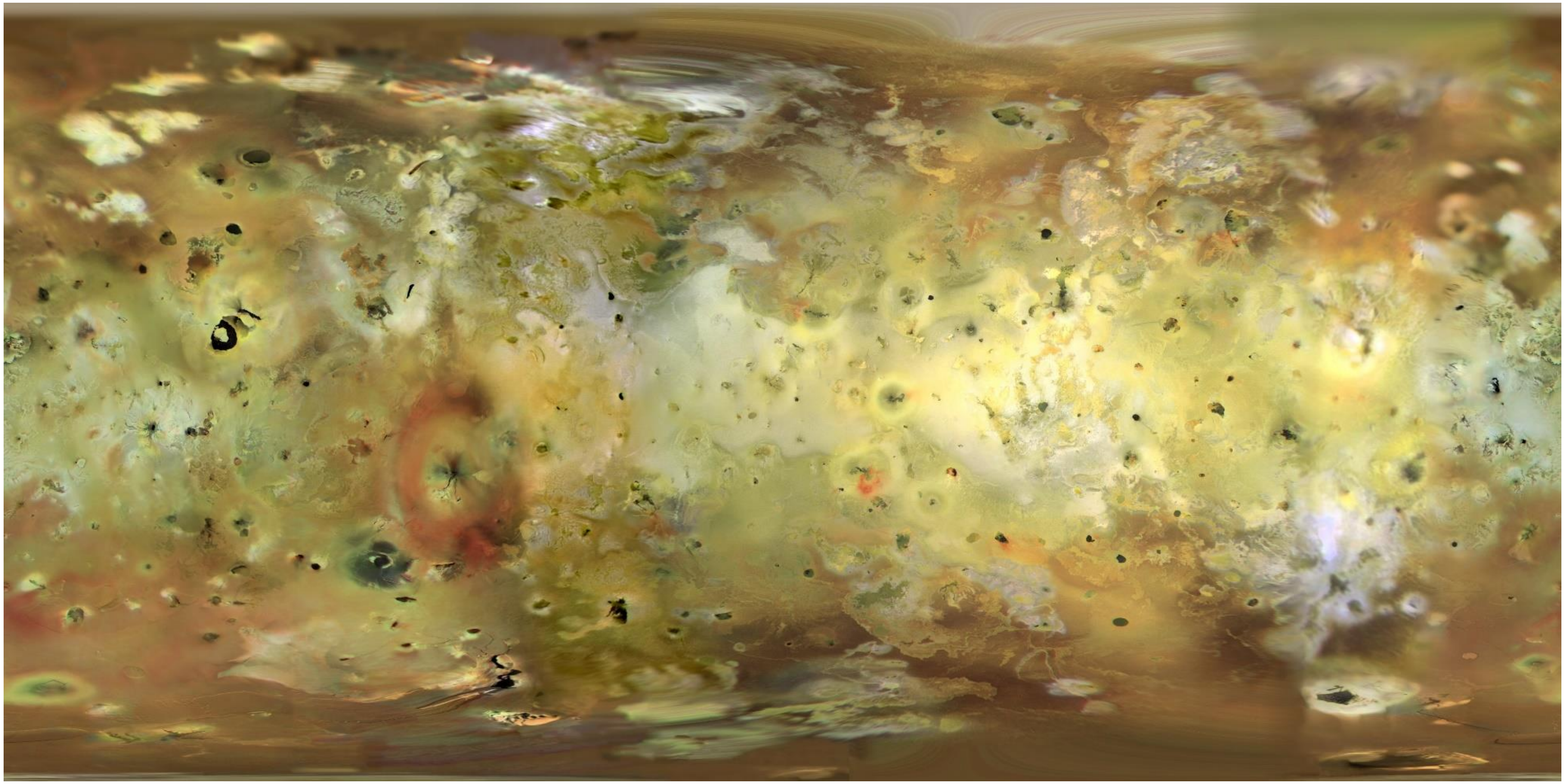
Examples for Spherical Maps

- **spherical** texture map = **cylindrical** map projection! the latter does not use cylindrical but **spherical textures**
- notice the distortion at the polar regions

Examples for Spherical Maps

- **spherical** texture map = **cylindrical** map projection! the latter does not use cylindrical but **spherical textures**
- notice the distortion at the polar regions

Examples for Spherical Maps



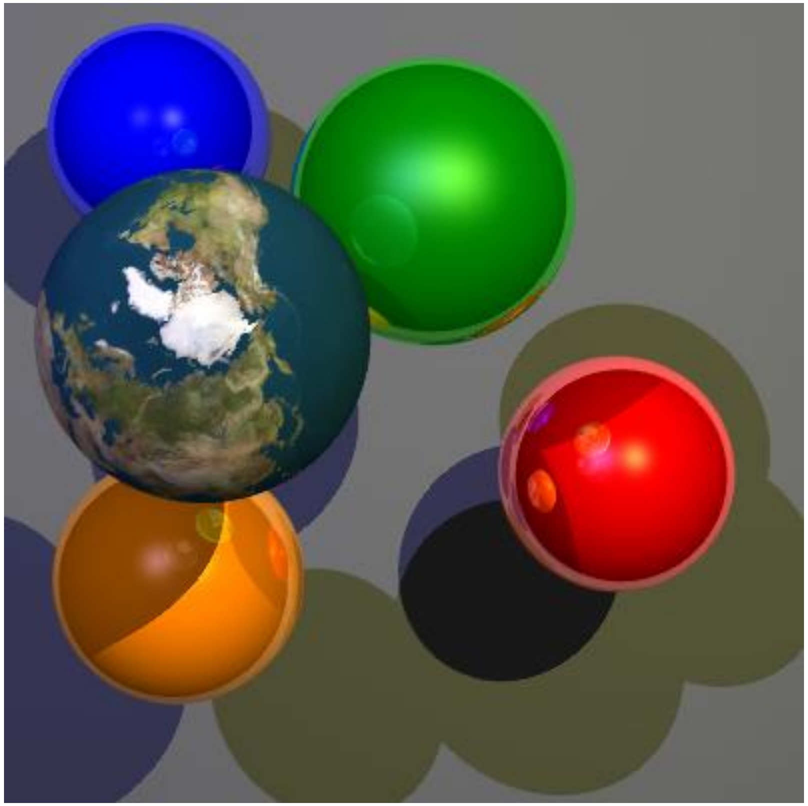
- **spherical** texture map = **cylindrical** map projection!
  the latter does not use cylindrical but **spherical textures**
- notice the distortion at the polar regions

Examples for Spherical Maps

- **spherical** texture map = **cylindrical** map projection!
  the latter does not use cylindrical but **spherical textures**
- notice the distortion at the polar regions

Examples for Spherical Maps

- but this distortion disappears if the spherical map is applied to a sphere

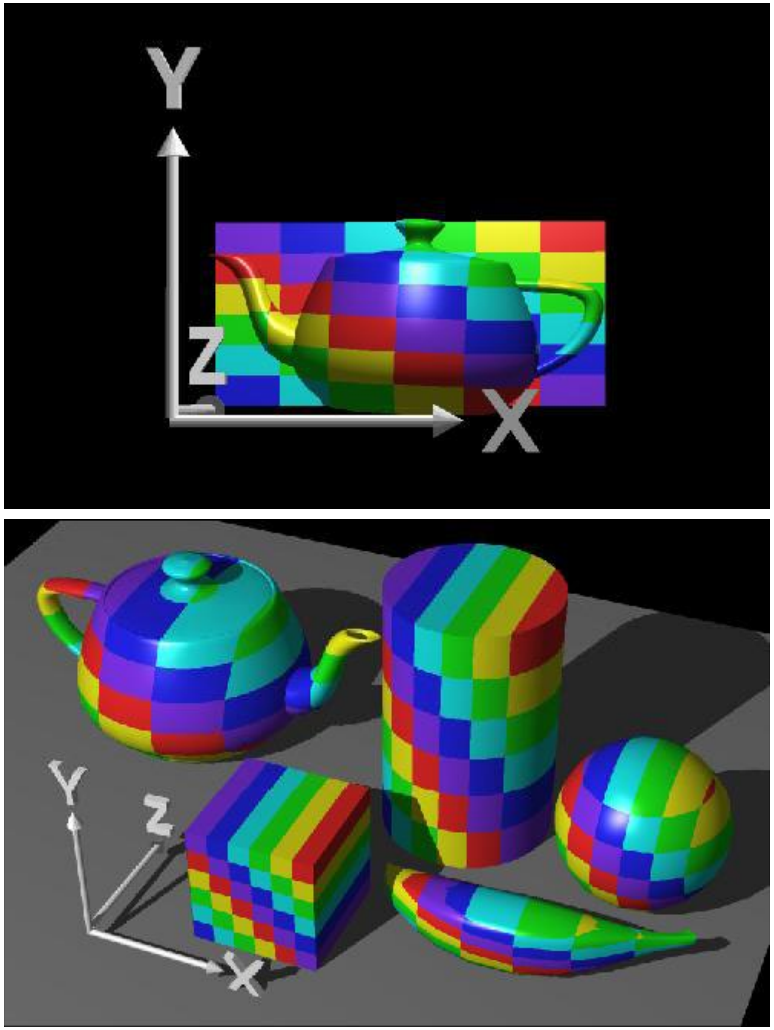


Planar Mapping

- mapping onto planar surface given by position vector $\vec{v}_0$ and two vectors $\vec{s}$ and $\vec{t}$

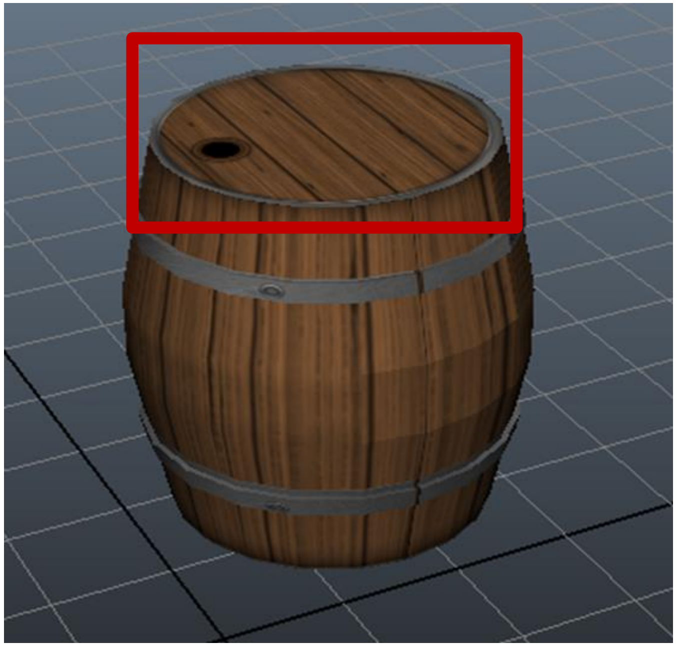
$$S : (x, y, z) \rightarrow (u, v) = \left(\frac{\vec{v} \cdot \vec{s}}{k}, \frac{\vec{v} \cdot \vec{t}}{k} \right)$$

- scaling factor k and $\vec{v} = \vec{P}_i - \vec{v}_0$ (describes point position w.r.t. the origin of the plane)



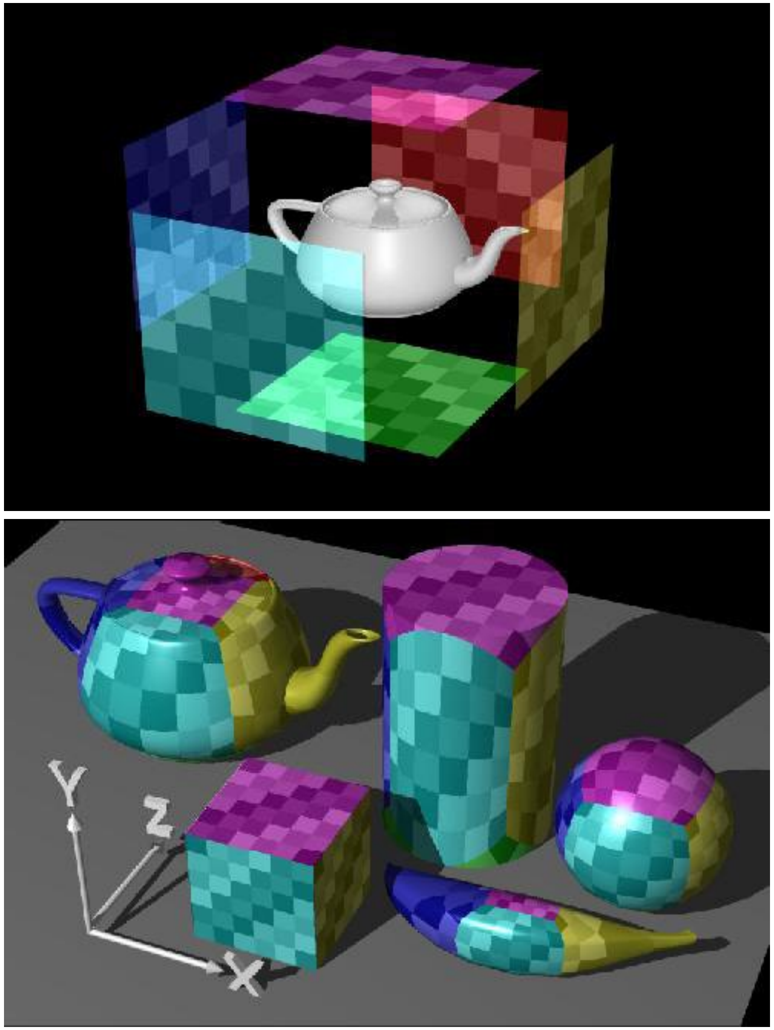
from R. Wolfe: *Teaching Texture Mapping*

Example of Planar Mapping



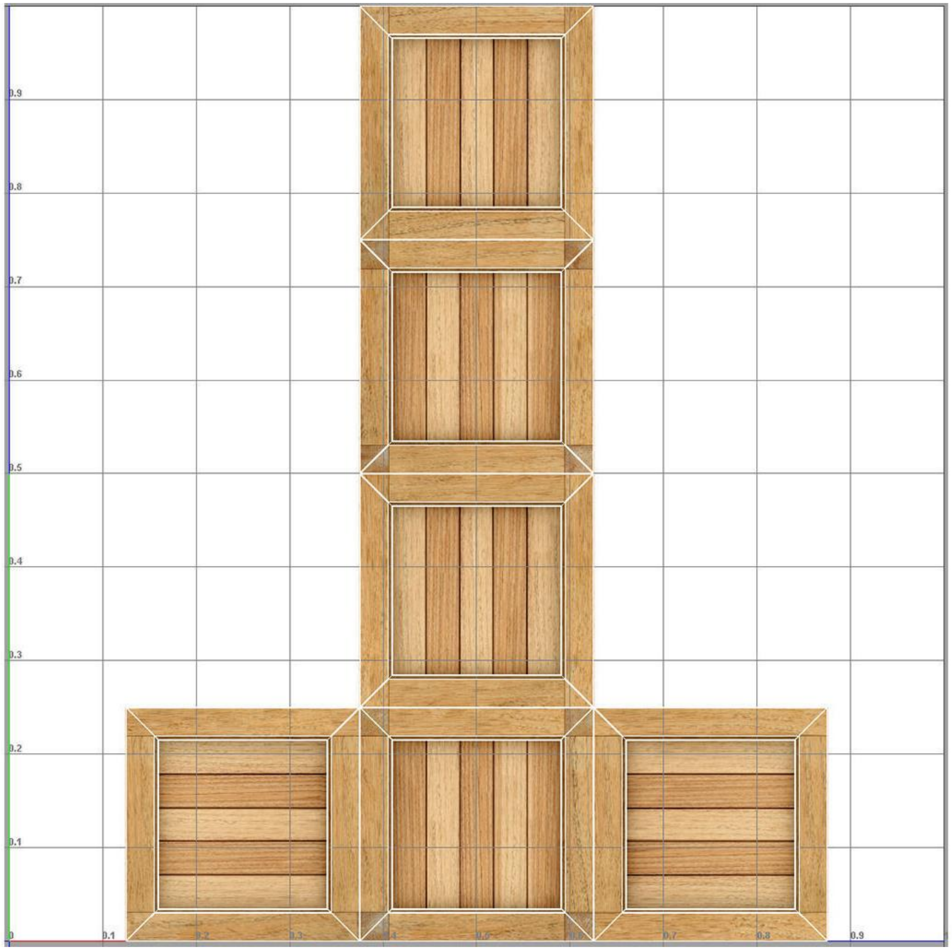
Box Mapping

- enclosing box is usually axis-parallel bounding box of object
- six rectangles onto which the texture is mapped
- similar to planar mapping



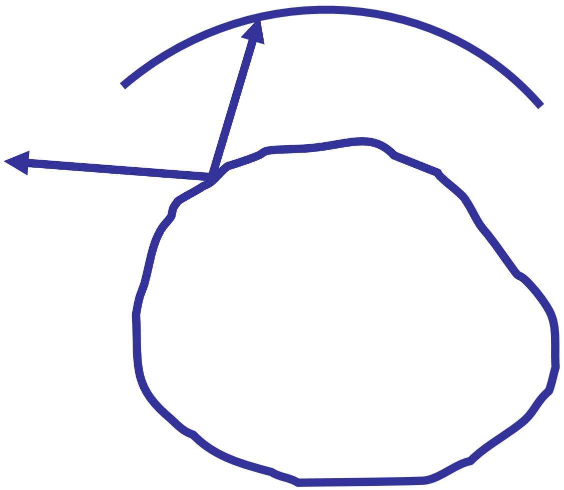
from R. Wolfe: *Teaching Texture Mapping*

Example of Box Mapping



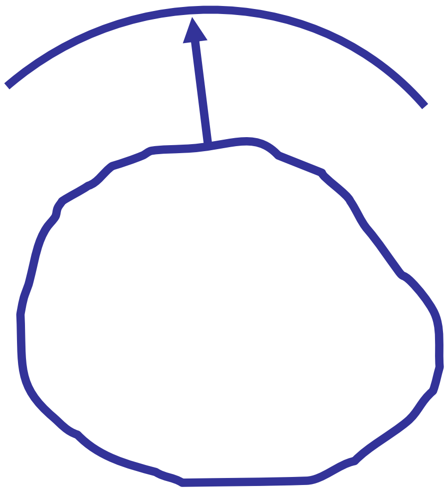
O Mapping: Object to Surface

- necessary for all named techniques
- four methods:
  reflected ray, *object normal*, *object center*, and *normal of intermediate surface*
  - ***reflected ray***:
    trace a ray from viewer to object and reflect it onto the intermediate surface



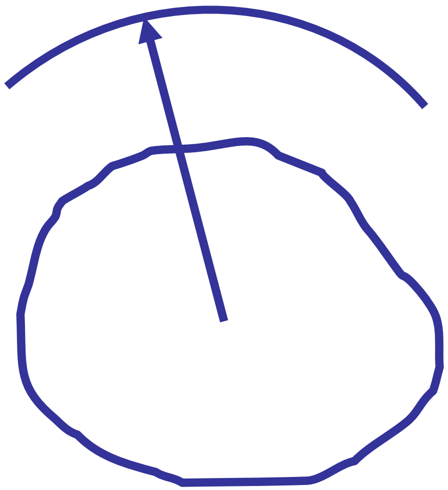
O Mapping: Object to Surface

- necessary for all named techniques
- four methods:
  reflected ray, *object normal*, *object center*, and *normal of intermediate surface*
  - ***object normal***: intersection of normal vector of object with intermediate surface



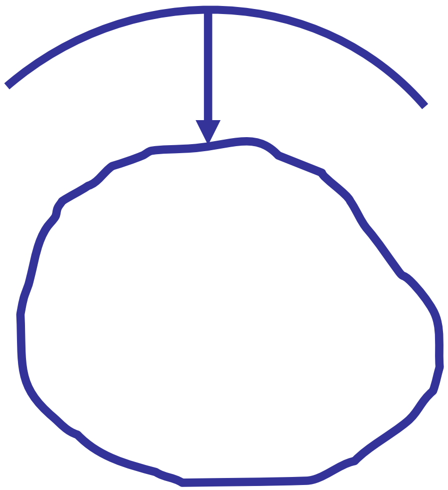
O Mapping: Object to Surface

- necessary for all named techniques
- four methods:
  reflected ray, *object normal*, *object center*, and *normal of intermediate surface*
  - ***object center***: intersection of ray from object center through the object surface with the intermediate surface

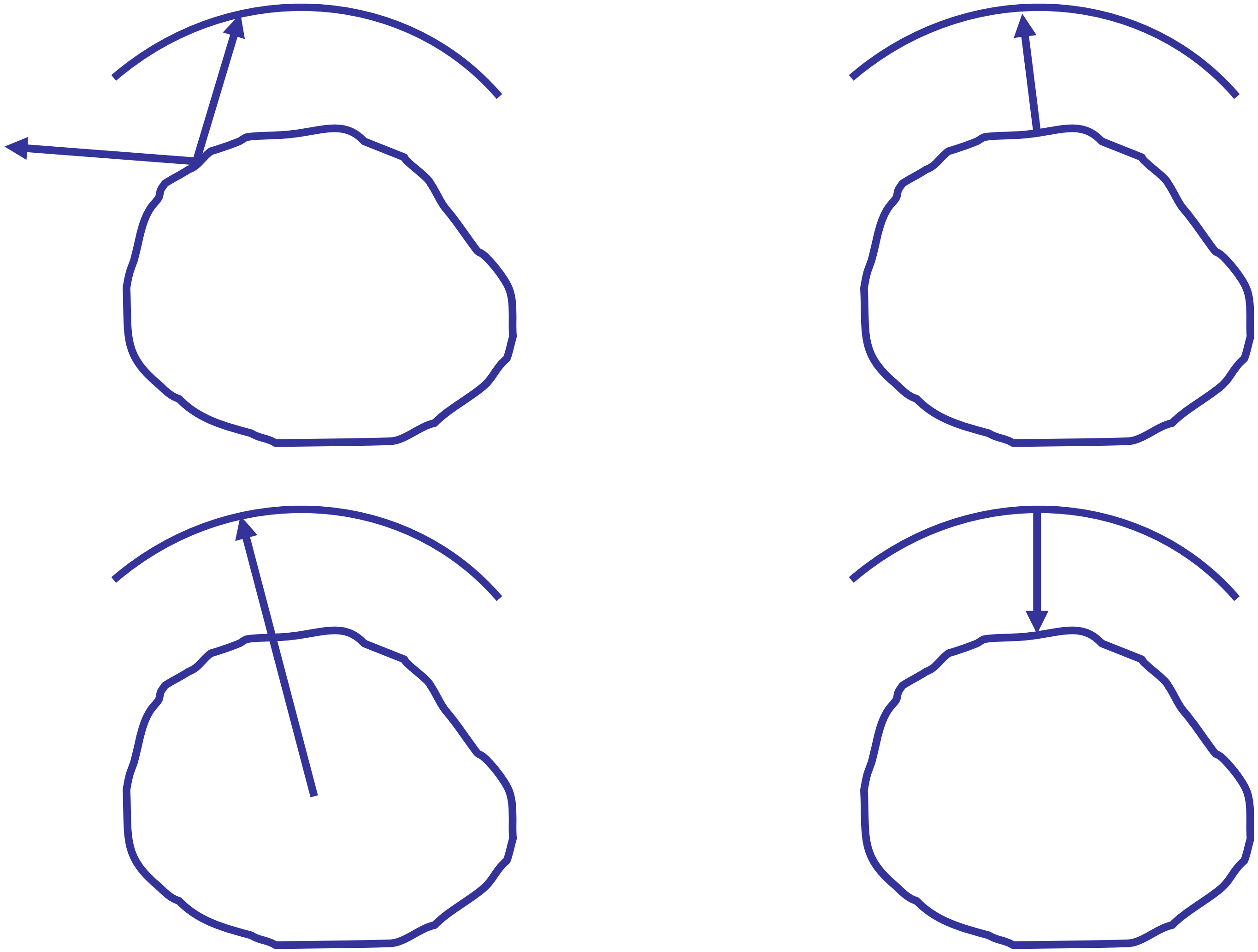


O Mapping: Object to Surface

- necessary for all named techniques
- four methods:
  reflected ray, *object normal*, *object center*, and *normal of intermediate surface*
  - ***normal of intermediate surface***:
    trace this normal vector towards the object and determine intersection with it

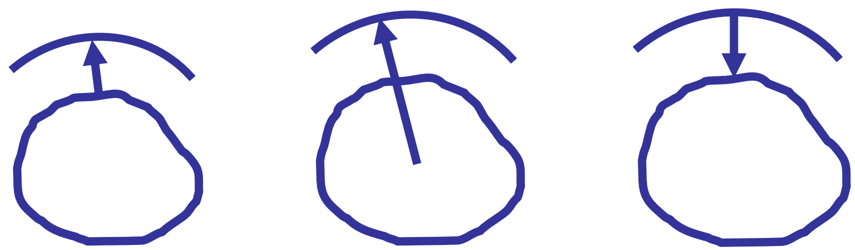


O Mapping: Object to Surface



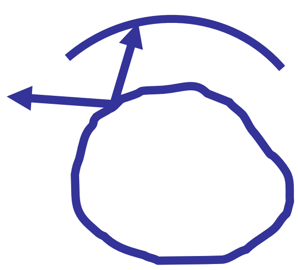
When to do the 2-step mapping?

- typically at model time, works for most schemes:



$\rightarrow$ uv coordinates stored with vertices/normals

- but reflection mapping depends on view direction, so needs to be computed at render time:

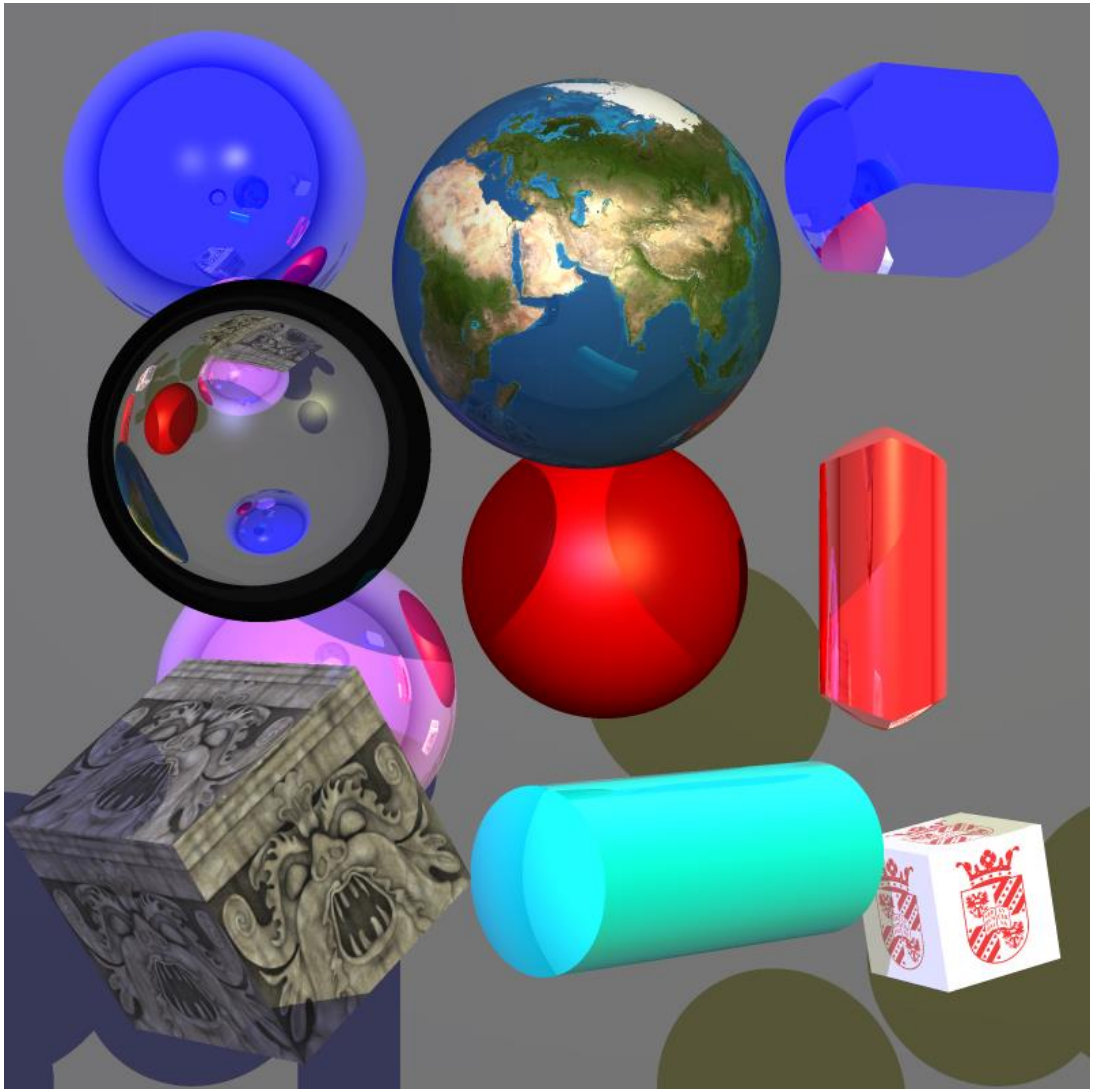


Application of Texture Values

- from an (x, y, z) position we derived an (r, g, b) color value from the texture, potentially with α transparence value
- is typically used to modify illumination
- methods:
  - replace: surface color value is replaced with texture color
  - decal: α blending of texture and original color
  - modulate: multiplication of original color value with texture color

Texture Mapping: Ambient & Diffuse

- done!
- well, almost ...



Texture Mapping

Affecting Other Properties

Texture Mapping: Bump Mapping

- recall: the brick wall
- texture captures visuals
- problem: illumination captured in the texture, conflicting visuals

- solution part 1: capture texture under diffuse light only – how?
- solution part 2: **bump mapping** → change illumination handling with texture

Texture Mapping: Bump Mapping

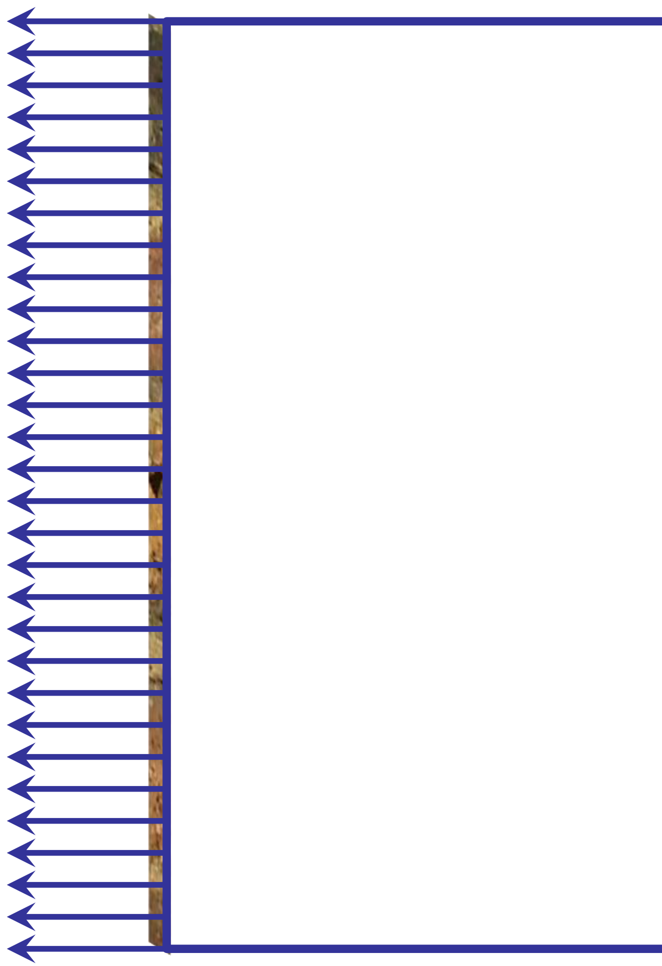
- bump maps: vector offsets to the normal vectors
- illumination computed as usual
- bump maps should match visual texture

Texture Mapping: Bump Mapping

- bump maps: vector offsets to the normal vectors
- illumination computed as usual
- bump maps should match visual texture

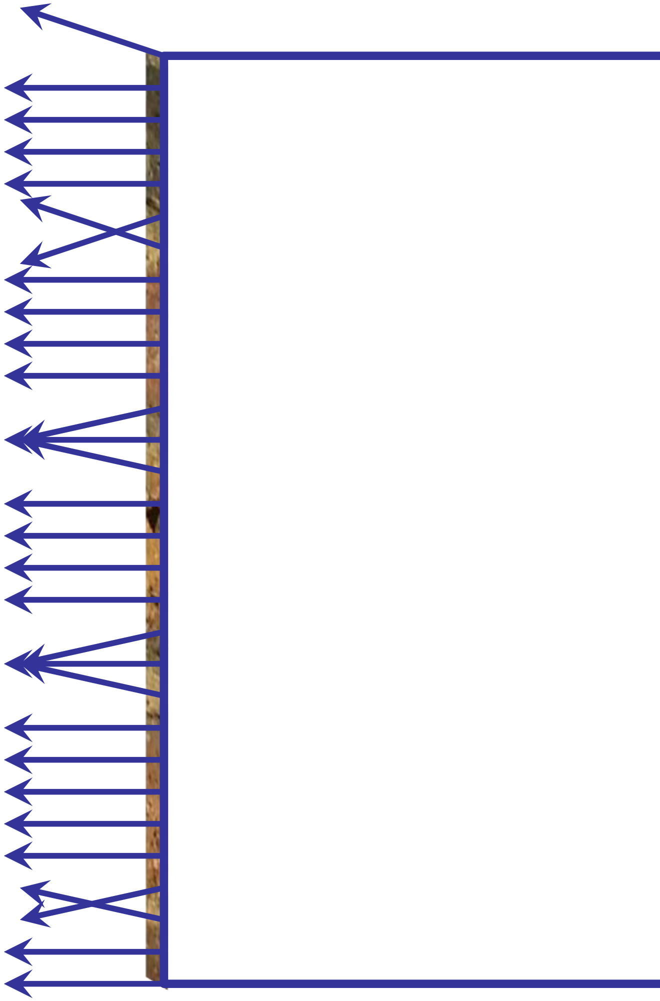
Texture Mapping: Bump Mapping

- bump maps: vector offsets to the normal vectors
- illumination computed as usual
- bump maps should match visual texture



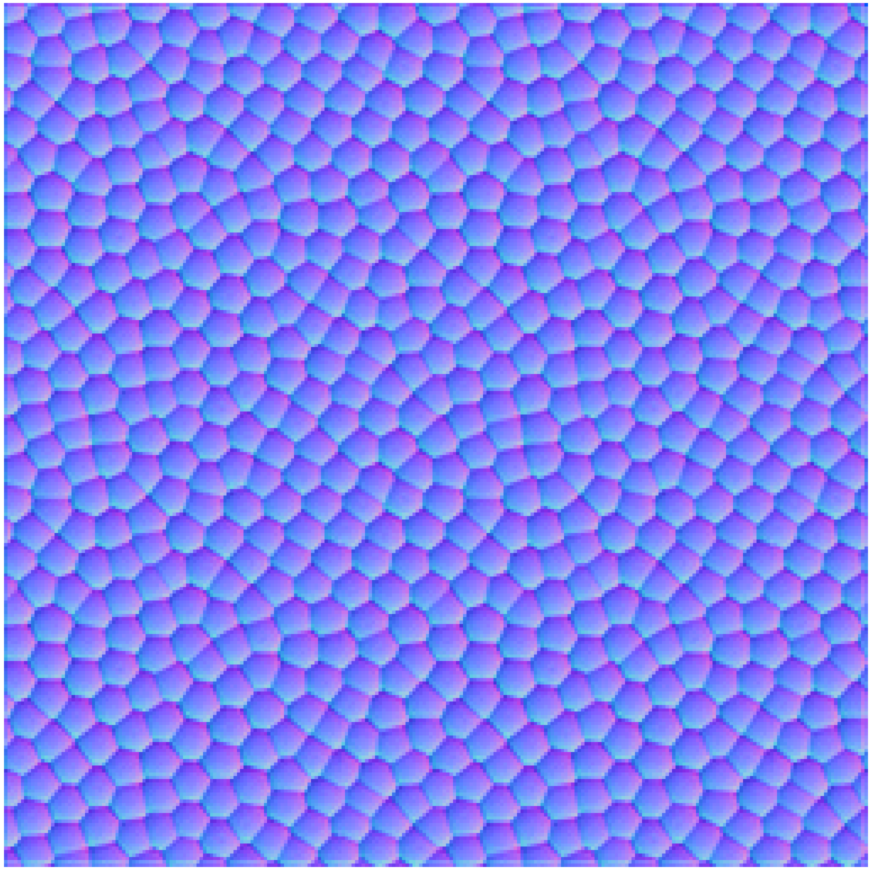
Texture Mapping: Bump Mapping

- bump maps: vector offsets to the normal vectors
- illumination computed as usual
- bump maps should match visual texture



Texture Mapping: Bump Mapping

- bump maps: vector offsets to the normal vectors
- illumination computed as usual
- bump maps should match visual texture

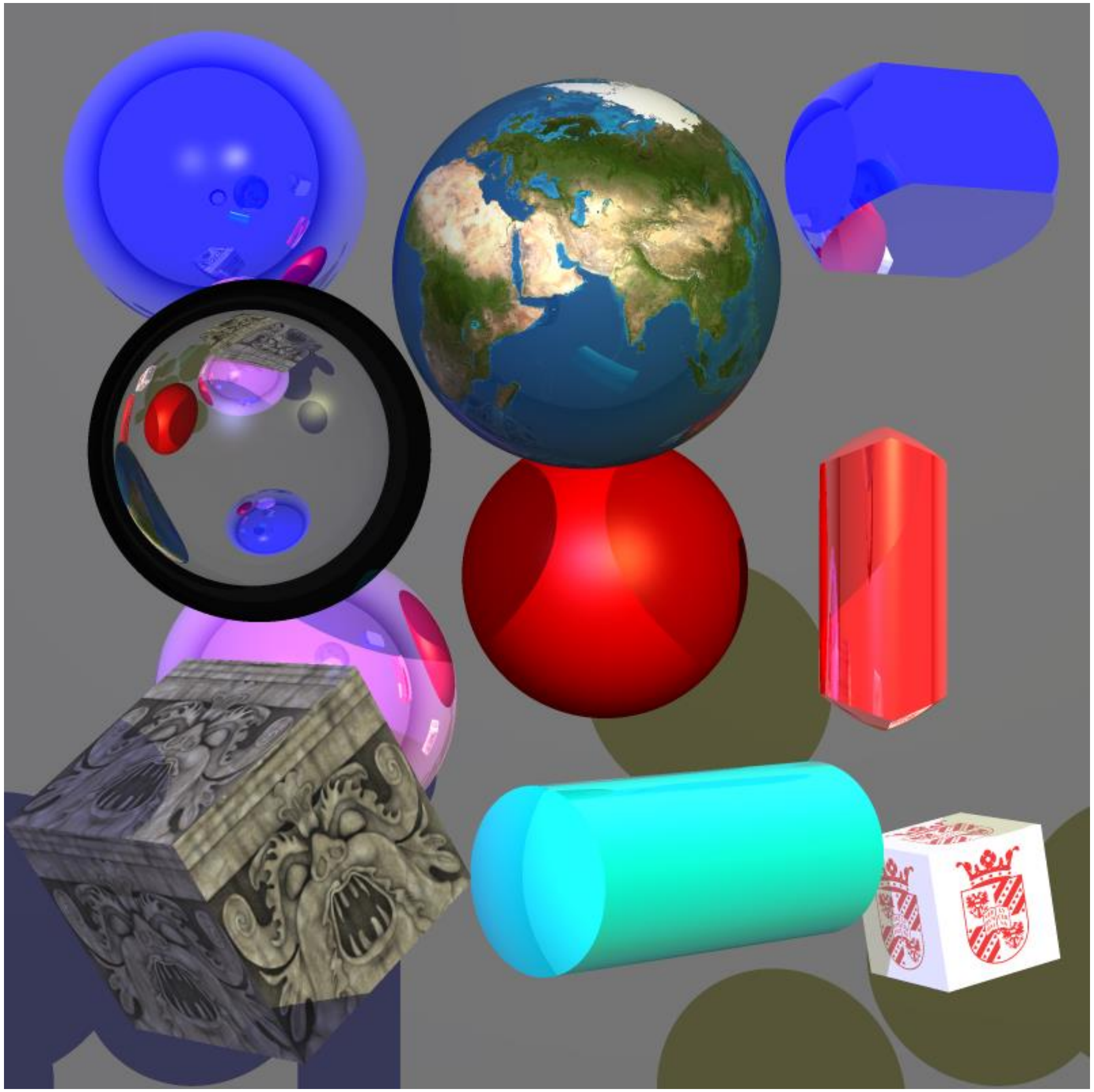


Texture Mapping: Bump Mapping

- bump maps: vector offsets to the normal vectors
- illumination computed as usual
- bump maps should match visual texture

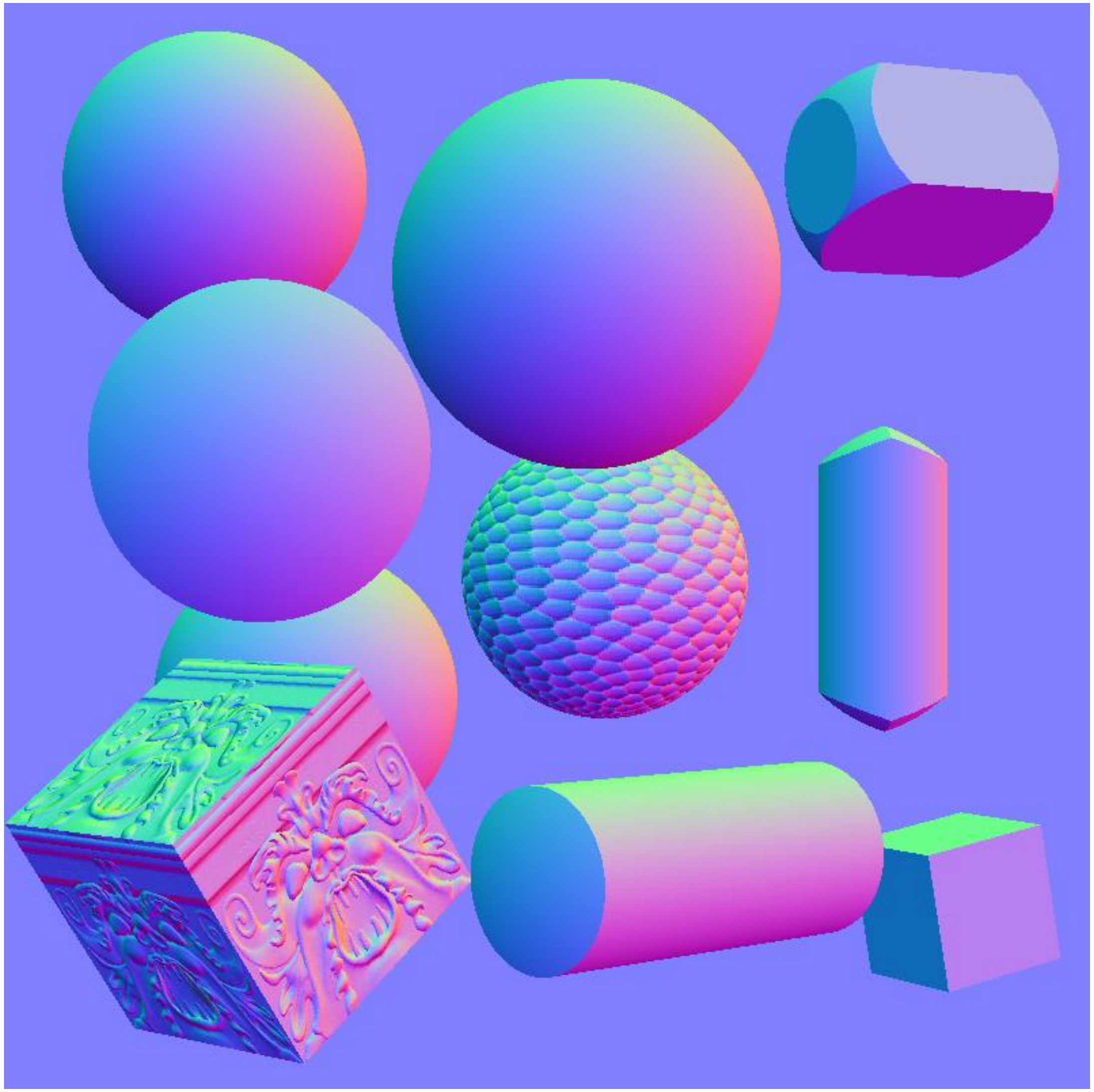

Texture Mapping: Bump Mapping

- bump maps: vector offsets to the normal vectors
- illumination computed as usual
- bump maps should match visual texture



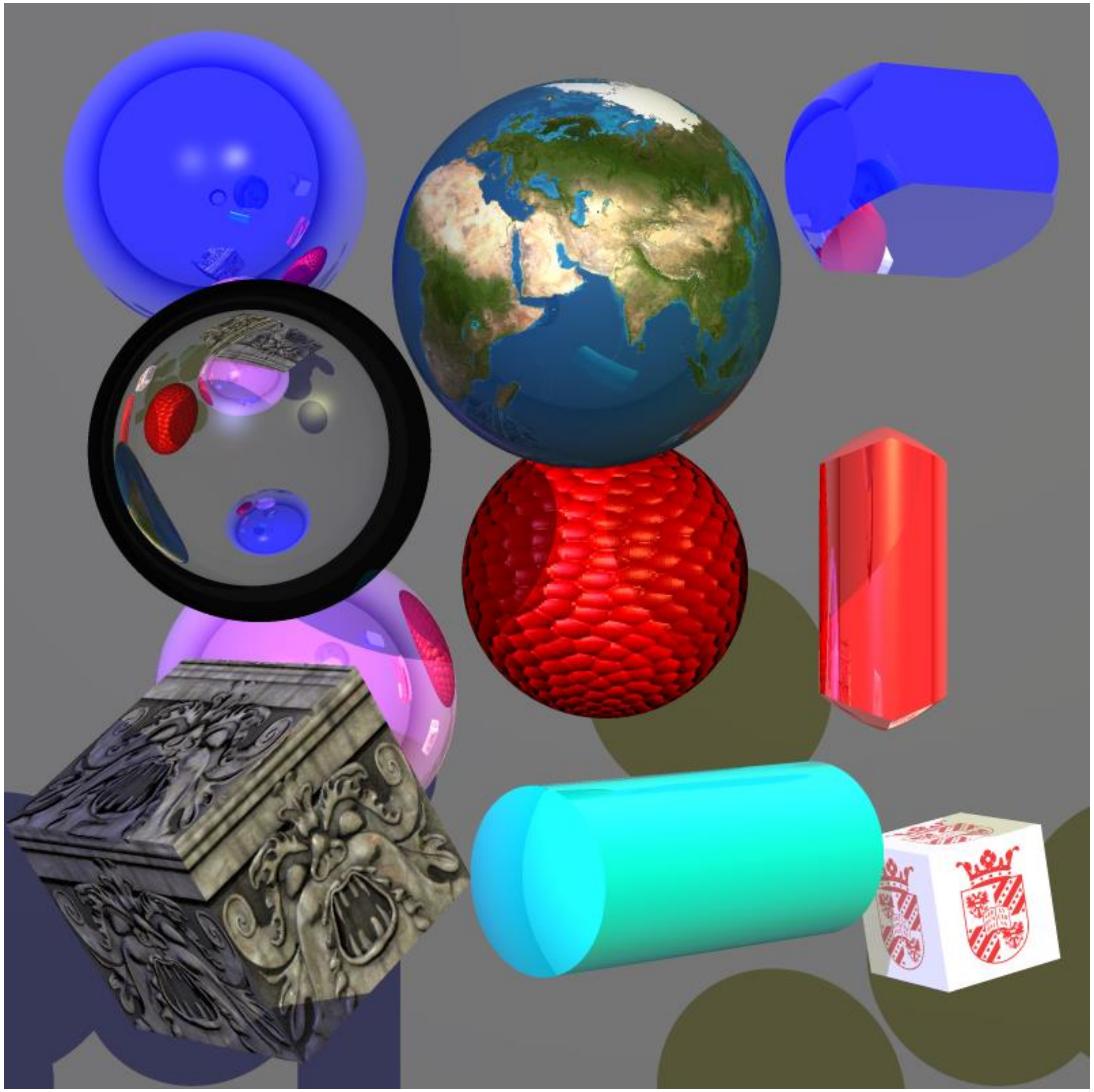
Texture Mapping: Bump Mapping

- bump maps: vector offsets to the normal vectors
- illumination computed as usual
- bump maps should match visual texture



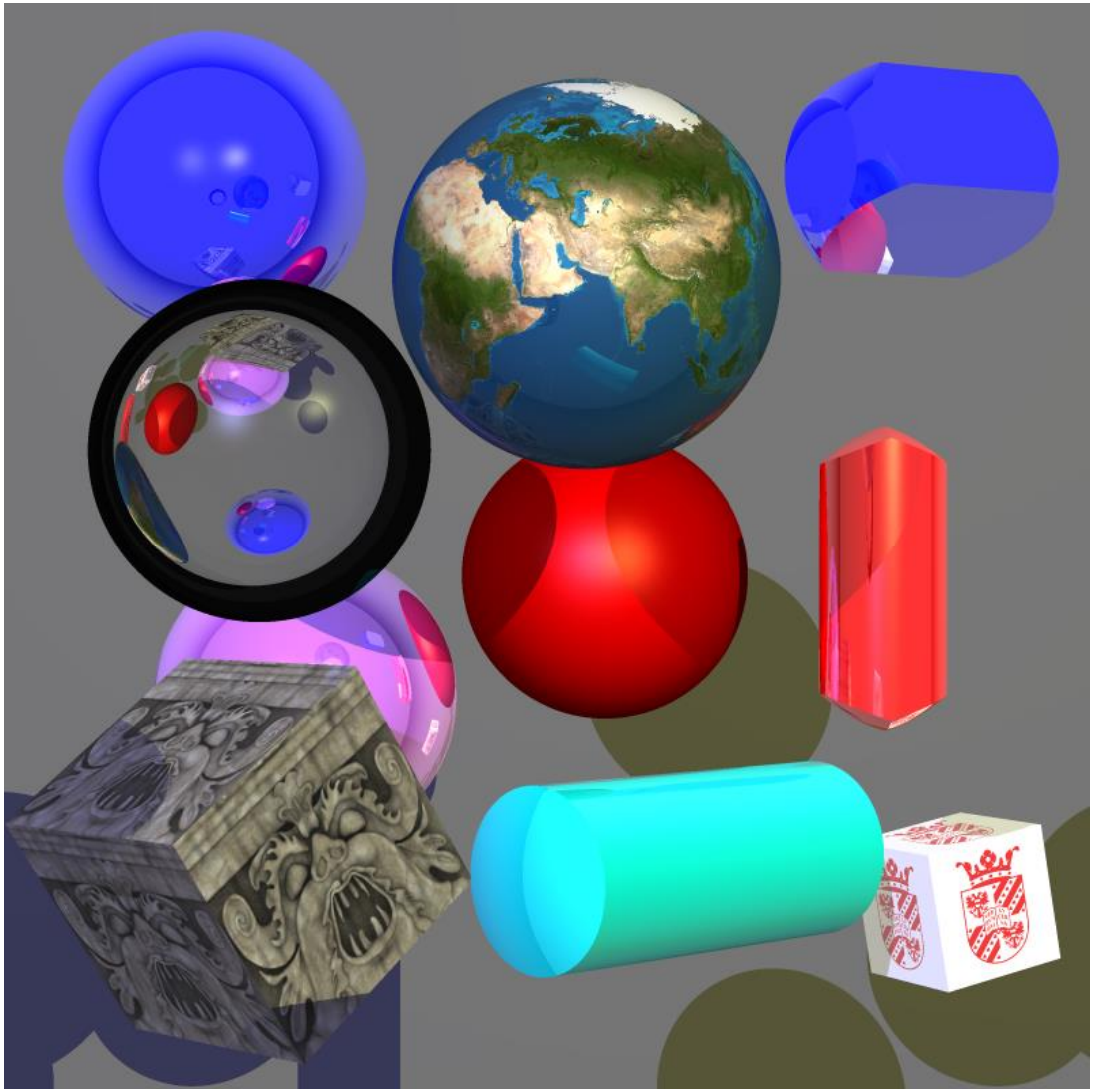
Texture Mapping: Bump Mapping

- bump maps: vector offsets to the normal vectors
- illumination computed as usual
- bump maps should match visual texture



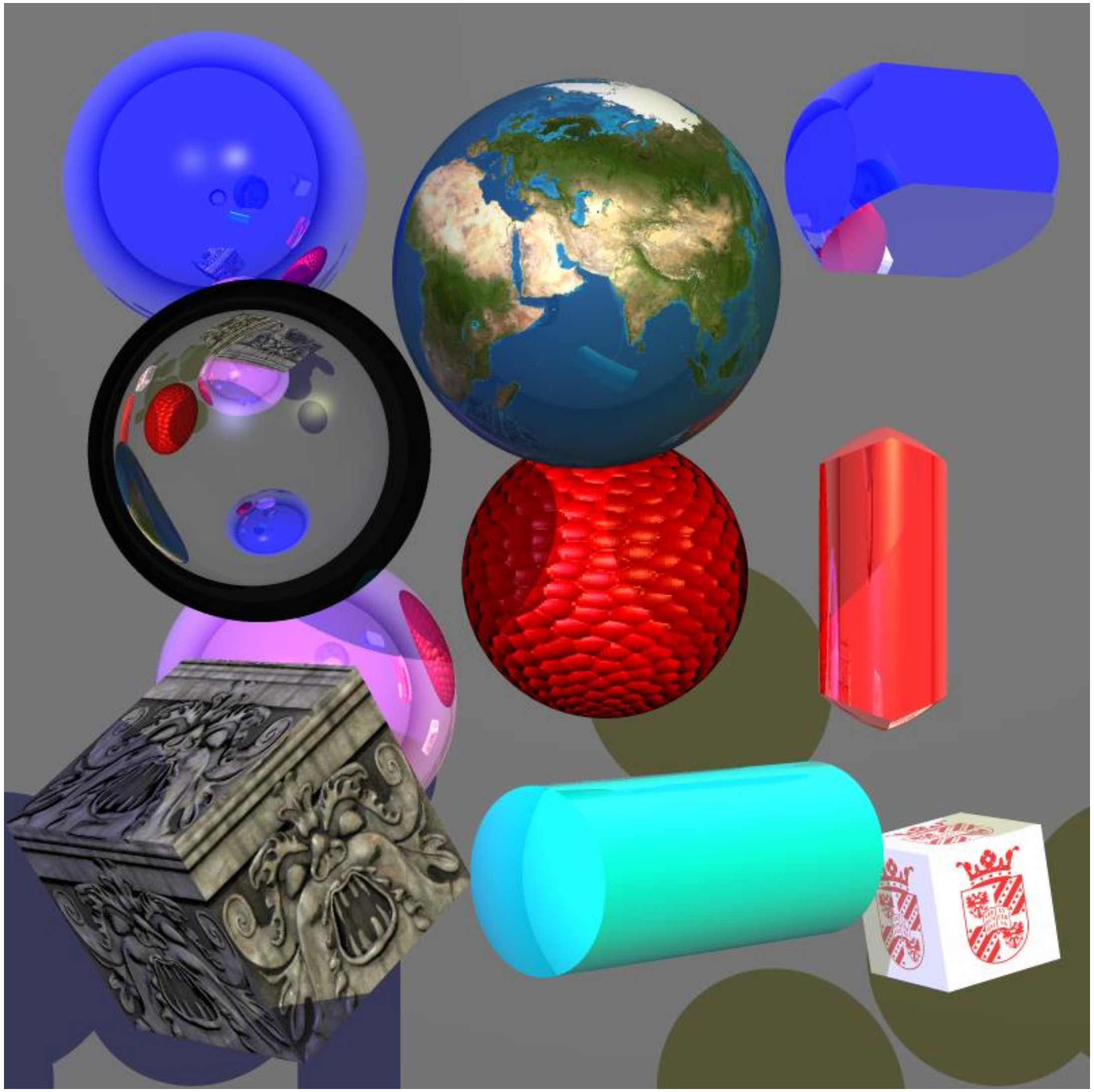
Texture Mapping: Bump Mapping

- bump maps: vector offsets to the normal vectors
- illumination computed as usual
- bump maps should match visual texture



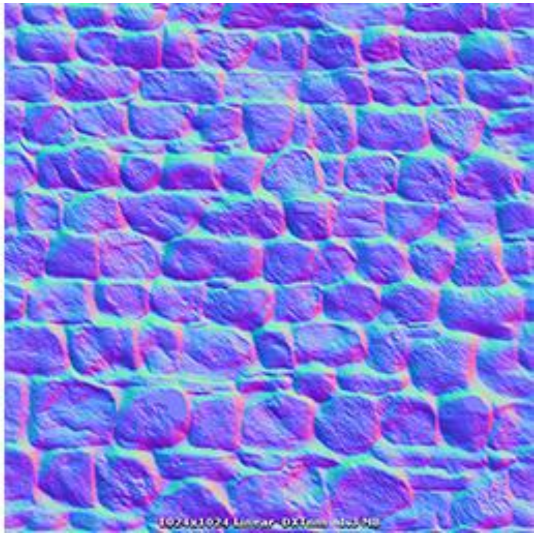
Texture Mapping: Bump Mapping

- bump maps: vector offsets to the normal vectors
- illumination computed as usual
- bump maps should match visual texture



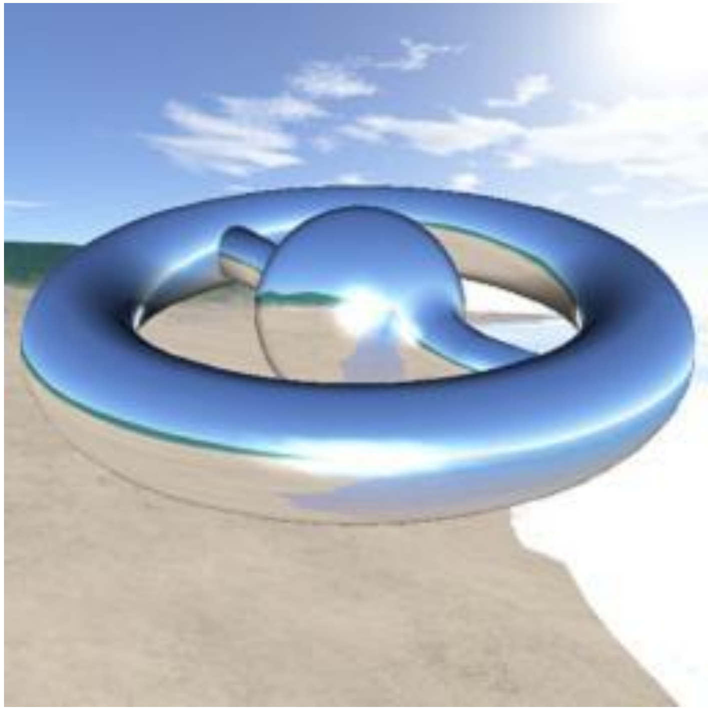
How to create bump maps?

1. Model/carve detailed object and “render” normals into a bump map.
2. Some image processing tools (e.g., Photoshop CC) allow one to create normal maps from regular images.

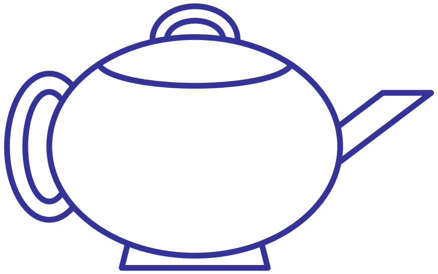


Affecting other Properties

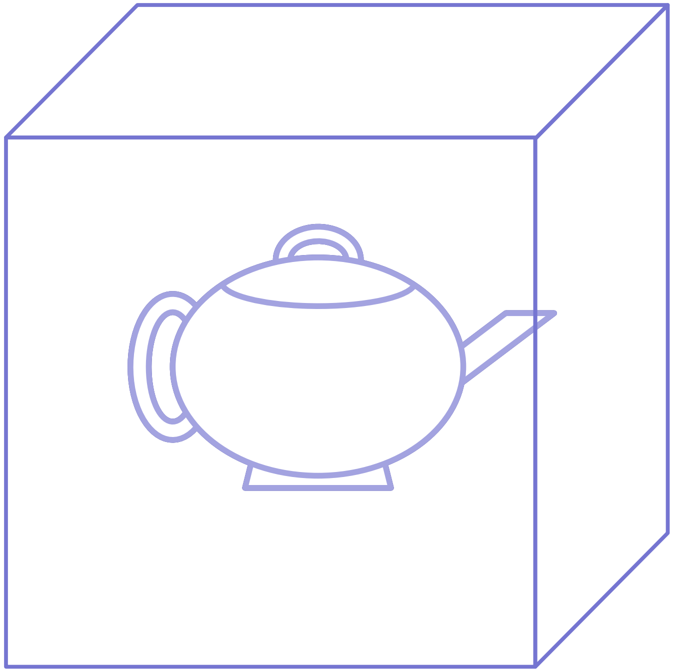
- actual surface positions (as opposed to normal vectors only): **displacement mapping**
- transparency
- simulation of reflection: **environment mapping**
- many more things with GPU processing



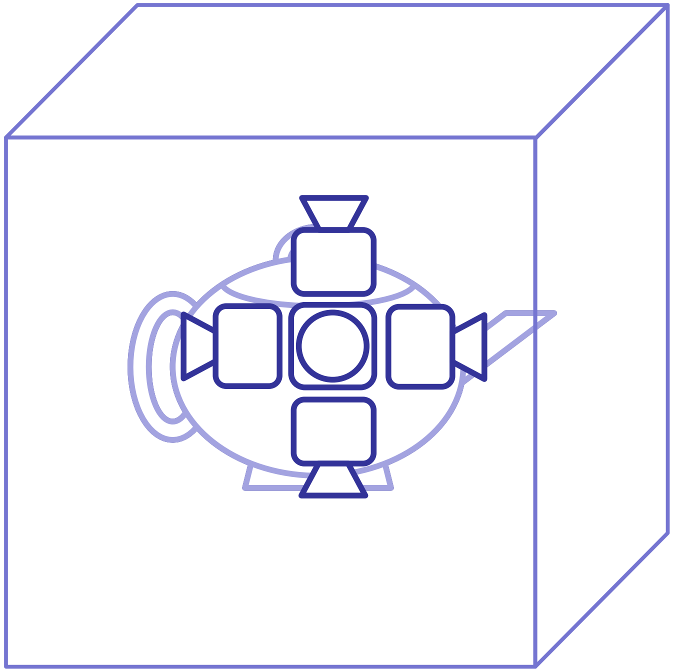
Environment Mapping



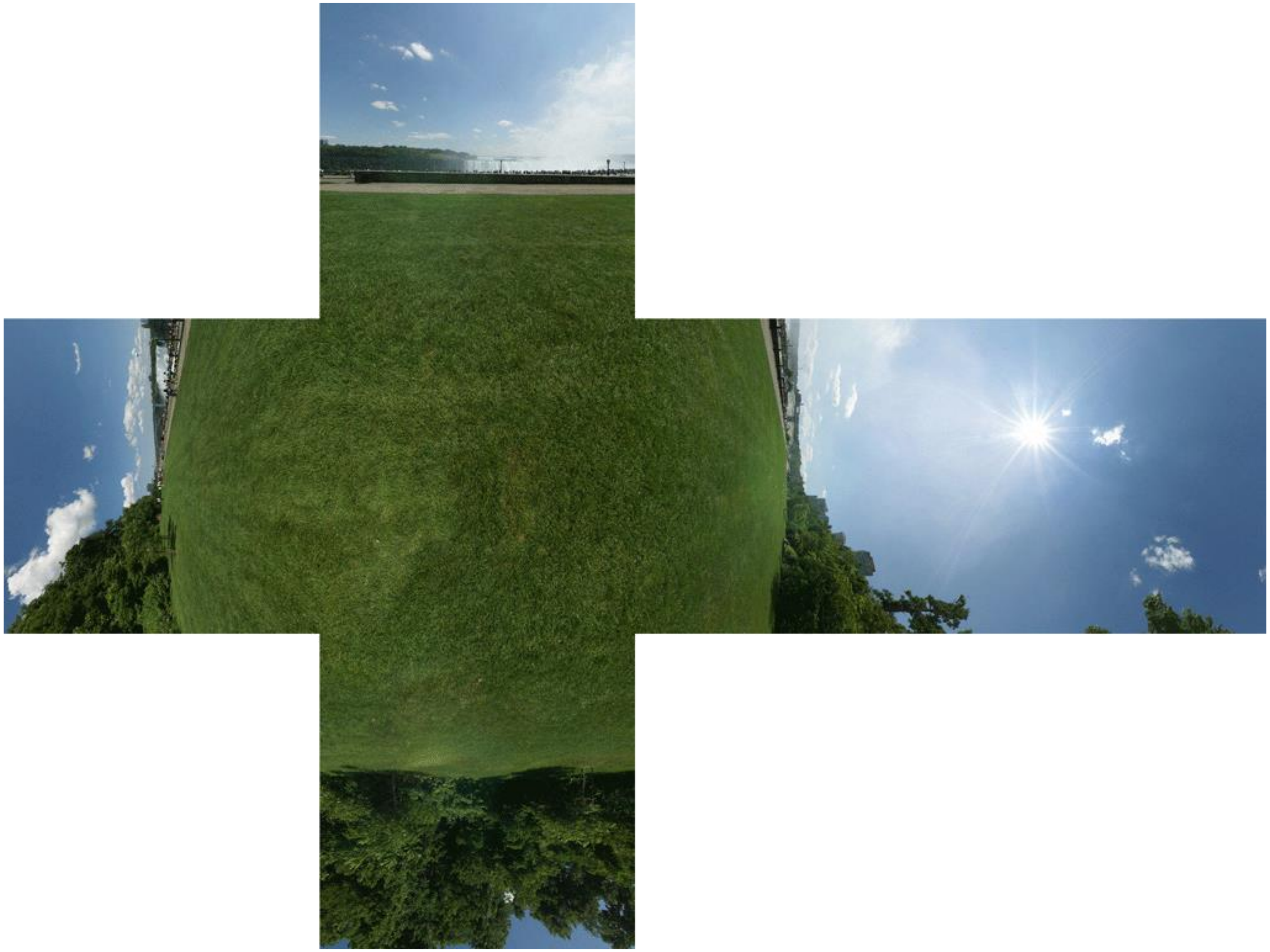
Environment Mapping



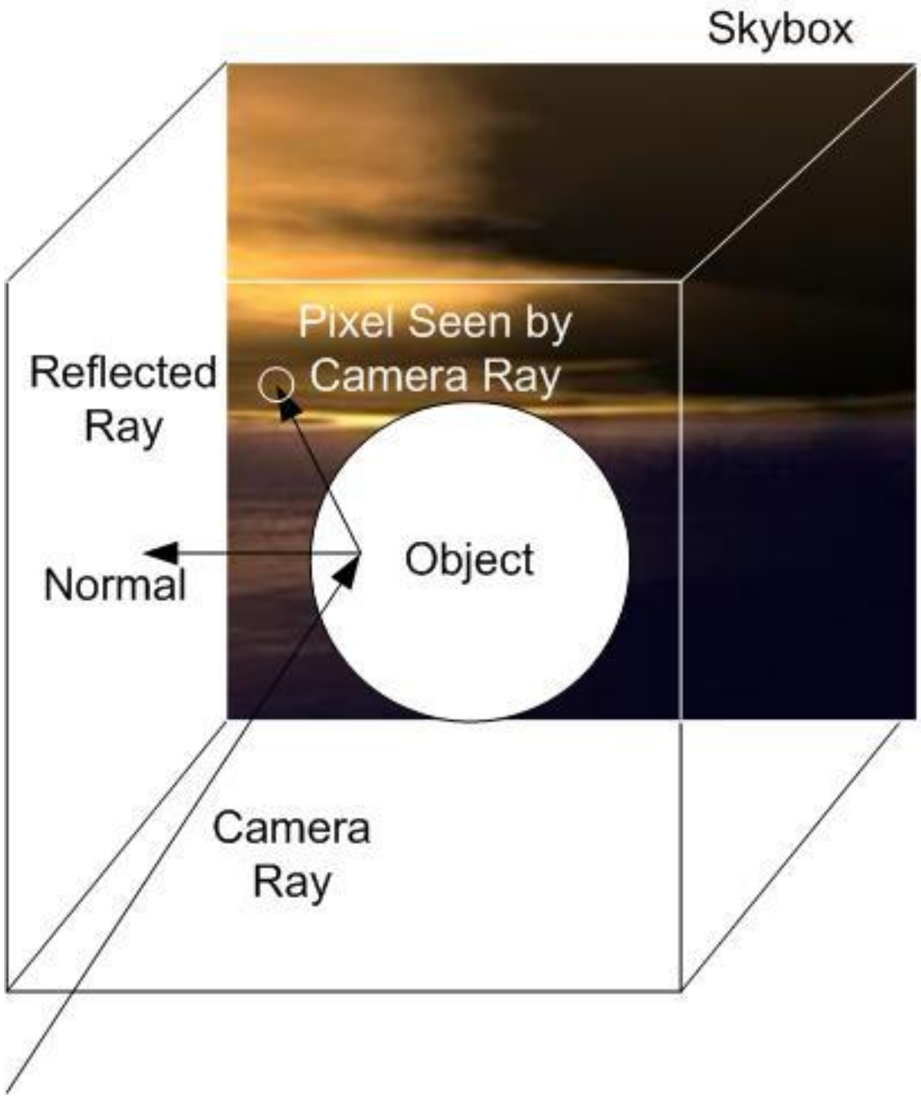
Environment Mapping

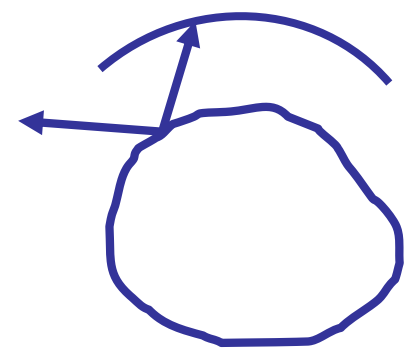


Environment Mapping

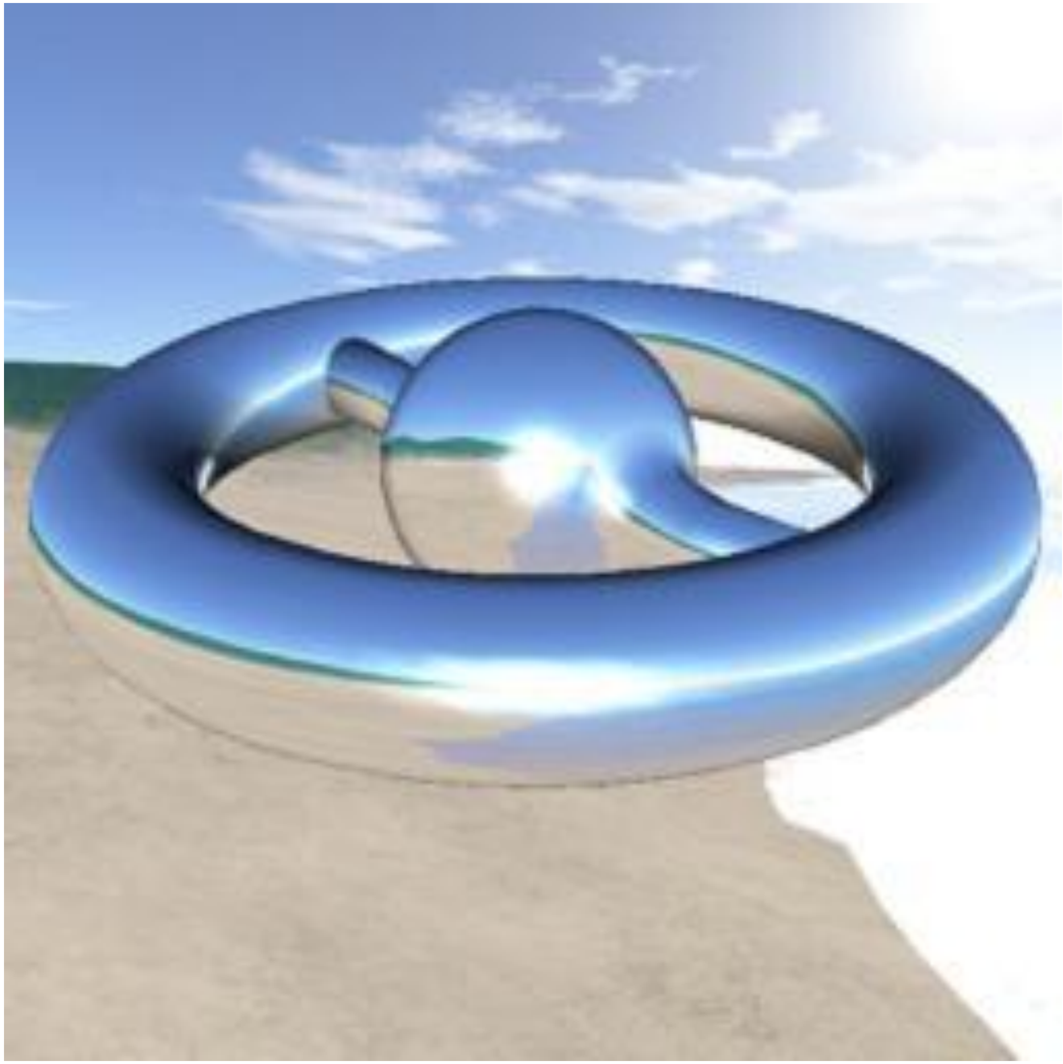


Environment Mapping





Environment Mapping

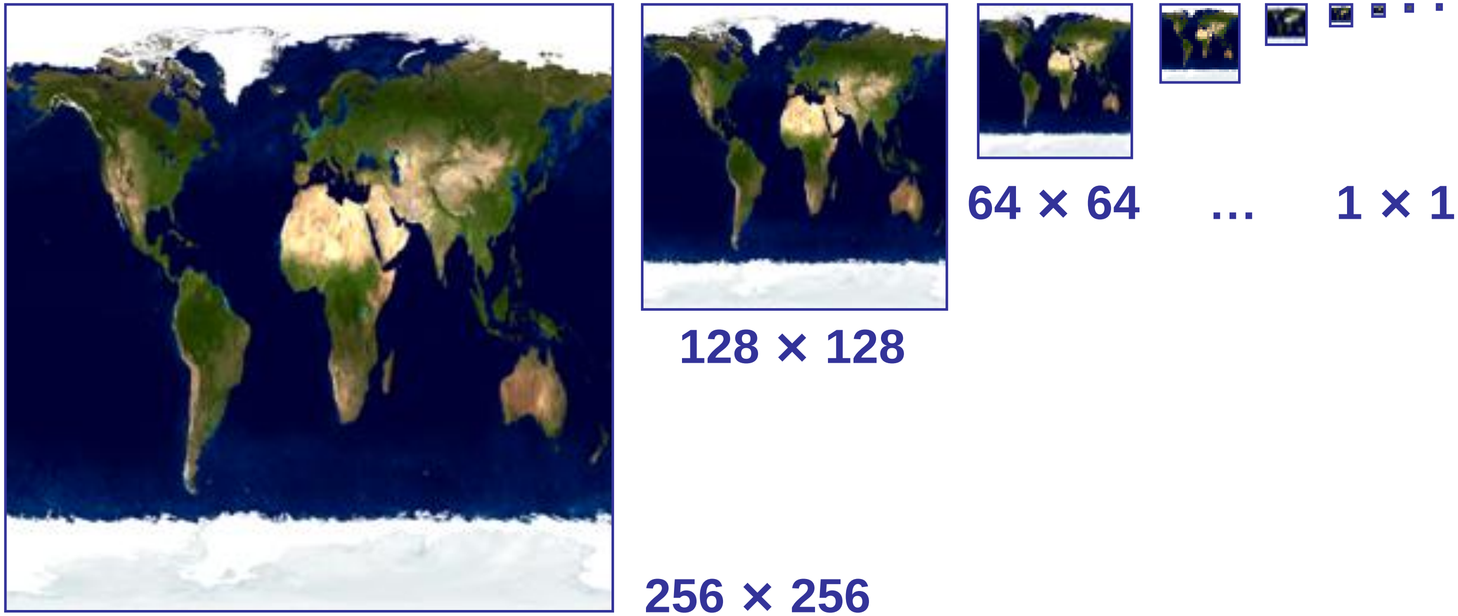


Texture Mapping

Quality Considerations

Texture Mapping: Mip Mapping

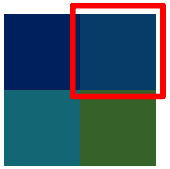
- optimal texture mapping (speed & quality): texel size ≈ pixel size
- idea: use stack of textures and select the most appropriate one w.r.t. situation



Texture Mapping: Mip Mapping

- optimal texture mapping (speed & quality): texel size ≈ pixel size
- interpolation: **GL_NEAREST_MIPMAP_NEAREST**

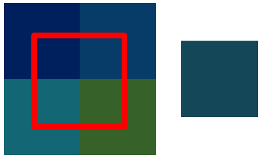
  select nearest mipmap level, select nearest pixel of 2×2 neighbourhood



Texture Mapping: Mip Mapping

- optimal texture mapping (speed & quality): texel size ≈ pixel size
- interpolation: **GL_LINEAR_MIPMAP_NEAREST**

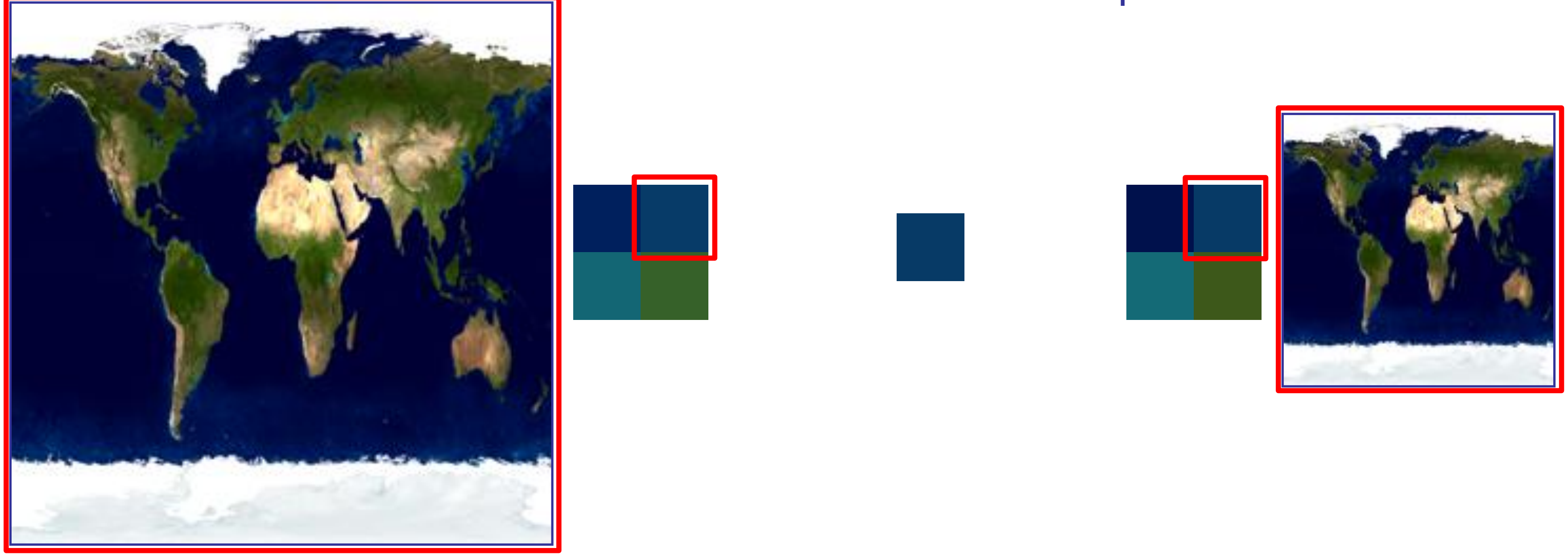
  select nearest mipmap level, linearly interpolate pixel in 2×2 neighbourhood



Texture Mapping: Mip Mapping

- optimal texture mapping (speed & quality): texel size ≈ pixel size
- interpolation: **GL_NEAREST_MIPMAP_LINEAR**

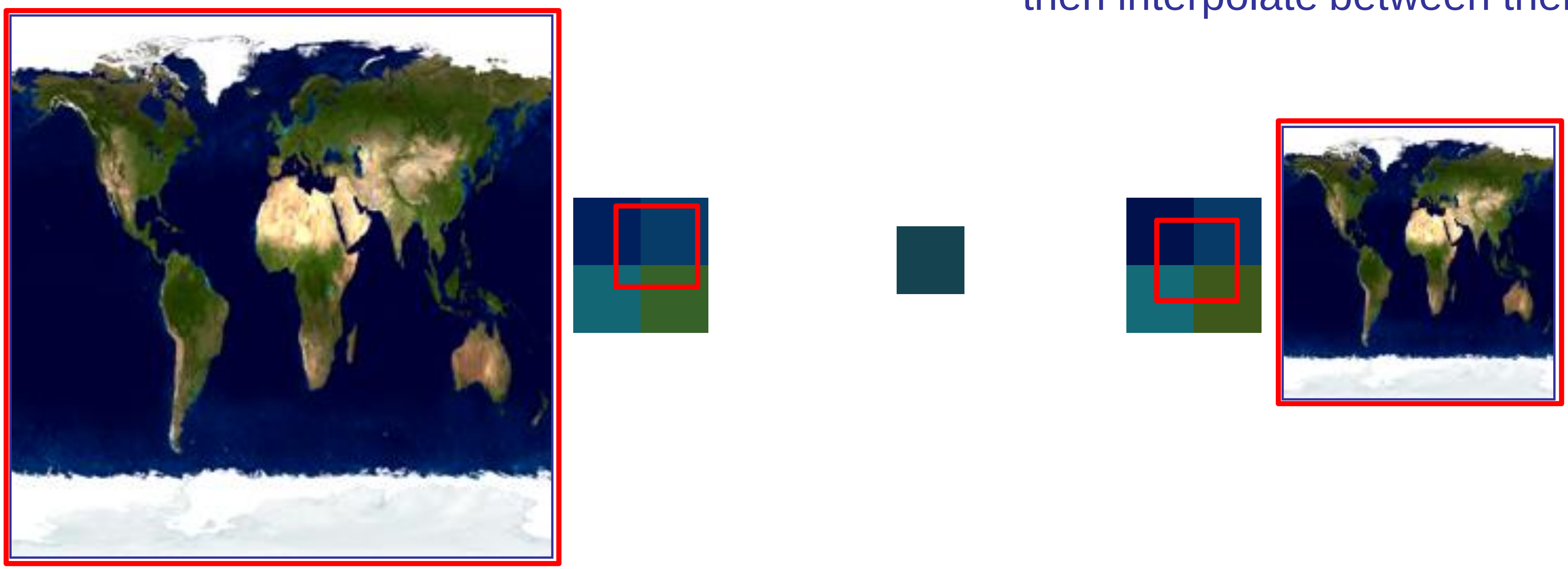
  select 2 adjacent mipmap levels, select nearest pixel in 2×2 neighbourhoods, then interpolate between them



Texture Mapping: Mip Mapping

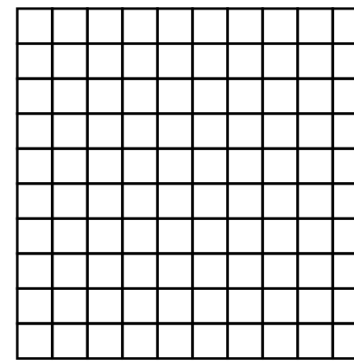
- optimal texture mapping (speed & quality): texel size ≈ pixel size
- interpolation: **GL_LINEAR_MIPMAP_LINEAR**

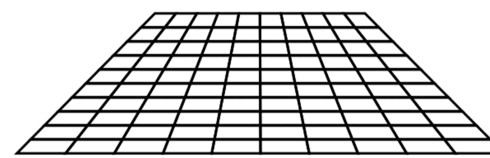
  select 2 adjacent mipmap levels, linearly interpolate pixel in 2×2 neighbourhoods, then interpolate between them



Texture Mapping: Anisotropic Filtering

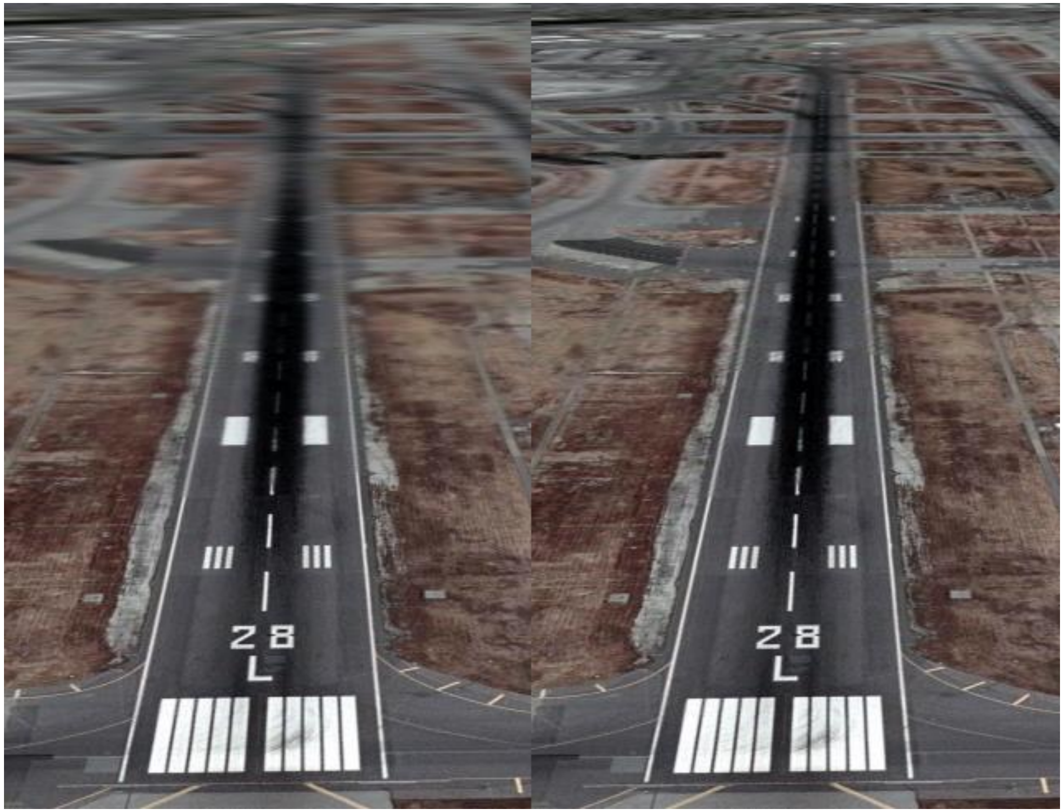
- large textures not perpendicular to viewing direction: blurring problems w.r.t. angle
- appropriate mip map selection not possible





Texture Mapping: Anisotropic Filtering

- large textures not perpendicular to viewing direction: blurring problems w.r.t. angle
- appropriate mip map selection not possible
- generate mip maps favoring one direction: 256×128, 256×64, 128×64, 128×32, ...



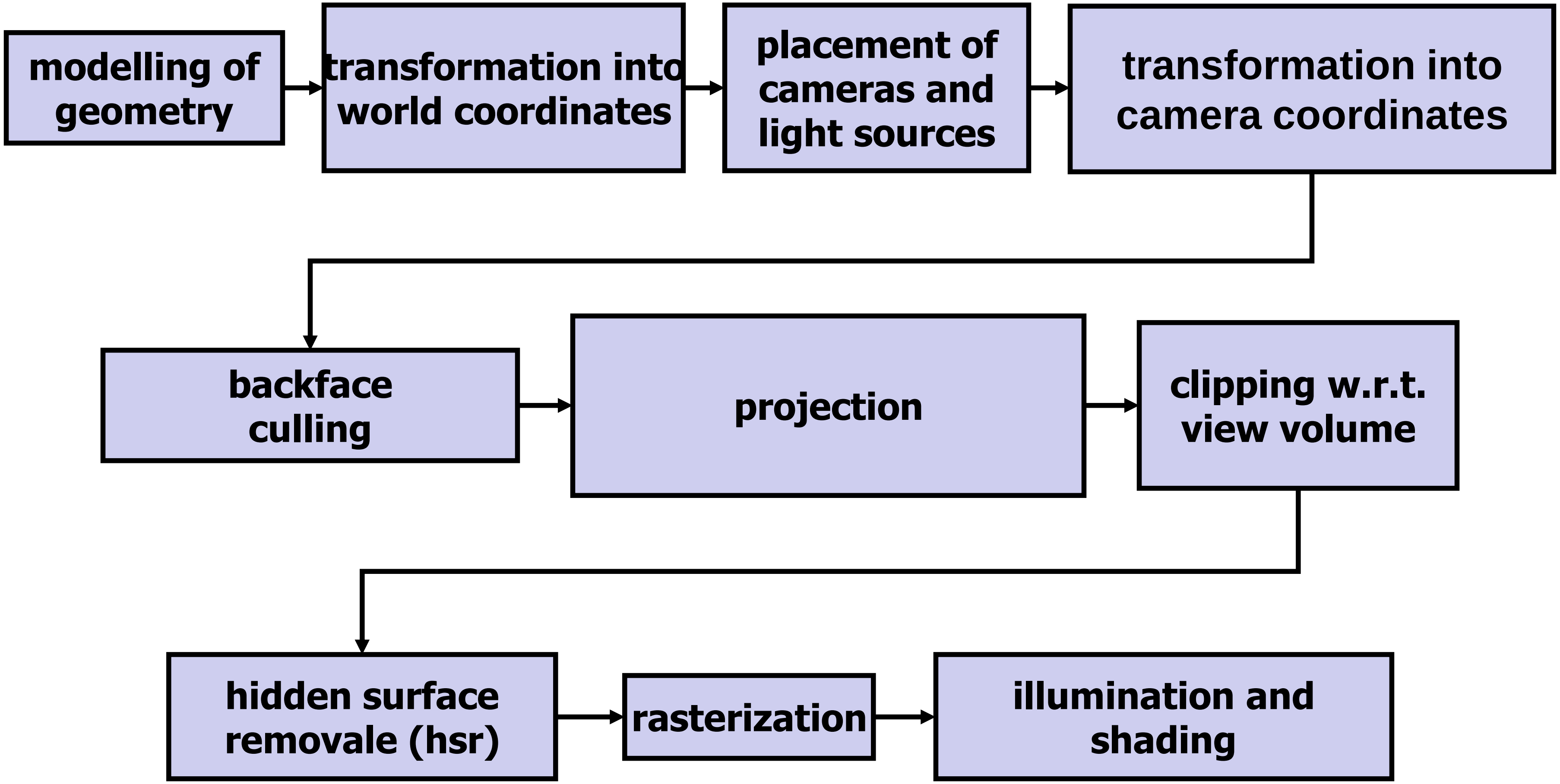
Texture Mapping: Summary

- a way to avoid having to model each detail using geometry and materials (modeling & rendering effort!)
- need textures and texture coordinates
- texture coordinates usually using 2 steps
- texture values typically affect diffuse/ ambient material color, but can change virtually anything in the rendering process

Pipeline-based Rendering

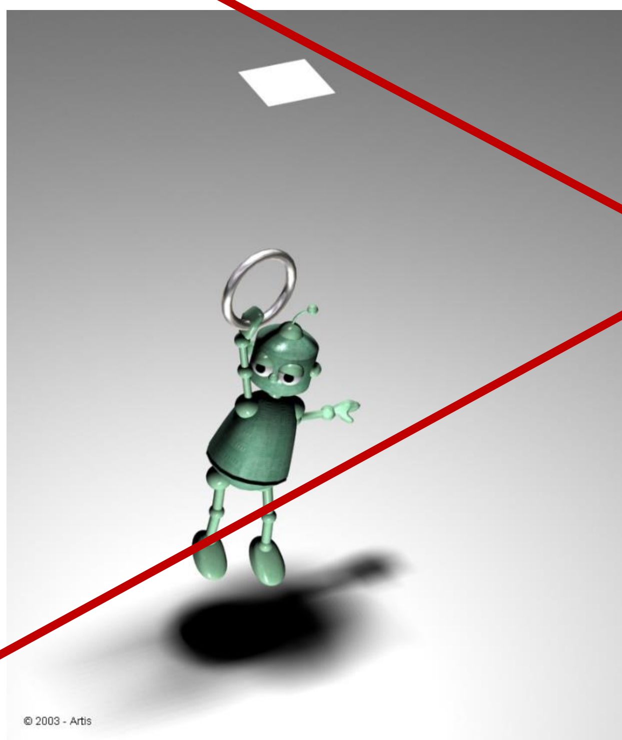
General Recaps

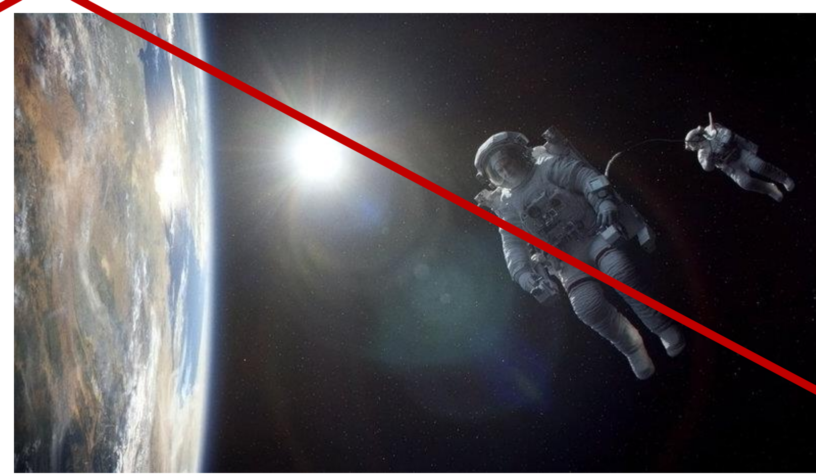
Recap: Rendering Pipeline



Recap: Differences from Cameras

- no shadows, no field of view, no lens flare, no motion blur

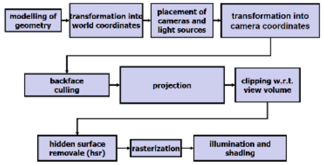


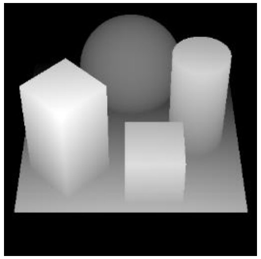


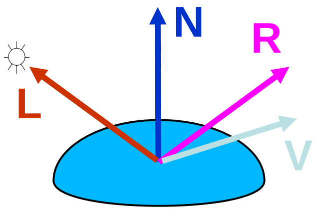
Recap: Efficiency & Effectiveness

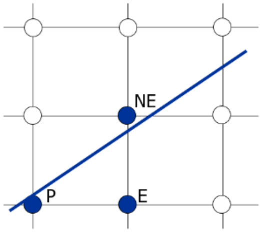
CG uses several “tricks”/strategies:

- only compute what is absolutely needed
- trade memory for speed
- trade precision for speed
- pre-“capture” data
- simplify, use heuristics
- reflect about the under-lying math & computation









Quality vs. Cost (GPU or Memory)

Quality vs. Cost (GPU or Memory)

Quality vs. Cost (GPU or Memory)

Quality vs. Cost (GPU or Memory)

Quality settings in games

Quality settings in games