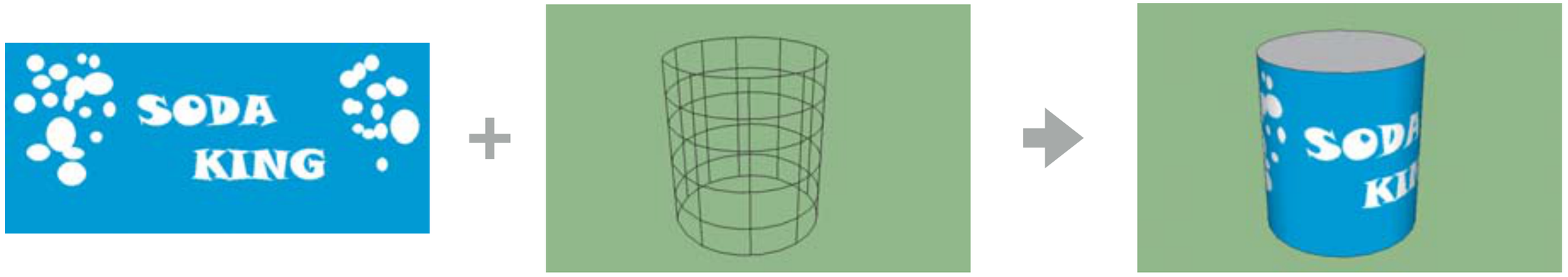
Source: http://www.bbb3viz.com

Texture Mapping

Introduction to Computer Graphics
CSE 533/333

Texture

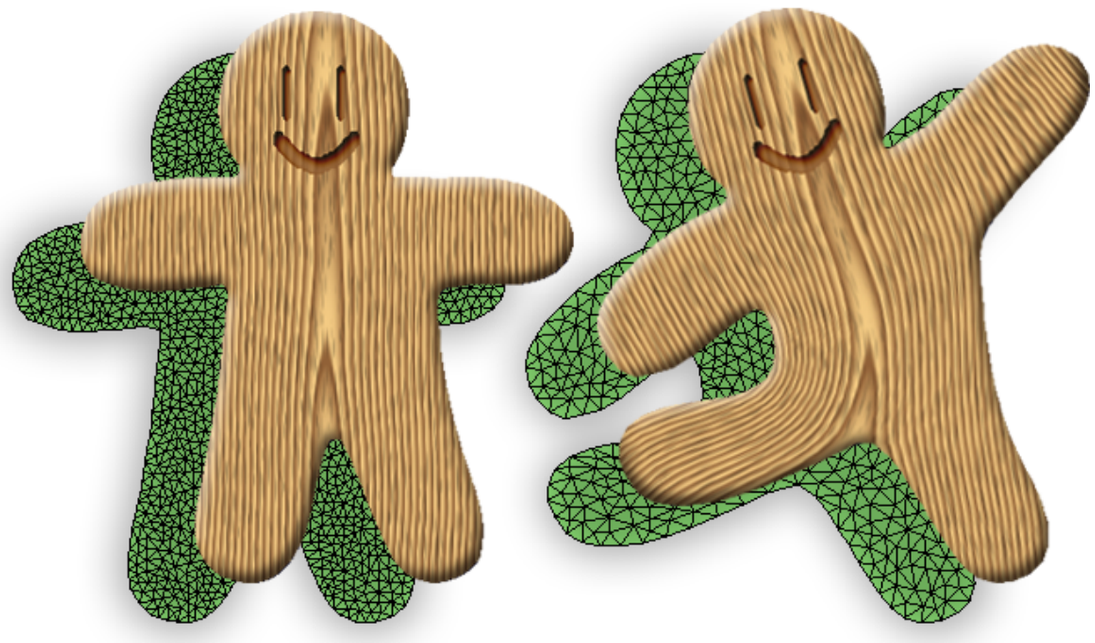
- Objects have variation in reflectance across the surface
- A common technique to handle such variations is to store the reflectance as a function or a pixel-based image and “map” it onto a surface



Source: Kefei Lei

Texture

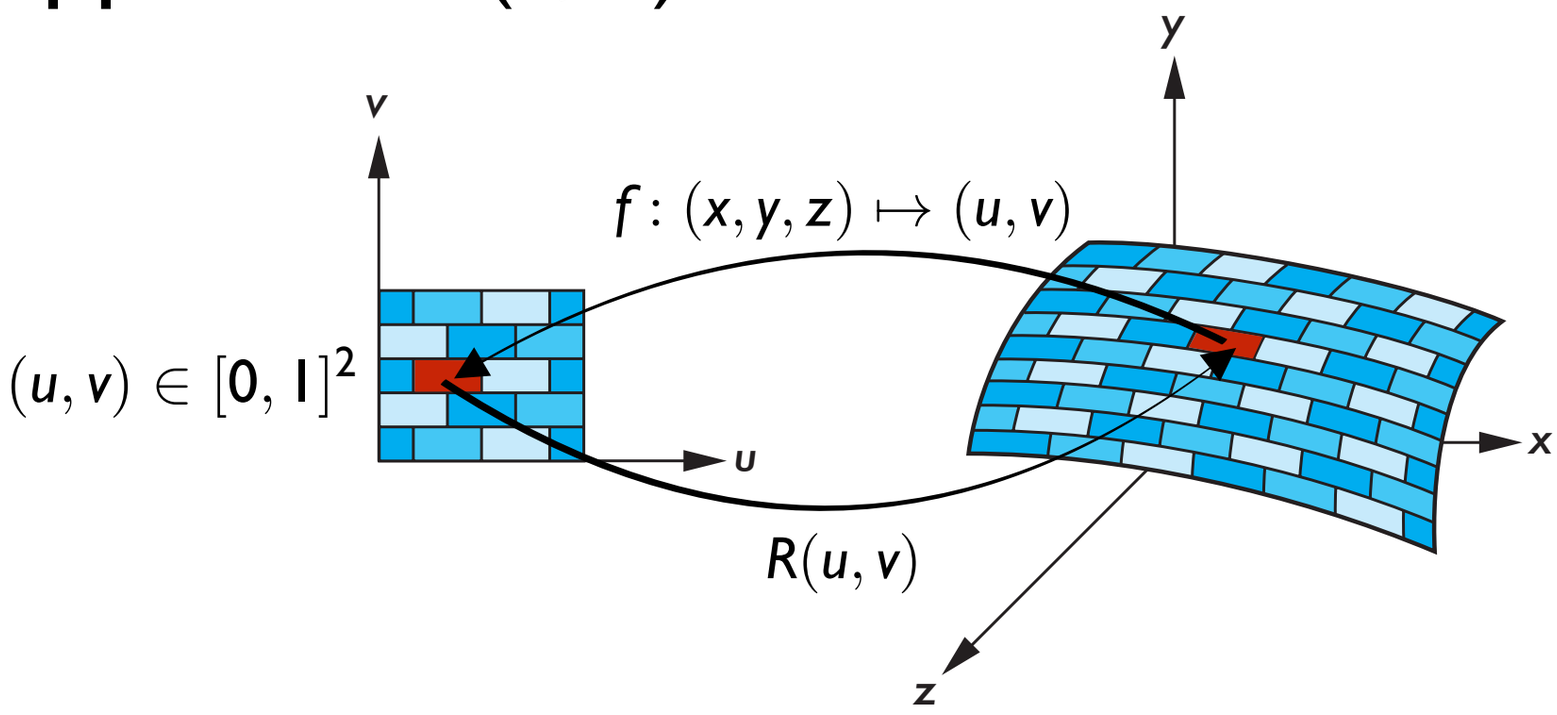
- The function or image is called *Texture*
- The process of controlling reflectance properties is called *texture mapping*



Source: Alec Jacobson, http://alecjacobson.com

2D Texture Mapping

- For 2D texture mapping, a local coordinate system called *uv* is used to create reflectance $R(u, v)$
- During texture mapping, each surface point (x, y, z) is mapped to a (u, v) coordinate to color it with $R(u, v)$

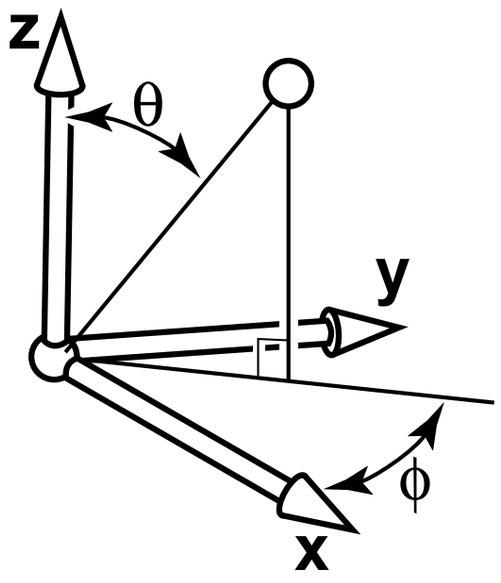


Texture Mapping on Sphere

Example

- To map (u, v) onto a sphere, we first compute spherical coordinates

- For a sphere with radius R and centre (x_c, y_c, z_c)



$$x = x_c + R\cos\phi\sin\theta$$

$$y = y_c + R\sin\phi\sin\theta$$

$$z = z_c + R\cos\theta$$

$$\theta = \arccos\left(\frac{z - z_c}{R}\right), \qquad \theta \in [0, \pi]$$

$$\phi = \arctan2\,(y - y_c, x - x_c) \qquad \phi \in [-\pi, \pi]$$

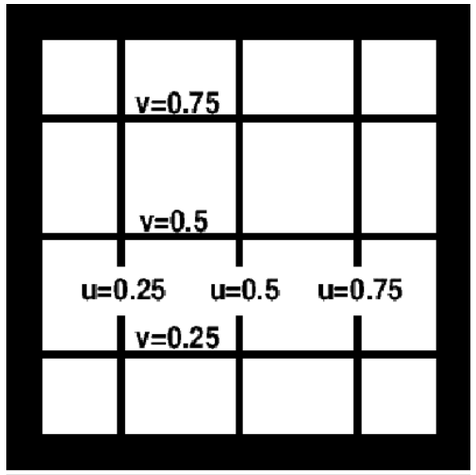
Texture Mapping on Sphere

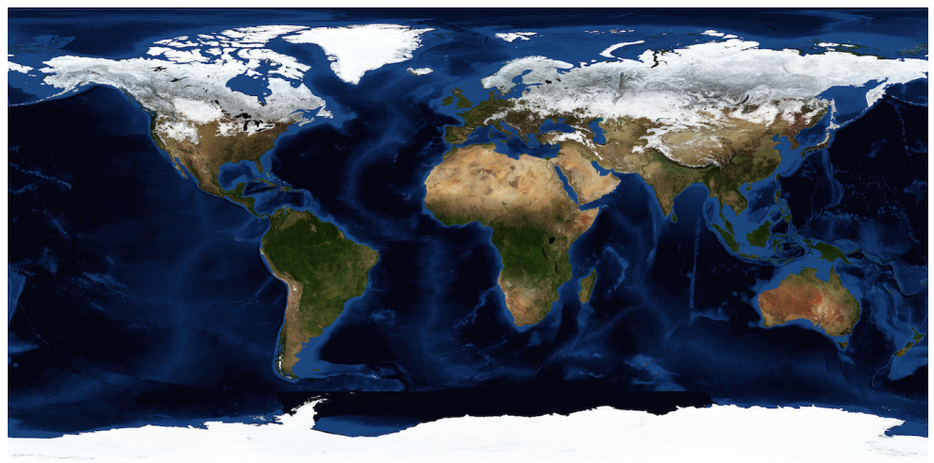
Example

Required mapping:

$$u = \frac{\phi}{2\pi},$$

$$v = \frac{\pi - \theta}{\pi}$$





Texture Mapping on Triangle

Example

- For surfaces represented by triangle meshes, texture coordinates are defined by storing (u, v) coordinates at each vertex

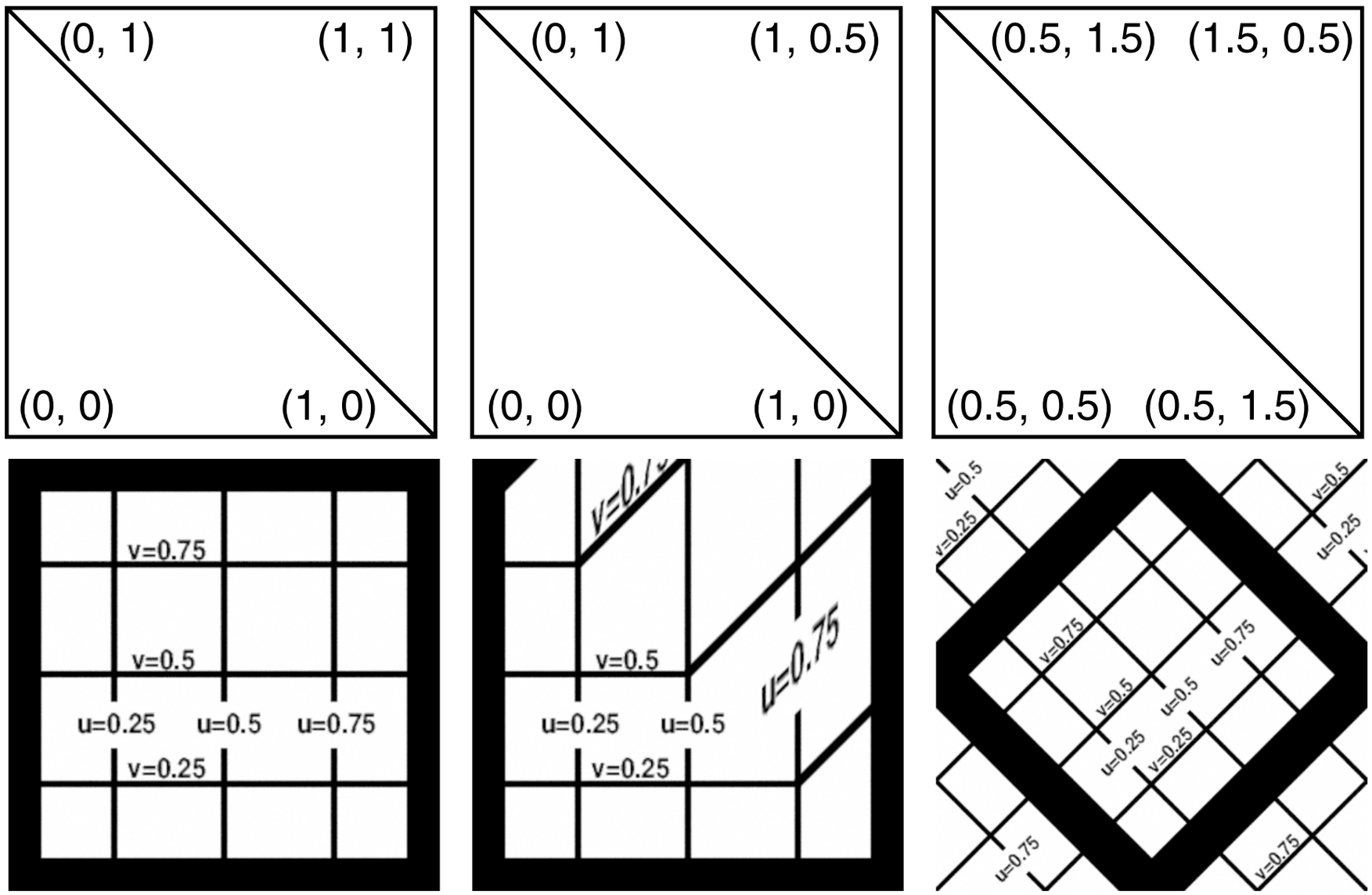
- At any point (β, γ) inside a triangle, the (u, v) coordinate is interpolated as

$$u(\beta, \gamma) = (1 - \beta - \gamma)u_b + \beta u_a + \gamma u_c$$
$$v(\beta, \gamma) = (1 - \beta - \gamma)v_b + \beta v_a + \gamma v_c$$

where a, b, and c are vertices of the triangle

Texture Mapping on Triangle

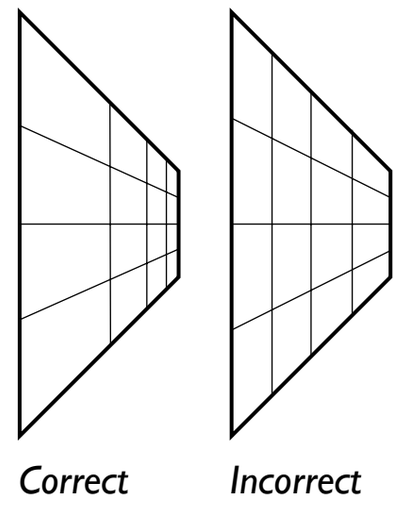
Example



Perspective-correct Textures

- Accounts for the actual 3D positions of vertices
- Screen space interpolation will result in distortions

$$f_x = (1-\alpha)\frac{x_0}{w_0} + \alpha\frac{x_1}{w_1}$$



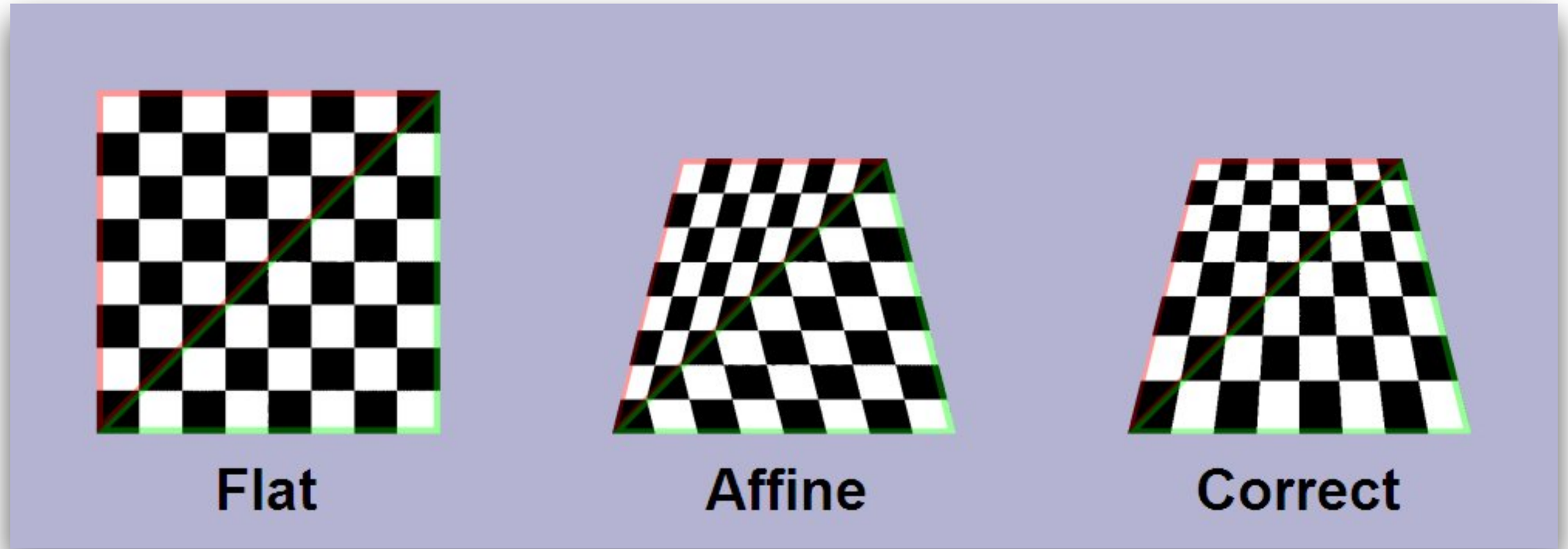
- Perspective correct interpolation:

$$f_x = \frac{(1-\beta)x_0 + \beta x_1}{(1-\beta)w_0 + \beta w_1}$$

- Post perspective, use:

$$\alpha = \frac{\beta w_1}{\beta w_1 + (1-\beta)w_0}$$

Perspective-correct Textures



Source: Wikipedia

Texture Coordinates

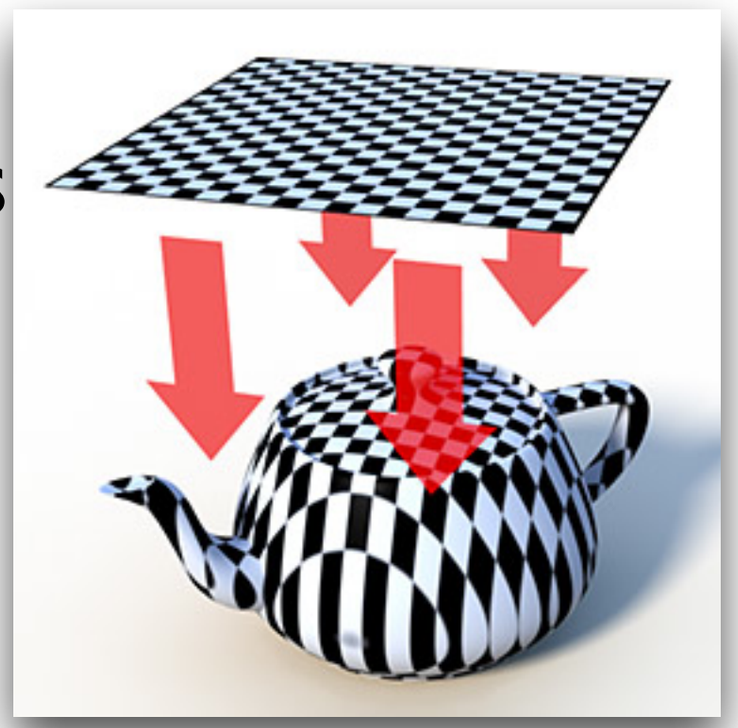
- How do we specify the mapping from an object to texture space?
- We want the following properties:
  - **Piecewise linear** (in order to use interpolation hardware)
  - **Invertible** (go back-forth b/w texture space and surface)
  - **Easy to compute**

Common Texture Mappings

- Linear/planar/hyperplanar projection
- Cylindrical
- Spherical
- Piecewise-linear or piecewise-bilinear on a plane from explicit per-vertex texture coordinates (UV)
- Normal-vector projection onto a sphere

Planar Texturing

- Similar to the notion of a movie projector
- Associated image is projected orthographically onto the surface
- Great for flat or nearly flat surfaces

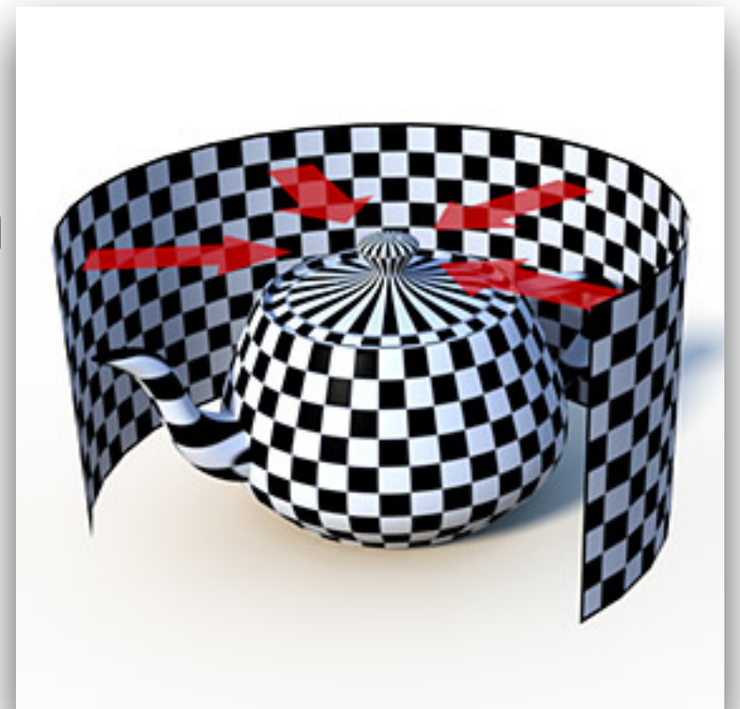


Source: Modo; Luxology, LLC

Cylindrical Texturing

- The object is surrounded by a large cylinder
- Project from the axis through the object to cylinder
- Great for objects that have some central axis
- If point (x, y, z) projects to (r, θ, z) in cylindrical coordinates, we assign (θ, z) texture coordinates

$$u = \frac{\theta}{2\pi}, v = \text{clamp}\left(\frac{z}{z_{max}}, -1, +1\right)$$



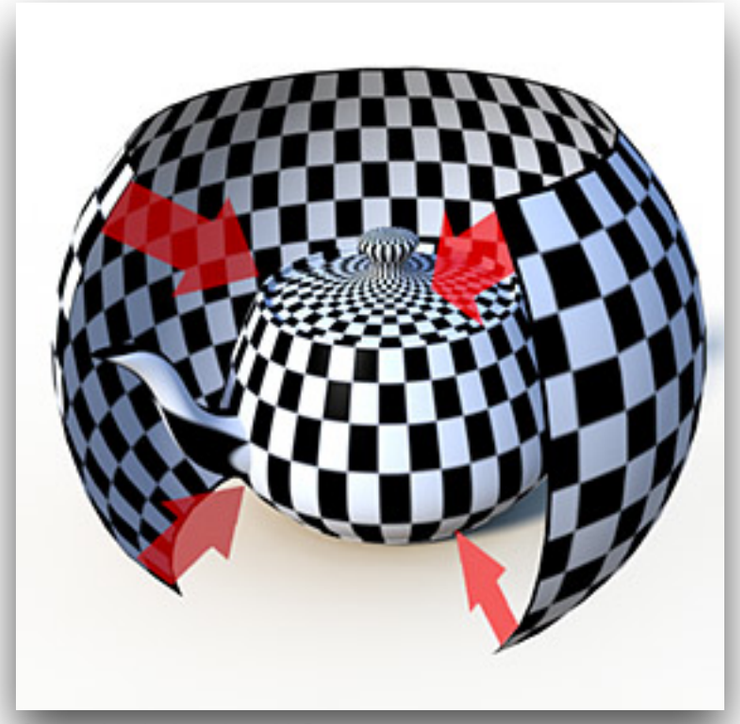
Source: Modo; Luxology, LLC

$$\text{clamp}(x, a, b) = \begin{cases} x & \text{if } a \le x \le b \\ a & \text{if } x < a \\ b & \text{if } x > b. \end{cases}$$

Spherical Texturing

- The object is surrounded by a large sphere
- Project from the centre to through the object to sphere
- Great for blobby objects
- If point (x, y, z) projects to (r, θ, ϕ) in spherical coordinates, we assign (θ, ϕ) texture coordinates

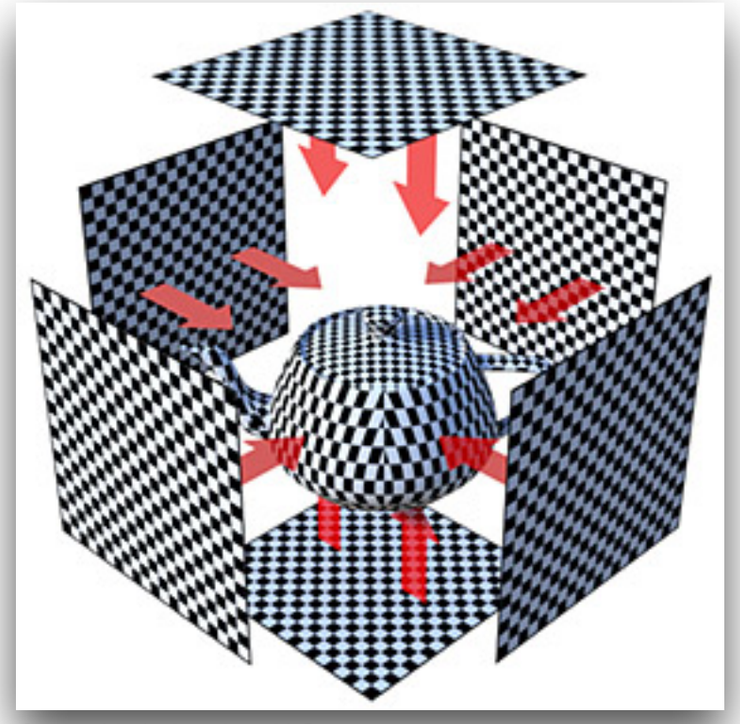
$$u = \frac{\phi}{2\pi}, v = \frac{\pi - \theta}{\pi}$$



Source: Modo; Luxology, LLC

Box Texturing

- The texture is *planar* projected through a surface from all six directions (like an inward facing cube)
- Great for cube shaped objects

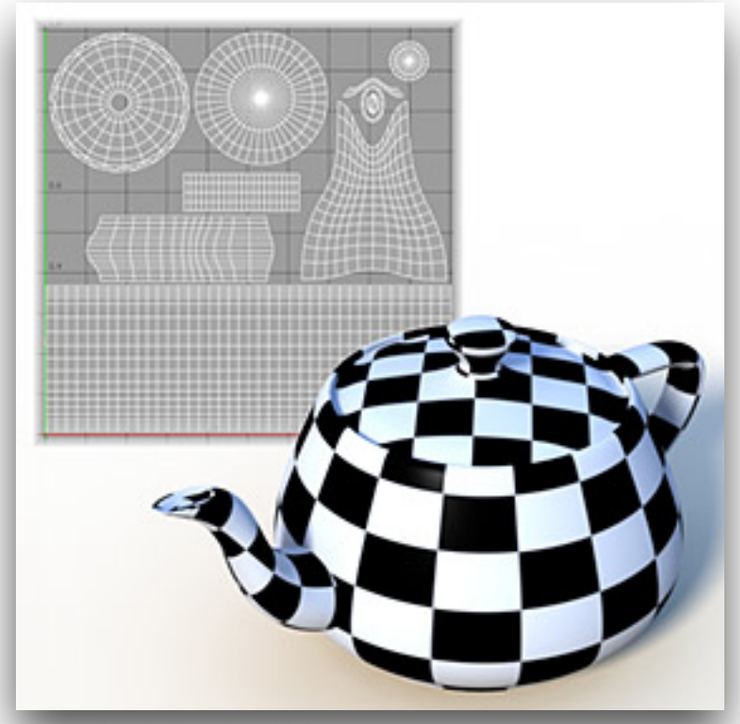


Source: Modo; Luxology, LLC

UV Texturing

- Piecewise-linear or piecewise-bilinear on a plane from explicit per-vertex texture coordinates (UV)
- Most common, but needs texture coord. assignment

- At least four ways:
  - Per vertex assignment
  - Texture parameterization
  - Breaking surface into patches
  - Have an artist "paint" the texture

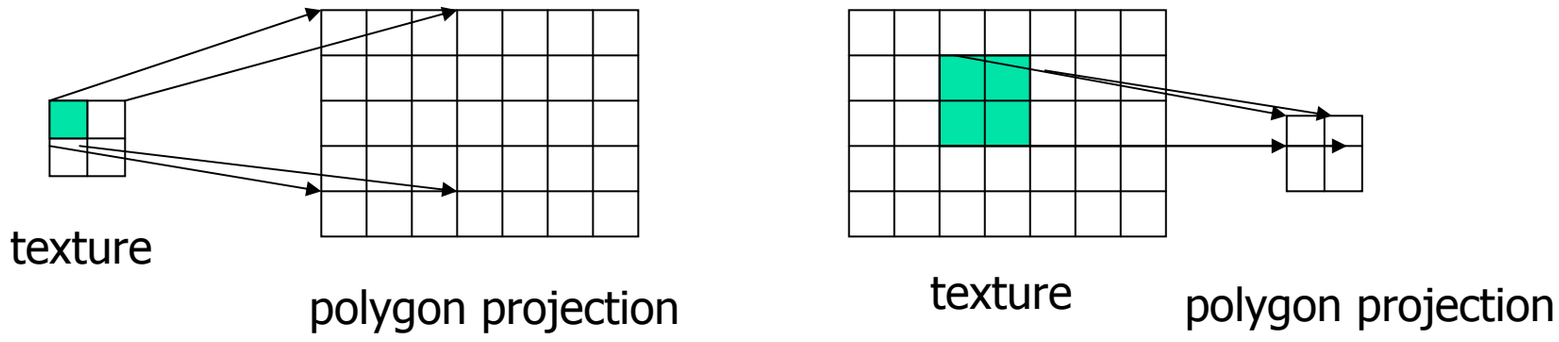


Source: Modo; Luxology, LLC

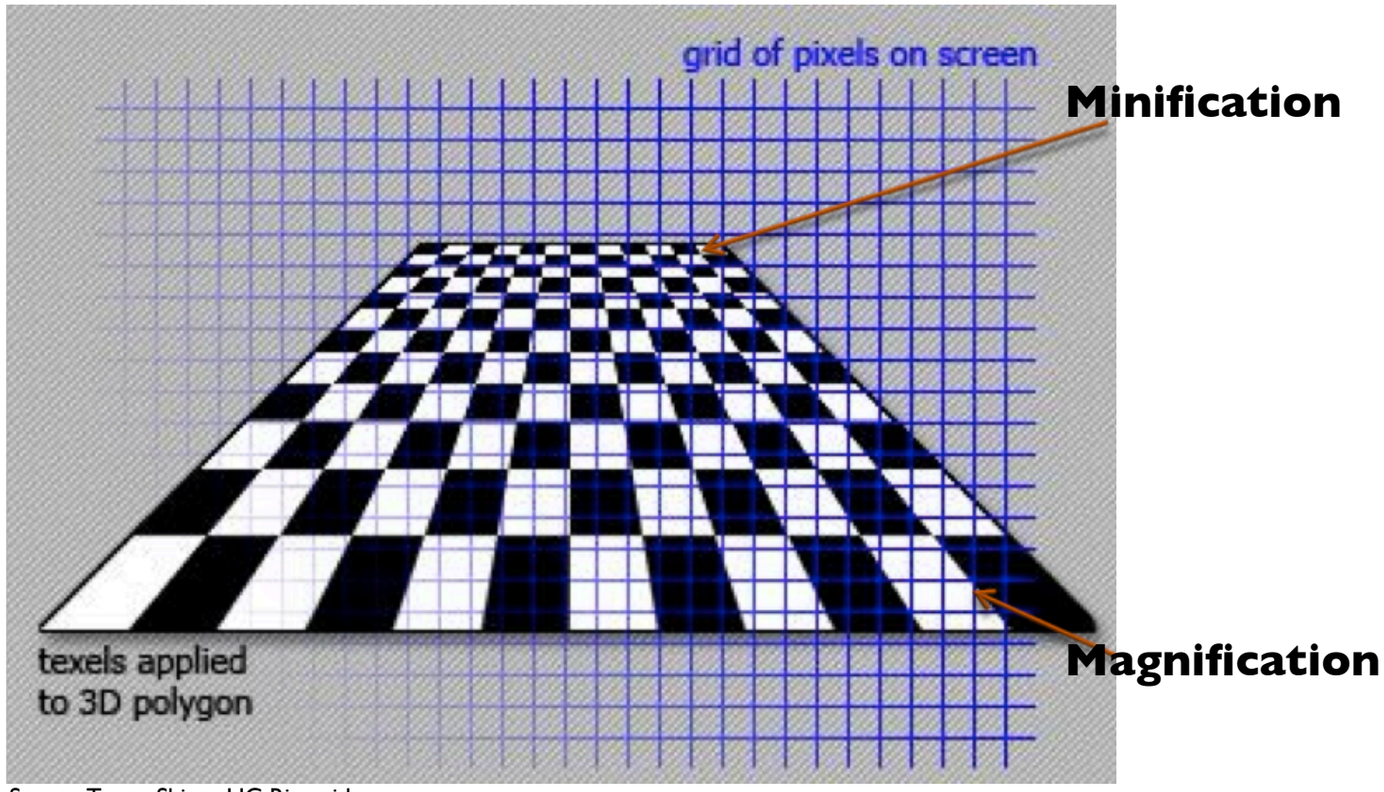
Texture Filtering

- During texture mapping, one pixel on a textured surface may not correspond to one *texel*
- Insufficient or incorrect filtering shows up as blockiness/jaggies
- Different correspondence between texel & pixel:
  - Each texel maps onto more than one pixel (*magnification*)
  - Each texel maps exactly onto one pixel
  - Each texel maps onto less than one pixel (*minification*)

Texture Filtering



Magnification *Minification*



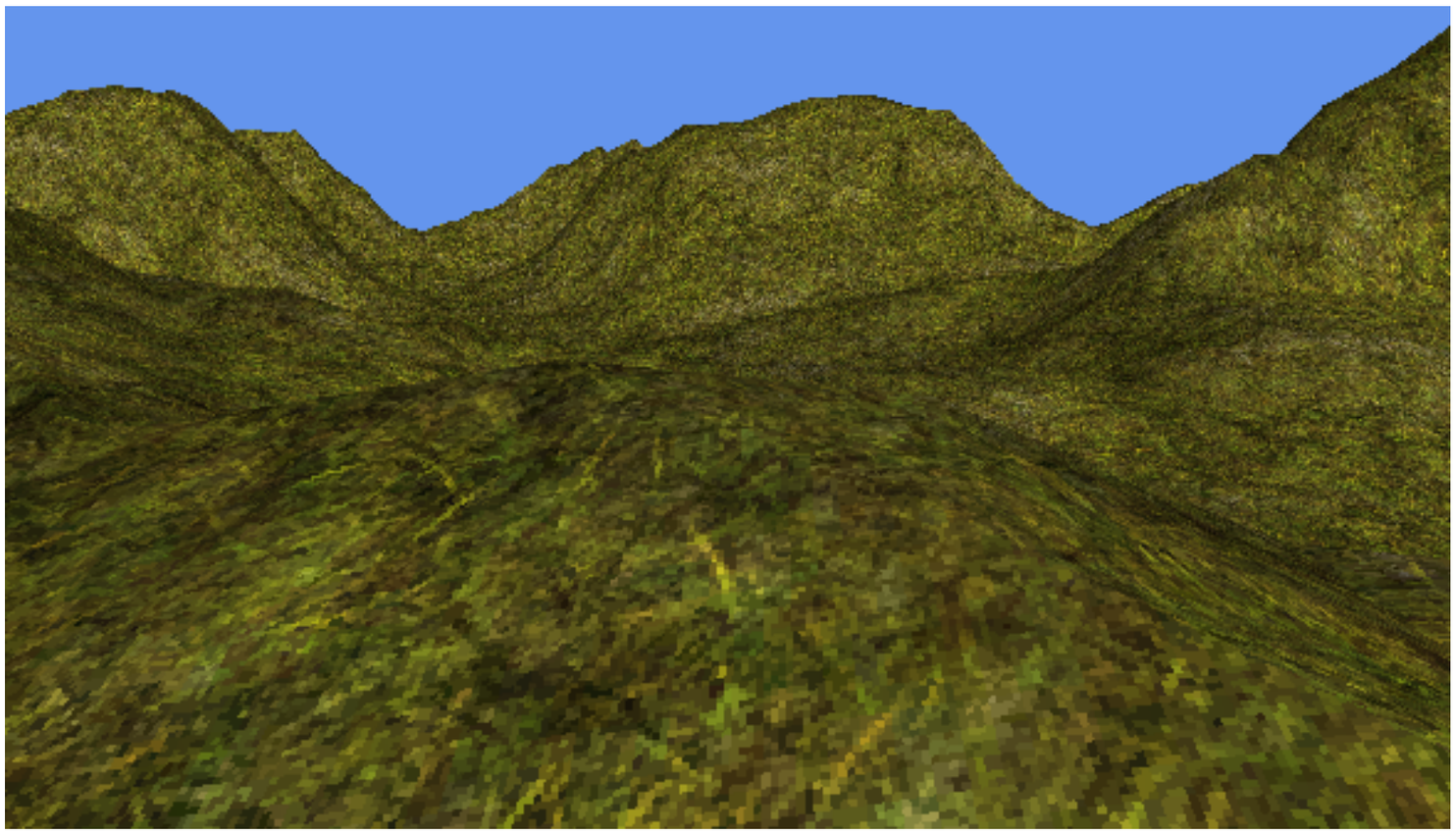
Source: Tamar Shinar, UC Riverside

Texture Filtering Modes

- Nearest Neighbour
- Bilinear filtering
- Mipmapping
- Anisotropic

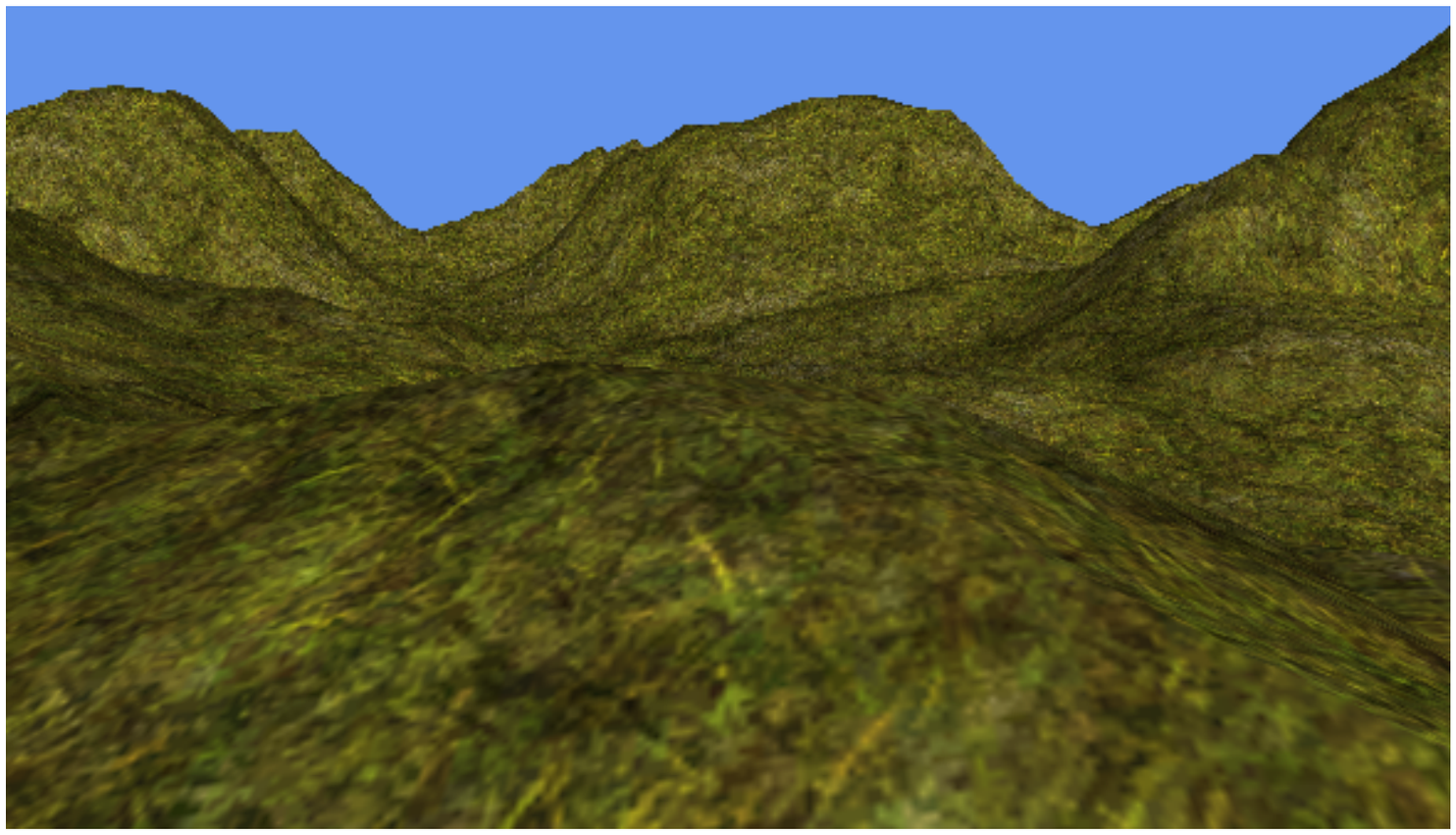
Nearest Neighbour

- For each destination pixel, round to the closest matching location in the source pixel grid and return its value



Bi-linear

- For each destination pixel, locate closest matching source pixel grid and interpolate between four neighbouring values in source



Mipmapping

- A mipmap is a pre-calculated copy of an image that has been shrunk to a lower resolution using a high quality filtering algorithm

Source: Texturing and Lighting in OpenGL, J.V. Oosten

Original = 256x256

Mip 1 = 128x128

Mip 2 = 64x64

Mip 3 = 32x32

Mip 4 = 16x16

Mip 5 = 8x8

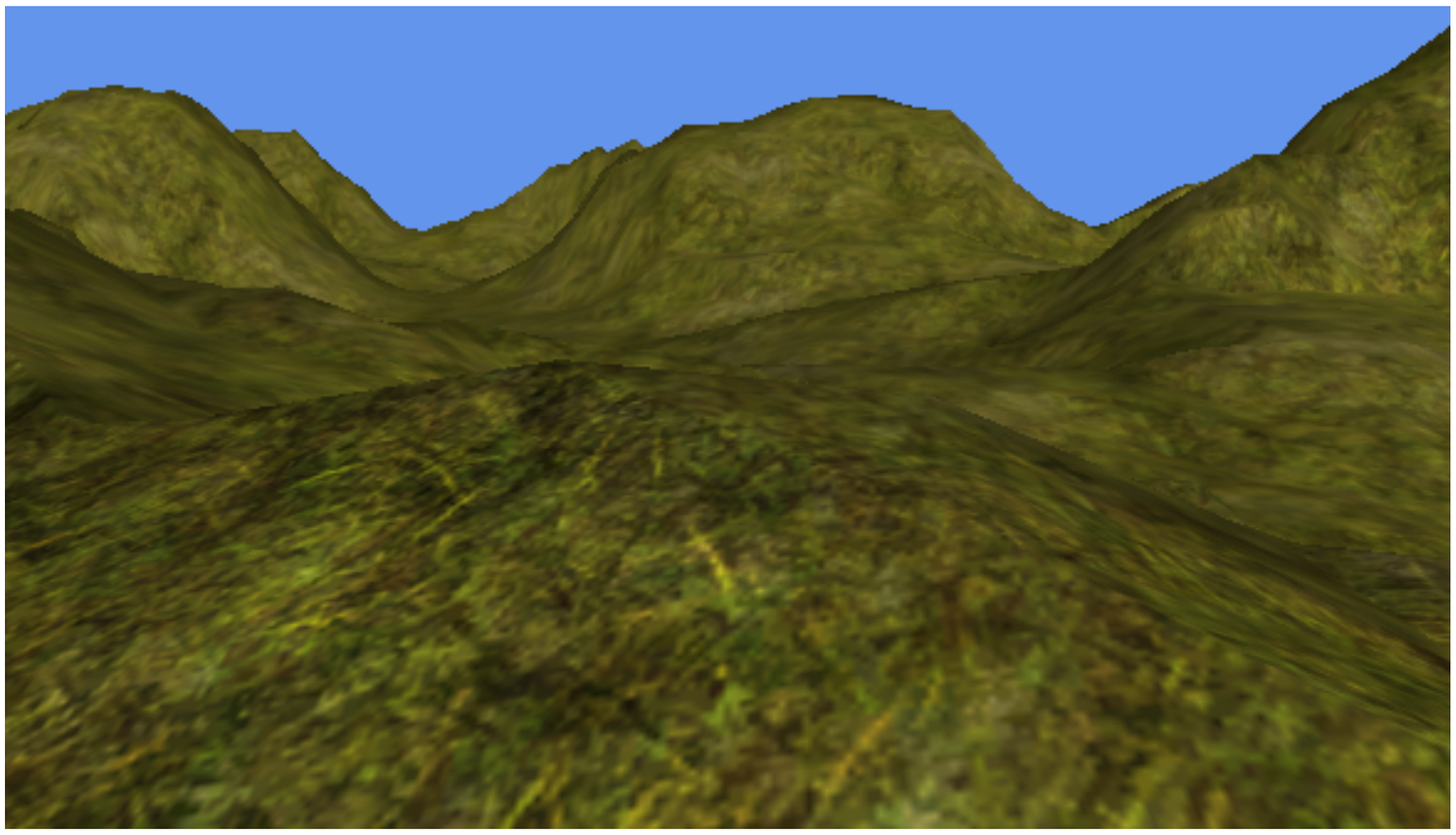
Mip 6 = 4x4

Mip 7 = 2x2

Mip 8 = 1x1

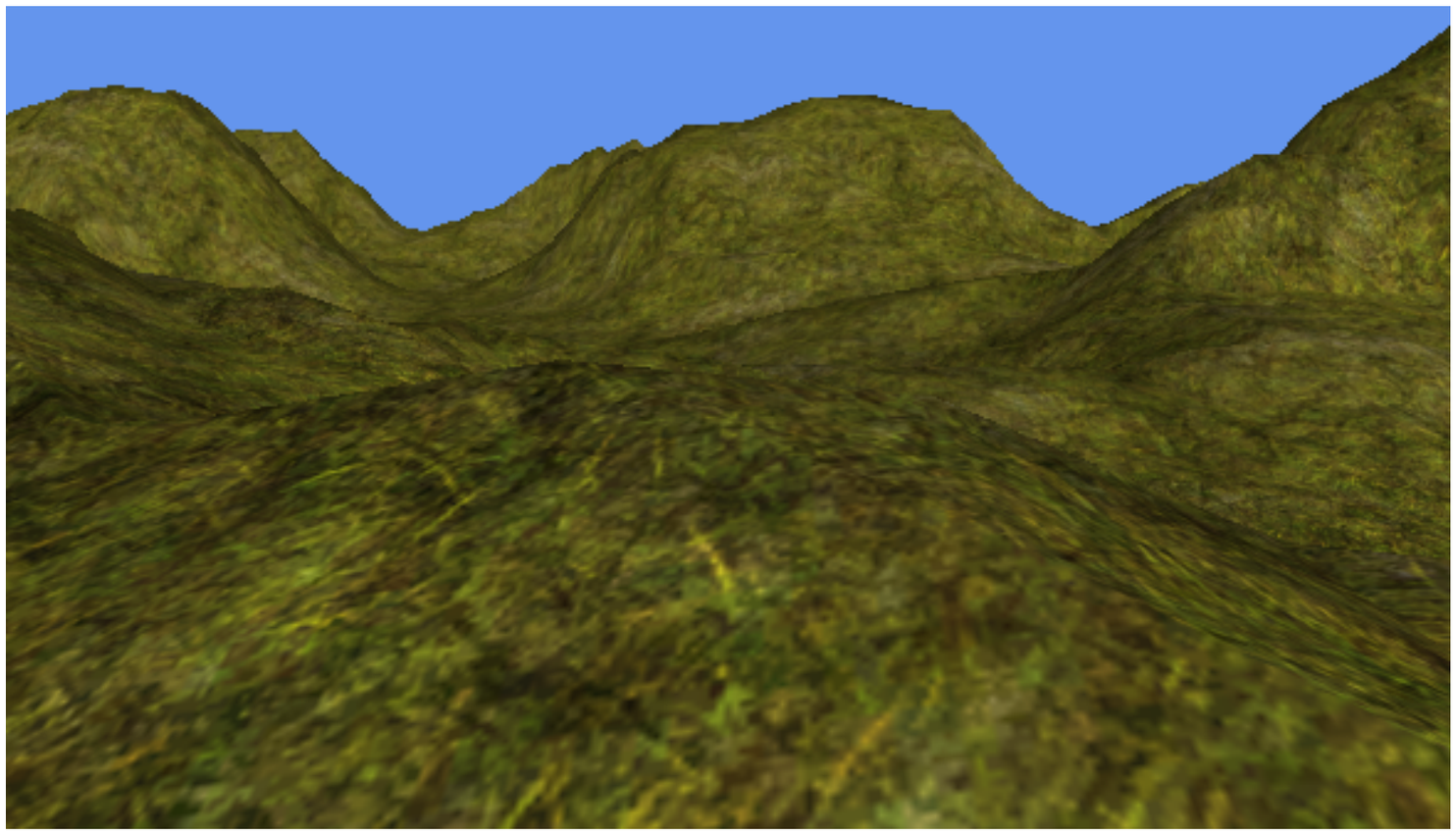
Mipmapping

- Either nearest-neighbour or bilinear filtering may be combined with mip-map filtering



Anisotropic

- Recognises when a texture is being scaled differently in different directions, and takes a larger number of samples to compensate



Texture Synthesis

Procedural texture: little program that computes colour as a function of x, y, z:

$$f(x, y, z) \mapsto \text{color}$$

- Fourier-like texture
- Perlin noise
- Reaction-diffusion textures
- Data driven texture synthesis

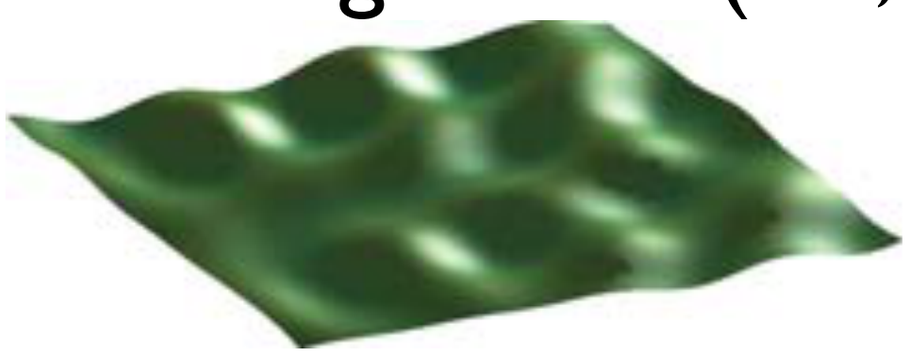
Source: Procedural wood texture, Blender

Fourier-like Synthesis

- Displacement function

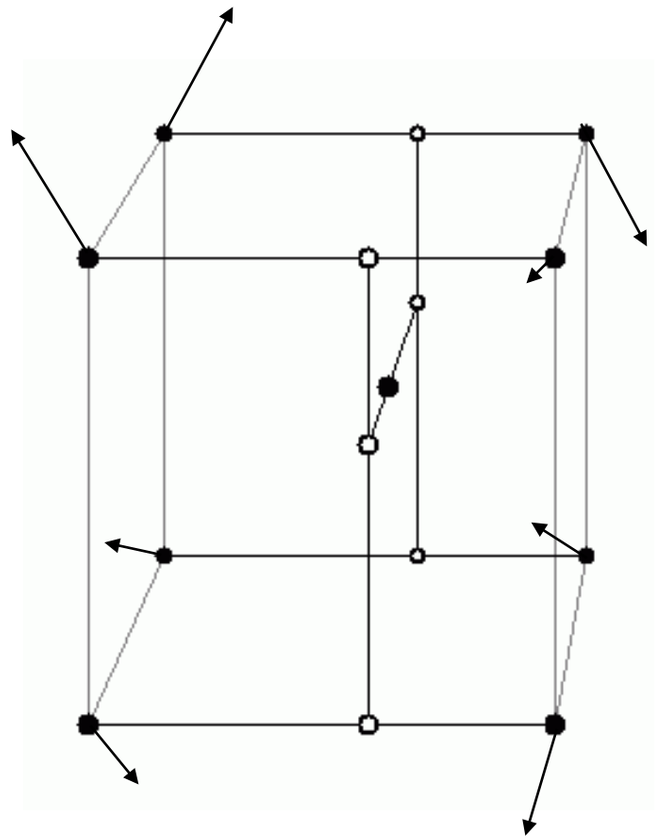
$$d(x, y) = \sum_{i=0}^{n-1} c_i \cos\left(a_i x + b_i y + c\right)$$

- (a_i, b_i) chosen randomly from annular region in the plane: $r^2 \leq a_i^2 + b_i^2 \leq R^2$
- Could be used for representing cloud density (in 3D), placement of vegetation (2D, with threshold)



Perlin Noise

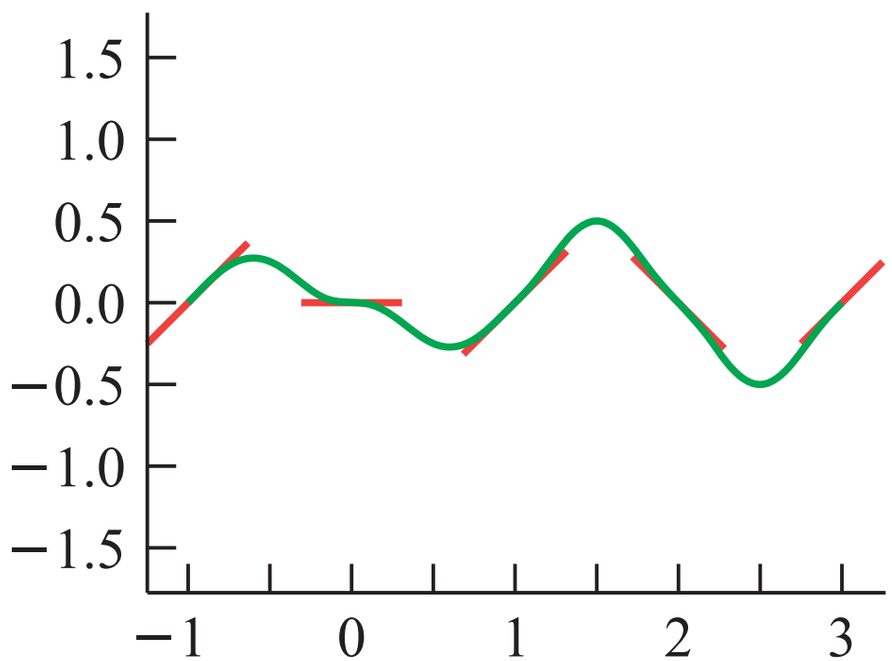
- Perlin (1985, 2002)
- Cube lattice
- Zero value at vertices
- Pseudo-random gradient at vertices
- Splines to interpolate the values at arbitrary 3D points



1D Noise

Example

- Zero at integer locations
- Pseudo-random gradient at integer locations
- Interpolate at any location



Perlin Noise

For a point $p(x, y, z)$ in the grid

$$n(x, y, z) = \sum_{i=\lfloor x \rfloor}^{\lfloor x \rfloor + 1} \sum_{j=\lfloor y \rfloor}^{\lfloor y \rfloor + 1} \sum_{k=\lfloor z \rfloor}^{\lfloor z \rfloor + 1} \Omega_{ijk}(x - i, y - j, z - k)$$

$$\Omega_{ijk}(u, v, w) = \omega(u)\omega(v)\omega(w)(\Gamma_{ijk} \cdot (u, v, w))$$

$$\omega(t) = \begin{cases} 2|t|^3 - 3|t|^2 + 1 & \text{if } |t| < 1, \\ 0 & \text{otherwise} \end{cases}$$

Perlin Noise

Γ_{ijk} is a random unit vector for the lattice point $(x, y, z) = (i, j, k)$

We can use a pseudorandom table for that:

$$\Gamma_{ijk} = G(\phi(i + \phi(j + \phi(k))))$$

Here, G is a precomputed array of n random unit vectors and $\phi(i) = P[i \bmod n]$

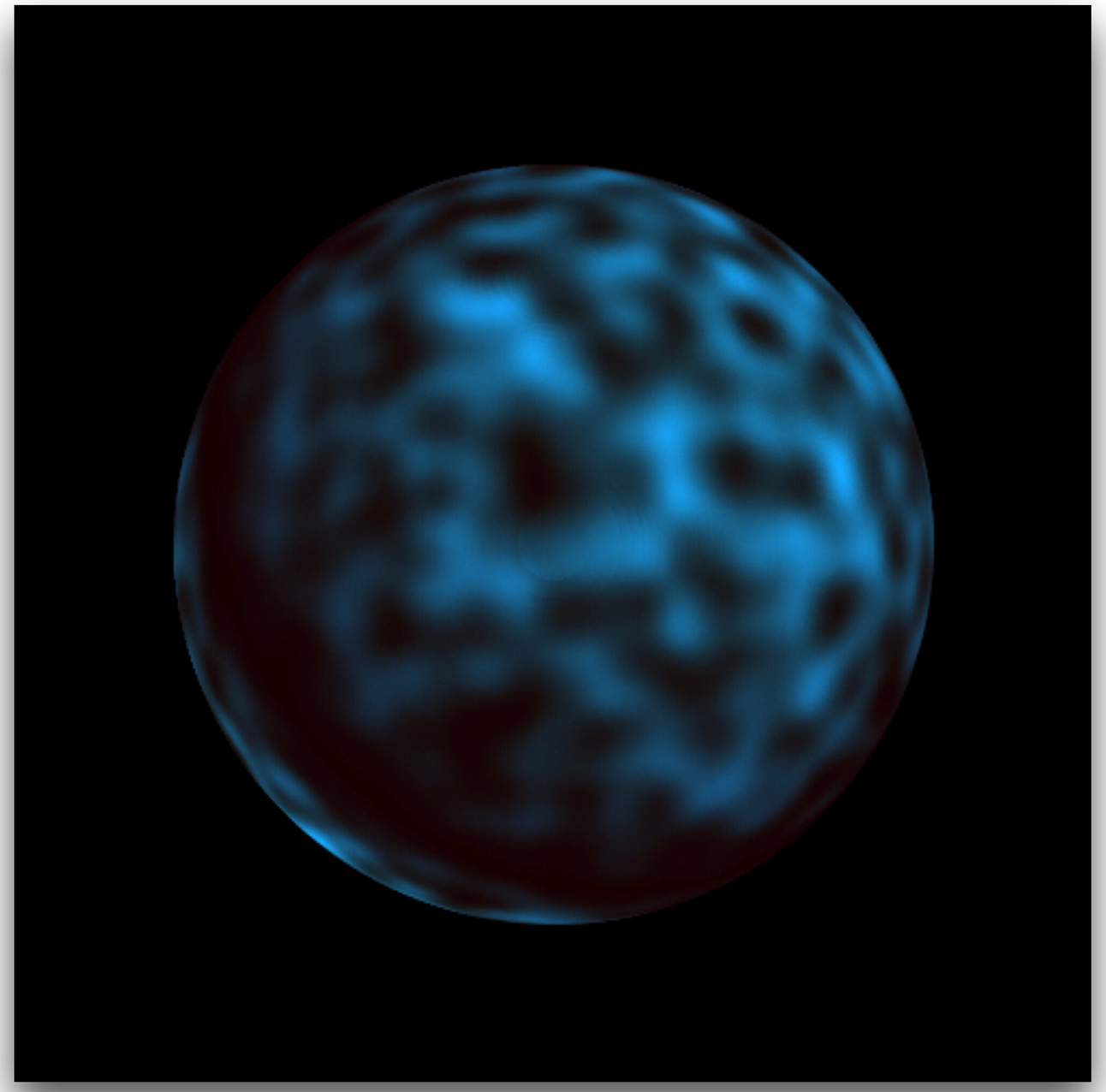
P is an array of length n containing a permutation of the integers 0 to n-1

E.g.: n=256 works well.

Algorithm

1. Given an input point (say p)
2. For each of its neighbouring grid points (q)
    I. Pick a "pseudo-random" gradient vector (g)
    II. Compute linear function (dot product: $g \cdot (p - q)$)
3. Take weighted sum, using *ease curves*
    1. Interpolate between 2^3 values using S-shaped cross-fade curve

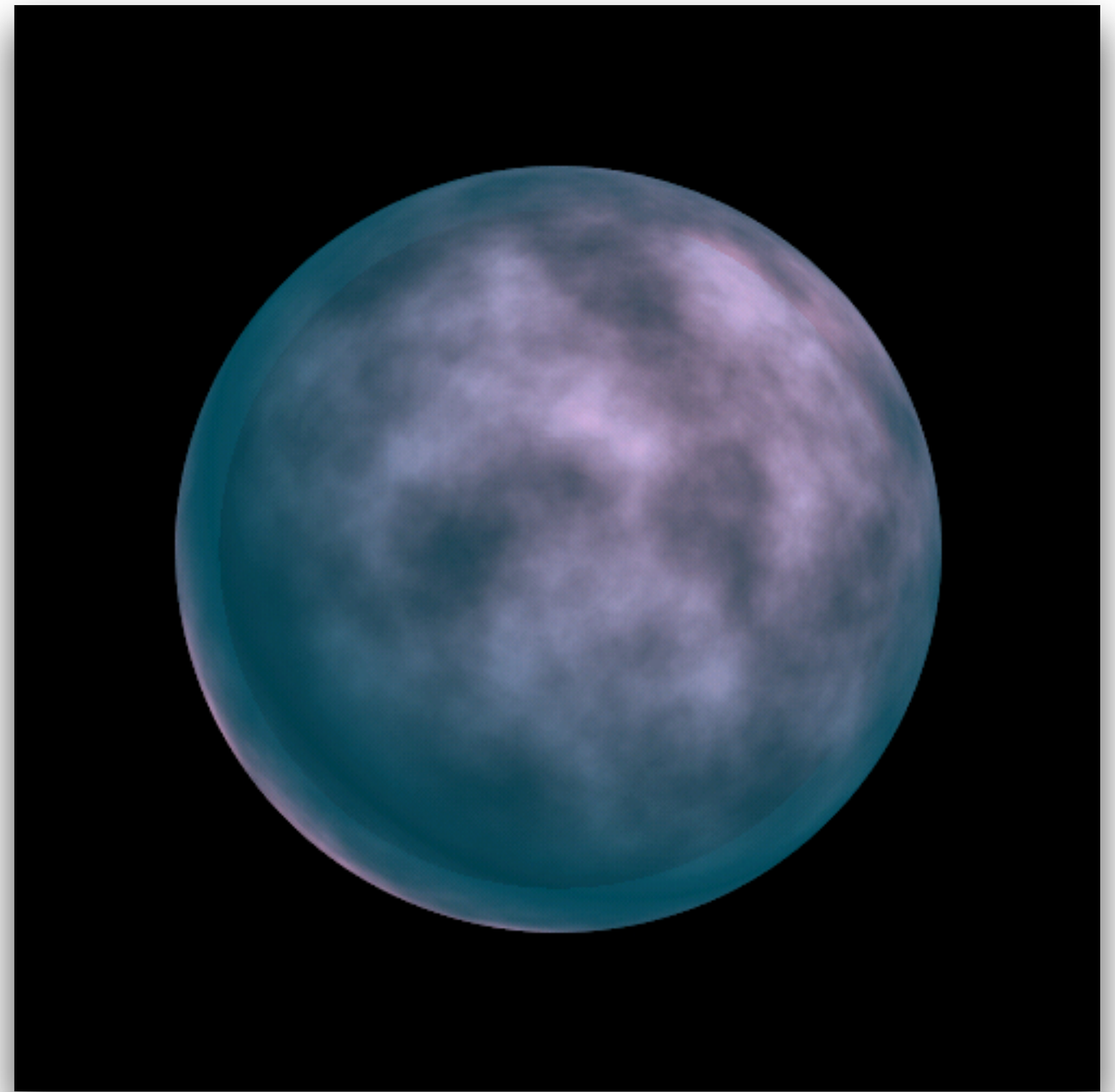
Noise



Source: Making Noise, Ken Perlin, http://www.noisemachine.com

Sum 1/f(noise)

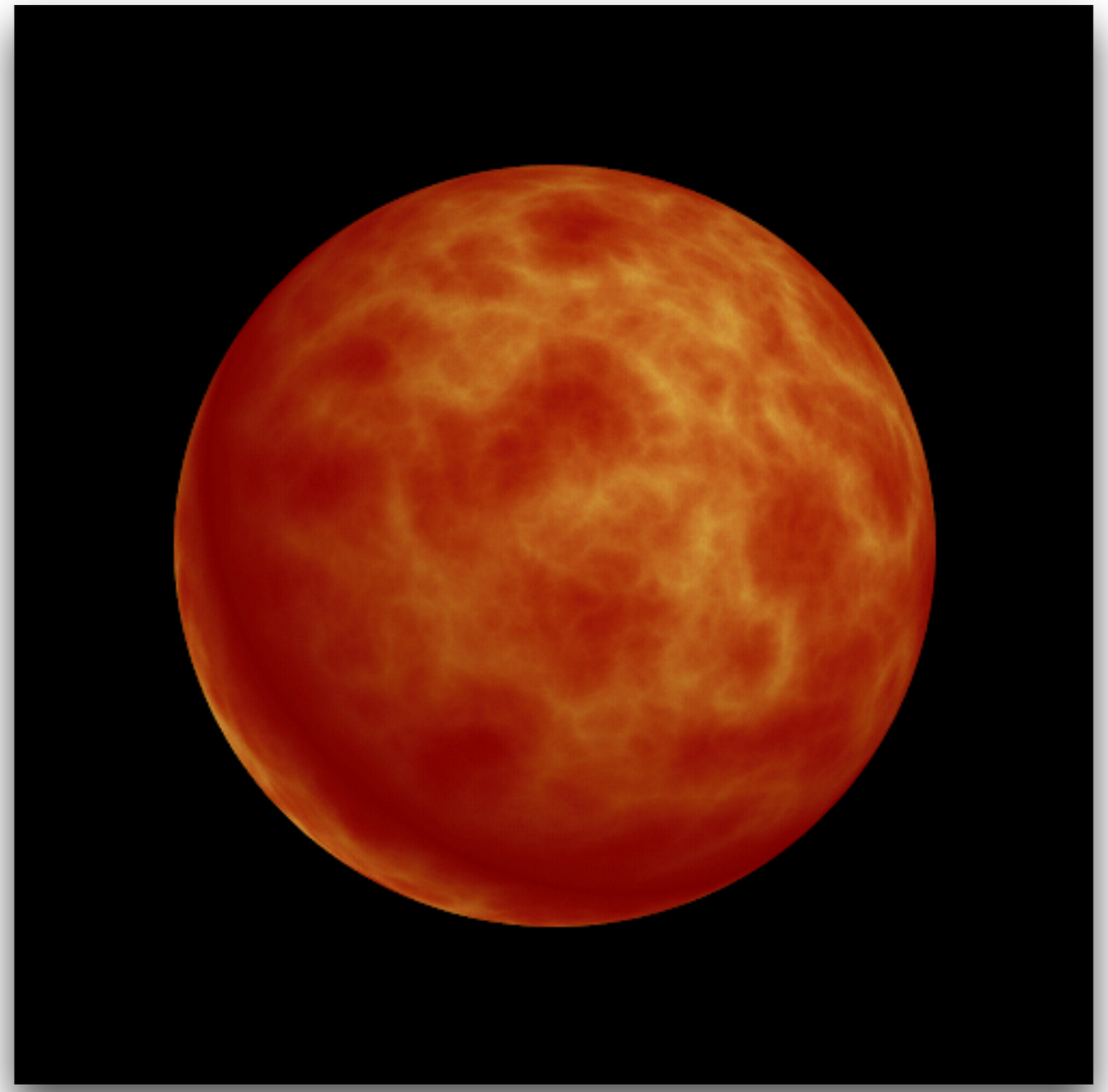
noise(p) + 1/2 noise(2p) + 1/4 noise(4p) + …



Source: Making Noise, Ken Perlin, http://www.noisemachine.com

Sum 1/f(|noise|)

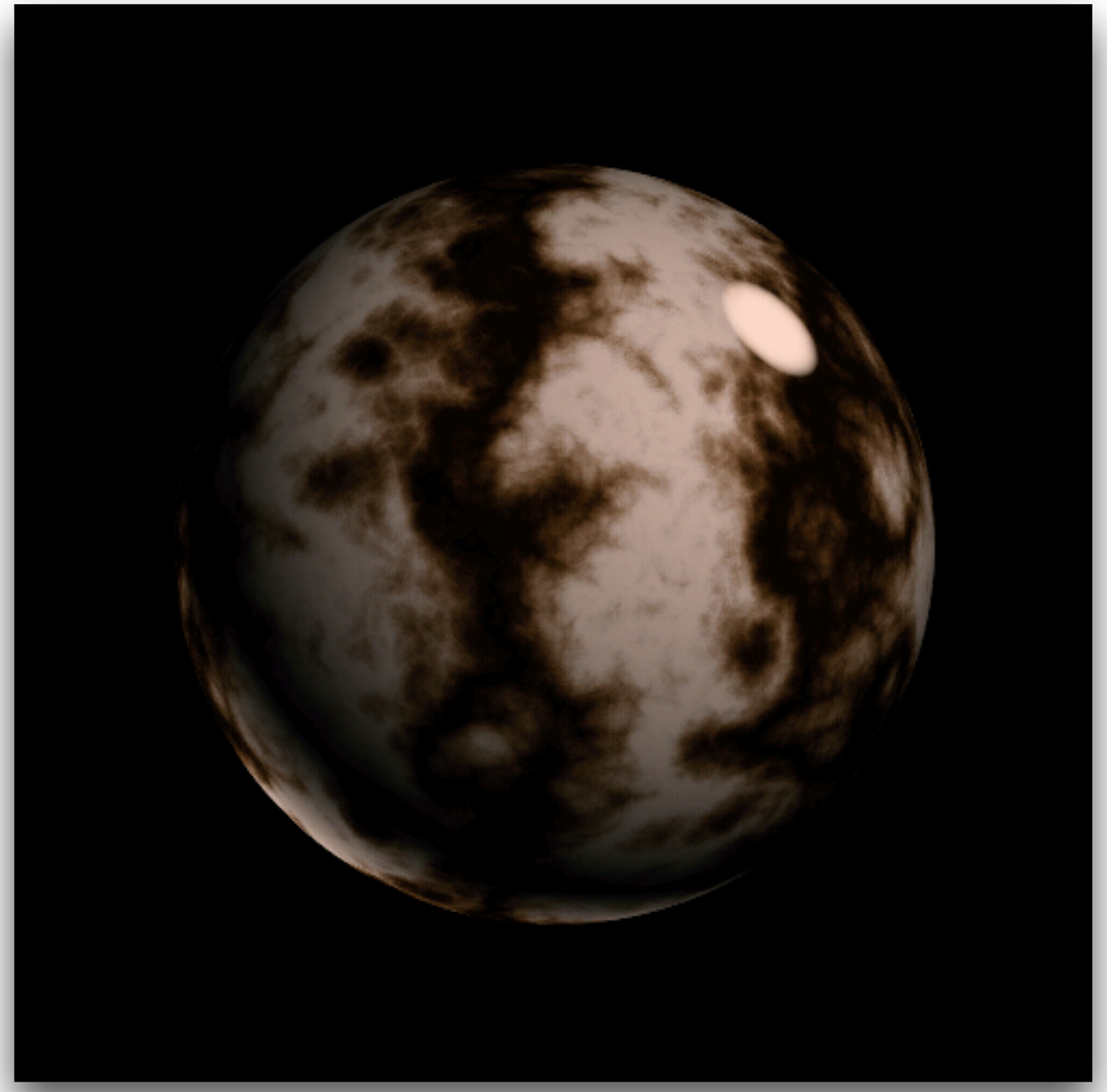
|noise(p)| + 1/2 |noise(2p)| + 1/4 |noise(4p)| + …



Source: Making Noise, Ken Perlin, http://www.noisemachine.com

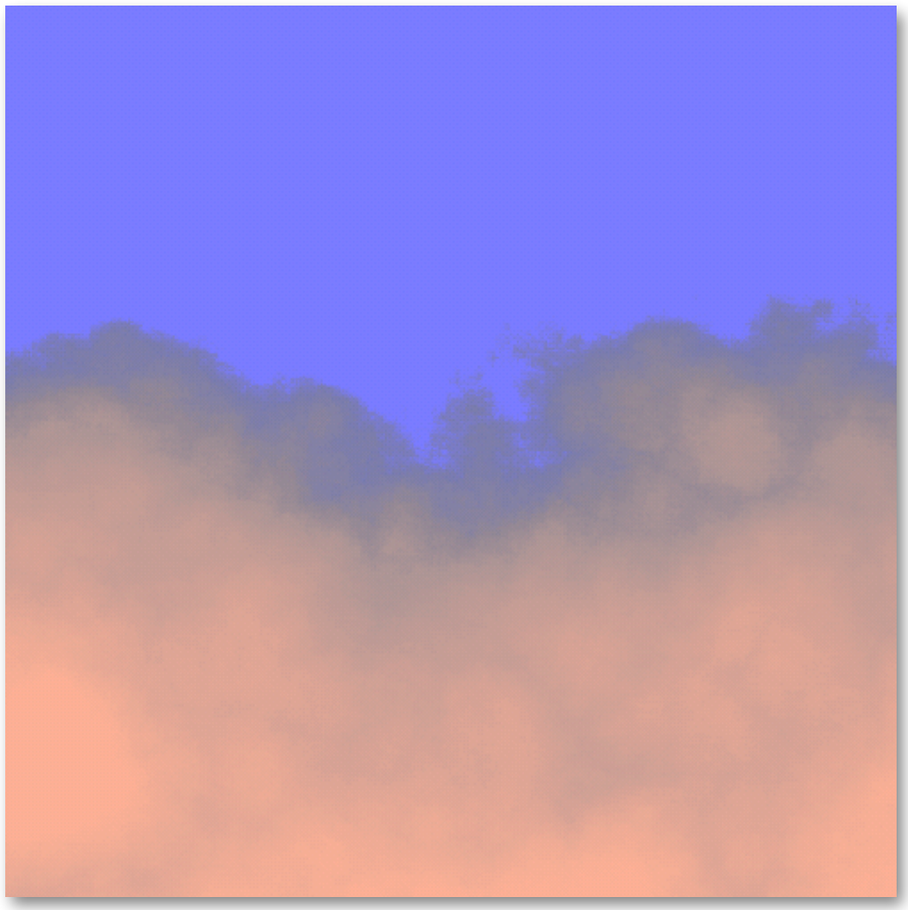
Sin(x + sin 1/f(|noise|)

sin(x + |noise(p)| + 1/2 |noise(2p)| + 1/4 |noise(4p)| + …)



Source: Making Noise, Ken Perlin, http://www.noisemachine.com

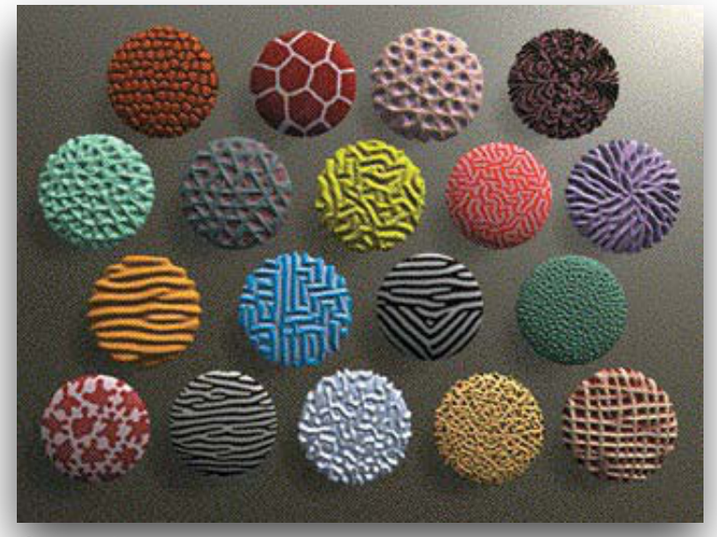
Flame and Clouds



Source: Making Noise, Ken Perlin, http://www.noisemachine.com

Reaction-diffusion Textures

- Turing (1952), Turk (1991), Kass & Witkin (1991)
- Simulating formation of patterns in nature (leopard spots, snake scales)
- Patterns evolve through *diffusion* and *reaction*

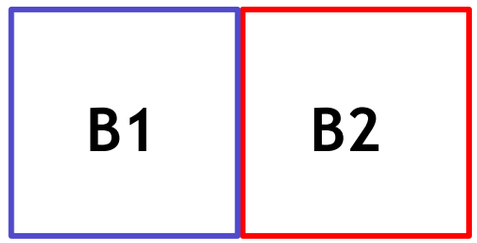


Source: Kass and Witkin

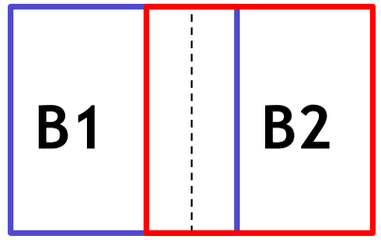
Source: Greg Turk

Data-driven Texture

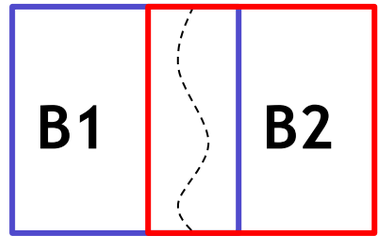
- Efros (2001): Synthesise a large texture (*target*) from a small sample texture (*source*)



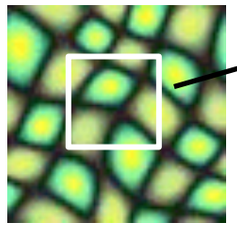
Random placement of blocks



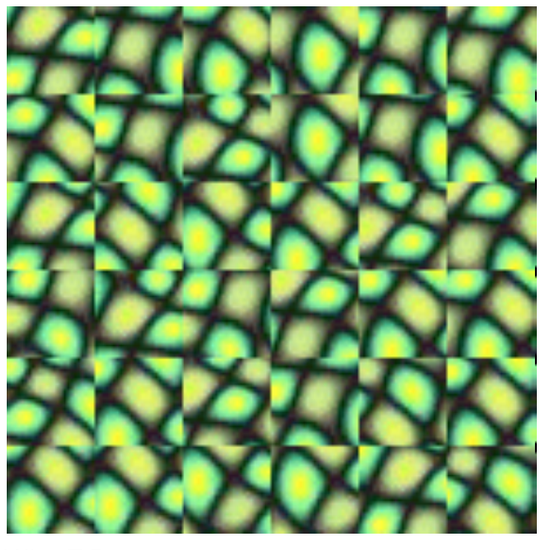
Neighboring blocks constrained by overlap

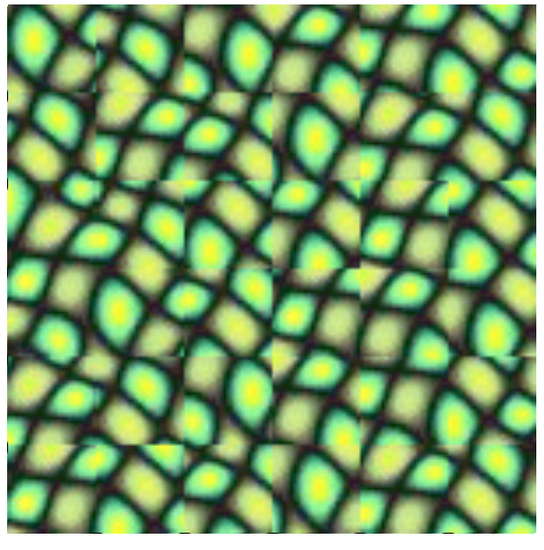


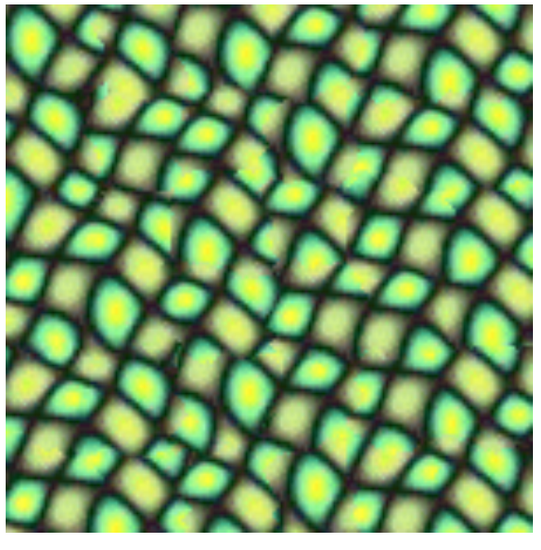
Minimal error boundary cut



Input texture

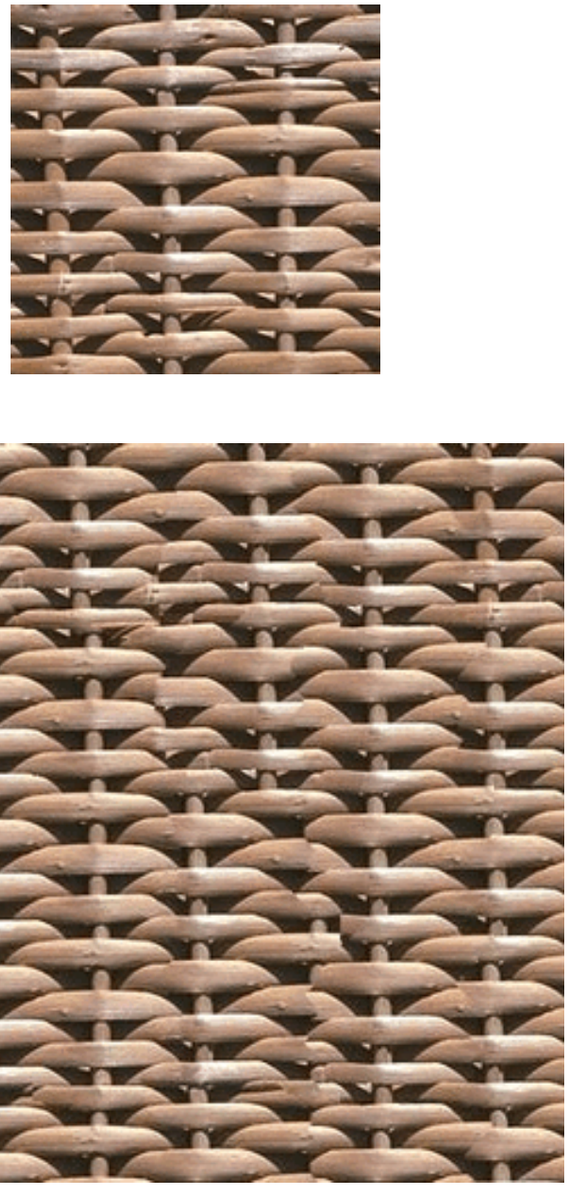






Source: Image Quilting for Texture Synthesis and Transfer, Alexei A. Efros and Willian T. Freeman

Data-driven Texture



Source: Image Quilting for Texture Synthesis and Transfer, Alexei A. Efros and Willian T. Freeman

Other Discrete Techniques

- Bump mapping
- Normal Mapping
- Displacement mapping
- Environment mapping

Bump Mapping

- Textures may as well be used to alter the surface detail by changing the surface normal
- This gives an illusion of fine-scale geometry
- A *bump map* perturbs the surface normal (Blinn, 1978) using a height map
  - Compute normal of height map at a point using finite difference method
  - Add this normal to the geometric normal and normalise

Bump Mapping

- Let $p(u, v)$ be a point on the a parameterised surface. The normal n is given by:

$$n = \frac{p_u \times p_v}{|p_u \times p_v|}$$

- Let the displacement be given by the function **bump** or **displacement function** $d(u,v)$

- Displaced surface can be written as:

$$p' = p + d(u, v)n$$

Bump Mapping

- Displaced normal is given by:

$$n' = p'_u \times p'_v$$

- Computing partial derivatives:

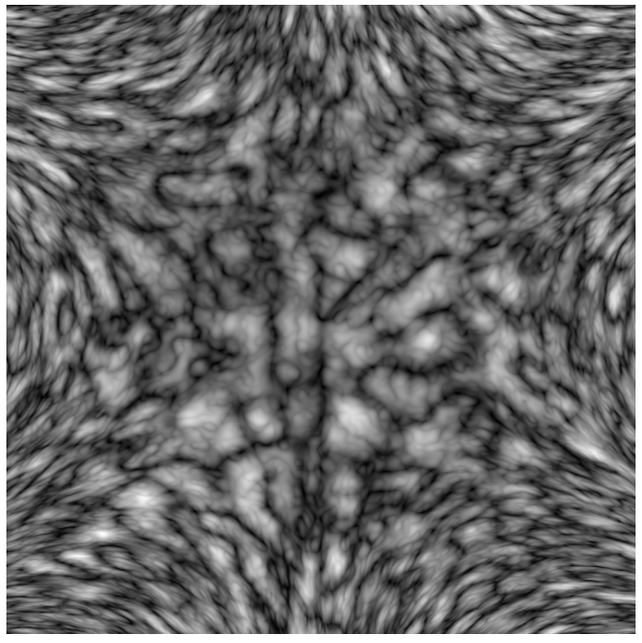
$$p'_u = p_u + \frac{\partial d}{\partial u} n + d(u,v) n_u,$$

$$p'_v = p_v + \frac{\partial d}{\partial v} n + d(u,v) n_v$$

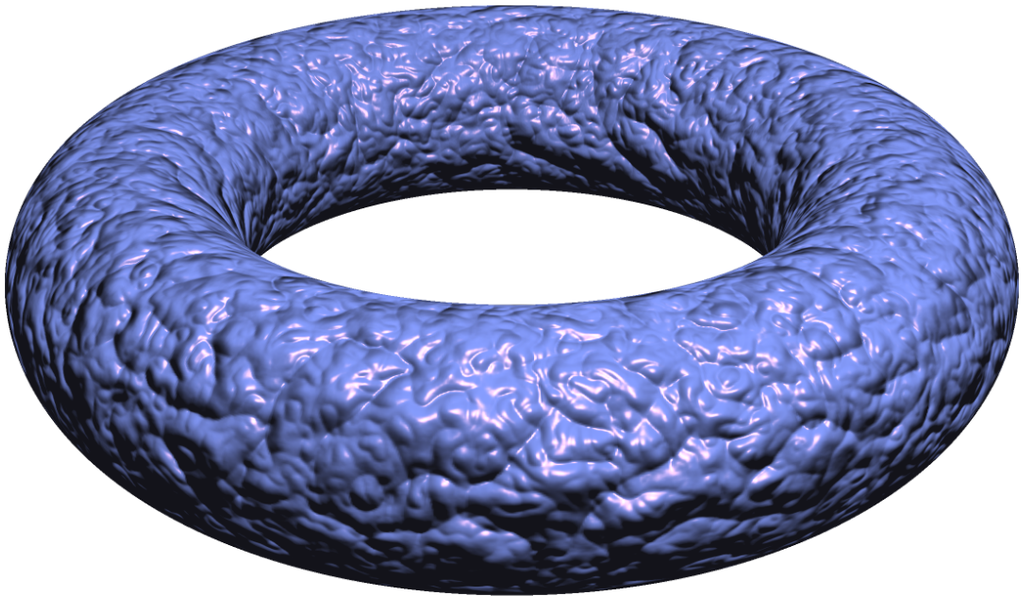
- For small d, the perturbed normal is given by:

$$n' \approx n + \frac{\partial d}{\partial u} n \times p_v + \frac{\partial d}{\partial v} n \times p_u$$

Bump Mapping



Bump map

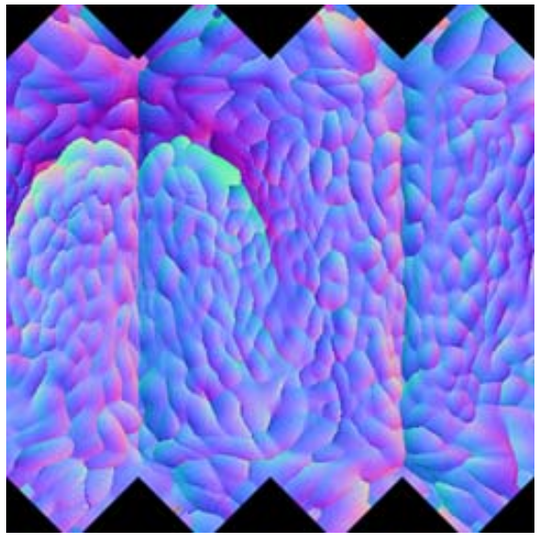


Bump mapped torus

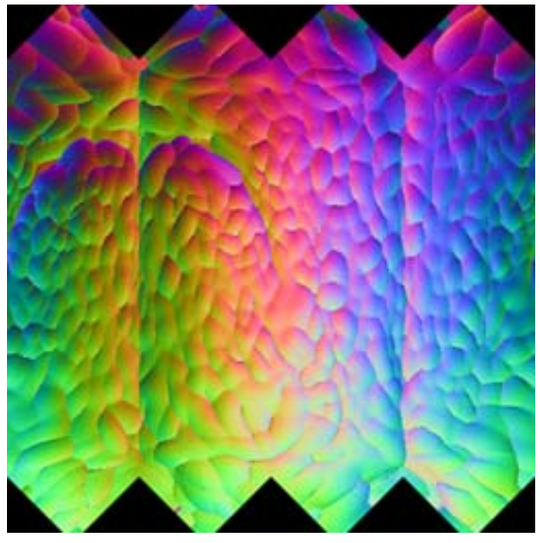
Normal Mapping

- Instead of computing the normal (as with bump mapping), it is stored in an RGB image
- The R, G, B components correspond to X Y Z coordinates of the surface normal

Tangent-space normal map



Object-space normal map



Source: Eric Chadwick, http://wiki.polycount.com

Normal Mapping

- Need to accurately compute tangent space

$$m = \frac{n}{|n|},$$

$$t = \frac{p_u}{|p_u|},$$

$$b = m \times t$$

m: normal vector

t: tangent vector

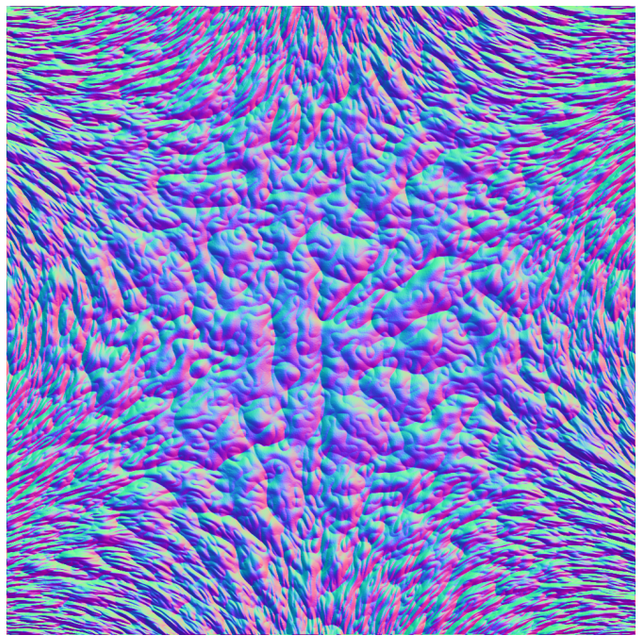
b: binormal vector

- Formulate the rotation matrix

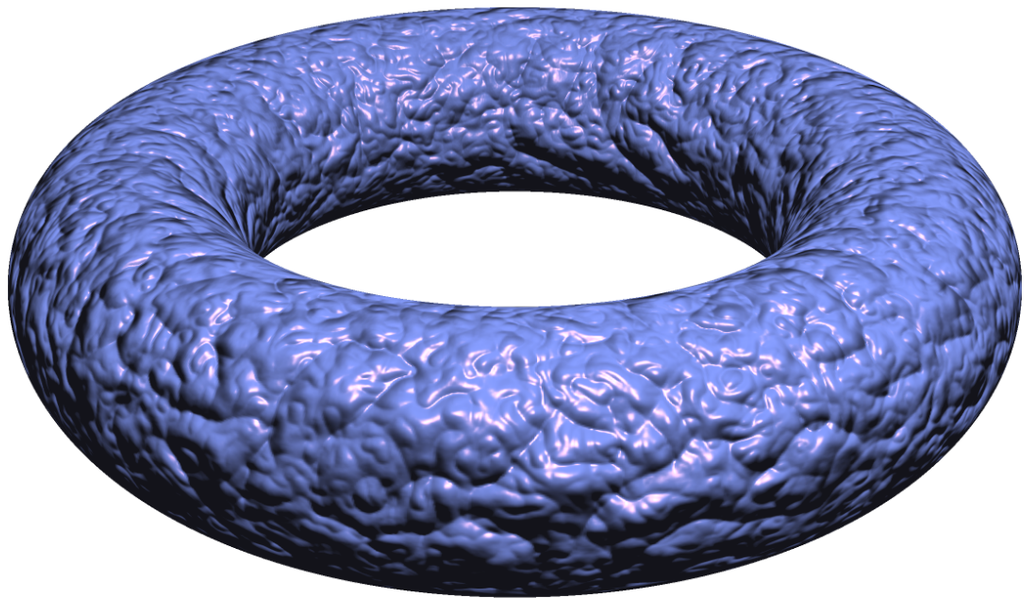
$$M = \begin{bmatrix} t & b & m \end{bmatrix}^T$$

that will convert representations in original space to the local tangent space

Normal Mapping



Normal map



Bump mapped torus

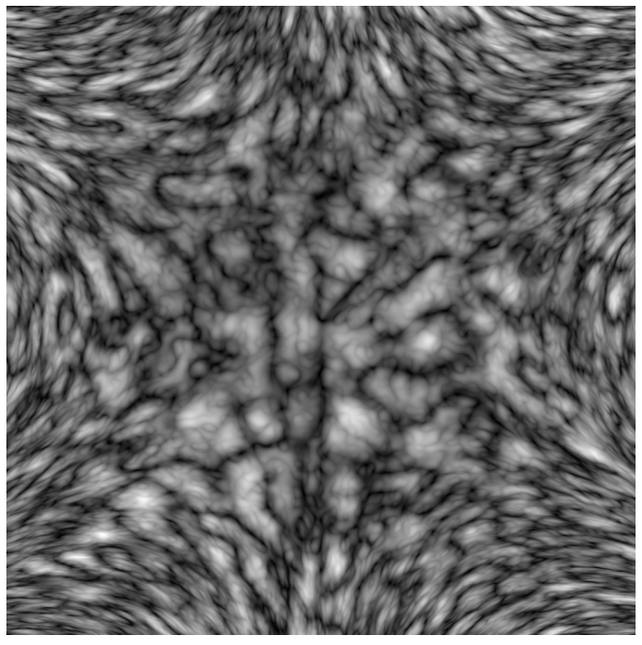
Displacement Mapping

- Bumps neither cast shadows nor affect the silhouette of the objects
- For more realism, we can apply a *displacement map* that actually changes the geometry

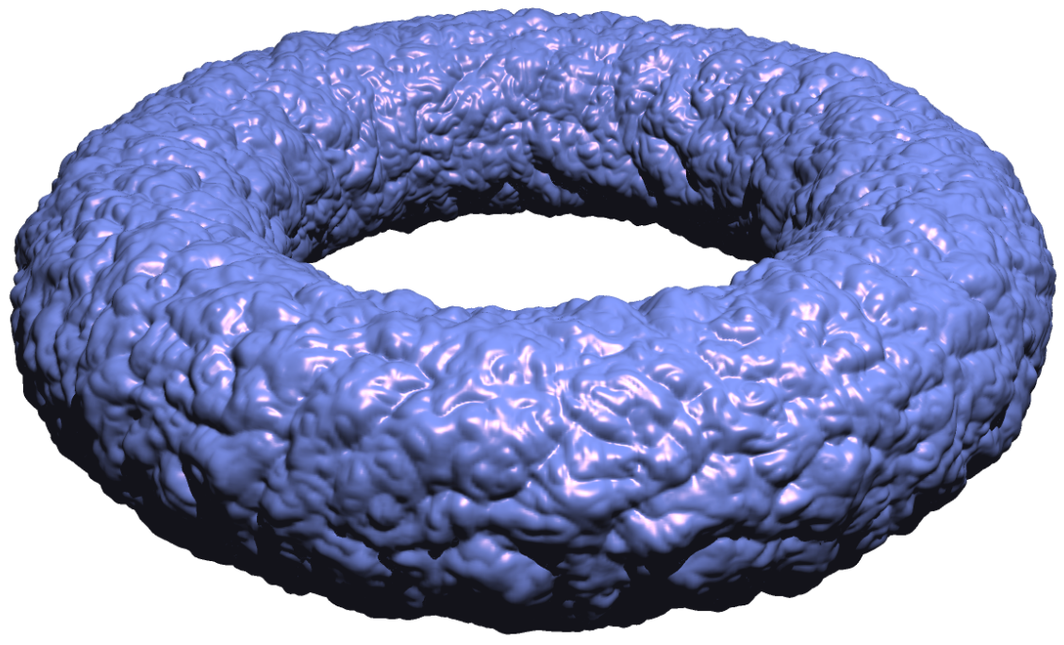
$$p' = p + d(u, v)n$$

- Correct normals to be used during illumination computation

Displacement Mapping



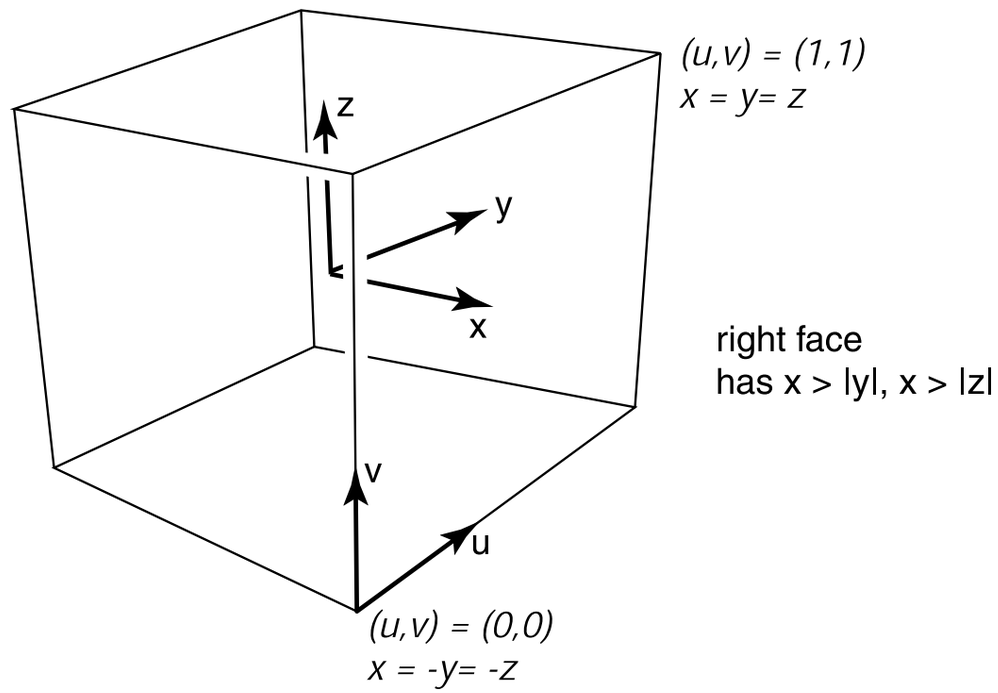
Displacement map



Displacement mapped torus

Environment Mapping

- Objects to have specular reflections of the surrounding environment
- The environment is modelled as a box textured with a cube map (comprising six textures)



Environment Mapping

- With an infinitely large textured cube, return what the reflected rays (from the object surface) see
- For any reflection ray *a*+ *tb*, return cubemap(*b*)

Reading

- FCG: 11
- CG: 20 (see notes: CG_Texure_Mapping_notes.pdf)
- On perspective correct texturing (see notes: perspectiveCorrect.pdf)

ICG: Interactive Computer Graphics, E. Angel, and D. Shreiner, 6th ed.
FCG: Fundamentals of Computer Graphics, P. Shirley, M. Ashikhmin, and S. Marschner, 3rd ed.
CG: Computer Graphics, principles and Practice, J. F. Hughes, et al.