Computer Graphics

- Texturing -

Philipp Slusallek
Ömercan Yazici

Overview

- **Last time**
  - Shading
  - BRDFs (will continue Monday)
- **Today**
  - Texture definition
  - Image textures
  - Procedural textures
  - Texture mapping
- **Next lecture**
  - Alias & signal processing

Texture

- **Textures modify the input for shading computations**
  - Either via (painted) images textures or procedural functions
- **Example texture maps for**
  - Reflectance, normals, shadow reflections, …

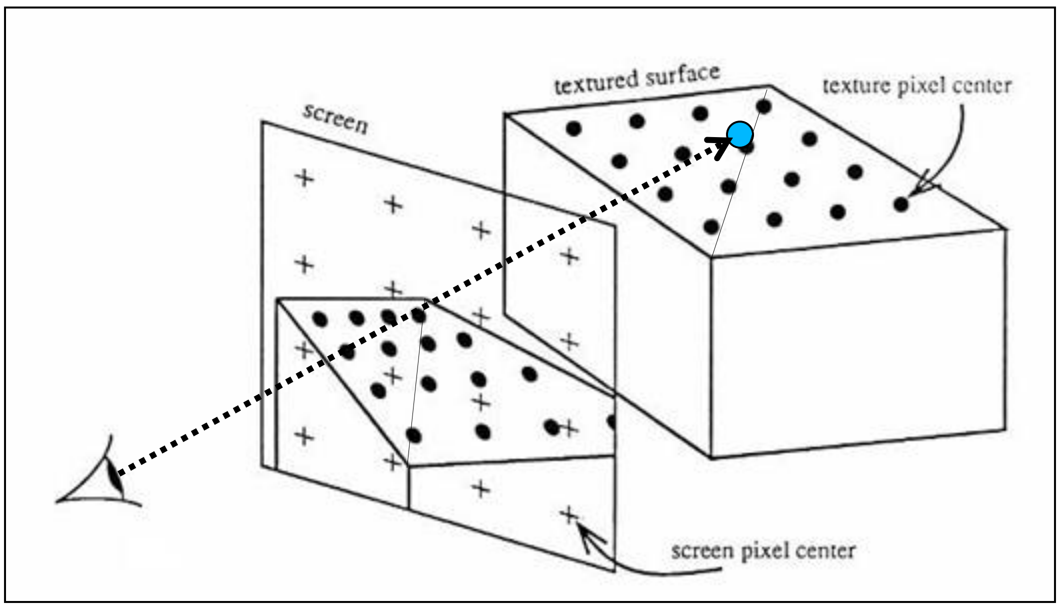
Definition: Textures

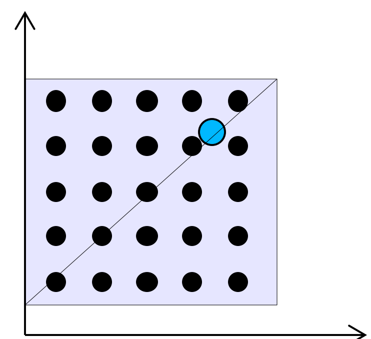
- **Texture maps texture coordinates to shading values**
  - Input: 1D/2D/3D texture coordinates
    - Explicitly given or derived via other data (e.g. position, direction, …)
  - Output: Scalar or vector value
- **Modified values in shading computations**
  - Reflectance
    - Changes the diffuse or specular reflection coefficient (k_d, k_s)
  - Geometry and Normal (important for lighting)
    - Displacement mapping $P' = P + \Delta P$
    - Normal mapping $N' = N + \Delta N$
    - Bump mapping $N' = N(P + tN)$
  - Opacity
    - Modulating transparency (e.g. for fences in games)
  - Illumination
    - Light maps, environment mapping, reflection mapping

IMAGE TEXTURES

Reconstruction Filter

- **Image texture**
  - Discrete set of sample values (given at texel centers!)
- **In general**
  - Hit point does not exactly hit a texture sample
- **Still want to reconstruct a continuous function**
  - Use reconstruction filter to find color for hit point

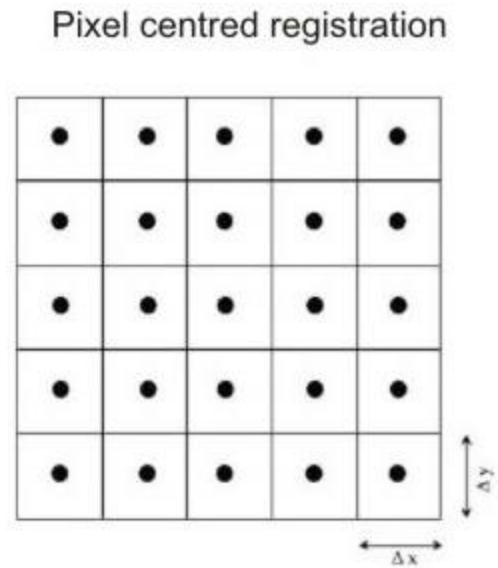


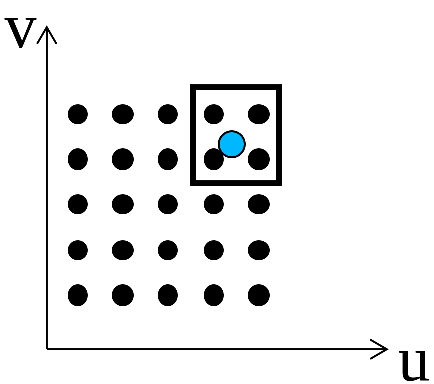


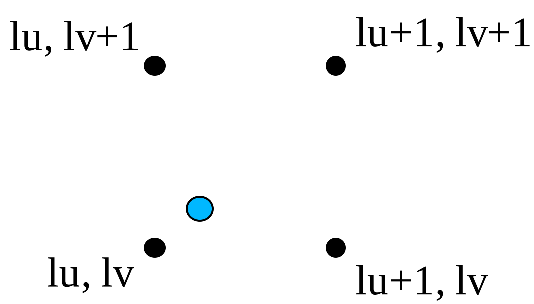
Texture Space

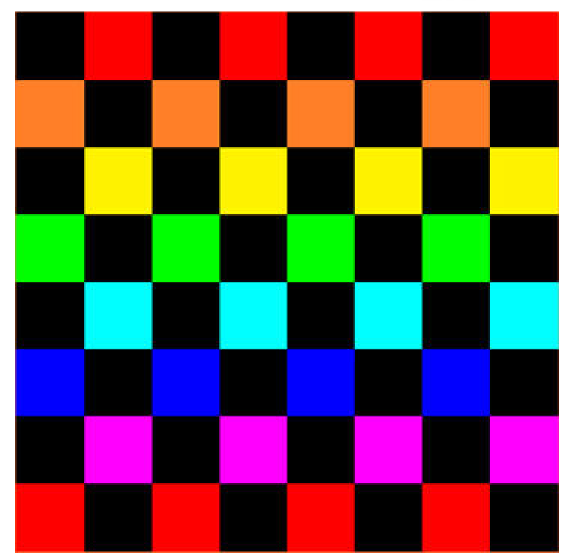
Nearest Neighbor

- **Local Coordinates**
  - Assuming cell-centered samples
  - u = tu * resU;
  - v = tv * resV;
- **Lattice Coordinates**
  - lu = min($\lfloor u \rfloor$, resU – 1);
  - lv = min($\lfloor v \rfloor$, resV – 1);
- **Texture Value**
  - return image[lu, lv];







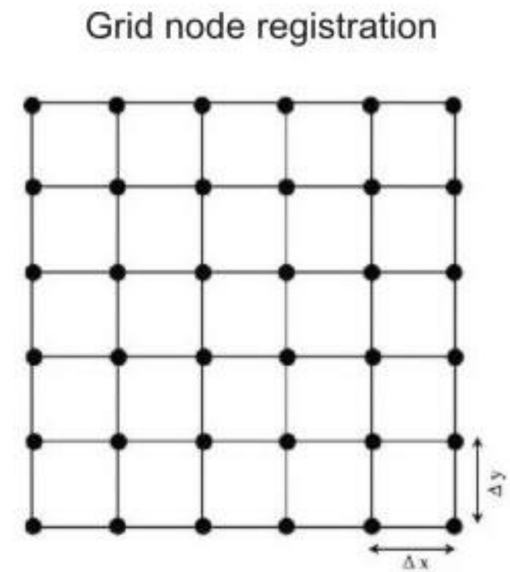


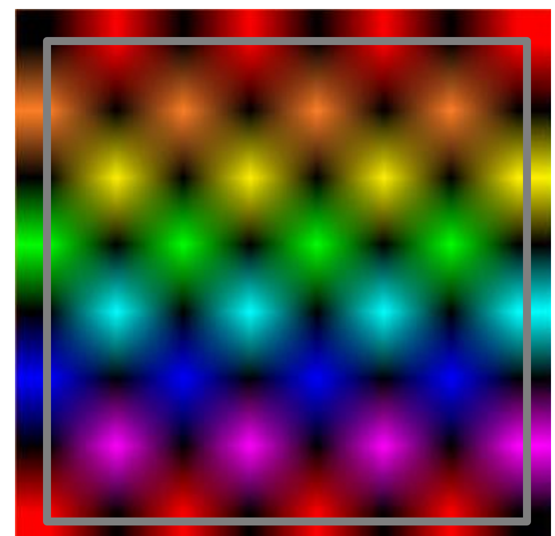
Bilinear Interpolation

- **Local Coordinates**
  - Assuming node-centered samples
  - u = tu * (resU – 1);
  - v = tv * (resV – 1);

- **Fractional Coordinates**
  - fu = u - $\lfloor$ u $\rfloor$;
  - fv = v - $\lfloor$ v $\rfloor$;

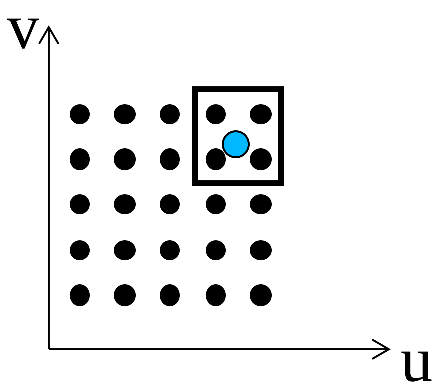
- **Texture Value**
  - return (1-fu) (1-fv) image[$\lfloor$u$\rfloor$, $\lfloor$v$\rfloor$]
    + (1-fu) (fv) image[$\lfloor$u$\rfloor$, $\lfloor$v$\rfloor$+1]
    + (fu) (1-fv) image[$\lfloor$u$\rfloor$+1, $\lfloor$v$\rfloor$]
    + (fu) (fv) image[$\lfloor$u$\rfloor$+1, $\lfloor$v$\rfloor$+1]

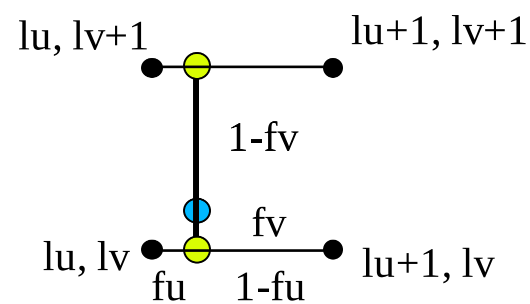




Bilinear Interpolation

- **Successive Linear Interpolations**
  - $u0 = (1\text{-}fv)\ \text{image}[\lfloor u \rfloor\ ,\ \lfloor v \rfloor\ ]$
    $+ (\quad fv)\ \text{image}[\lfloor u \rfloor\ ,\ \lfloor v \rfloor+1];$
  - $u1 = (1\text{-}fv)\ \text{image}[\lfloor u \rfloor+1, \lfloor v \rfloor\ ]$
    $+ (\quad fv)\ \text{image}[\lfloor u \rfloor+1, \lfloor v \rfloor+1];$
  - return $(1\text{-}fu)\ u0$
    $+ (\quad fu)\ u1;$





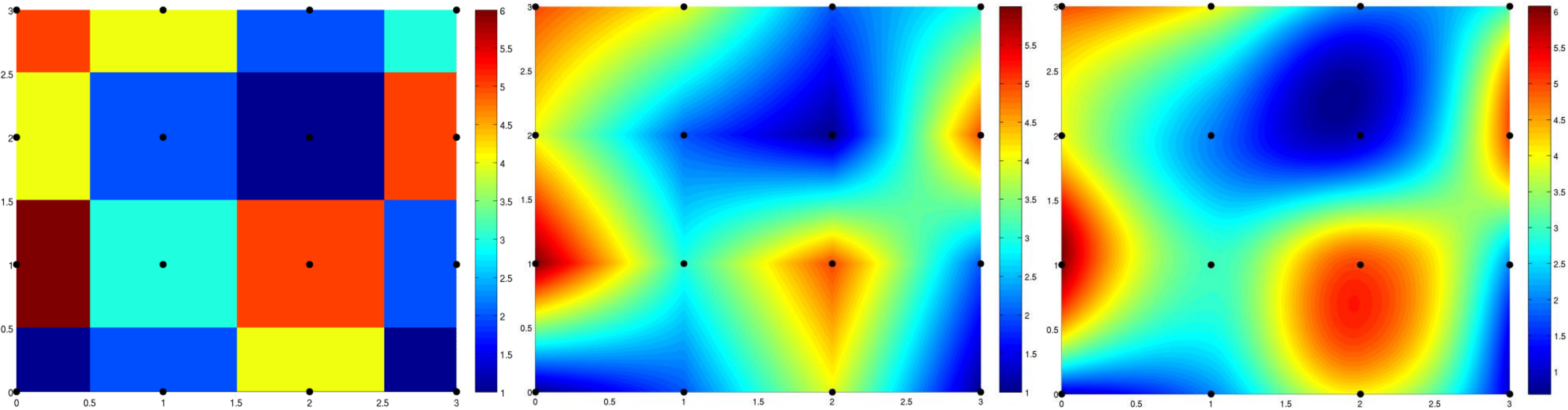
Nearest vs. Bilinear Interpolation

GL_NEAREST

GL_LINEAR

Bicubic Interpolation

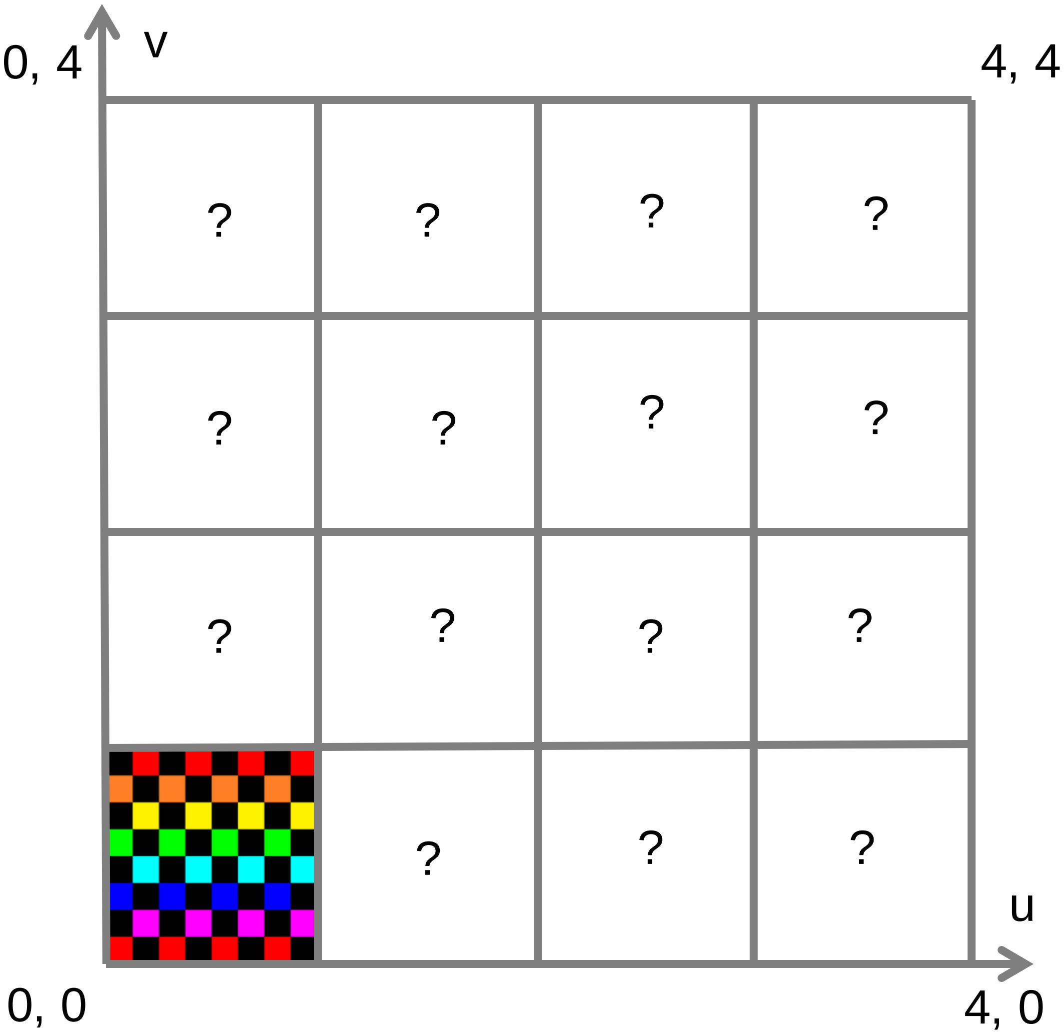
- **Properties**
  - Assuming node-centered samples
  - Essentially based on cubic splines (see later)
- **Pros**
  - Even smoother
- **Cons**
  - More complex & expensive (4x4 kernel)
  - Overshoot



Wrap Mode

- **Texture Coordinates**
  - (u, v) in [0, 1] x [0, 1]

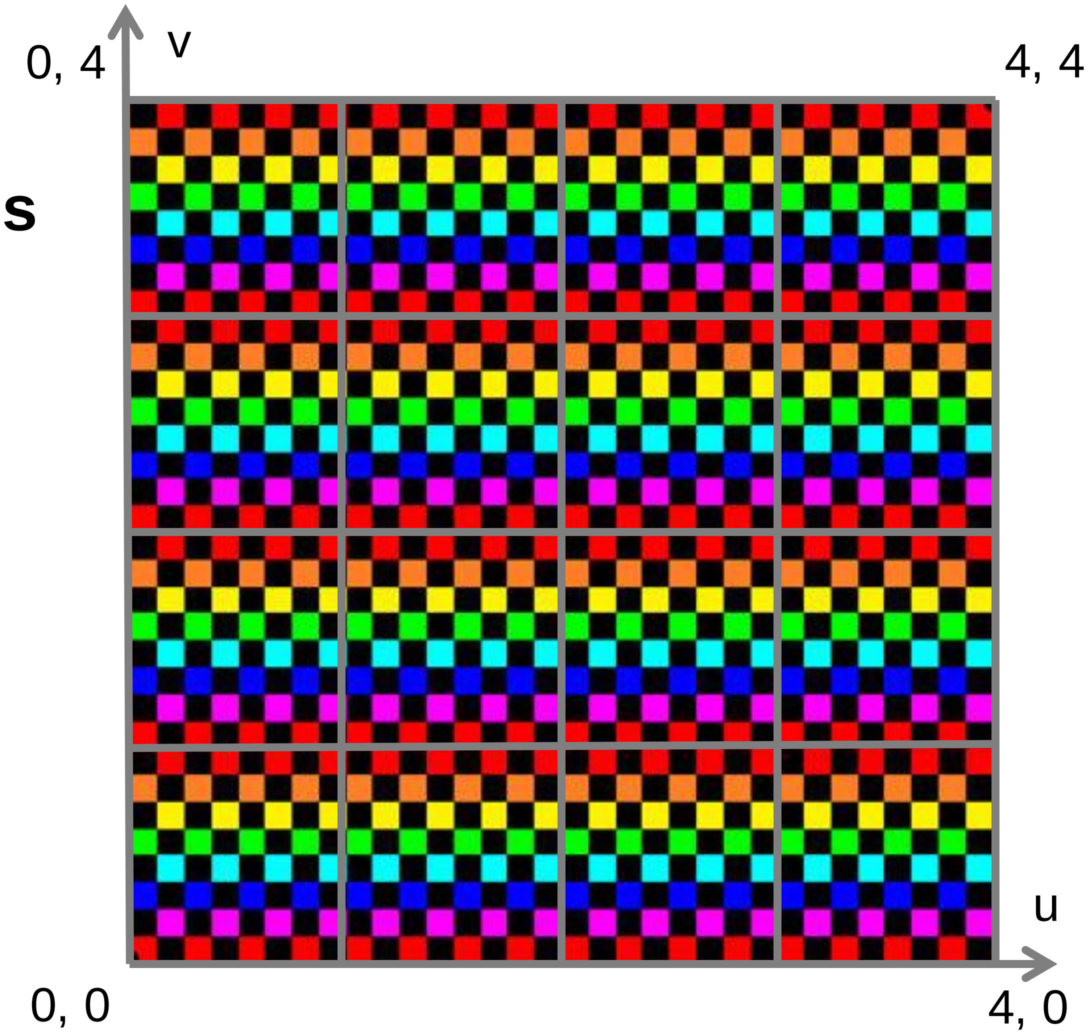
- **What if?**
  - (u, v) not in unit square?



Wrap Mode

- **Repeat**

- **Fractional Coordinates**
  - $t_u = u - \lfloor u \rfloor$
  - $t_v = v - \lfloor v \rfloor$



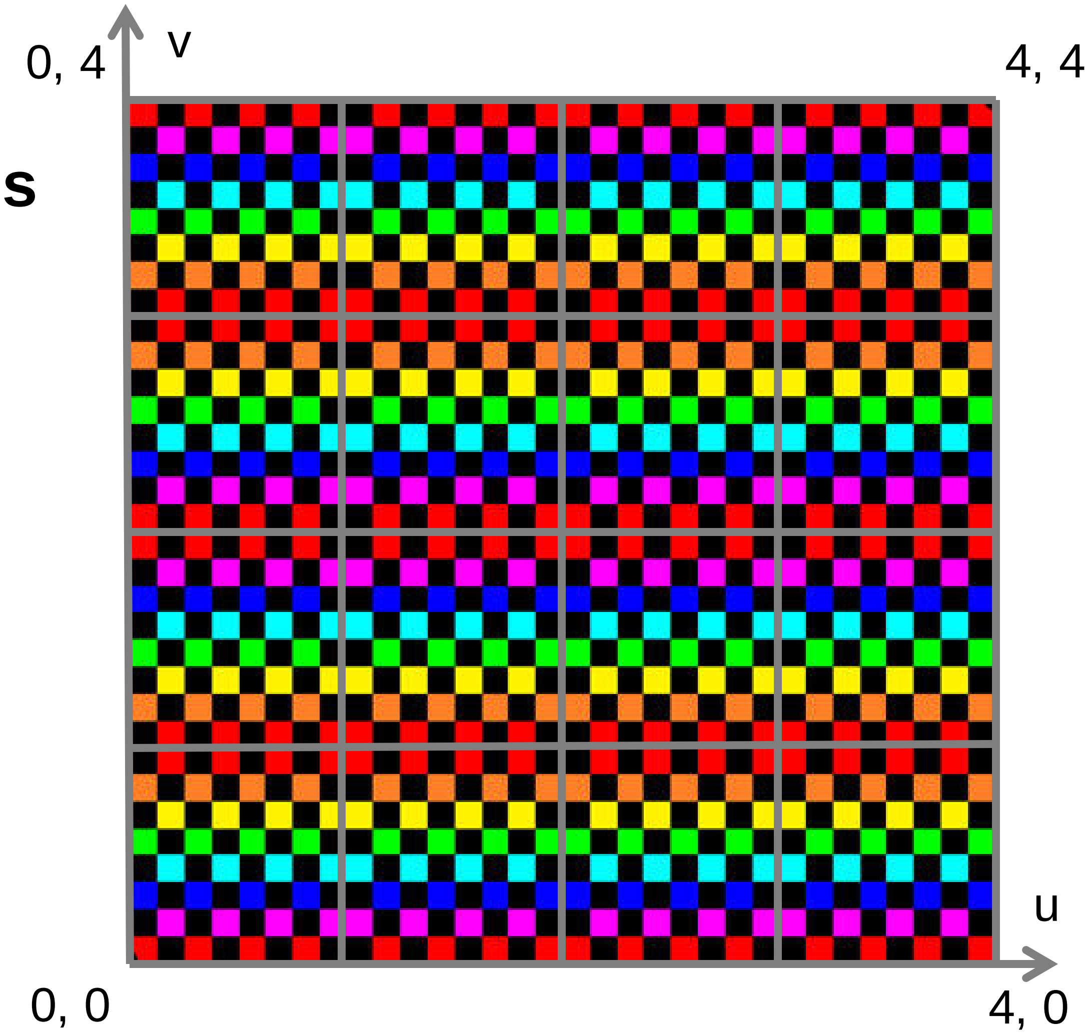
Wrap Mode

- **Mirror**

- **Fractional Coordinates**
  - $t_u = u - \lfloor u \rfloor$
  - $t_v = v - \lfloor v \rfloor$

- **Lattice Coordinates**
  - $l_u = \lfloor u \rfloor$
  - $l_v = \lfloor v \rfloor$

- **Mirror if Odd**
  - if (l_u % 2 == 1)
    t_u = 1 - t_u
  - if (l_v % 2 == 1)
    t_v = 1 - t_v



Wrap Mode

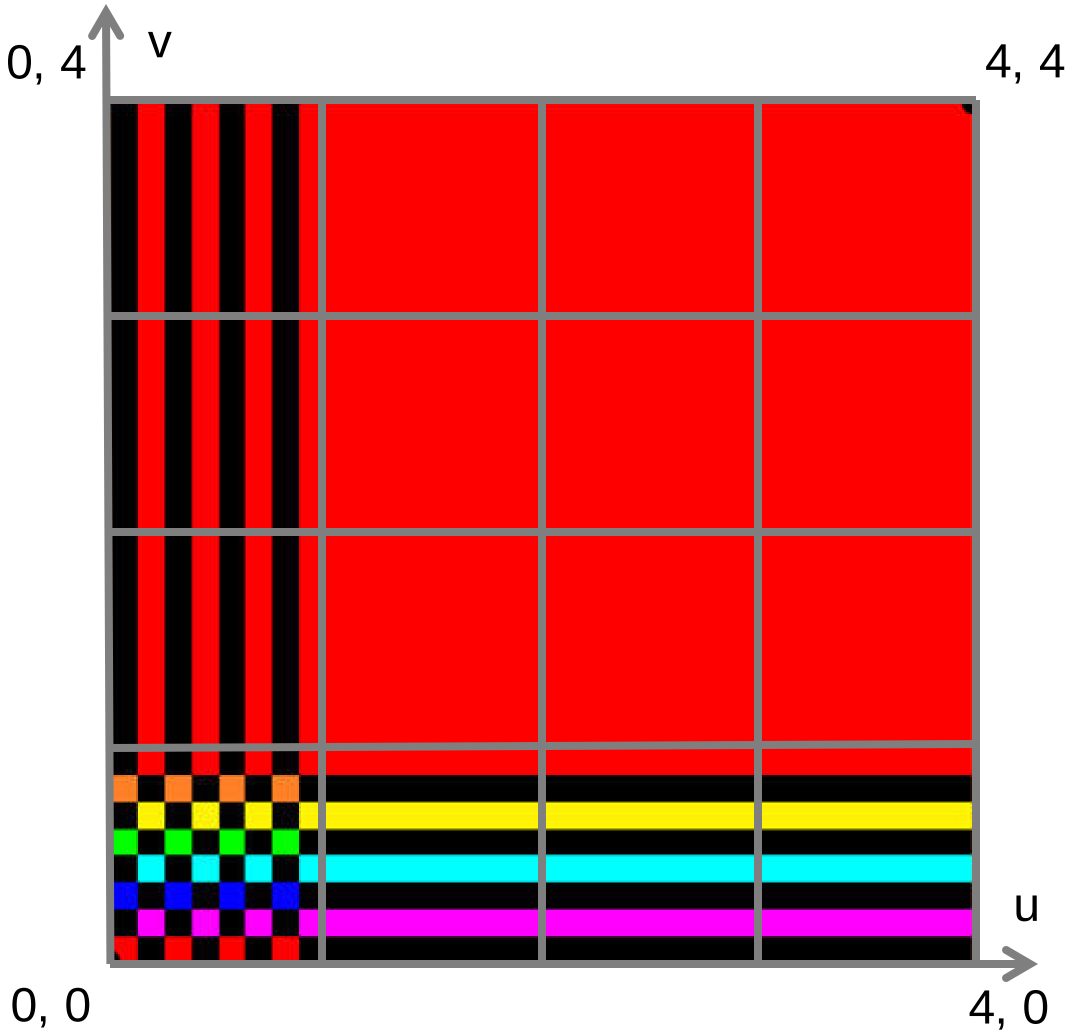
- **Clamp**

- **Clamp u to [0, 1]**

```
if          (u < 0) tu = 0;
else if (u > 1) tu = 1;
else                 tu = u;
```

- **Clamp v to [0, 1]**

```
if          (v < 0) tv = 0;
else if (v > 1) tv = 1;
else                 tv = v;
```



Wrap Mode

- **Border**

- **Check Bounds**

if (u < 0 || u > 1

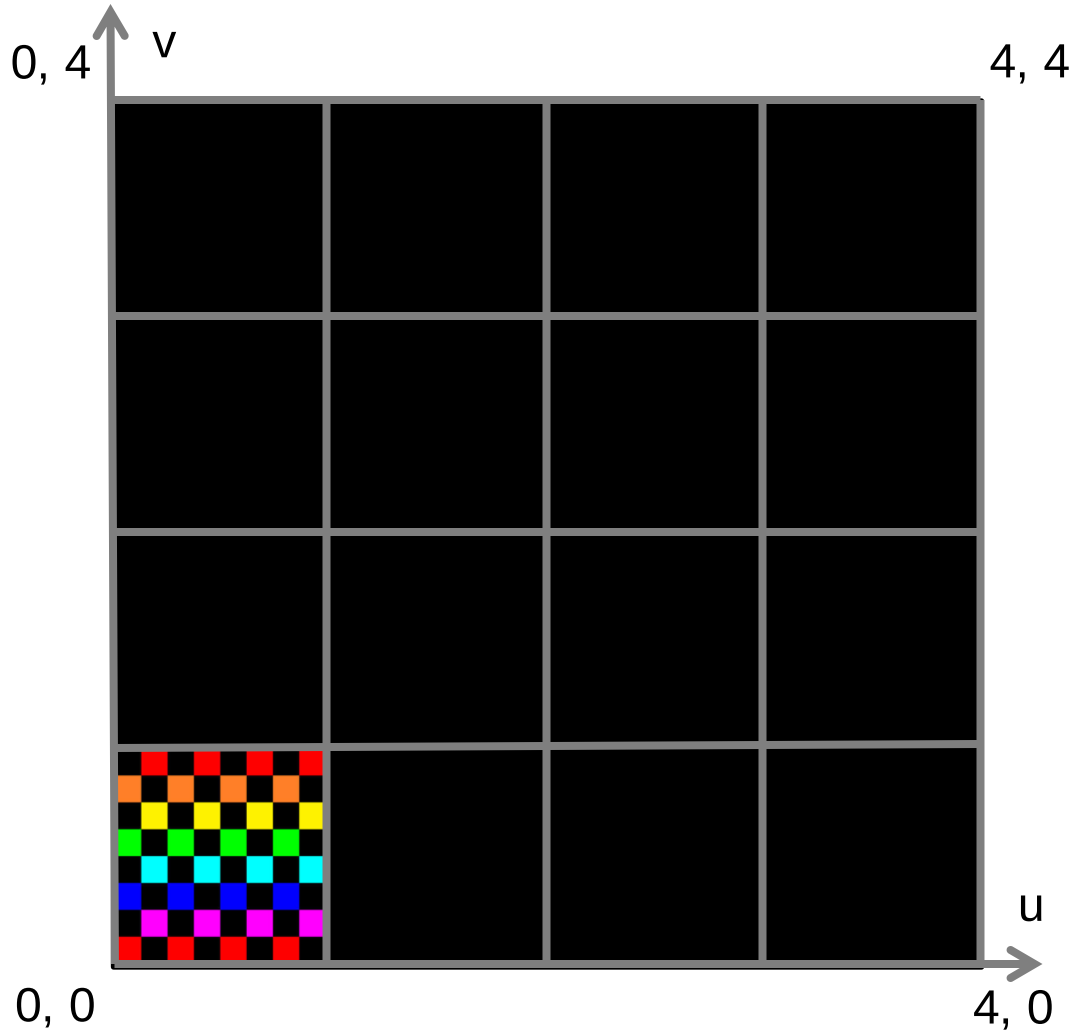
|| v < 0 || v > 1)

return backgroundColor;

else

tu = u;

tv = v;



Wrap Mode

- **Comparison**
  – With OpenGL texture modes

Discussion: Image Textures

- **Pros**
  - Simple generation
    - Painted, simulation, ...
  - Simple acquisition
    - Photos, videos

- **Cons**
  - Illumination “frozen” during acquisition
  - Limited resolution
  - Susceptible to aliasing
  - High memory requirements (often HUGE for films, 100s of GB)
  - Issues when mapping 2D image onto 3D object

PROCEDURAL TEXTURES

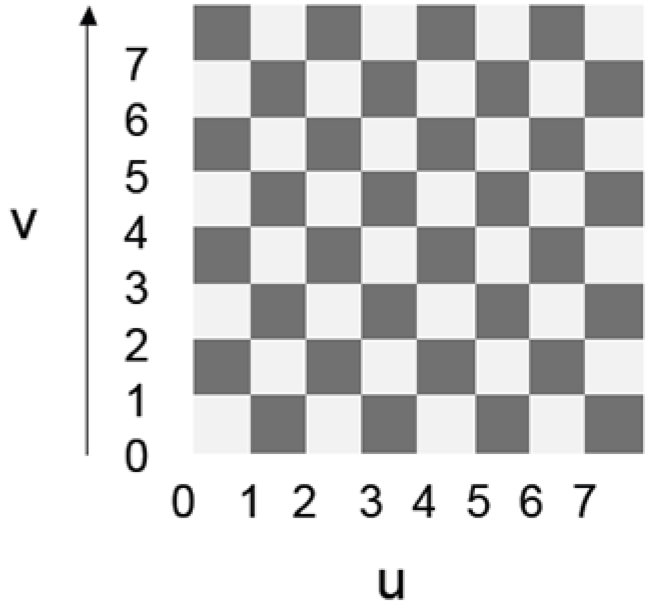
Discussion: Procedural Textures

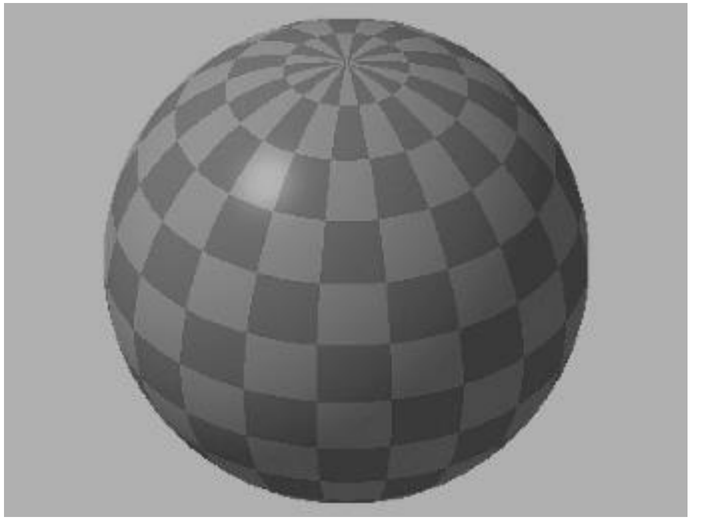
- **Cons**
  - Sometimes hard to achieve specific effect
  - Possibly non-trivial programming

- **Pros**
  - Flexibility & parametric control
  - Unlimited resolution
  - Anti-aliasing possible
  - Low memory requirements
  - May be directly defined as 3D “image” mapped to 3D geometry
  - Low-cost visual complexity

2D Checkerboard Function

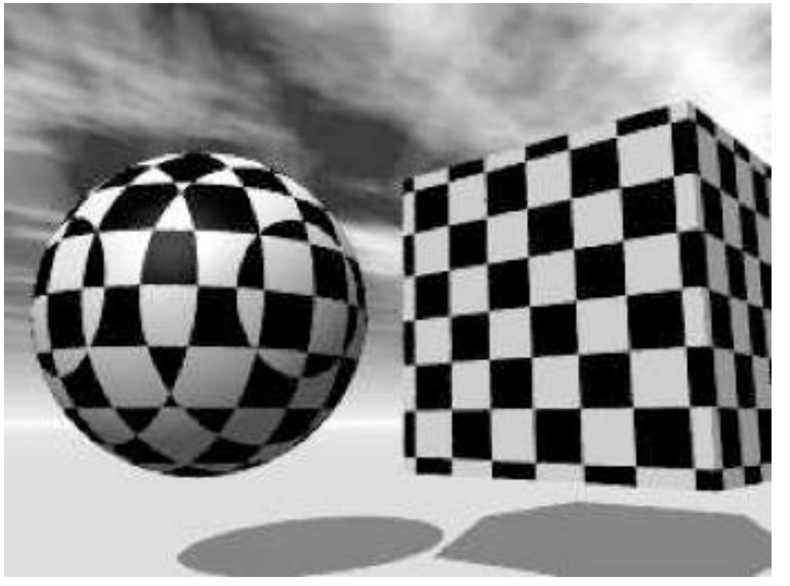
- **Lattice Coordinates**
  - lu $= \lfloor u \rfloor$
  - lv $= \lfloor v \rfloor$
- **Compute Parity**
  - parity = (lu + lv) % 2;
- **Return Color**
  - if (parity == 1)
    - return color1;
  - else
    - return color0;



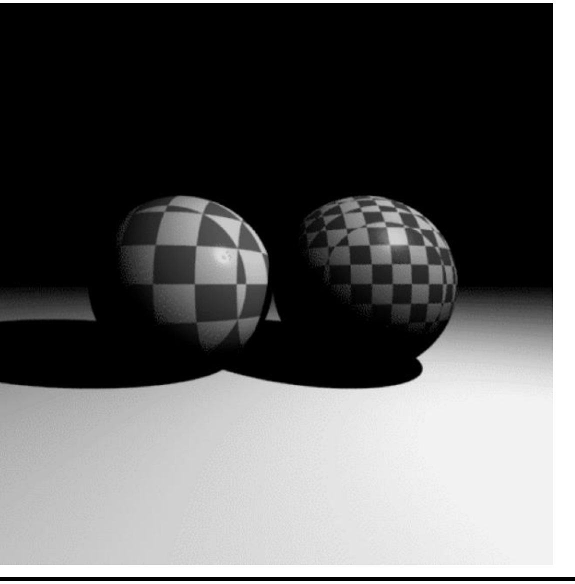


3D Checkerboard - Solid Texture

- **Lattice Coordinates**
  - $lu = \lfloor u \rfloor$
  - $lv = \lfloor v \rfloor$
  - $lw = \lfloor w \rfloor$



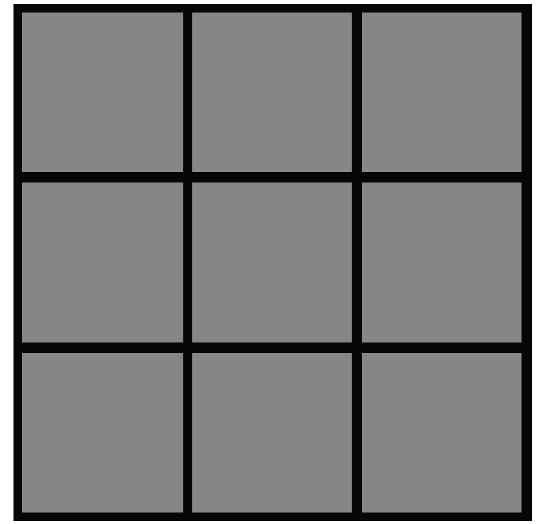
- **Compute Parity**
  - parity = (lu + lv + lw) % 2;

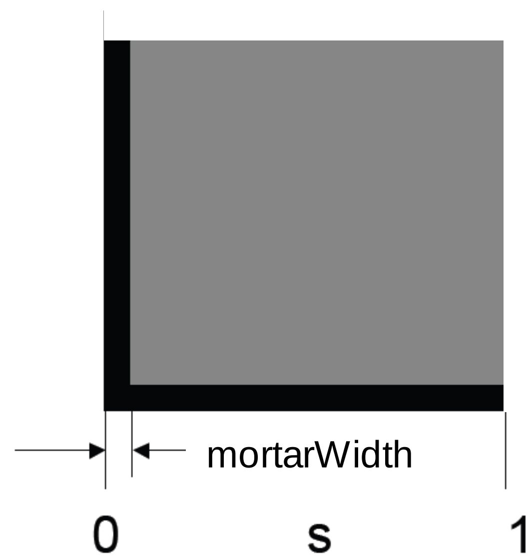


- **Return Color**
  - if (parity == 1)
    - return color1;
  - else
    - return color0;

Tile

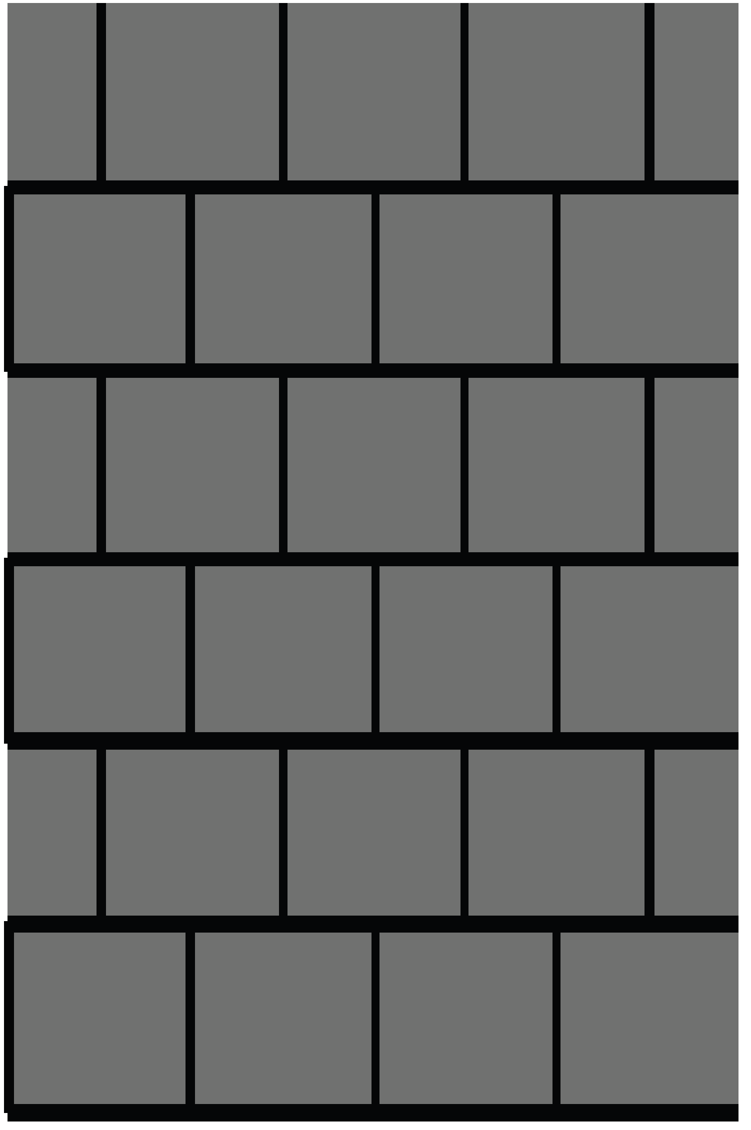
- **Fractional Coordinates**
  - $fu = u - \lfloor u \rfloor$
  - $fv = v - \lfloor v \rfloor$
- **Compute Booleans**
  - bu = fu < mortarWidth;
  - bv = fv < mortarWidth;
- **Return Color**
  - if (bu || bv)
    - return mortarColor;
  - else
    - return tileColor;



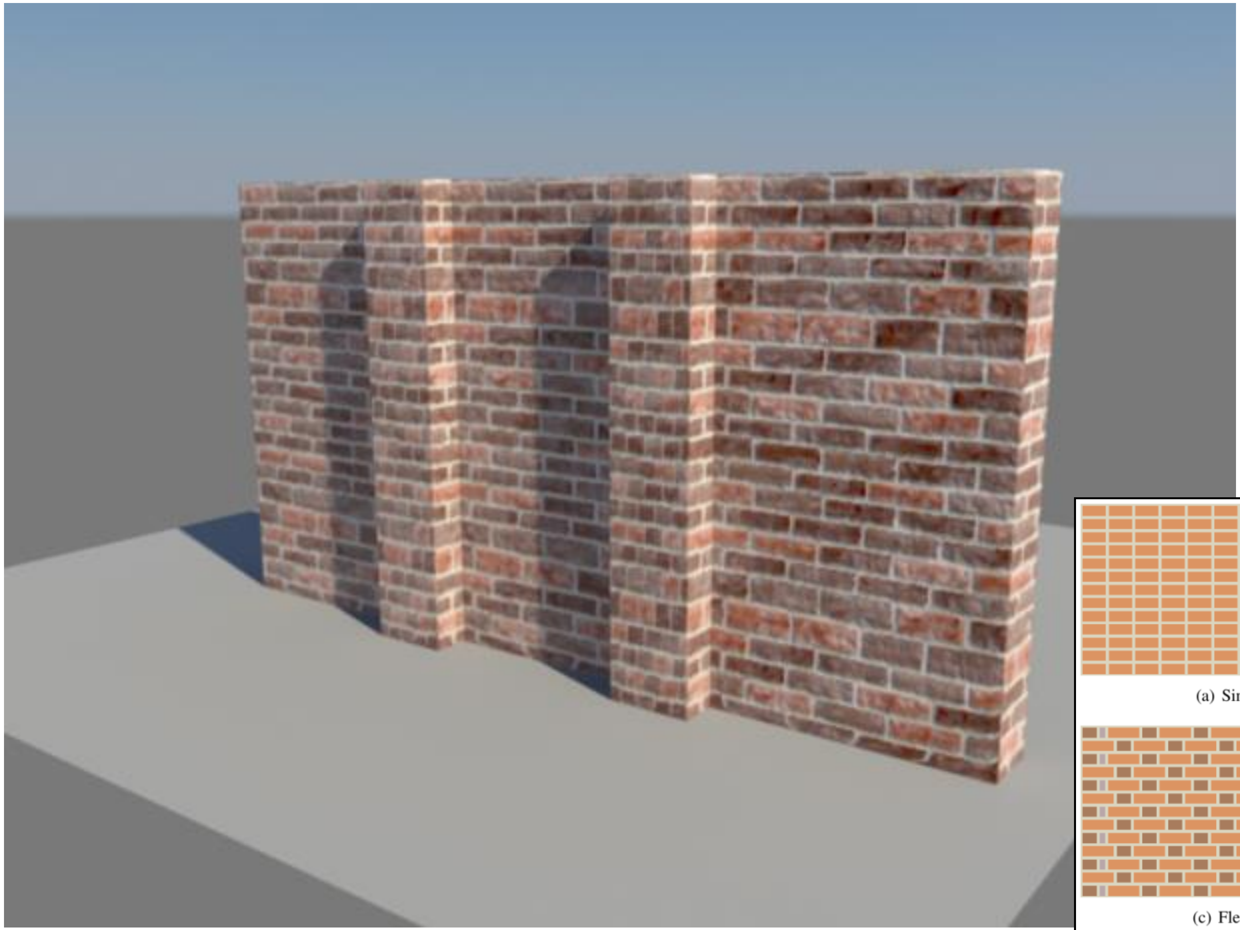


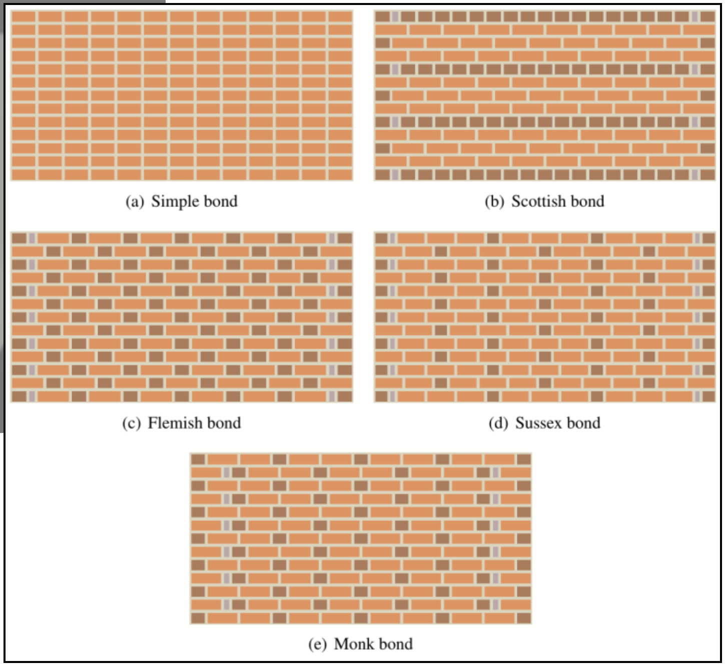
Brick

- **Shift Column for Odd Rows**
  - parity = $\lfloor v \rfloor$ % 2;
  - u -= parity * 0.5;
- **Fractional Coordinates**
  - fu = u - $\lfloor u \rfloor$
  - fv = v - $\lfloor v \rfloor$
- **Compute Booleans**
  - bu = fu < mortarWidth;
  - bv = fv < mortarWidth;
- **Return Color**
  - if (bu || bv)
    - return mortarColor;
  - else
    - return brickColor;



More Variation





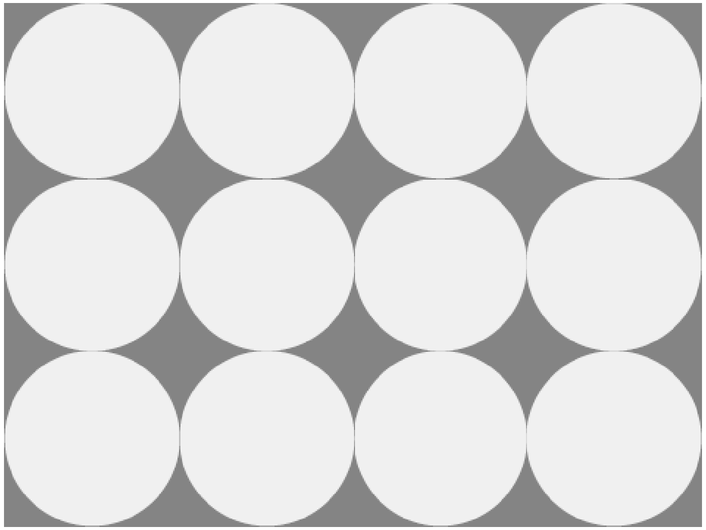
(a) Simple bond (b) Scottish bond

(c) Flemish bond (d) Sussex bond

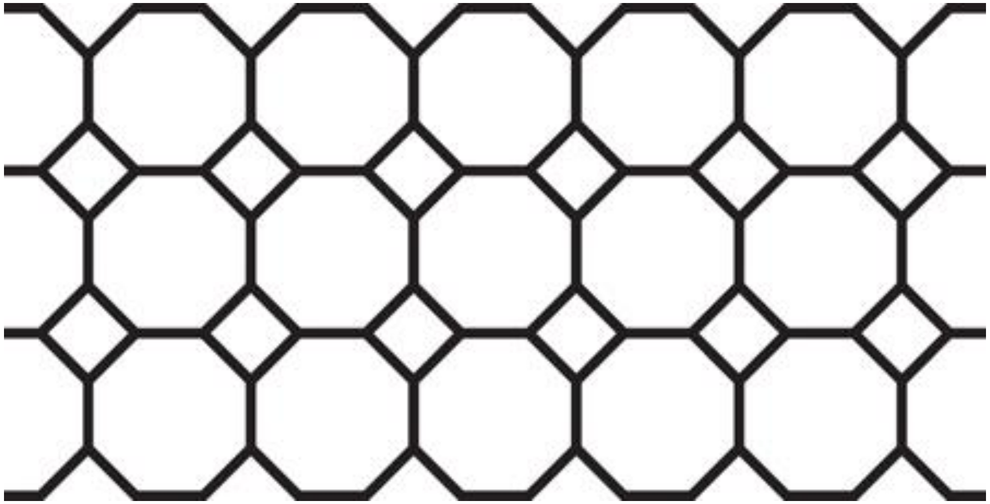
(e) Monk bond

Other Patterns

- **Circular Tiles**



- **Octagonal Tiles**



- **Use your imagination!**

Perlin Noise

- **Natural Patterns**
  - Similarity between patches at different locations
    - Repetitiveness, coherence (e.g., skin of a tiger or zebra)
  - Similarity on different resolution scales
    - Self-similarity
  - But never completely identical
    - Additional disturbances, turbulence, noise

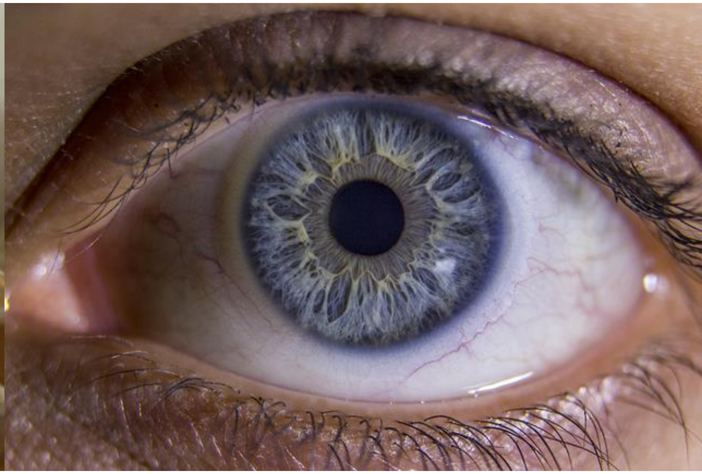
- **Mimic Statistical Properties**
  - Purely empirical approach
  - Looks convincing, but has nothing to do with material’s physics

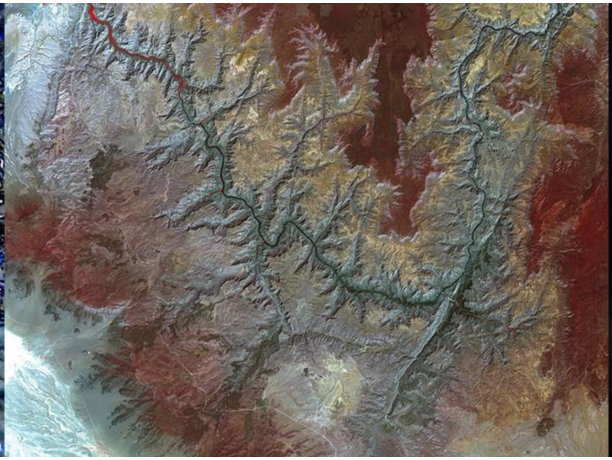
- **Perlin Noise is essential for adding “natural” details**
  - Used in many texture functions

Perlin Noise

- **Natural Fractals**

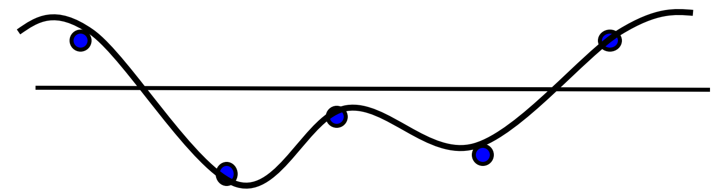




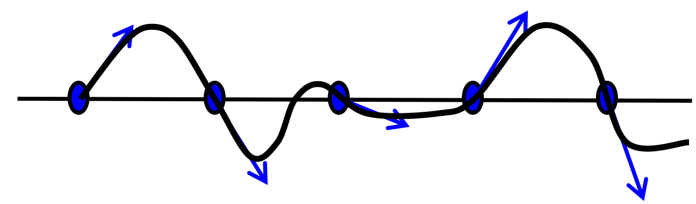


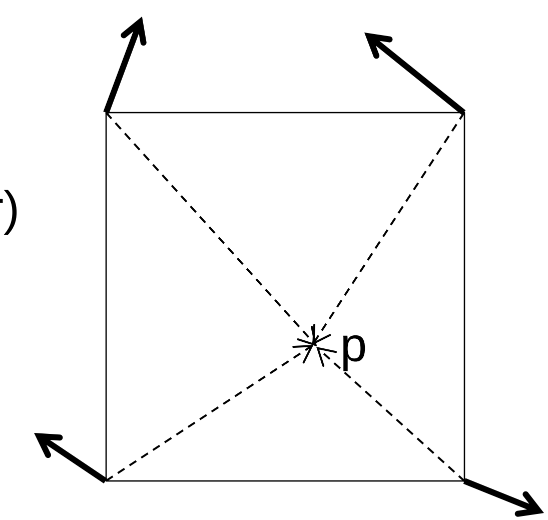

Noise Function

- **Noise(x, y, z)**
  - Statistical invariance under rotation
  - Statistical invariance under translation
  - Roughly fixed frequency of ~1 Hz



- **Integer Lattice (i, j, k)**
  - Value noise
    - Random value at lattice points
  - Gradient noise
    - Random gradient vector at lattice point
  - Interpolation
    - Bi-/tri-linear or cubic (Hermite spline, → later)
  - Hash function to map vertices to values
    - Randomized look up
    - Virtually infinite extent and variation with finite array of values

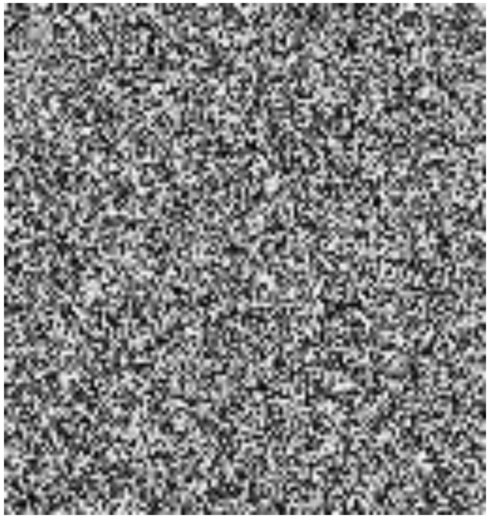




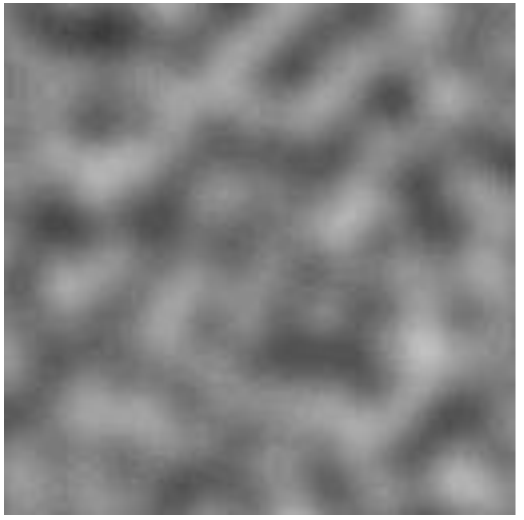
Noise vs. Noise

- **Value Noise vs. Gradient Noise**
  - Gradient noise has lower regularity artifacts
  - More high frequencies in noise spectrum

- **Random Values vs. Perlin Noise**
  - Stochastic vs. deterministic



Random values
at each pixel

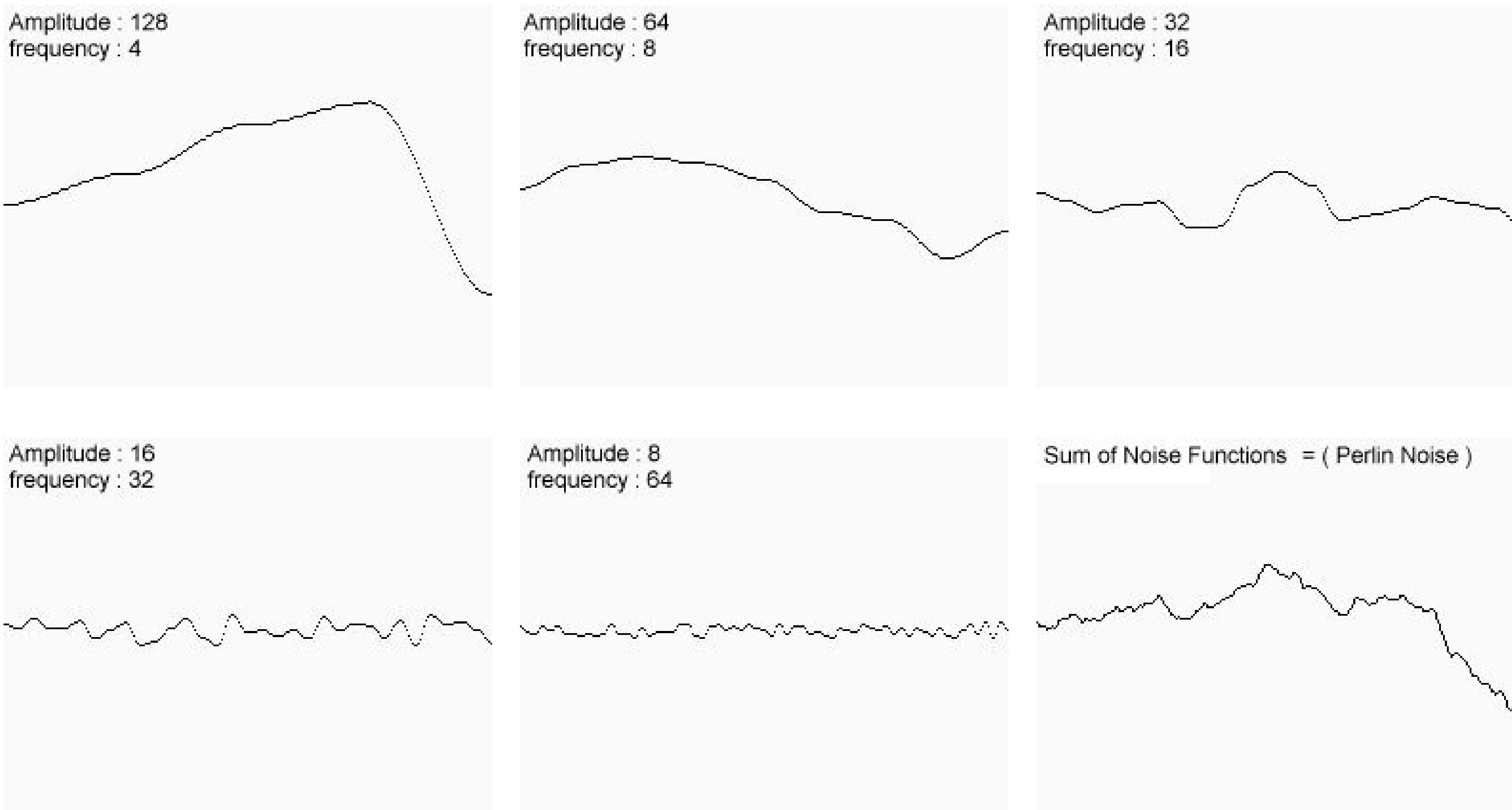


Gradient noise

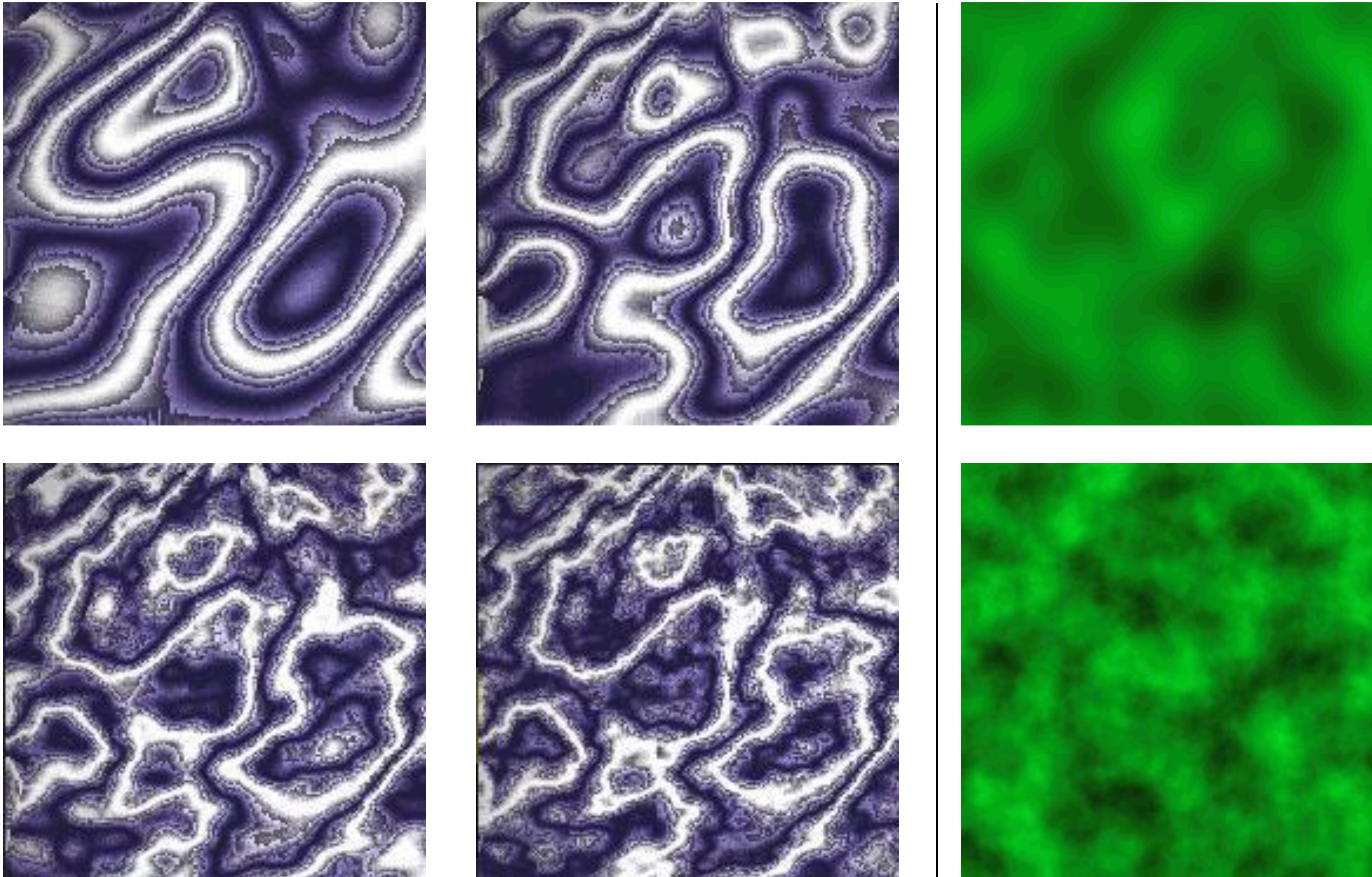
Turbulence Function

- **Noise Function**
  - Single spike in frequency spectrum (single frequency, see later)
- **Natural Textures**
  - Mix of different frequencies
  - Decreasing amplitude for high frequencies
- **Turbulence from Noise**
  - $Turbulence(x) = \sum_{i=0}^{k} |a_i * noise(f_i\, x)|$
    - Frequency: $f_i = 2^i$
    - Amplitude: $a_i = 1 / p^i$
    - Persistence: *p* typically *p=2*
    - Power spectrum : $a_i = 1 / f_i$
    - Brownian motion: $a_i = 1 / f_i^2$
  - Summation truncation
    - 1st term: noise(x)
    - 2nd term: noise(2x)/2
    - …
    - Until period $(1/f_k)$ < 2 pixel-size (band limit, see later)

Synthesis of Turbulence (1-D)

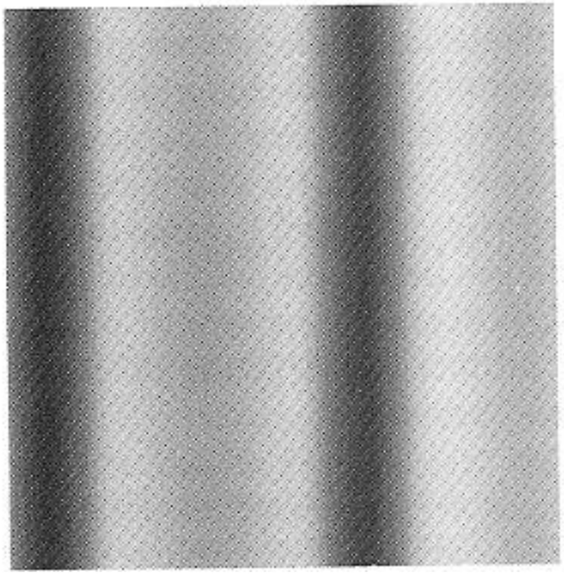


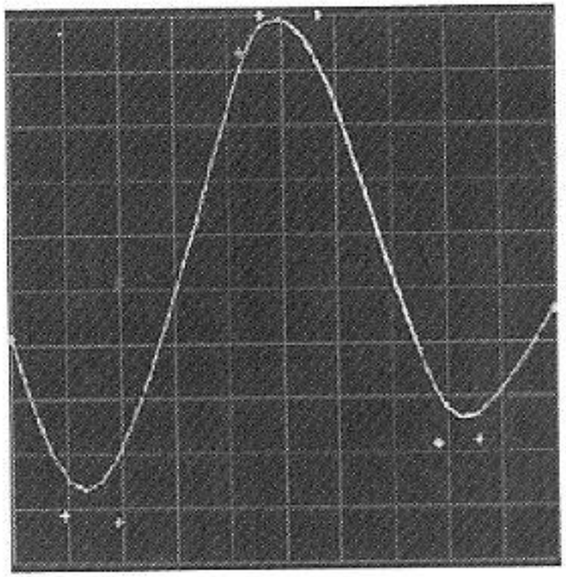
Synthesis of Turbulence (2-D)



Example: Marble

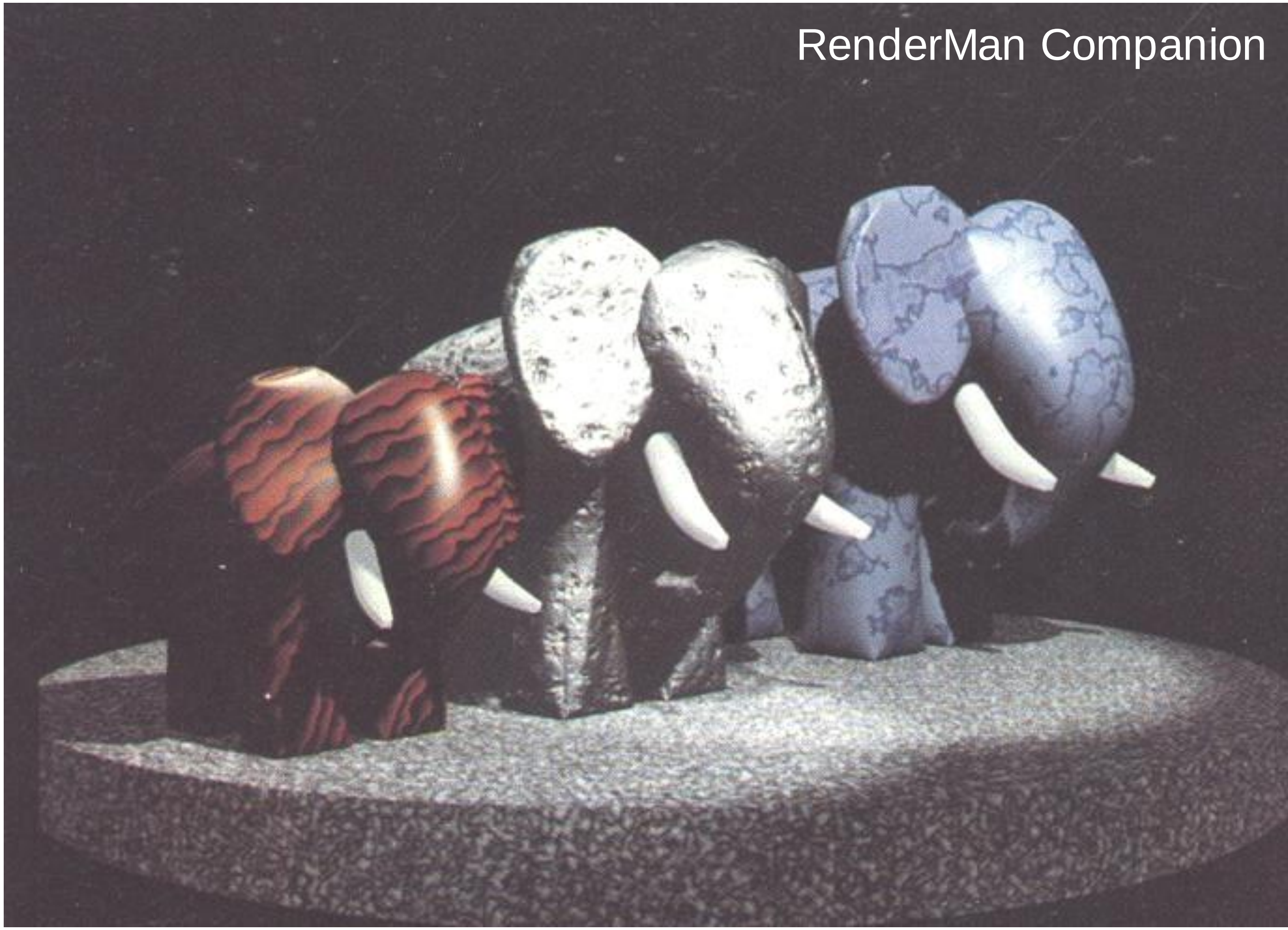
- **Overall Structure**
  - Smoothly alternating layers of different marble colors
  - $f_{marble}(x,y,z) := marble\_color(\sin(x))$
  - marble_color : transfer function (see lower left)
- **Realistic Appearance**
  - Simulated turbulence
  - $f_{marble}(x,y,z) := marble\_color(\sin(x + turbulence(x, y, z)))$





Solid Noise

- **3D Noise Texture**
  - Wood
  - Erosion
  - Marble
  - Granite
  - …



Other Applications

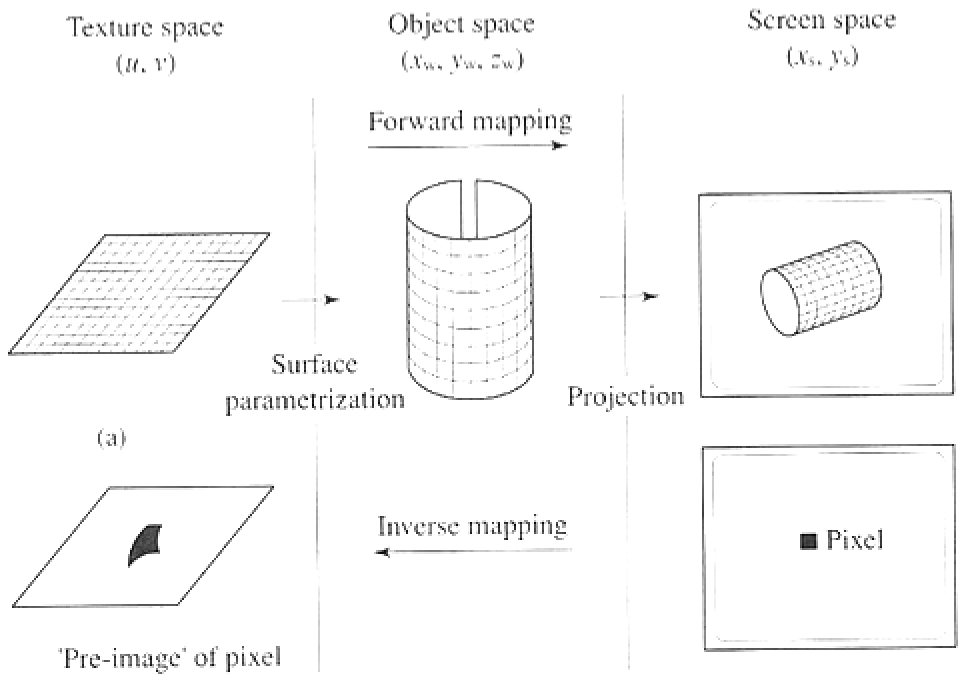
- **Bark**
  - Turbulated saw-tooth function
- **Clouds**
  - White blobs
  - Turbulated transparency along edge
- **Animation**
  - Vary procedural texture function's parameters over time

Shading Languages

- **Small program fragments (plugins)**
  - Compute certain aspects of the rendering process
    - Executing at innermost loop, must be extremely efficient
  - Executed at each intersection
- **Typical shaders**
  - Material/surface shaders: Compute reflected color
  - Light shaders: Compute illumination from light source at some point
  - Volume shader: Compute interaction in participating medium
  - Displacement shader: Compute changes to the geometry
  - Camera shader: Compute rays for each pixel
- **Shading languages**
  - RenderMan (the mother of all shading languages)
  - HLSL (DX only), GLSL (OpenGL only), CG (Nvidia only)
  - OSL (Modern approach)
  - Currently no portable shading format usable for exchange
    - But Material Definition Language (MDL, Nvidia), shade.js (UdS)
- **More details later**

TEXTURE MAPPING

2D Texture Mapping

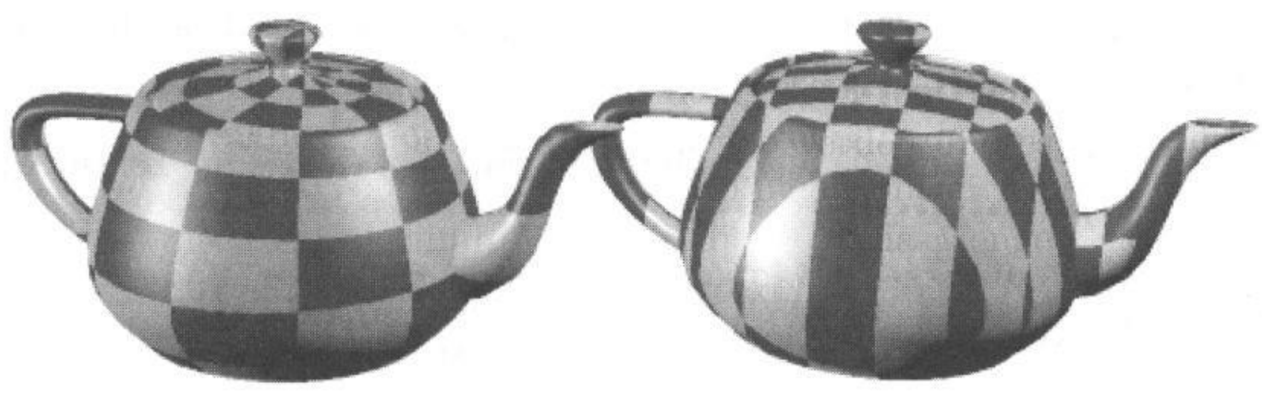


- **Forward mapping**
  - Object surface parameterization
  - Projective transformation
- **Inverse mapping**
  - Find corresponding pre-image/footprint of each pixel in texture
  - Integrate over pre-image

Surface Parameterization

- **To apply textures we need 2D coordinates on surfaces**

  → Parameterization
- **Some objects have a natural parameterization**
  - Sphere: spherical coordinates $(\varphi, \theta) = (2\pi\, u, \pi\, v)$
  - Cylinder: cylindrical coordinates $(\varphi, h) = (2\,\pi\, u, H\, v)$
  - Parametric surfaces (such as B-spline or Bezier surfaces → later)
- **Parameterization is less obvious for**
  - Polygons, implicit surfaces, teapots, …



Triangle Parameterization

- **Triangle is a planar object**
  - Has implicit parameterization (e.g., barycentric coordinates)
  - But we need more control: Placement of triangle in texture space
- **Assign texture coordinates *(u,v)* to each vertex (x_o,y_o,z_o)**
- **Apply viewing projection $(x_o,y_o,z_o) \rightarrow (x,y)$ (details later)**
- **Yields full texture transformation (warping) *(u,v)* → *(x,y)***

$$x = \frac{au + bv + c}{gu + hv + i} \qquad y = \frac{du + ev + f}{gu + hv + i}$$

  - In homogeneous coordinates (by embedding (u,v) as (u,v,1))

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} u' \\ v' \\ q \end{bmatrix}; (x,y) = \left(\frac{x'}{w}, \frac{y'}{w}\right), (u,v) = \left(\frac{u'}{q}, \frac{v'}{q}\right)$$

  - Transformation coefficients determined by 3 pairs *(u,v)*→*(x,y)*
    - Three linear equations
    - Invertible if neither set of points is collinear

Triangle Parameterization (2)

- **Given** $$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} u' \\ v' \\ q \end{bmatrix}$$

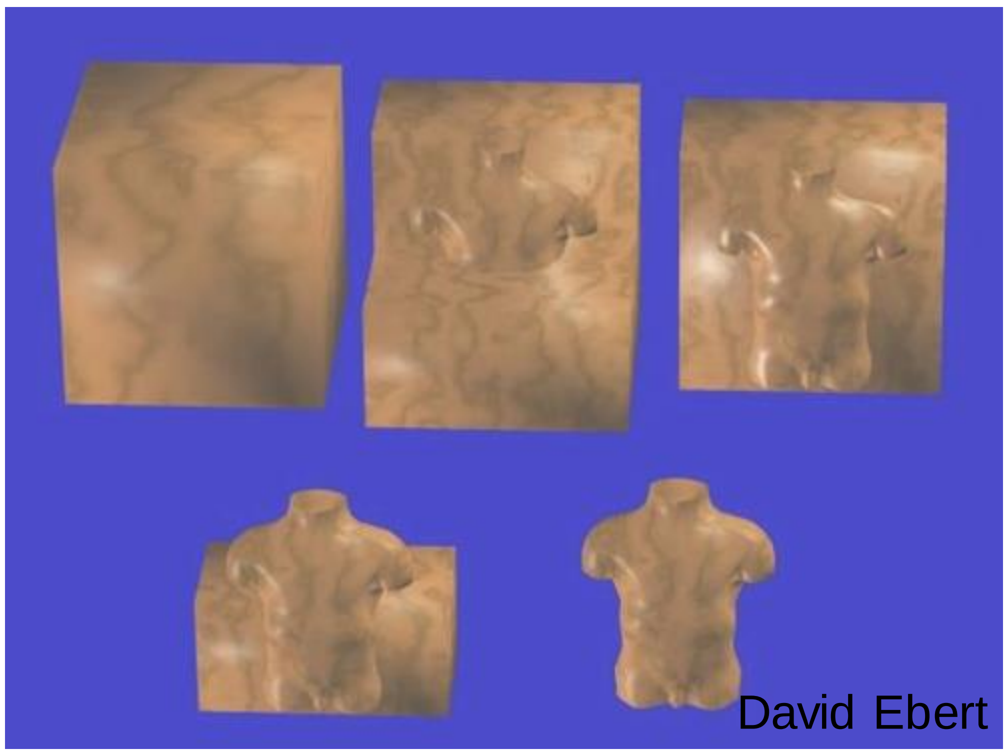
- **The inverse transform *(x,y)*→*(u,v)* is**

$$\begin{bmatrix} u' \\ v' \\ q \end{bmatrix} = \begin{bmatrix} ei - fh & ch - bi & bf - ce \\ fg - di & ai - cg & cd - af \\ dh - eg & bg - ah & ae - bd \end{bmatrix} \begin{bmatrix} x' \\ y' \\ w \end{bmatrix}$$

- **Coefficients must be calculated for each triangle**
  - Rasterization
    - Incremental bilinear update of *(u',v',q)* in screen space
    - Using the partial derivatives of the linear function (i.e. constants)
  - Ray tracing
    - Evaluated at every intersection (via barycentric coordinates)
- **Often (partial) derivatives are needed as well**
  - Explicitly given in matrix (colored for $\partial u/\partial x$, $\partial v/\partial x$, $\partial q/\partial x$)

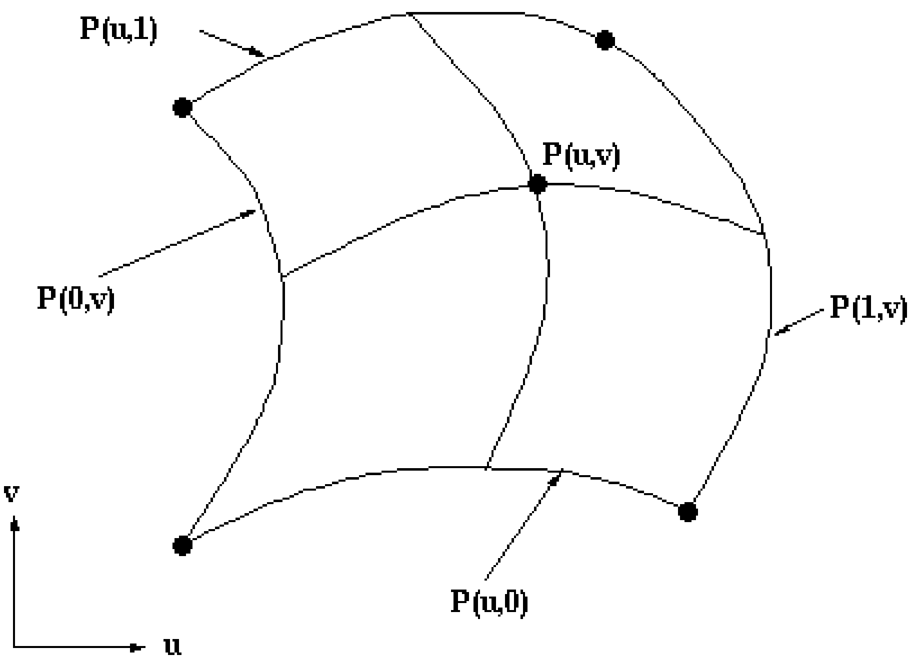
Textures Coordinates

- **Solid Textures**
  - 3D world/object (x,y,z) coords → 3D (u,v,w) texture coordinates
  - Similar to carving object out of material block
- **2D Textures**
  - 3D Cartesian (x,y,z) coordinates → 2D (u,v) texture coordinates?



Parametric Surfaces

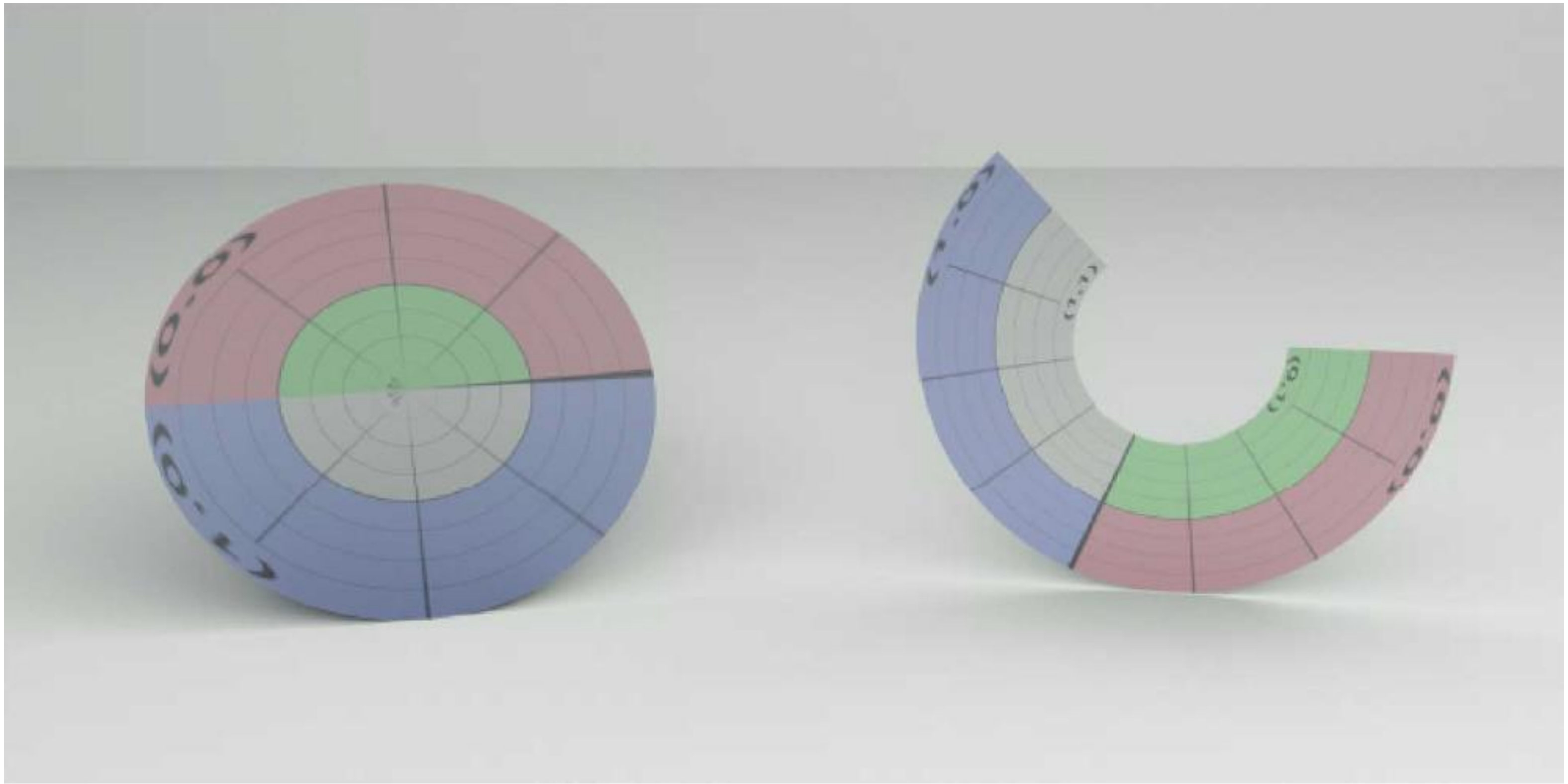
- **Definition (more detail later)**
  - Surface defined by parametric function
    - (x, y, z) = p(u, v)
  - Input
    - Parametric coordinates: (u, v)
  - Output
    - Cartesian coordinates: (x, y, z)



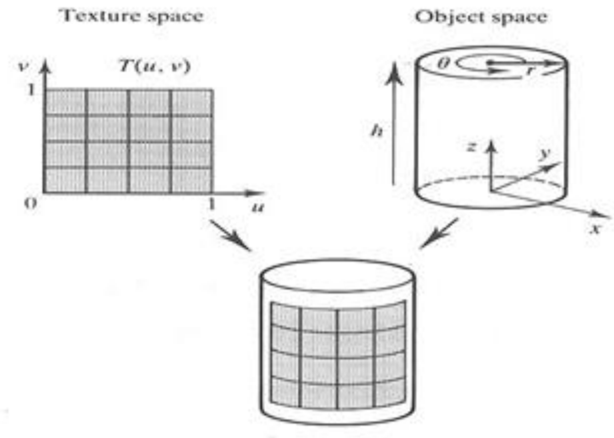
- **Texture Coordinates**
  - Directly derived from surface parameterization
  - Invert parametric function
    - From world coordinates to parametric coordinates
    - Usually computed implicitly anyway (e.g., in ray tracing)

Parametric Surfaces

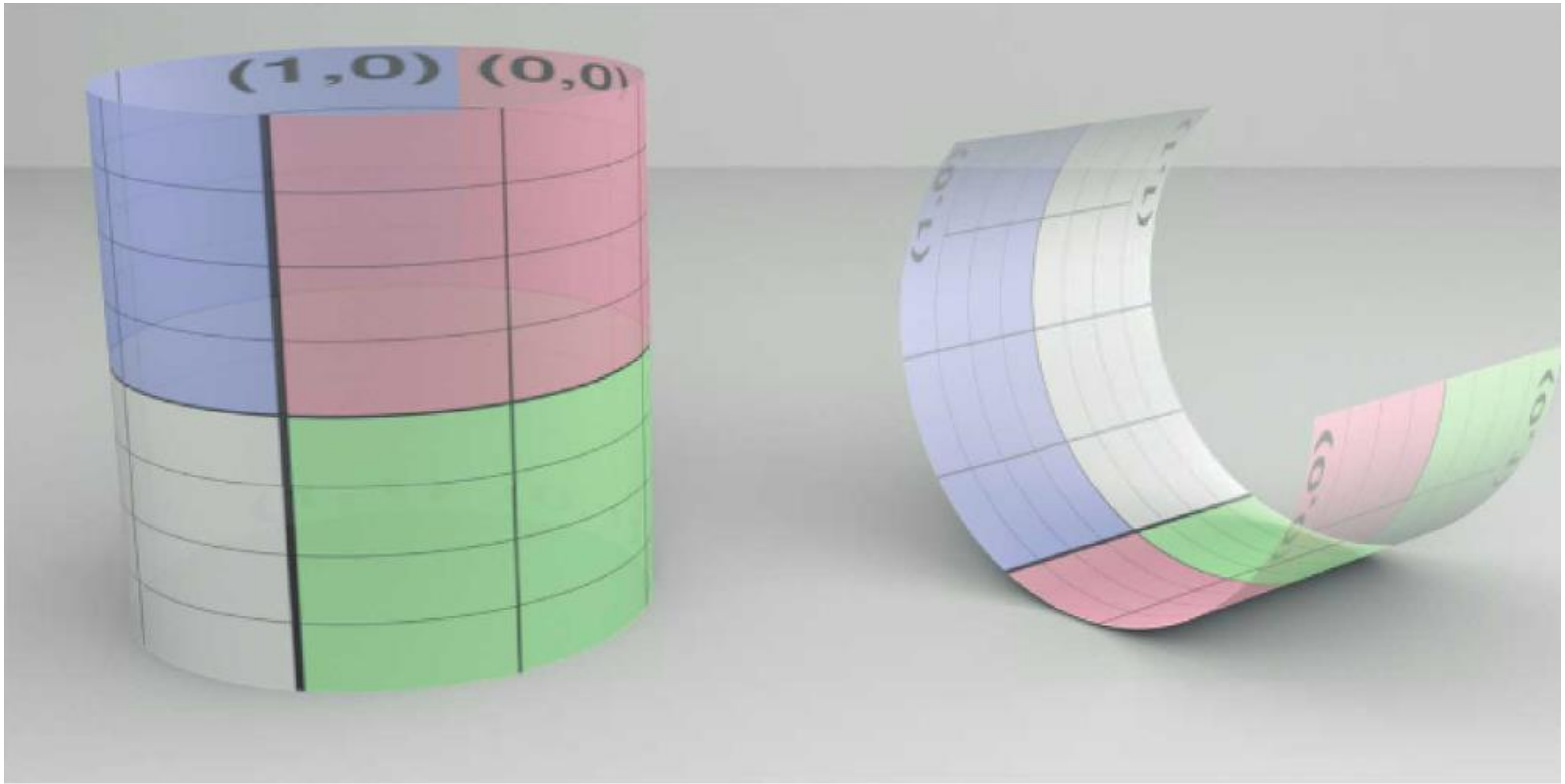
- **Polar Coordinates**
  - (x, y, 0) = Polar2Cartesian(r, φ)
- **Disc**
  - p(u, v) = Polar2Cartesian(R v, 2 π u) // disc radius R



Parametric Surfaces

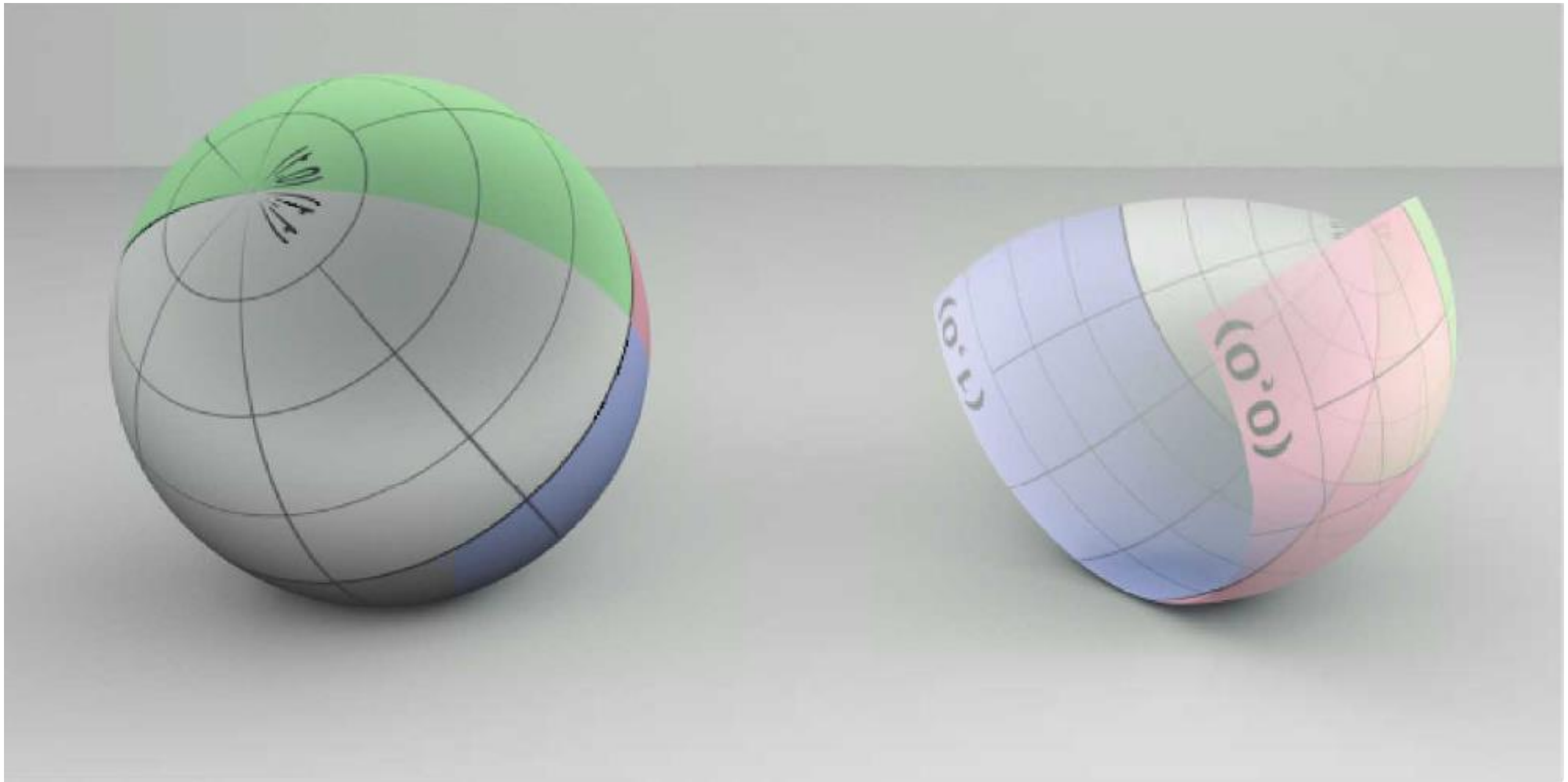


- **Cylindrical Coordinates**
  - (x, y, z) = Cylindrical2Cartesian(r, φ, z)
- **Cylinder**
  - p(u, v) = Cylindrical2Cartesian(r, 2 π u, H v) // cylinder height H



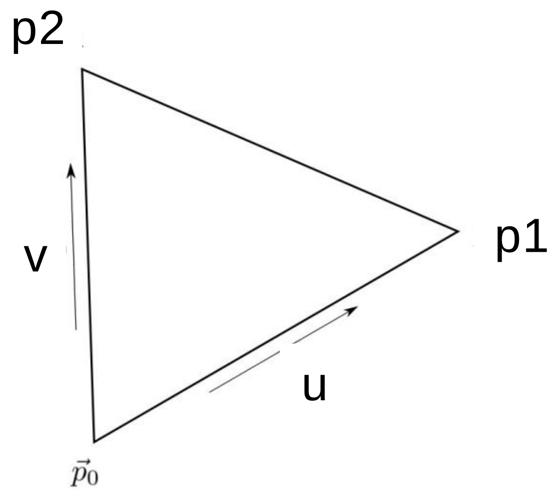
Parametric Surfaces

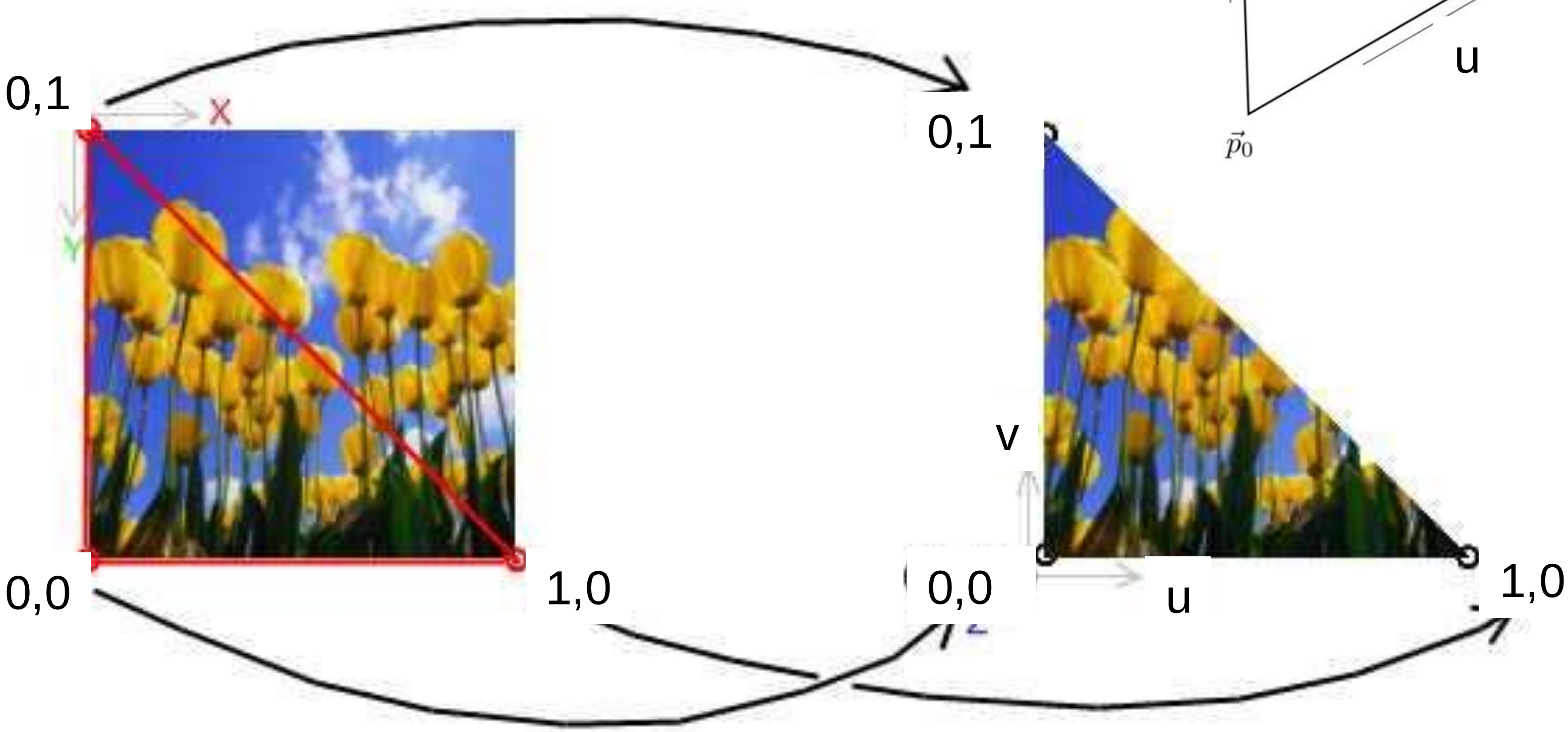
- **Spherical Coordinates**
  - (x, y, z) = Spherical2Cartesian(r, θ, φ)
- **Sphere**
  - p(u, v) = Spherical2Cartesian(r, π v, 2 π u)



Parametric Surfaces

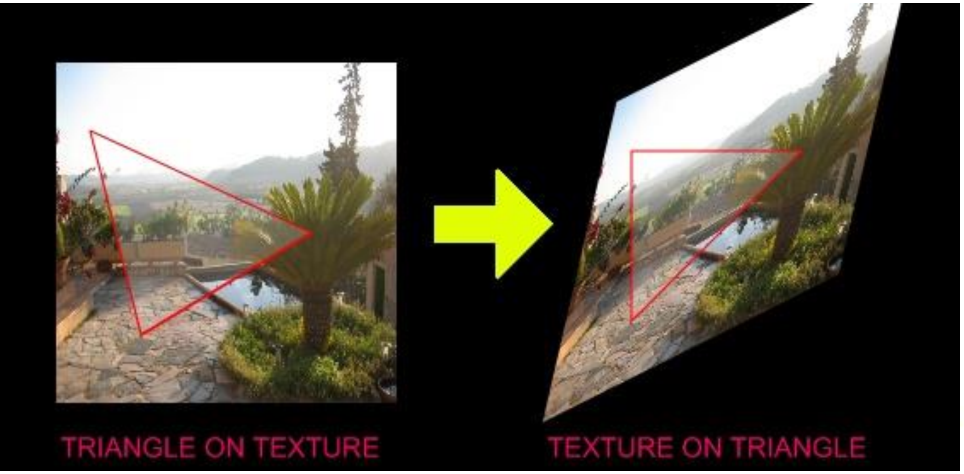
- **Triangle**
  - Use barycentric coordinates directly
  - $p(u,v) = (1 - u - v)p_0 + up_1 + v\,p_2$

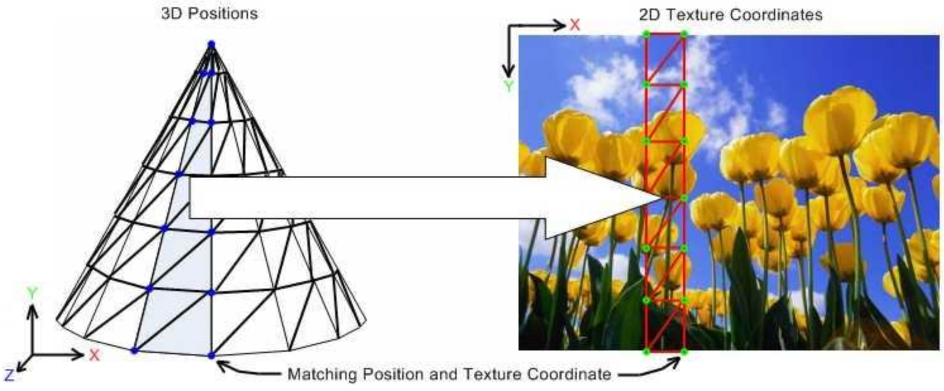




Parametric Surfaces

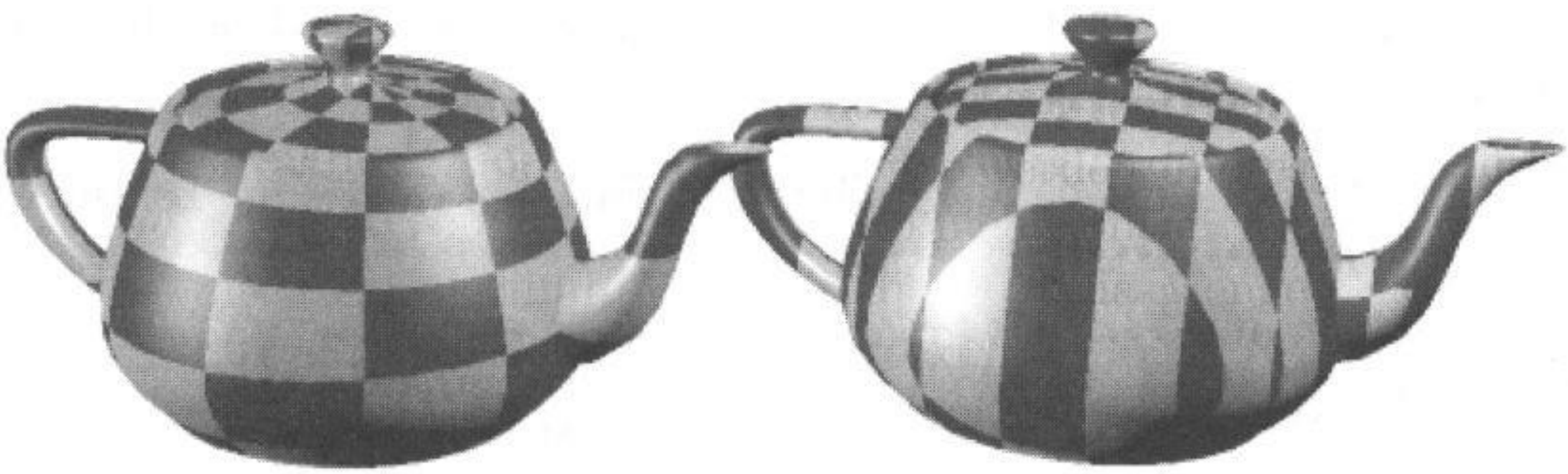
- **Triangle Mesh**
  - Associate a predefined texture coordinate to each triangle vertex
    - Interpolate texture coordinates using barycentric coordinates
    - $u = \lambda_0 p_{0u} + \lambda_1 p_{1u} + \lambda_2 p_{2u}$
    - $v = \lambda_0 p_{0v} + \lambda_1 p_{1v} + \lambda_2 p_{2v}$
  - Texture mapped onto manifold
    - Single texture shared by many triangles





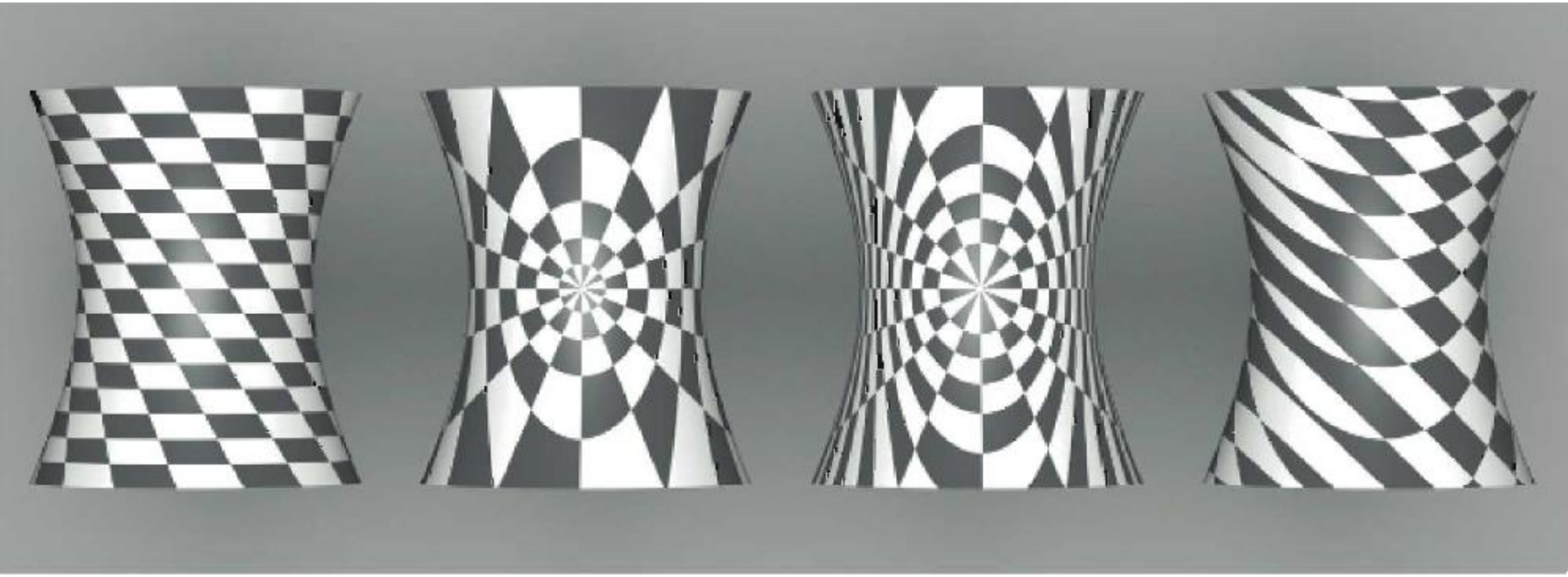
Surface Parameterization

- **Other Surfaces**
  - No intrinsic parameterization??



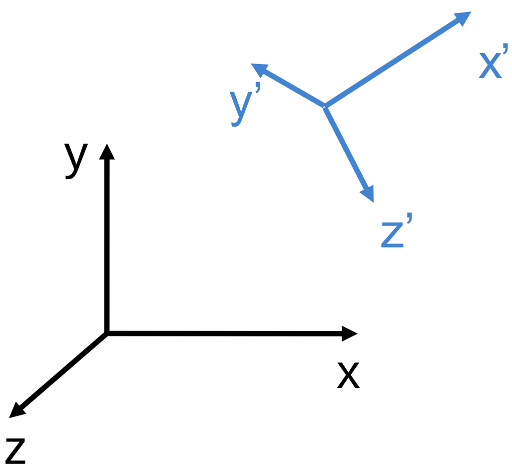
Intermediate Mapping

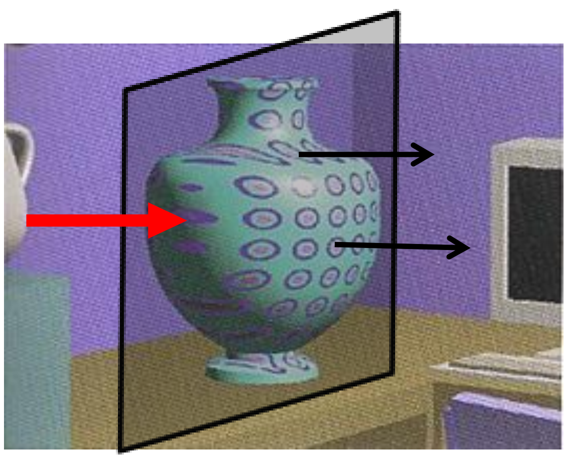
- **Coordinate System Transform**
  - Express Cartesian coordinates into a given coordinate system
- **3D to 2D Projection**
  - Drop one coordinate
  - Compute u and v from remaining 2 coordinates



Intermediate Mapping

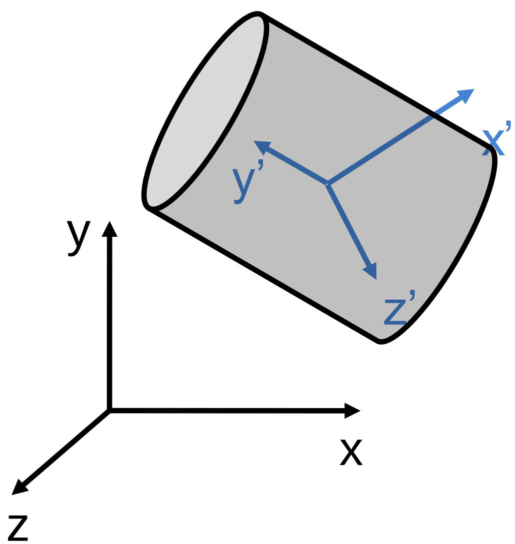
- **Planar Mapping**
  – Map to different Cartesian coordinate system
  – (x’, y’, z’) = AffineTransformation(x, y, z)
    - Orthogonal basis: translation + row-vector rotation matrix
    - Non-orthogonal basis: translation + inverse column-vector matrix
  – Drop z’, map u = x’, map v = y’
  – E.g.: Issues when surface normal orthogonal to projection axis

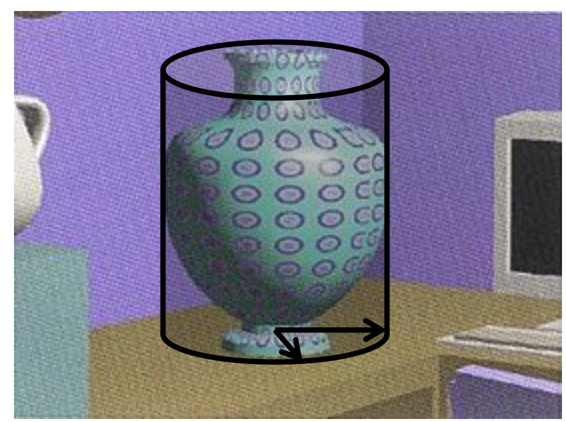




Intermediate Mapping

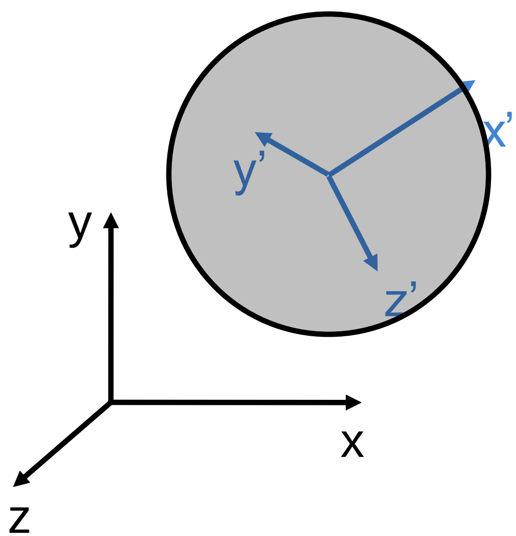
- **Cylindrical Mapping**
  - Map to cylindrical coordinates (possibly after translation/rotation)
  - $(r, \varphi, z) = \text{Cartesian2Cylindrical}(x, y, z)$
  - Drop r, map $u = \varphi / 2\pi$, map $v = z / H$
  - Extension: add scaling factors: $u = \alpha \varphi / 2\pi$
  - E.g.: Similar topology gives reasonable mapping

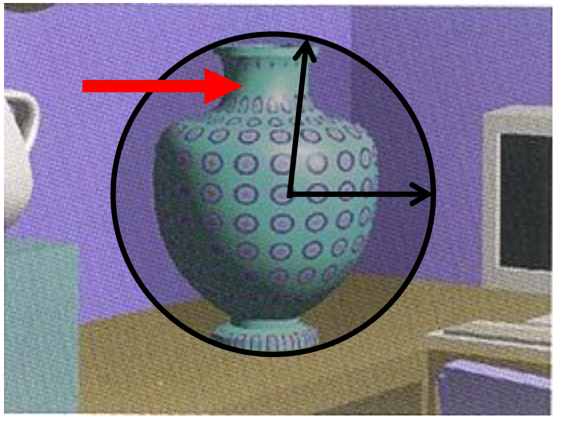




Intermediate Mapping

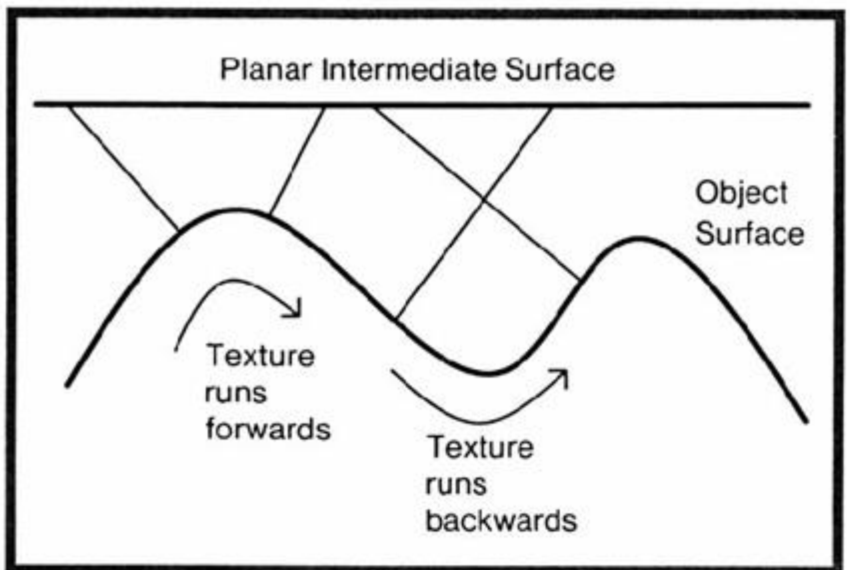
- **Spherical Mapping**
  - Map to spherical coordinates (possibly after translation/rotation)
  - (r, θ, φ) = Cartesian2Spherical(x, y, z)
  - Drop r, map $u = \varphi / 2\pi$, map $v = \theta / \pi$
  - Extension: add scaling factors to both u and v
  - E.g.: Issues in concave regions





Two-Stage Mapping: Problems

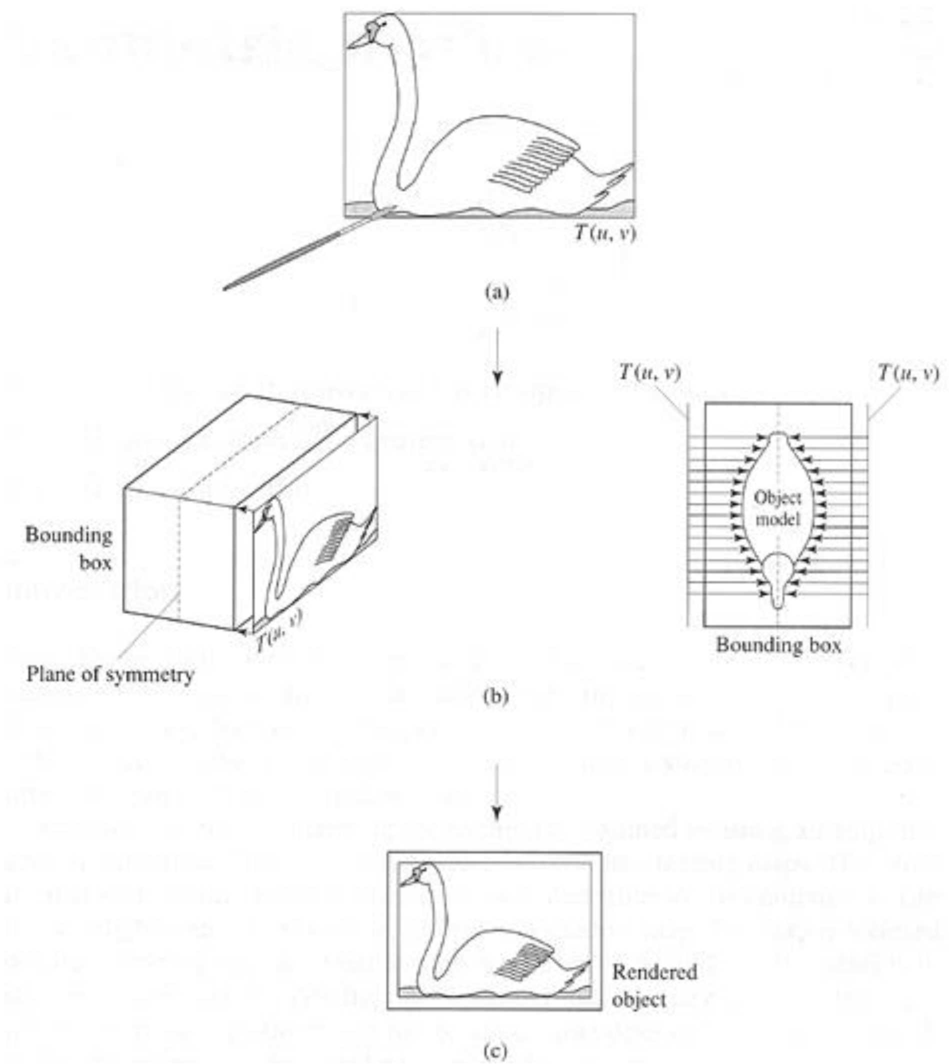
- **Problems**
  - May introduce undesired texture distortions if the intermediate surface differs too much from the destination surface
  - Still often used in practice because of its simplicity



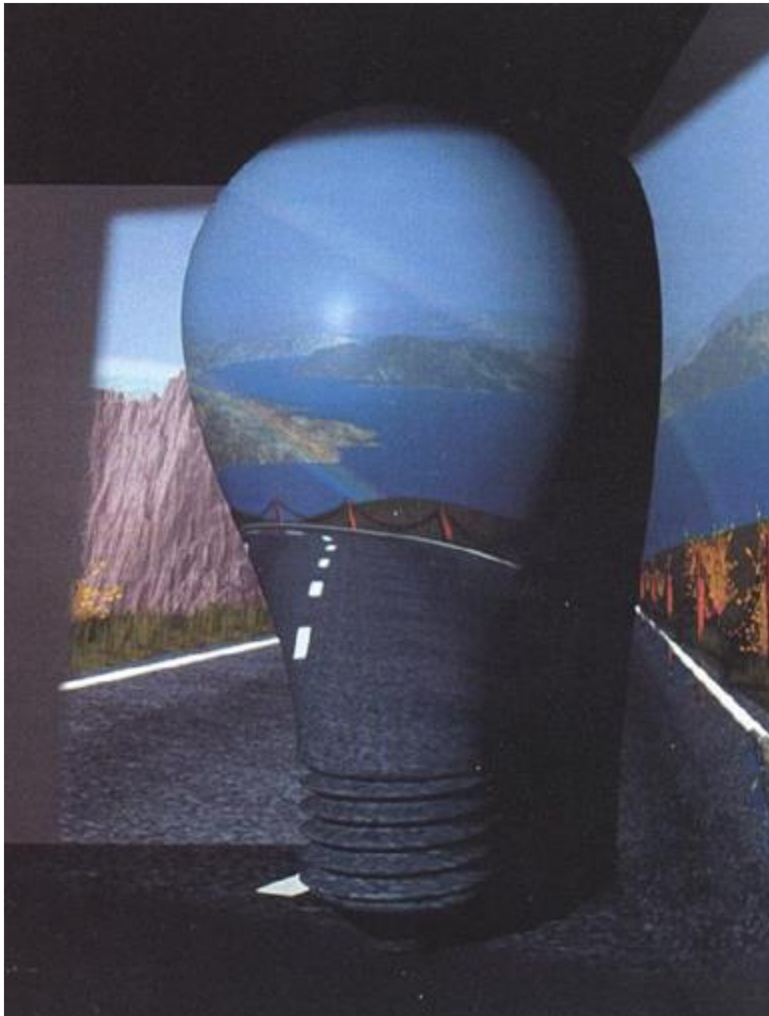
Surface concavities can cause the texture pattern to reverse if the object normal mapping is used.

Projective Textures

- **Project texture onto object surfaces**
  - Slide projector
- **Parallel or perspective projection**
- **Use photographs (or drawings) as textures**
  - Used a lot in film industry!
- **Multiple images**
  - View-dependent texturing (advanced topic)
- **Perspective Mapping**
  - Re-project photo on its 3D environment

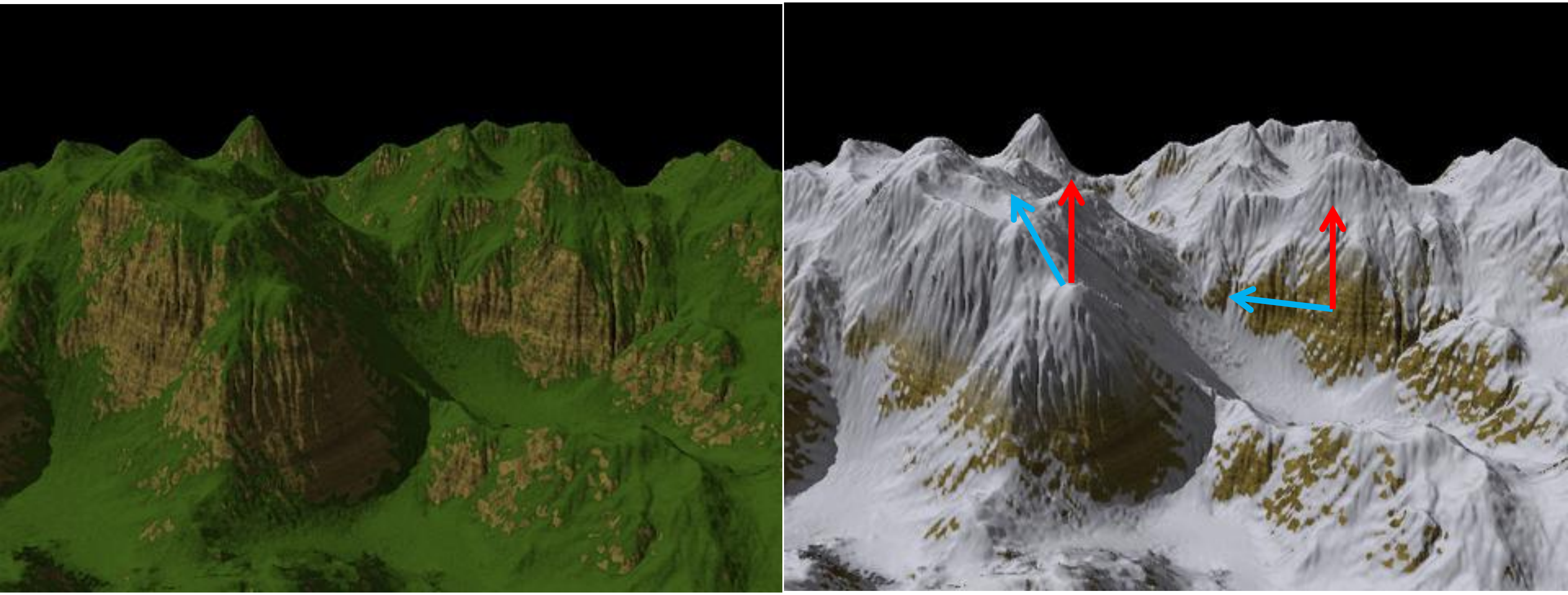


Projective Texturing: Examples



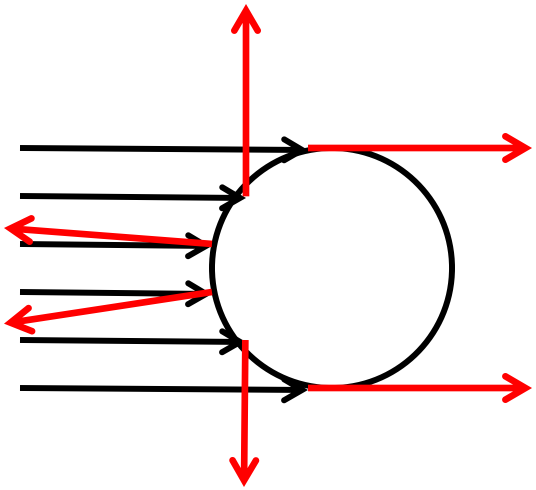
Slope-Based Mapping

- **Definition**
  - Depends on surface normal and predefined vector
- **Example**
  - α = n ω
  - return α flatColor + (1 - α) slopeColor;



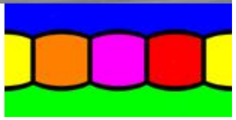
Environment Map

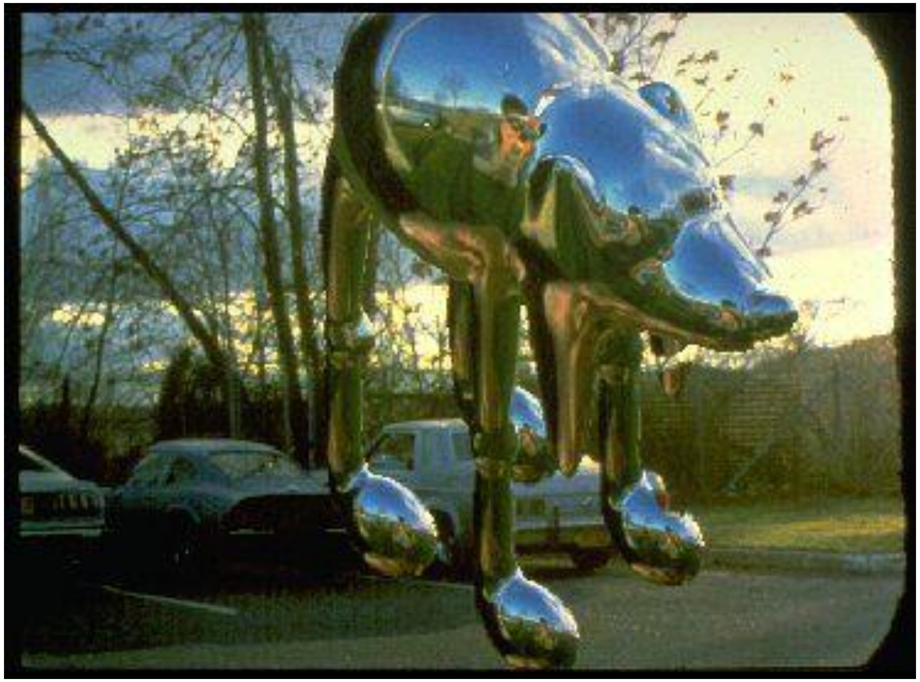
- **Spherical Map**
  - Photo of a reflective sphere (gazing ball)
  - Photos with a fish-eye camera
    - Only gives hemi-sphere mapping



Environment Map

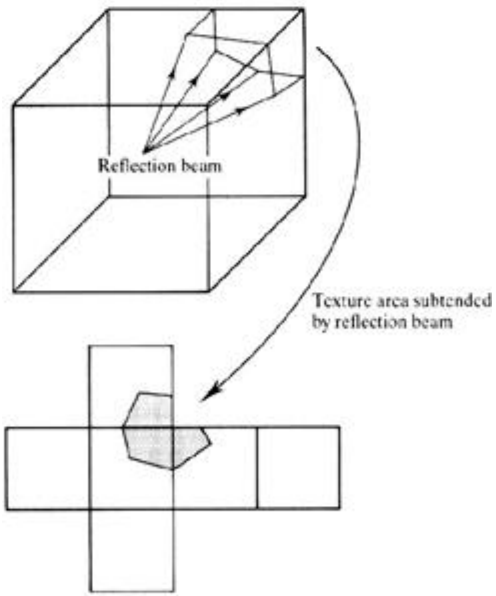
- **Latitude-Longitude Map**
  - Remapping 2 images of reflective sphere
  - Photo with an environment camera
- **Algorithm**
  - If no intersection found, use ray direction to find background color
  - Cartesian coords of ray dir. → spherical coords → uv tex coords

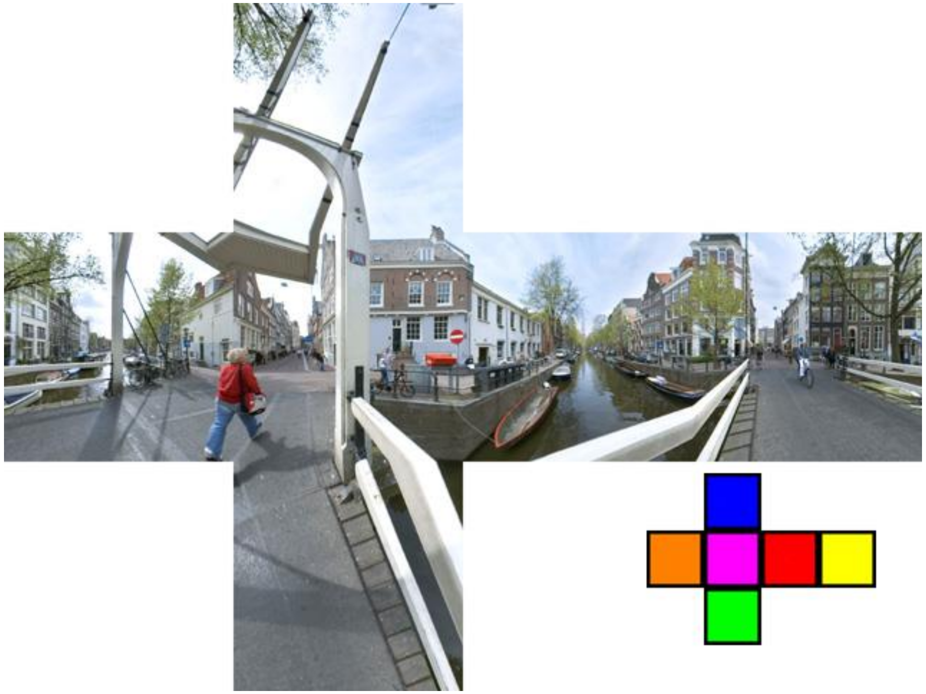




Environment Map

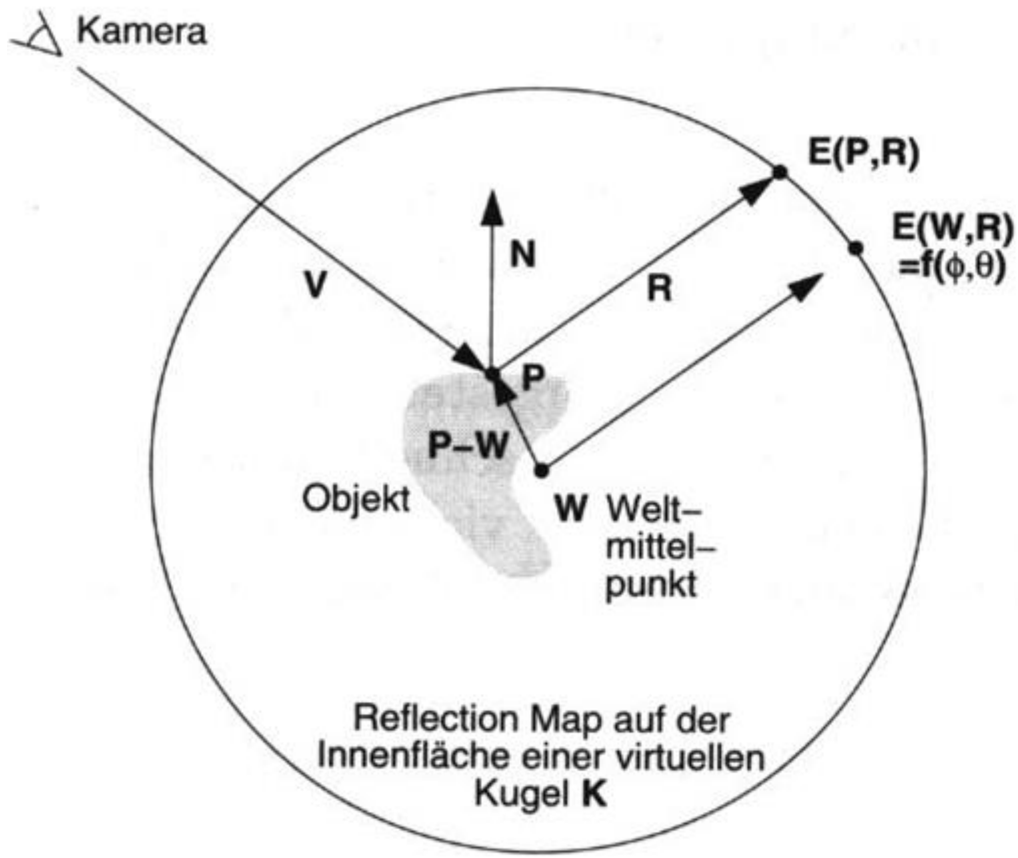
- **Cube Map**
  - Remapping 2 images of reflective sphere
  - Photos with a perspective camera
- **Algorithm**
  - Find main axis (-x, +x, -y, +y, -z, +z) of ray direction
  - Use other 2 coordinates to access corresponding face texture
    - Akin to a 90° projective light





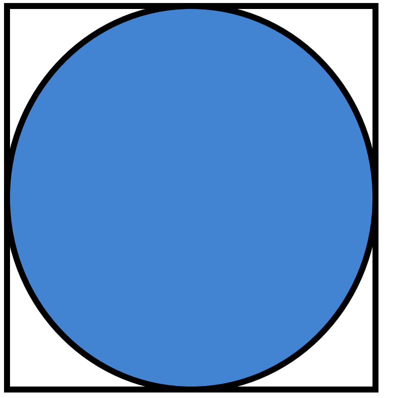
Reflection Map Rendering

- **Spherical parameterization**
- **O-mapping using reflected view ray intersection**

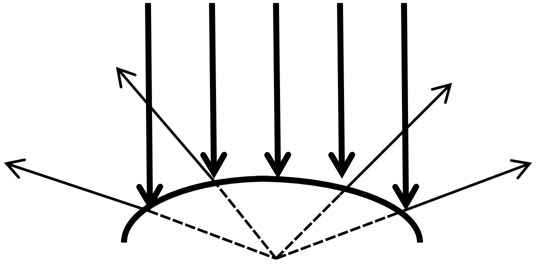


Reflection Map Parameterization

- **Spherical mapping**
  - Single image
  - Bad utilization of the image area
  - Bad scanning on the edge
  - Artifacts, if map and image do not have the same viewpoint



- **Double parabolic mapping**
  - Yields spherical parameterization
  - Subdivide in 2 images (front-facing and back-facing sides)
  - Less bias near the periphery
  - Arbitrarily reusable
  - Supported by OpenGL extensions



Reflection Mapping Example

Terminator II motion picture

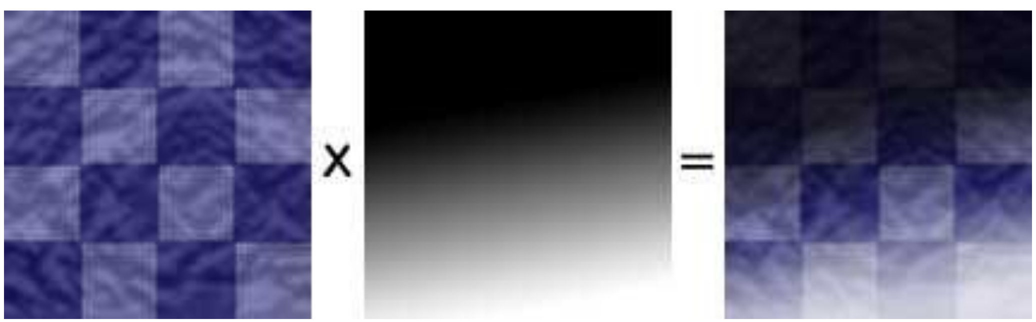
Reflection Mapping Example II

- **Reflection mapping with Phong reflection**
  - Two maps: diffuse & specular
  - Diffuse: index by surface normal
  - Specular: indexed by reflected view vector

RenderMan
Companion

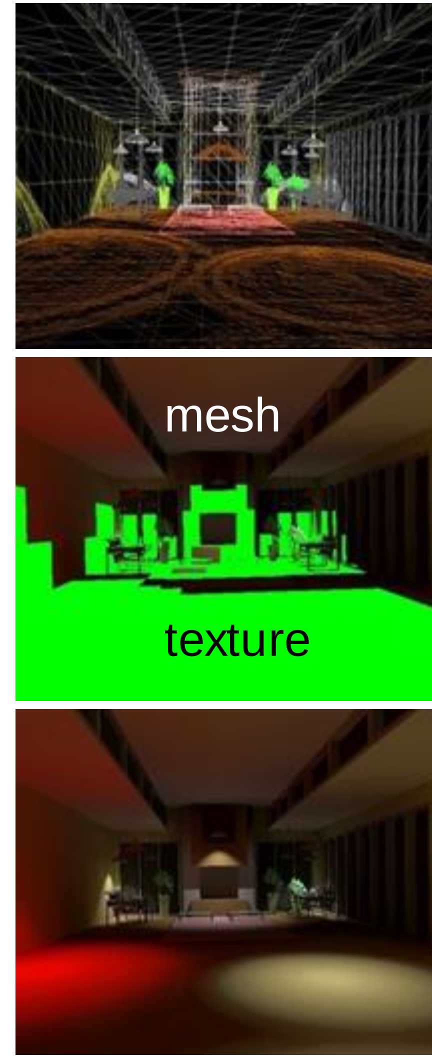
Light Maps

- **Light maps (e.g., in Quake)**
  - Pre-calculated illumination (local irradiance)
    - Often very low resolution: smoothly varying
  - Multiplication of irradiance with base texture
    - Diffuse reflectance only
  - Provides surface radiosity
    - View-independent out-going radiance
  - Animated light maps
    - Animated shadows, moving light spots, etc…



Reflectance Irradiance Radiosity

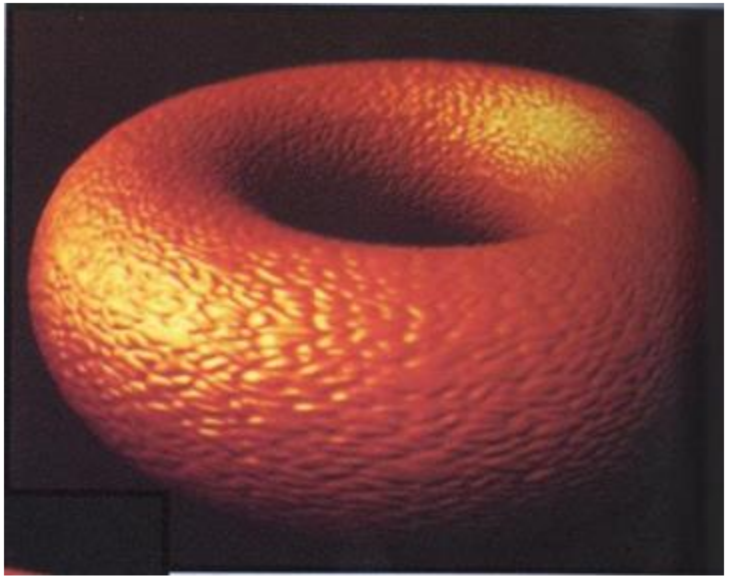
$$B(x) = \rho(x)\, E(x) = \pi L_o(x)$$



Representing radiosity in a mesh or texture

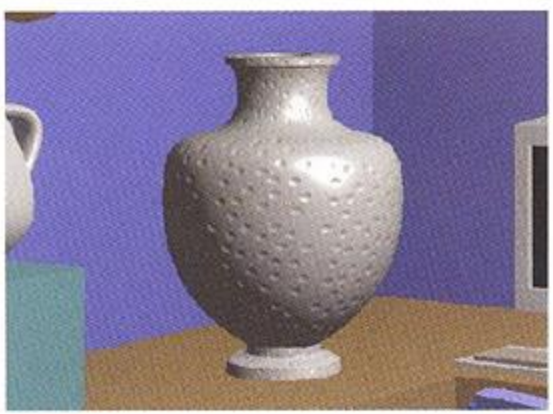
Bump Mapping

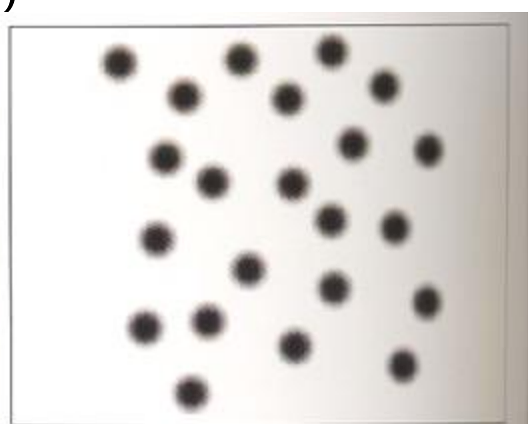
- **Modulation of the normal vector**
  - Surface normals changed only
    - Influences shading only
    - No self-shadowing, contour is **not** altered



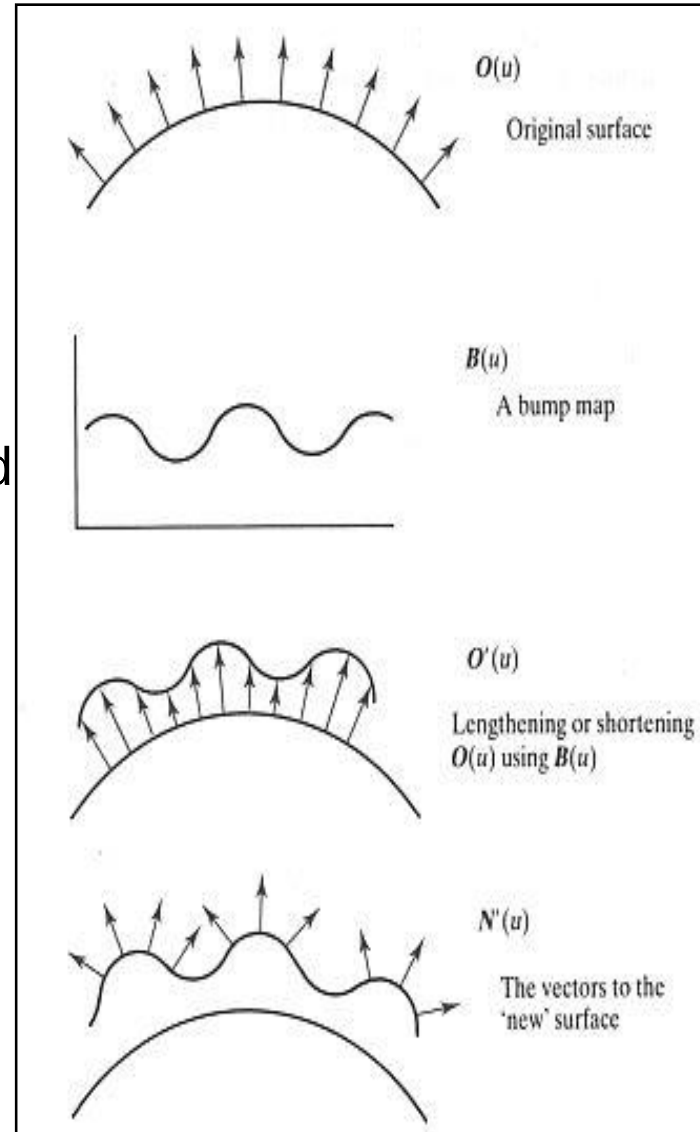
Bump Mapping

- **Original surface:** $O(u,v)$
  - Surface normals are known
- **Bump map:** $B(u,v) \in \mathbb{R}$
  - Surface is offset in normal direction according to bump map intensity
  - New normal directions $N'(u,v)$ are calculated based on virtually displaced surface $O'(u,v)$
  - Original surface is rendered with new normals $N'(u,v)$





Grey-valued texture used for bump height



Bump Mapping

- **Displaced surface:**

$$O'(u,v) = O(u,v) + B(u,v)\,N(u,v)$$

- **Computing the normal:**
  - Normal is cross-product of derivatives:

$$N'(u,v) = O'_u \times O'_v$$

  - Where:

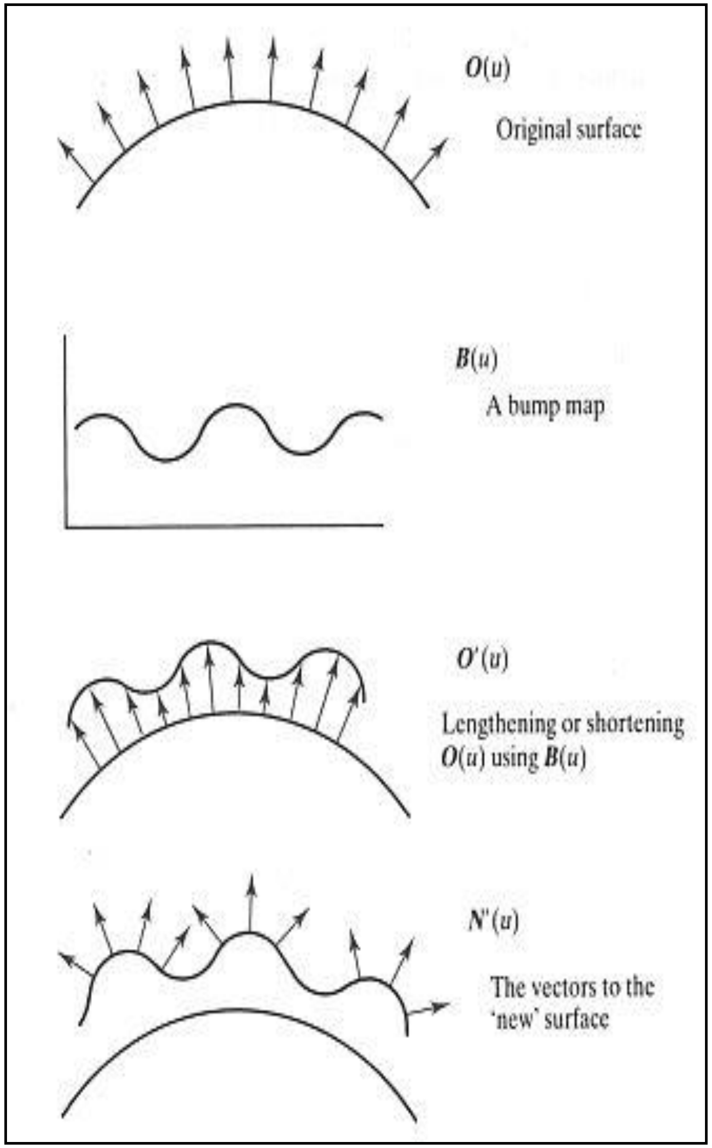
$$O'_u = O_u + B_u N + B N_u$$
$$O'_v = O_v + B_v N + B N_v$$

  - If *B* is small the last term in each equation can be ignored, yielding:

$$N'(u,v) = O_u \times O_v + B_u(N \times O_v) + B_v(O_u \times N) + B_u B_v (N \times N)$$

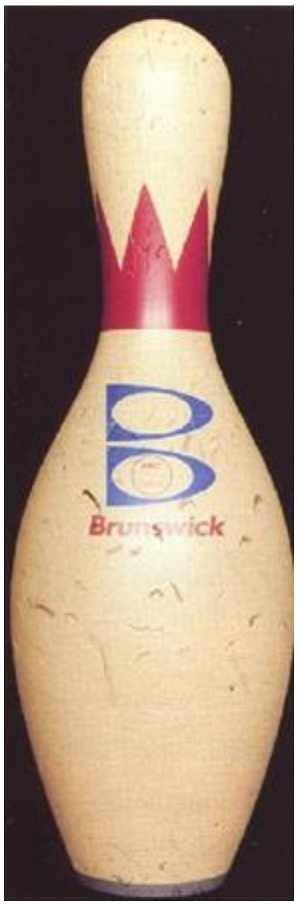
  - The first term is the normal to the surface and the last is zero, giving:

$$D = B_u(N \times O_v) - B_v(N \times O_u)$$
$$N' = N + D$$



Texture Examples

- **Complex optical effects**
  - Combination of multiple texture effects



RenderMan Companion

Billboards

- **Single textured polygons**
  - Often with opacity texture
  - Rotates, always facing viewer
  - Used for rendering distant objects
  - Best results if approximately radially or spherically symmetric
- **Multiple textured polygons**
  - Azimuthal orientation: different view-points
  - Complex distribution: trunk, branches, …