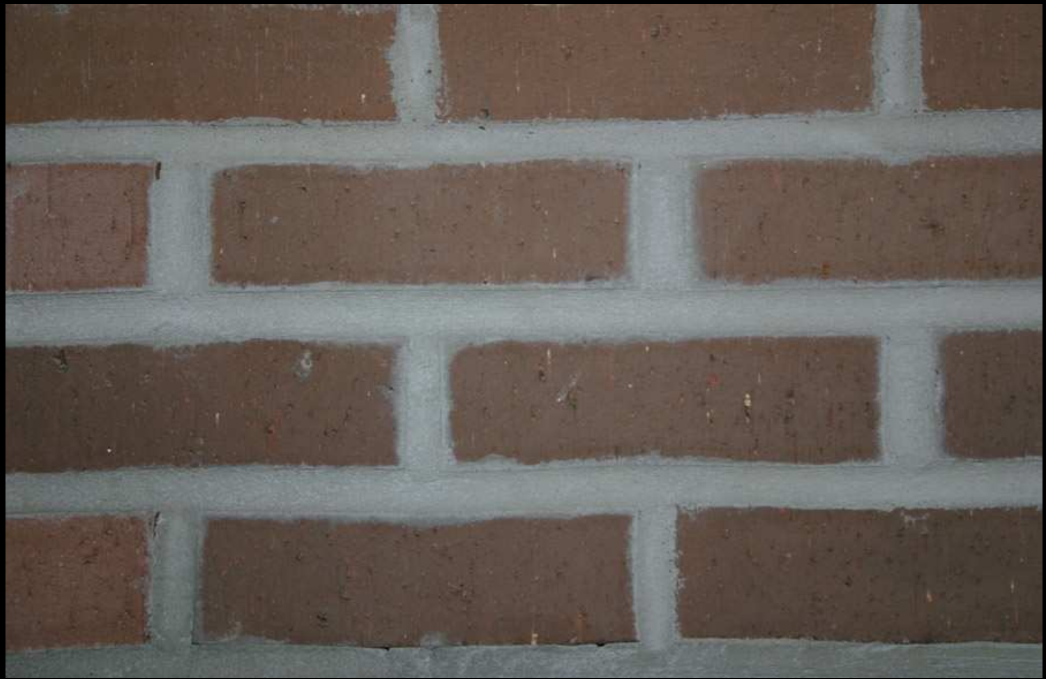
# Texture



# Overview

- Objectives and introduction
- Painted textures
- Bump mapping
- Environment mapping
- Three-dimensional textures
- Functional textures
- Antialiasing textures
- OpenGL details

# Objectives

- Why texture mapping?
- Introduce mapping methods
  - Texture Mapping
  - Environment Mapping
  - Bump Mapping
- Consider basic strategies
  - Forward vs backward mapping
  - Point sampling vs area averaging

# Why texture mapping?

# Limits of Geometric Modeling

- Graphics cards
  - Render over 10 million polygons per second
- Still insufficient for many phenomena
  - Clouds
  - Grass
  - Terrain
  - Skin
- Few real objects are smooth

# Textures

- Can handle
  - Large repetitions
    - Brick wall
    - Stripes
  - Small scale patterns
    - Brick's roughness
    - Concrete
    - Wood grain
- Typically 2-D images of what's being simulated

# E.g., problem: Modeling an orange (fruit)

- Start with orange-colored sphere
  - Too simple
- Replace sphere with more complex shape
  - Does not capture surface characteristics
    - Small dimples
  - Too many polygons to model all dimples

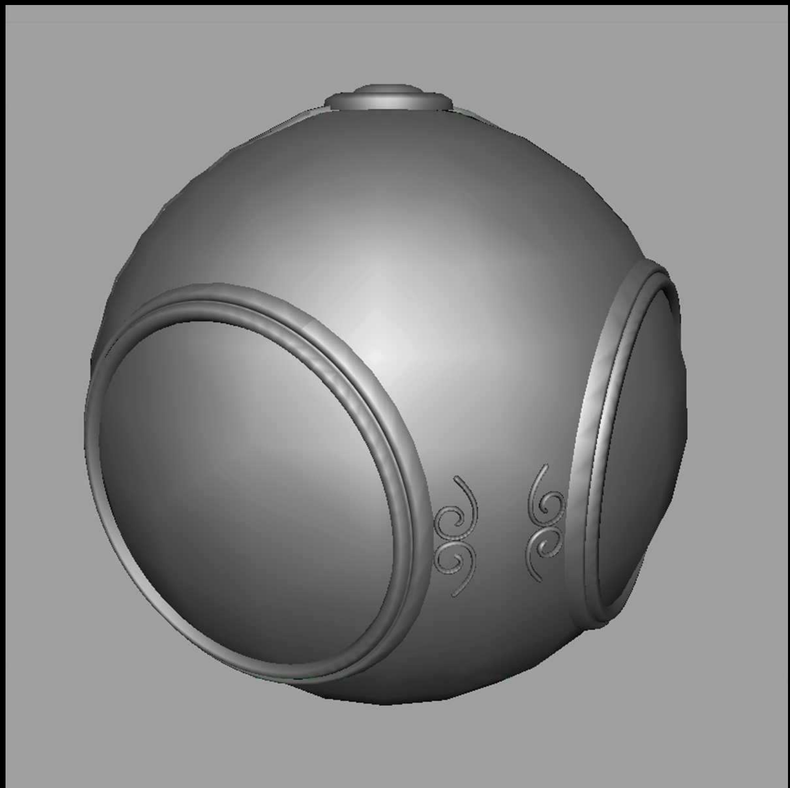
# Modeling an orange (2)

- Take picture of real orange
  - Scan
  - “paste” onto simple geometric model
  - Process known as “texture mapping”
- Still not sufficient
  - Resulting surface smooth
  - Need to change local shape
    - Bump mapping

# Three types of mapping

- Texture Mapping
  - Uses images to fill inside of polygons
- Environment (reflection) mapping
  - Uses picture of environment for texture maps
  - Allows simulation of highly specular surfaces
- Bump mapping
  - Emulates altering normal vectors during rendering

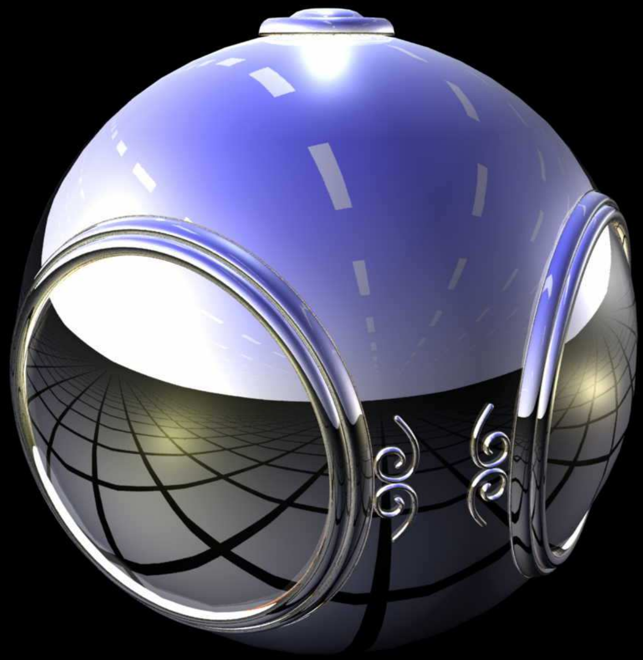
# Texture mapping



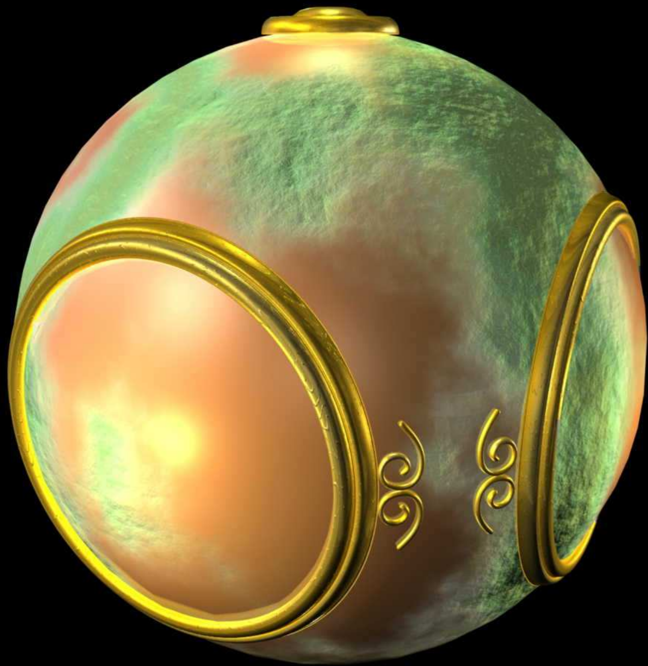
geometric model

texture mapped

# Environment (reflection) mapping

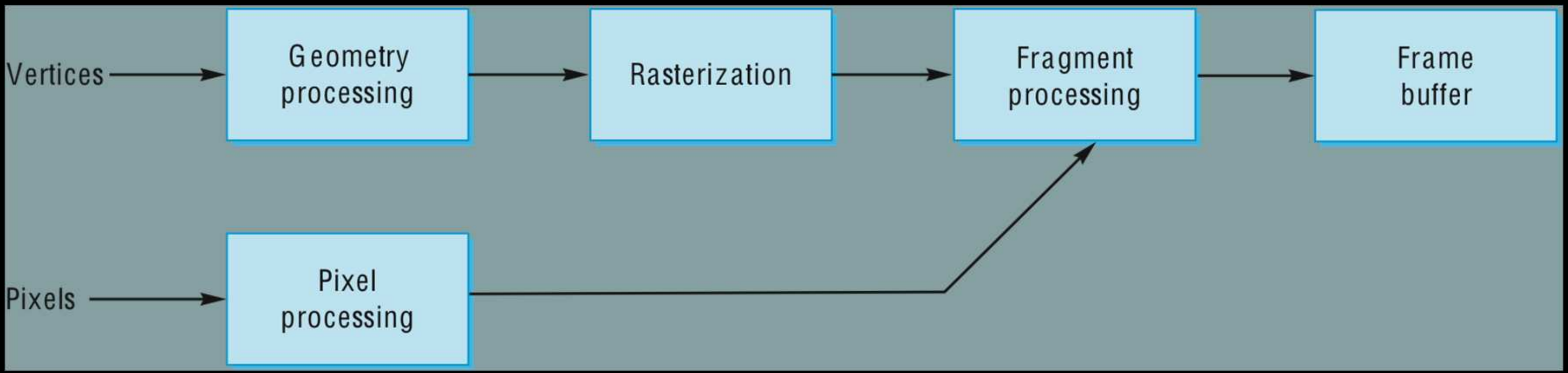


# Bump mapping



# Where does mapping take place?

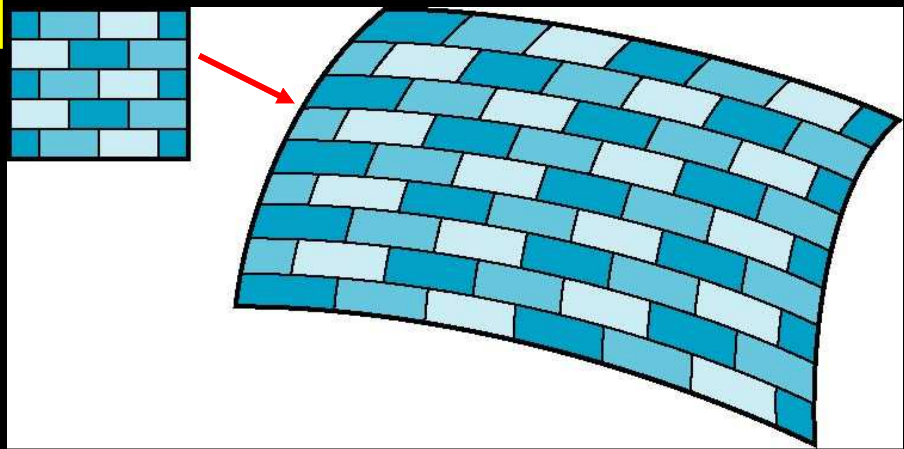
- Mapping techniques implemented at end of rendering pipeline
  - Very efficient because few polygons make it past clipper



# Is it simple?

- Idea: simple
  - Map image to surface
- But, 3 or 4 coordinate systems involved

2D image

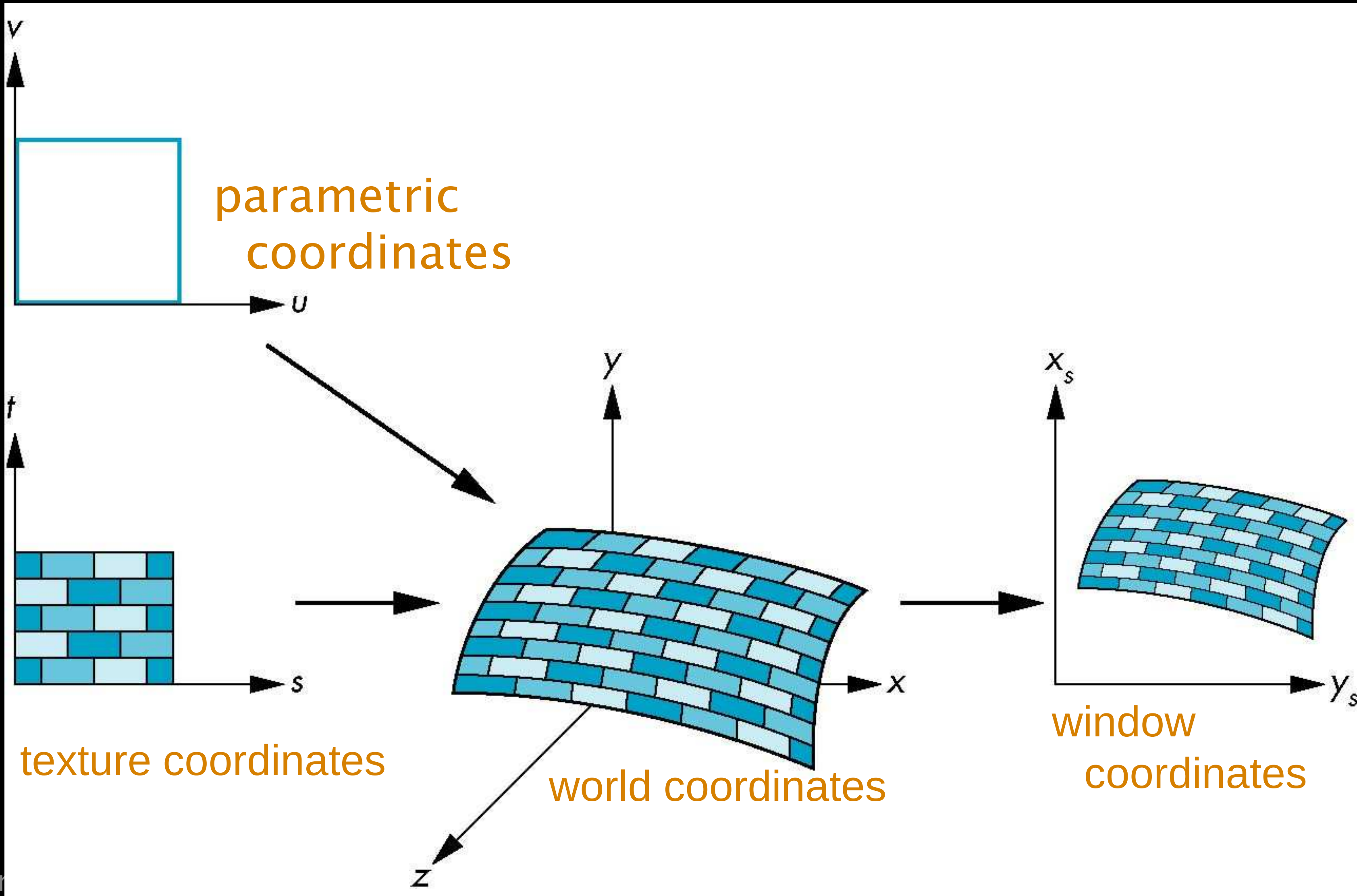


3D surface

# Coordinate systems

- Parametric coordinates
  - Model curves and surfaces
- Texture coordinates
  - Identify points in image to be mapped
- Object or World Coordinates
  - Conceptually, where mapping takes place
- Window Coordinates
  - Where final image really produced

# Texture mapping



# Mapping functions

- Basic problem: how to find the maps
- Consider mapping from texture coo's to point on surface
- Appear to need three functions

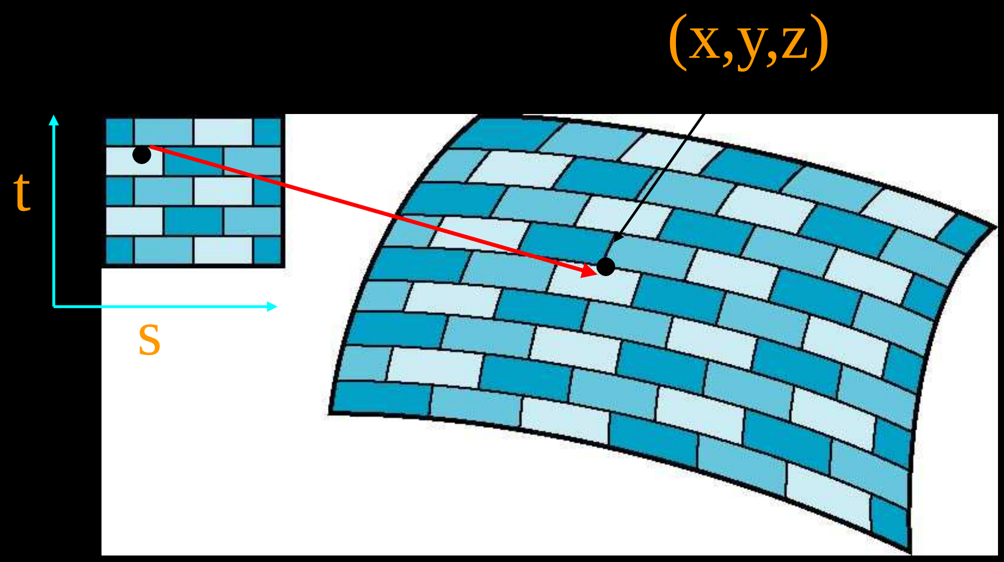
$x = x(s,t)$

$y = y(s,t)$

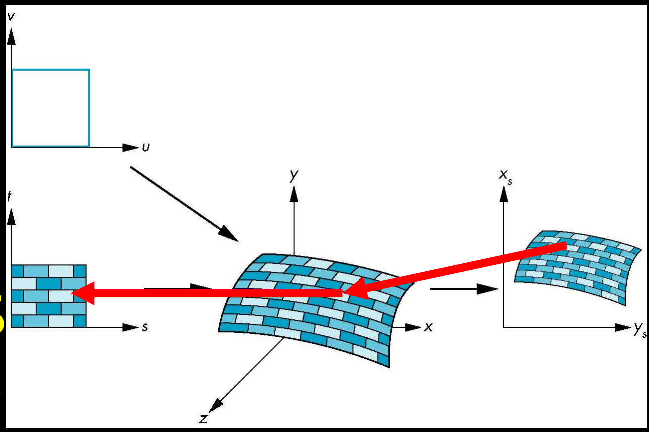
$z = z(s,t)$

- But really want

to go other way



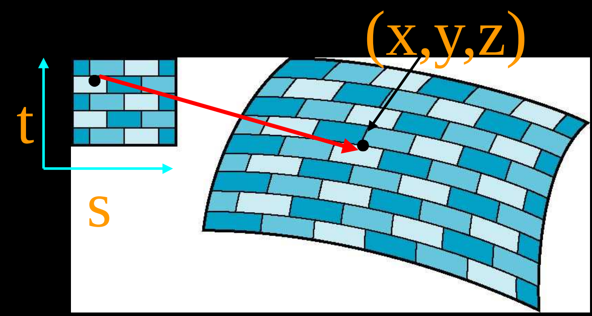
# Backward mapping




- Really want to go backwards
  - Given pixel, want to know
    - to which point on object it corresponds
  - Given point on object, want to know
    - to which point in texture it corresponds
- Need map of form

$$s = s(x,y,z)$$

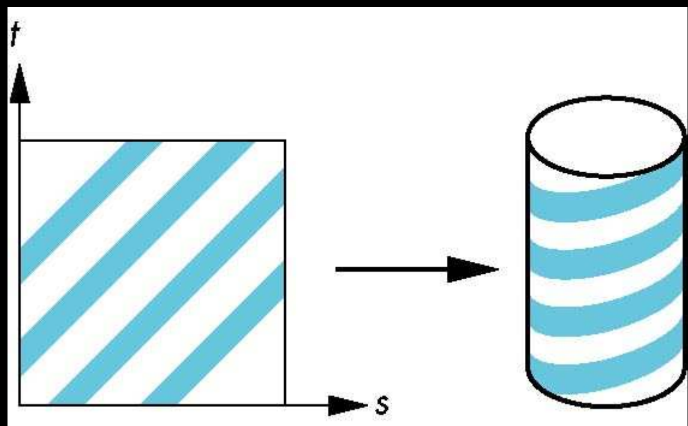
$$t = t(x,y,z)$$




- Such functions difficult to find in general

# Two-part mapping

- One solution to mapping problem
  - first map texture to simple intermediate surface
- Example: map to cylinder



# Cylindrical mapping

parametric cylinder

$$x = r \cos 2\pi\, u$$
$$y = r \sin 2\pi u$$
$$z = v/h$$

maps rectangle in u,v space to cylinder of radius r and height h in world coordinates

$$s = u$$
$$t = v$$

maps from texture space

# Spherical map

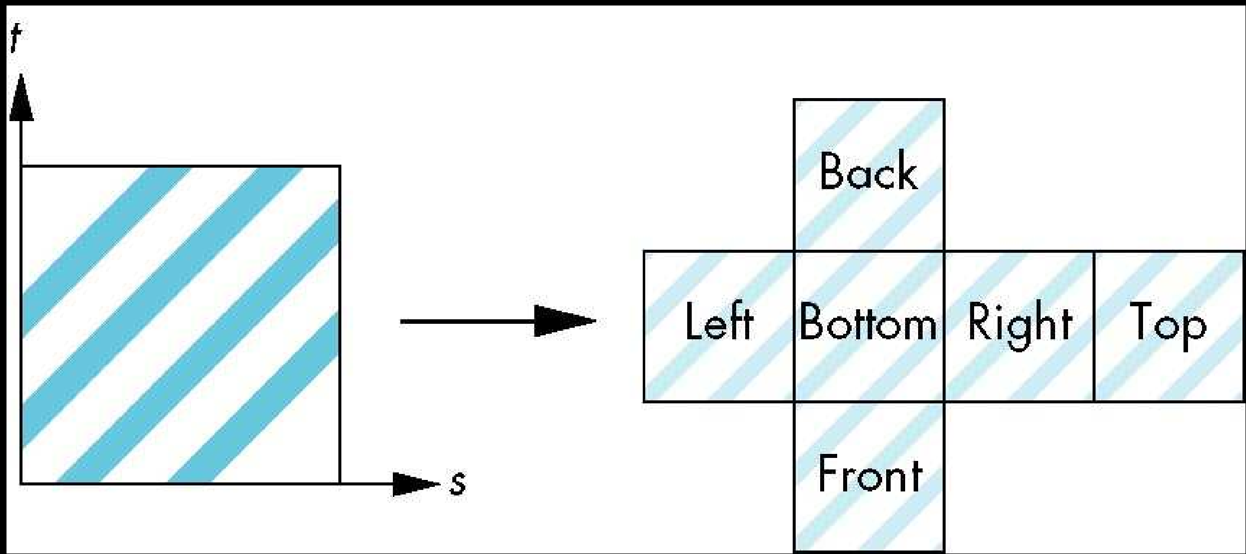
- Can use parametric sphere

$$x = r \cos 2\pi u$$
$$y = r \sin 2\pi u \cos 2\pi v$$
$$z = r \sin 2\pi u \sin 2\pi v$$

- Similar manner to cylinder
  - But have to decide where to put distortion
- Spheres used in environmental maps

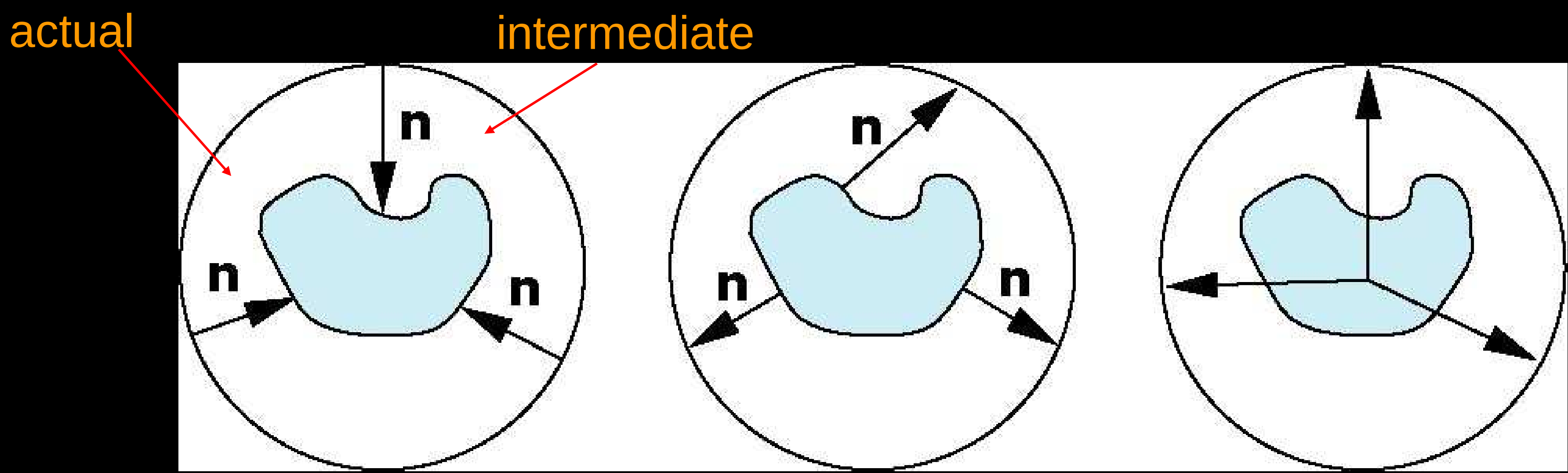
# Box mapping

- Easy to use with simple orthographic projection
- Also used in environment maps



# Second mapping

- Map from intermediate object to actual object
  - Normals from intermediate to actual
  - Normals from actual to intermediate
  - Vectors from center of intermediate

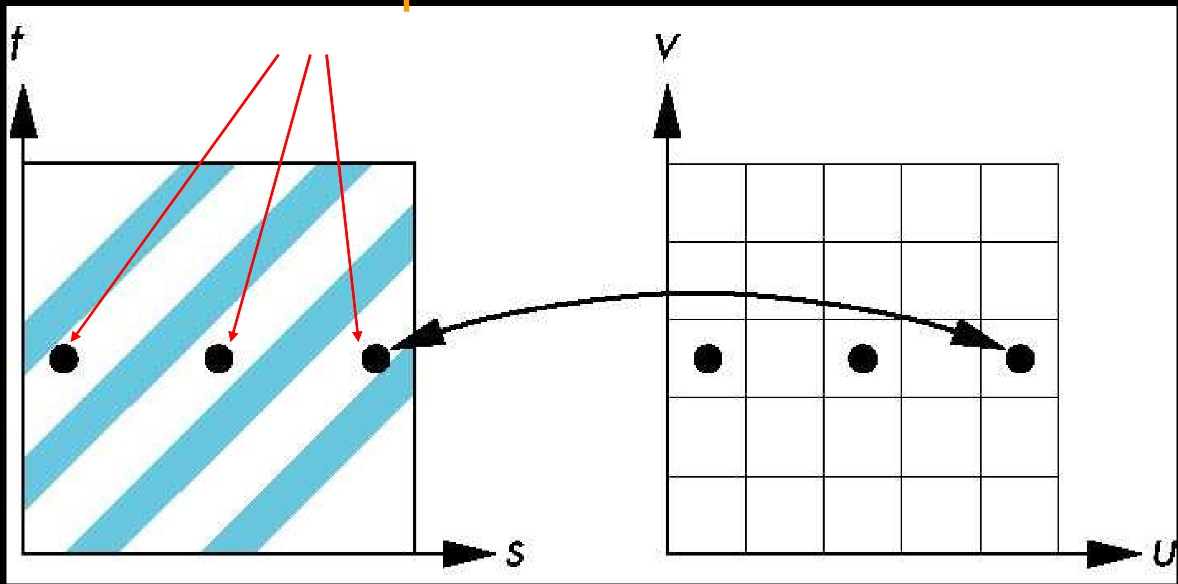


# Aliasing

- Point sampling of texture can lead to aliasing errors

point samples in u,v (or x,y,z) space

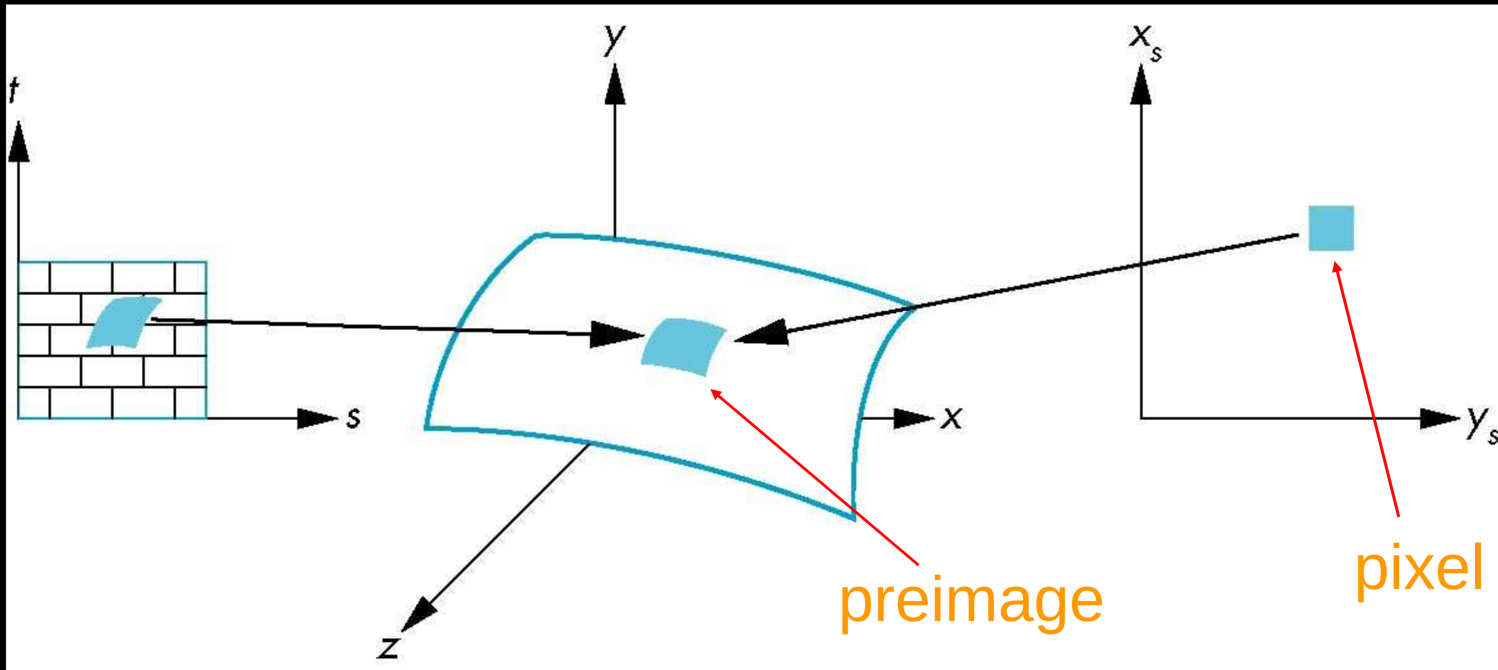
miss blue stripes




point samples in texture space

# Area averaging

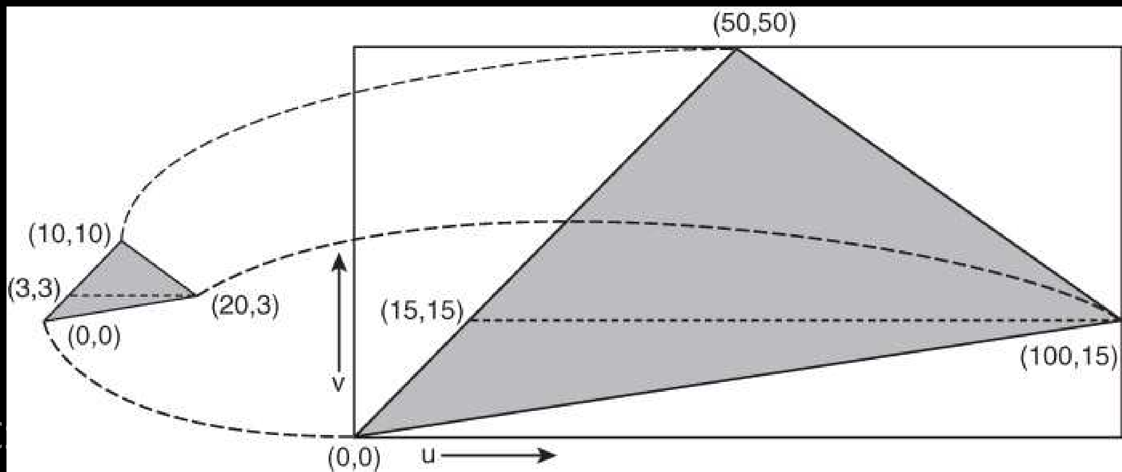
- Better but slower option




Note that *preimage* of pixel is curved

# Painted textures -- details

- Map locations on object into locations in texture
- Value from texture replaces diffuse component in shading calculations
- If texture > object,
  - only part of texture used, or
  - texture is shrunk
- If object > texture,
  - texture repeated across object or
  - texture stretched



# Painted textures

- If object location maps between texture locations, value between them can be interpolated

$$f_1 = \mathrm{fract}(u)$$

$$f_2 = \mathrm{fract}(v)$$

$$T_1 = (1.0 - f_1) * \mathrm{texture}[\mathrm{trunc}(u)][\mathrm{trunc}(v)] + f_1 * \mathrm{texture}[\mathrm{trunc}(u)+1][\mathrm{trunc}(v)]$$

$$T_2 = (1.0 - f_1) * \mathrm{texture}[\mathrm{trunc}(u)][\mathrm{trunc}(v)+1] + f_1 * \mathrm{texture}[\mathrm{trunc}(u)+1][\mathrm{trunc}(v)+1]$$

$$\mathrm{result} = (1.0 - f_2) * T_1 + f_2 * T_2$$

# Repeating textures

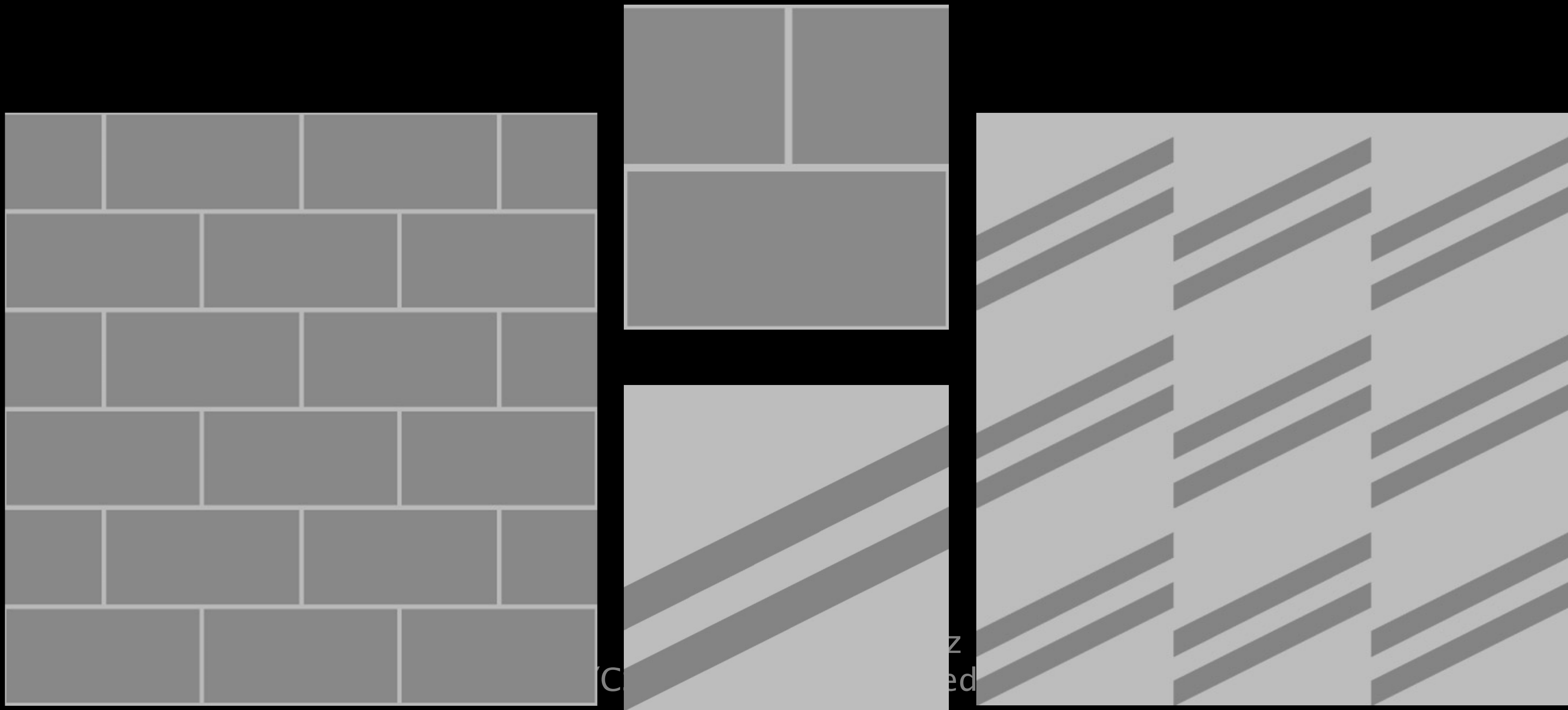
- A texture can be repeated across an object with the equations:

$$u = fract\left(\frac{repeat_x}{(x_{\max} - x_{\min})} * (x - x_{\min})\right) * T_u$$

$$v = fract\left(\frac{repeat_y}{(y_{\max} - y_{\min})} * (y - y_{\min})\right) * T_v$$

# Repeating textures

- Repeated texture must be continuous along edges to prevent obvious seams

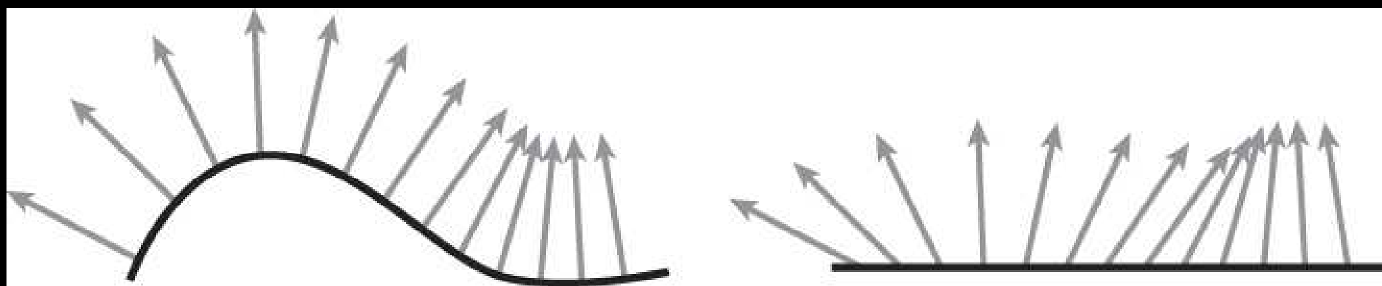


# Painted texture problem

- Texture values don't alter specular highlights
  - Specular highlighting may be inconsistent with texture appearance

# Bump Mapping

- For real bump, surface normal changes when moving across bump
- Similar appearance if similar change made to surface normal for a plane



# Bump Mapping

- ==> texture image called 'bump map'
- bump map used to alter surface normal
- altered surface normal used for color calculations

$$\mathbf{N_B} = \text{bump}(u, v, \mathbf{N})$$
$$\mathbf{R_B} = 2 * \mathbf{N_B} * (\mathbf{N_B} \cdot \mathbf{V}) - \mathbf{V}$$
$$C_r = k_{ar} + I_L * [k_{dr} * \mathbf{L} \cdot \mathbf{N_B} + k_s * (\mathbf{R_B} \cdot \mathbf{V})^n]$$
$$C_g = k_{ag} + I_L * [k_{dg} * \mathbf{L} \cdot \mathbf{N_B} + k_s * (\mathbf{R_B} \cdot \mathbf{V})^n]$$
$$C_b = k_{ab} + I_L * [k_{db} * \mathbf{L} \cdot \mathbf{N_B} + k_s * (\mathbf{R_B} \cdot \mathbf{V})^n]$$

# Bump Mapping

- bump function based on gradient of bump map
  - gradient = measure of how much bump map values change at chosen location
- modified normal calculated using gradients in two directions ($B_u$ and $B_v$) and two vectors tangent to surface in those directions (**$S_u$** and **$S_v$**)

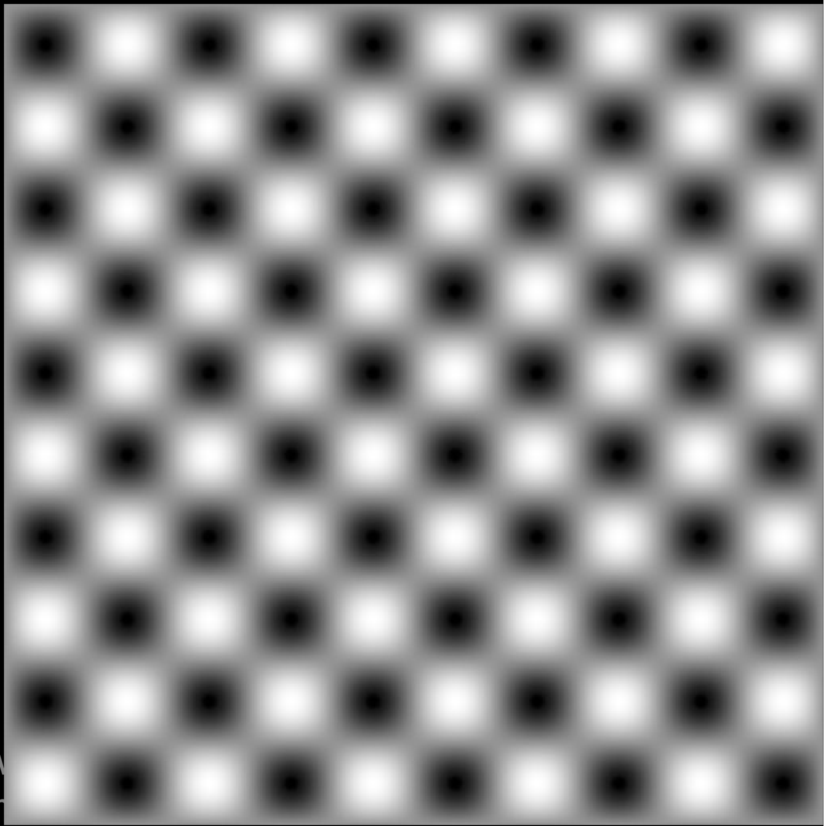
$$N' = N + \left( \frac{B_u * S_v}{|S_v|} + \frac{B_v * S_u}{|S_u|} \right)$$

- parenthesized expression can also be multiplied by factor to control appearance of size of bumps

# Bump Mapping Examples

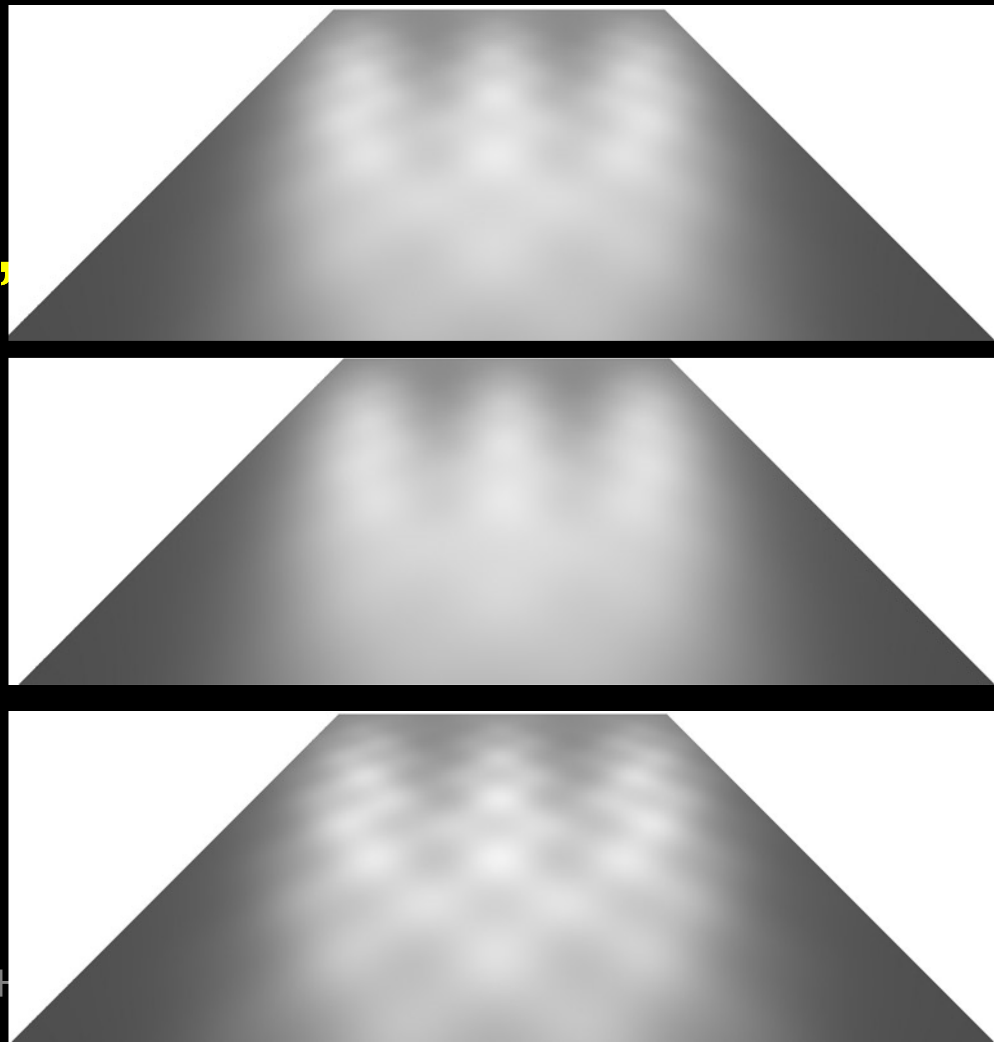
- Bump map based on product of sine taken of two coordinates:

  $\sin(10 * u * \pi / 1024)$
  $* \sin(10 * v * \pi / 1024)$



# Bump Mapping Examples

- Bump mapped plane using factors (h) of 1.0, 0.5, and 2.0
- 3 light sources illuminate surface patch

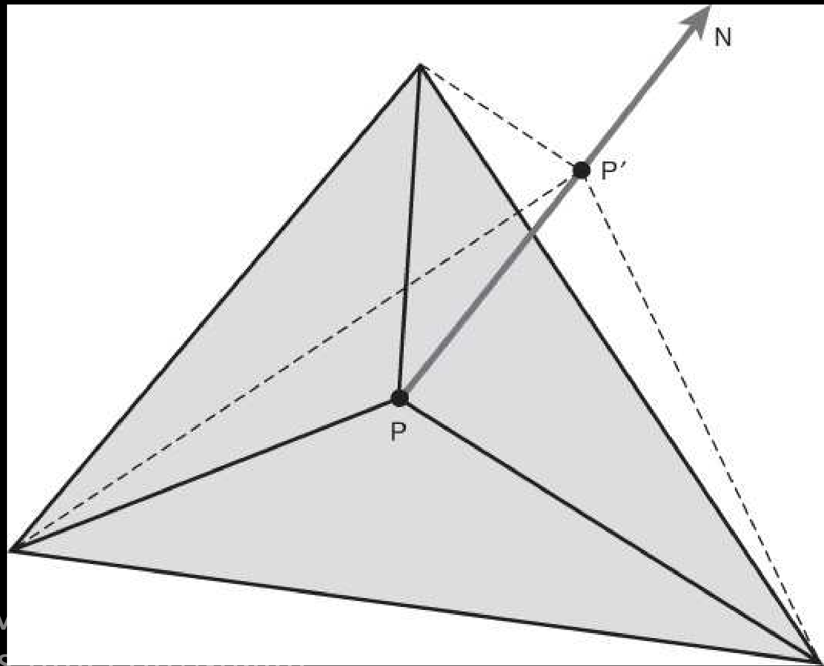


# Bump mapping problem

- Surface shape not changed ==> “bumps” near silhouette will not change silhouette shape
- If bump-mapped object rotated, bumps will disappear when they reach object profile

# Displacement mapping

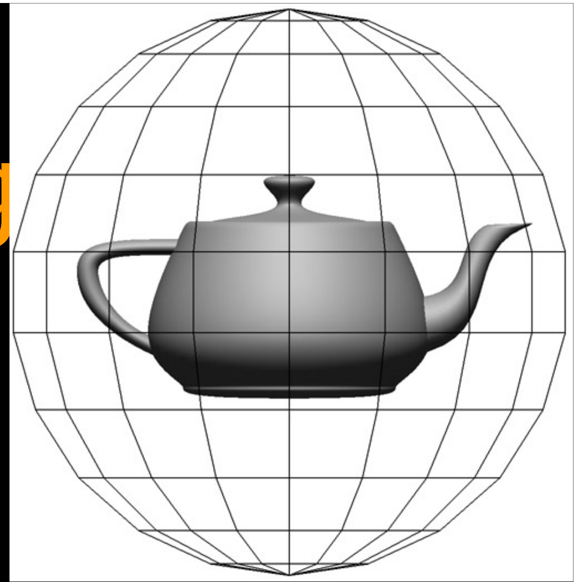
- In displacement mapping, object is subdivided into many very small polygons
- Texture values cause vertices to be displaced in direction of surface normal
- In this case, texture actually changes object shape



# Environment mapping

- Environment mapping simulates reflective objects
- Environment map
  - = rendering of scene from inside reflective object
  - used to determine what would be seen in reflection direction

# Environment mapping



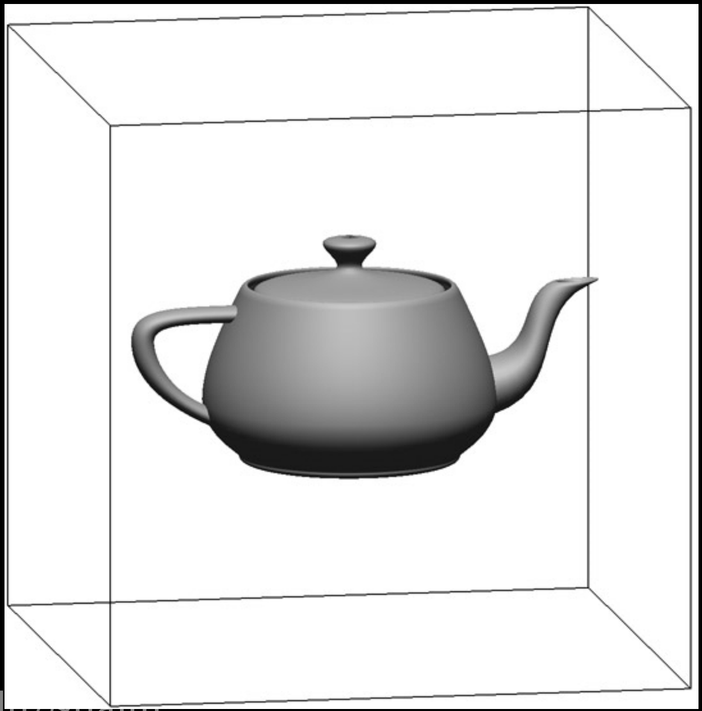
- Environment map can be
  - Spherical,
    - require calculation to convert direction through sphere into 2-dim matrix location
  - or
  - cube shaped …

$$u = \frac{1}{2}\left(1 + \frac{1}{\pi} * \arctan\left(\frac{R_x}{R_y}\right)\right)$$

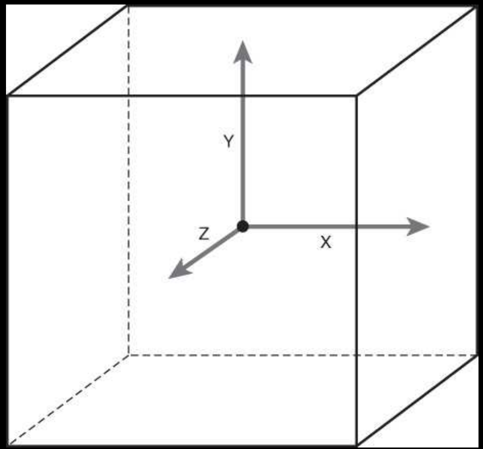
$$v = \frac{R_z + 1}{2}$$

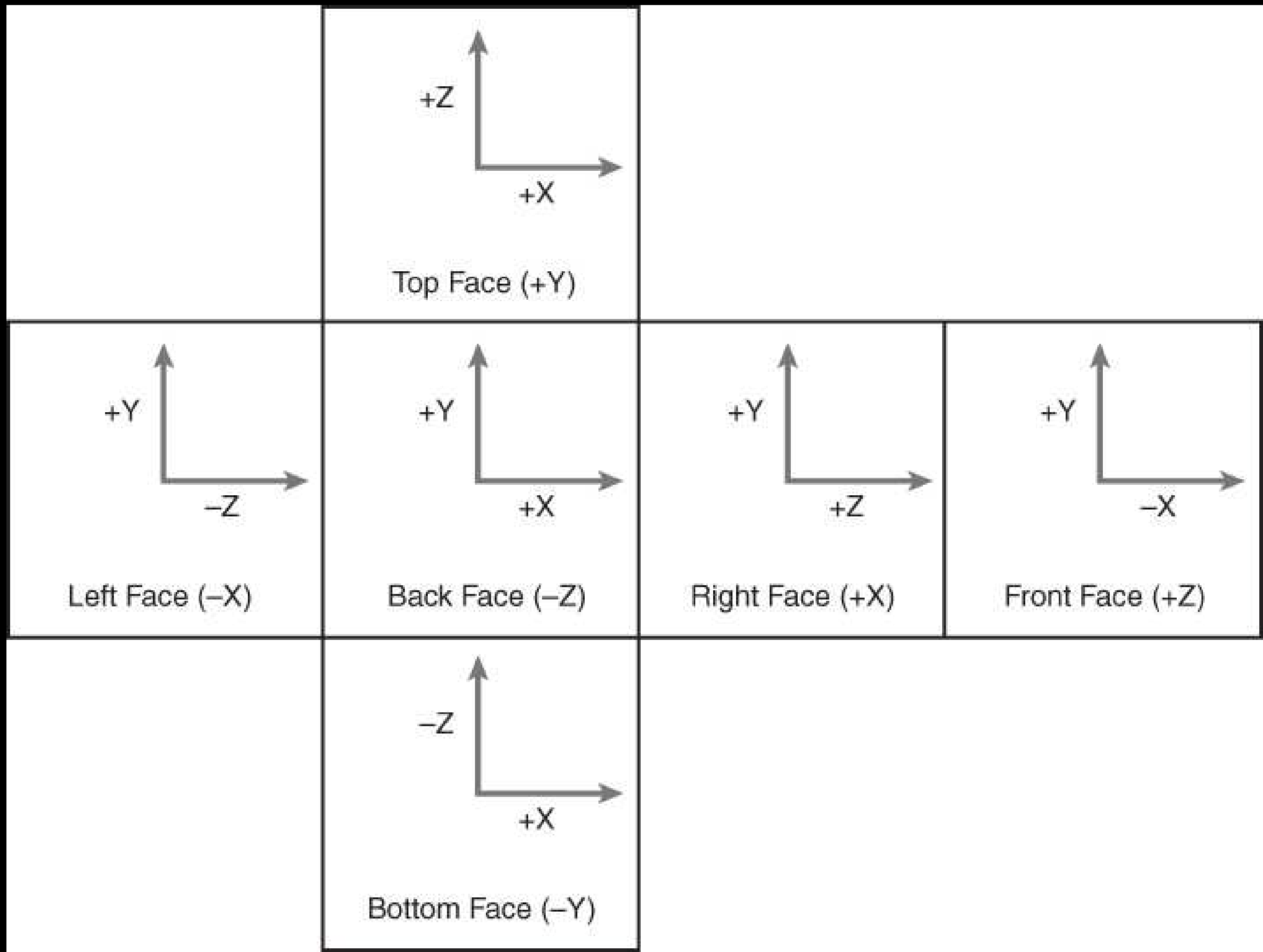
# Environment mapping

- Cube shaped environment map
  - renders scene from center of object onto faces of a cube
- component of reflection vector with largest absolute value determines face reflection vector goes through



# Environment Mapping



# Environment Mapping

- Location on correct face is calculated by:

$$u = \frac{a + c}{2c}$$

$$v = \frac{b + c}{2c}$$

- Where
  - a = component of reflection vector shown on horizontal axis,
  - b = component shown for vertical axis, and
  - c = component for chosen face

# Three-Dimensional Textures

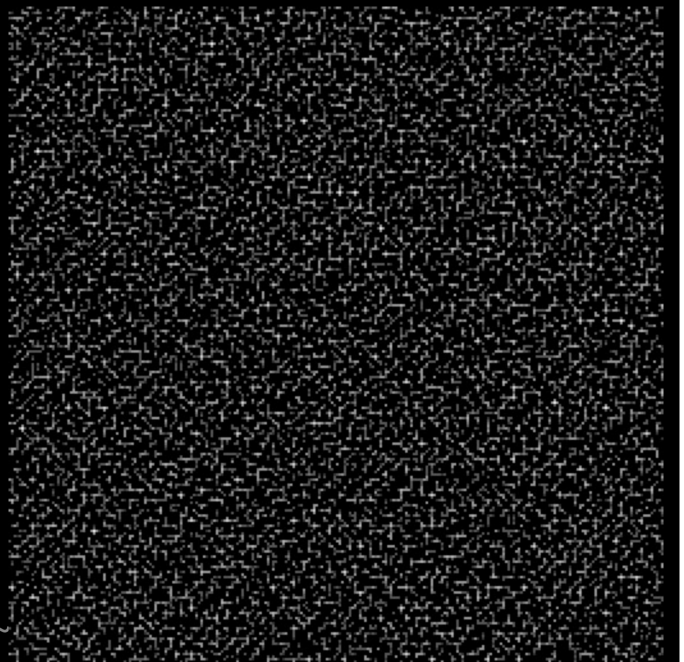
- Very difficult (impossible?) to create texture that can be wrapped around irregularly shaped object without visible discontinuity
- 2-D texture applied to surface ==>
  - object can be thought to be carved out of 3-D texture
- Storage requirements for 3-D texture extremely large
- ==> 3-D textures closely related to functional textures

# Functional Textures

- Calculation based on texture location
  - Instead of stored image as texture
- Function continuous in every direction
  - ==> texture continuous in every direction
- Space <---> computation time
- Change how texture calculated
  - ==> many different textures can be created

# Noise

- 1st step : noise function
- True white noise is highly random
- For graphics, pseudo random noise that is repeatable is important
  - Truly random noise ==> texture would change
    - every time image is rendered
    - or for every scene of animation



# Perlin Noise

- Perlin: developed noise function
  - used in many Hollywood movies
- This noise function has:
  - No statistical variance when rotated
  - No statistical variance when translated
  - Narrow range of values

# Turbulence

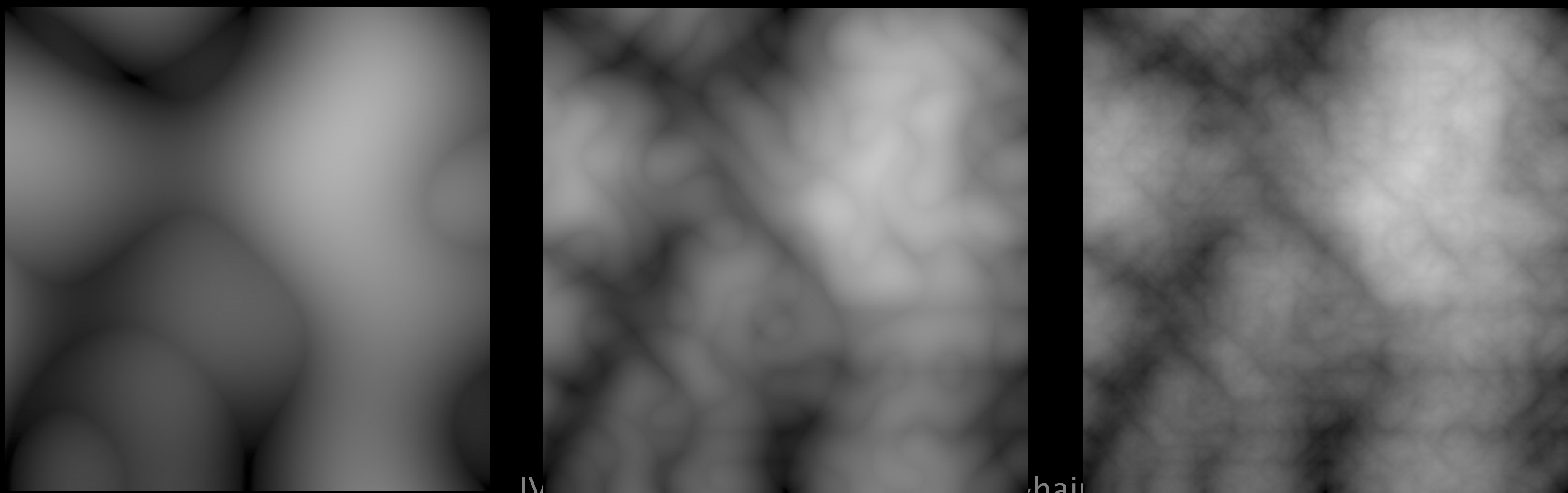
- 2nd step : build turbulence function on top of noise function
- Functional textures then built on top of turbulence function
- Turbulence function takes multiple samples of noise function at many different frequencies
- Different researchers frequently develop own turbulence function

# Peachy's Turbulence Function

```
float turb(float x, float y, float z, float minFreq,
  float maxFreq)
{
  float result = 0.0;
  for (float freq = minFreq; freq < maxFreq; freq =
  2.0*freq)
  {
        result += fabs( noise(x*freq, y*freq, z*freq ) /
  freq );
  }
  return result;
}
```

# Peachy's Turbulence Results

- Samples have frequency ranges of [1.0, 4.0], [1.0, 16.0], and [1.0, 256.0]

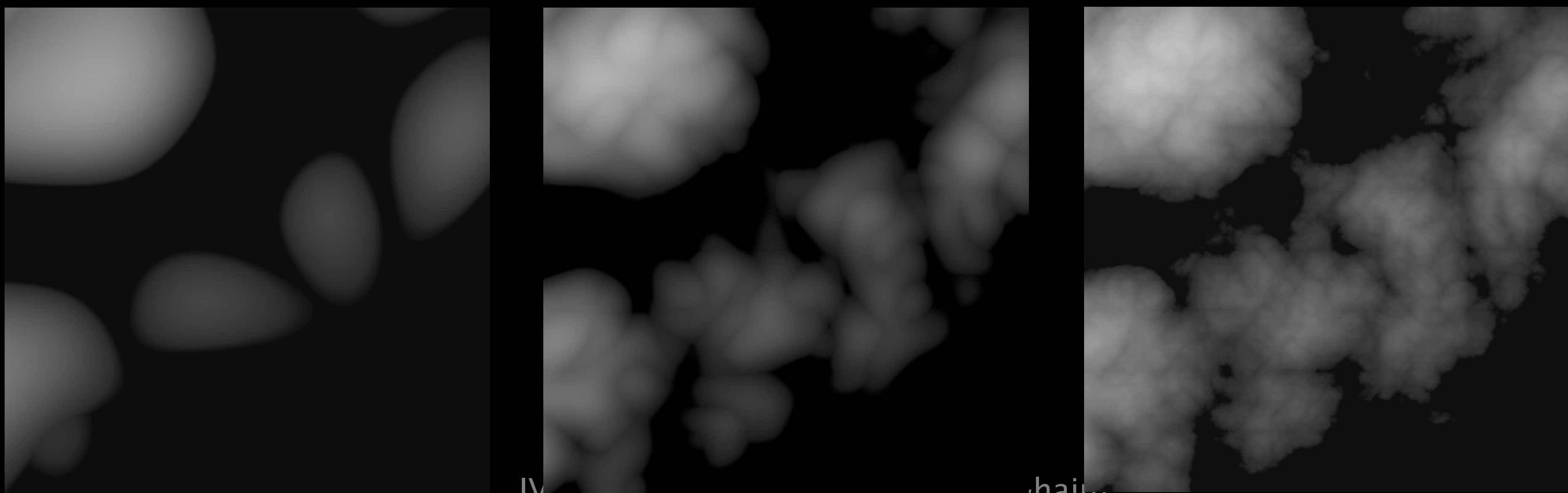


# Perlin's Turbulence Function

```
float turb(float x, float y, float z, float minFreq, float
  maxFreq) // minFreq = lowest allowed; maxFreq =
  image resolution
{
  float result = 0.0;
  x = x + 123.456;
  for (float freq = minFreq; freq < maxFreq; freq =
  2.0*freq)
  {
        result += fabs( noise(x, y, z ) ) / freq;
        x *= 2.0;
        y *= 2.0;
        z *= 2.0;
  }
  // return the result adjusted so the mean is 0.0
  return result-0.3;
}
```

# Perlin's Turbulence Results

- Samples have frequency ranges of [1.0, 4.0], [1.0, 16.0], and [1.0, 256.0]



# Marble Texture

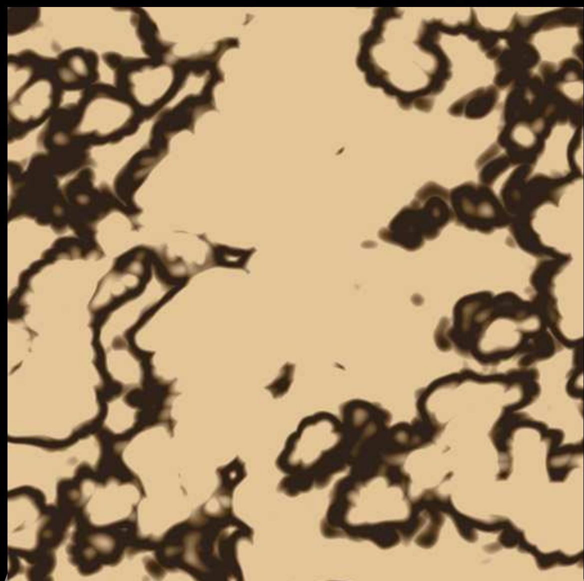
- Perlin/Ebert marble function uses turbulence value to determine color for location

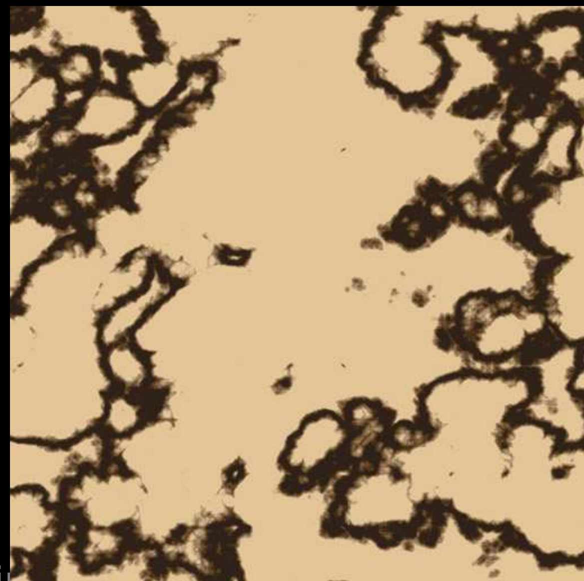
```
void marble(float x, float y, float z, float color[3])
{
  float value = x + 3.0 * turb(x, y, z, minFreq, maxFreq);
  marbleColor( sin(π * value), color );
}
```

- Variations how marbleColor function implemented
  - Linear interpolation between light and dark color
  - Spline-based interpolation between light and dark color

# Linear Interpolation Marble Examples

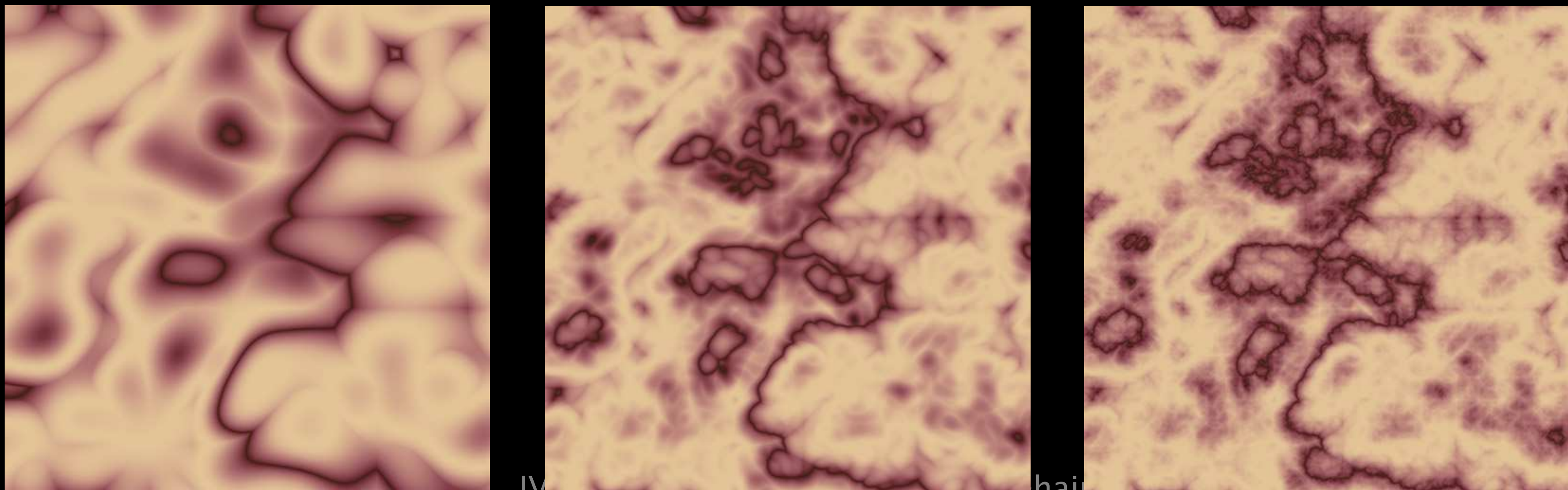
- Samples have frequency ranges of [1.0, 4.0], [1.0, 16.0], and [1.0, 256.0]





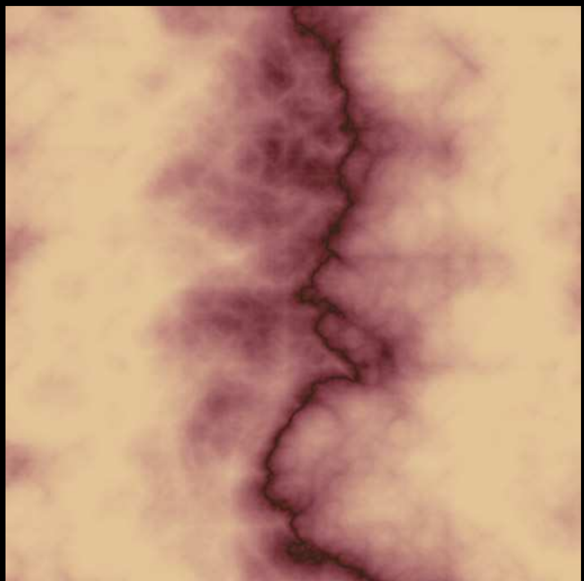
# Spline Interpolation Marble Examples

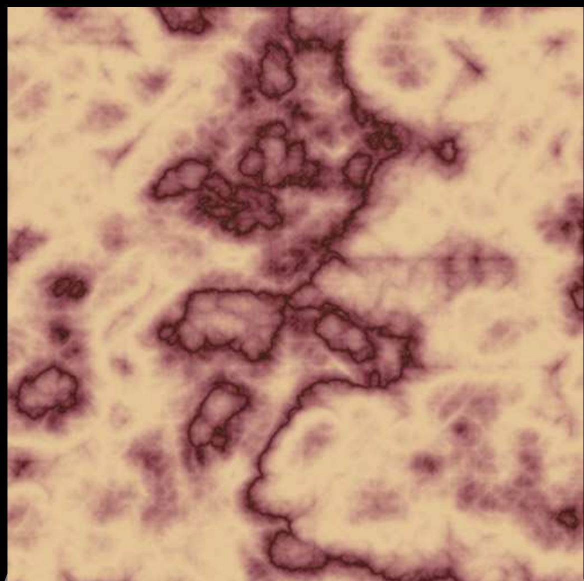
- Samples have frequency ranges of [1.0, 4.0], [1.0, 16.0], and [1.0, 256.0]

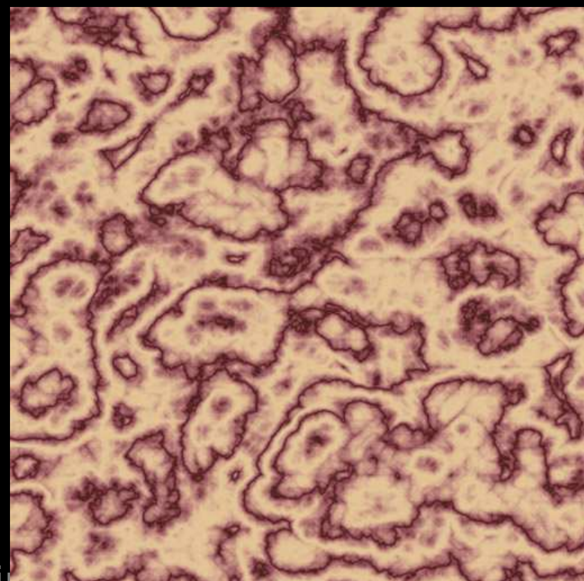


# Spline Interpolation Marble Examples

- Samples have a frequency range of [1.0, 256.0] and have no multiplier, and multipliers of 3 and 7







# Cosine Textures

- Based on summations of cosine curves
- Parameters in equation provide control on result
- 2-D equation is:

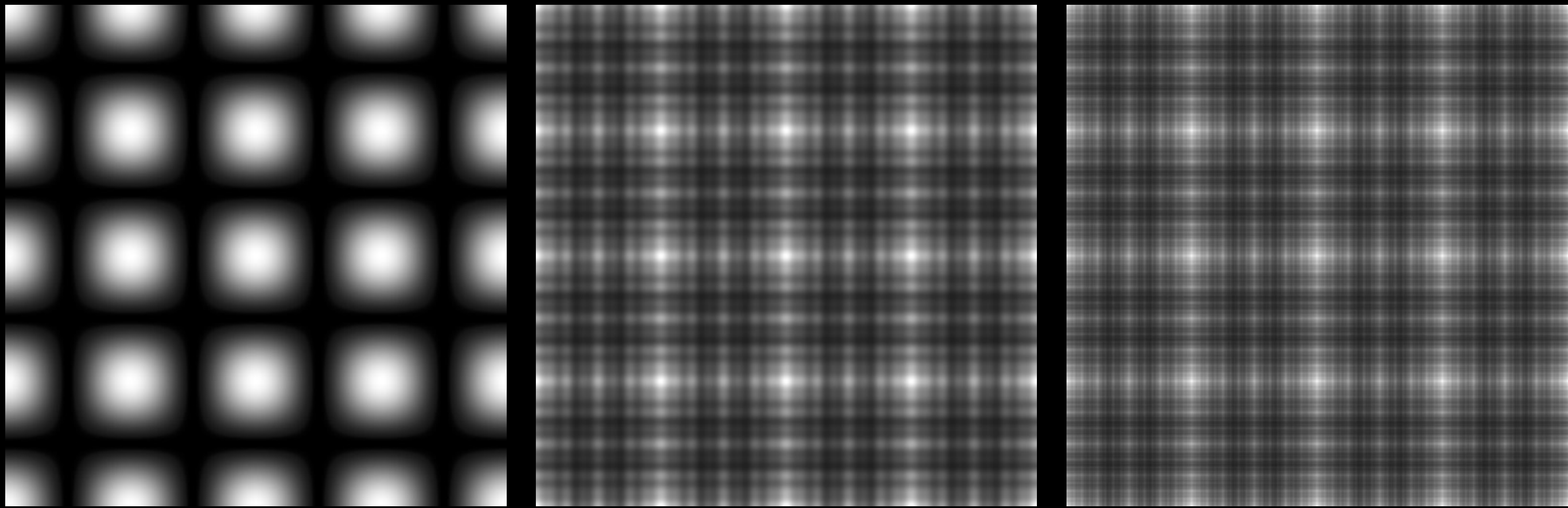
$$f(x,y) = \sum_{i=1}^{N} C_i * \left[\cos\left(\omega_{x_i} * \left(x + \Phi_{Gx}\right) + \phi_{x_i}\right) + A_0\right] * \sum_{i=1}^{N} C_i * \left[\cos\left(\omega_{y_i} * \left(y + \Phi_{Gy}\right) + \phi_{y_i}\right) + A_0\right]$$

# Cosine Texture Parameters

- N controls # cos terms
  - Typically 4 to 7
- Cons'ts $\Phi_{Gx}$ $\Phi_{Gy}$ = global phase shifts
  - move pattern
- $C_i$ terms change amp of various cos com's
- $A_0$ terms shift pattern but related to $C_i$ terms
- $\omega_{x1}$ & $\omega_{y1}$ determine # times pattern repeats
- $\phi_{x_i}$ and $\phi_{y_i}$ = phase values
  - interdependent with base periods of x & y

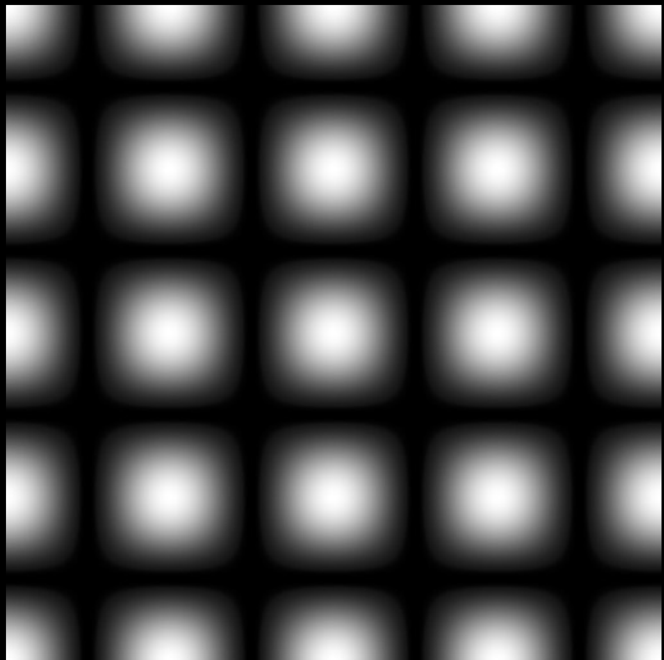
# Number of Cosines

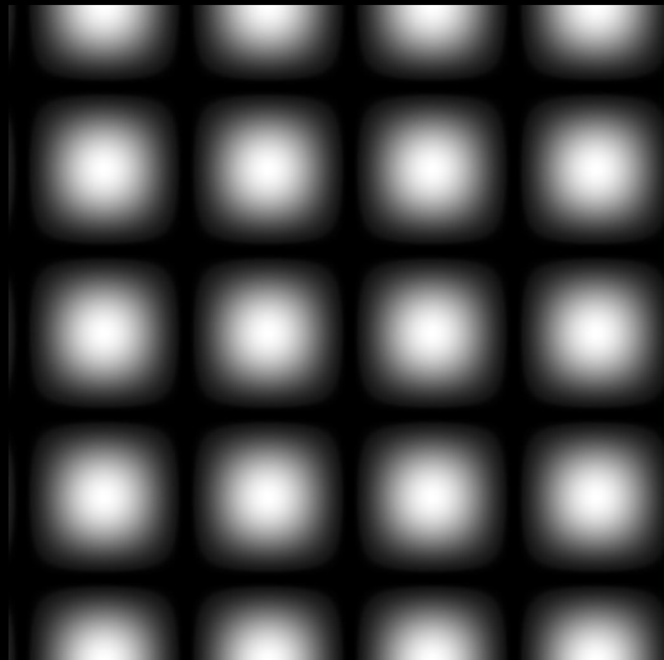
- Examples show the results of using 1, 4, and 7 cosines



# Global Phase Shift

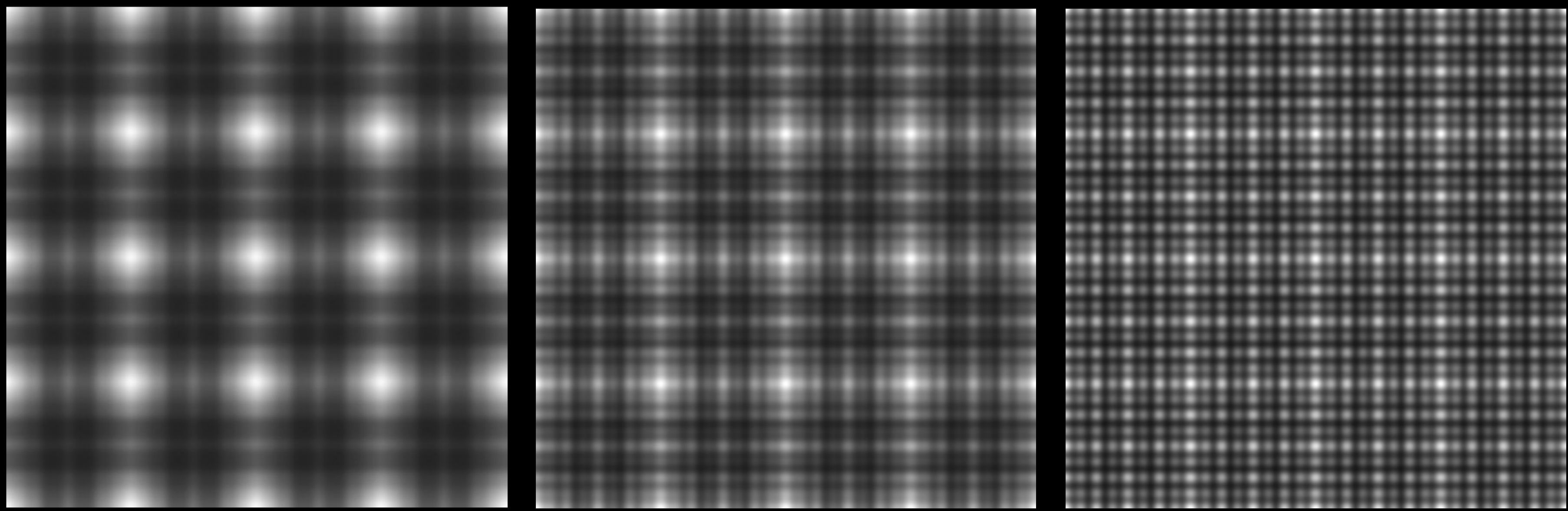
- Examples with x global phase shifts of 0 and 150 show that global phase shift moves the pattern





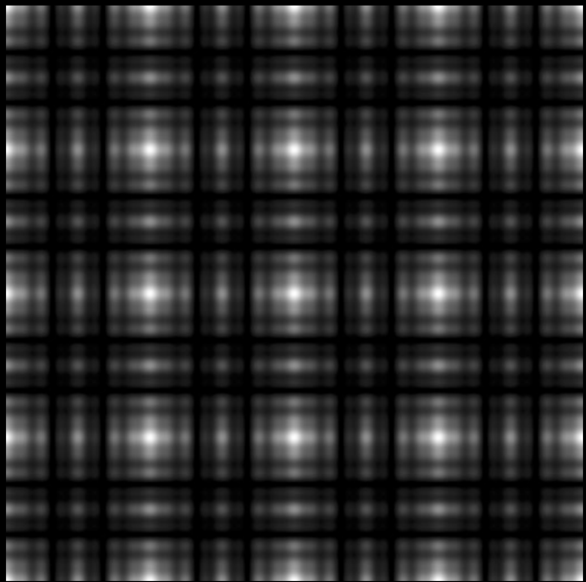
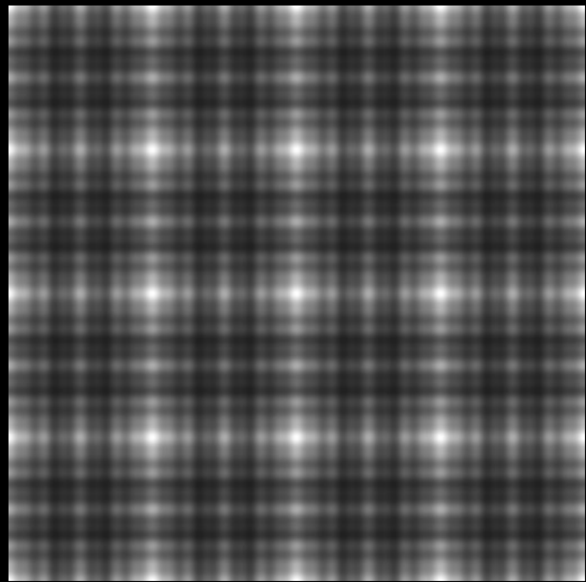
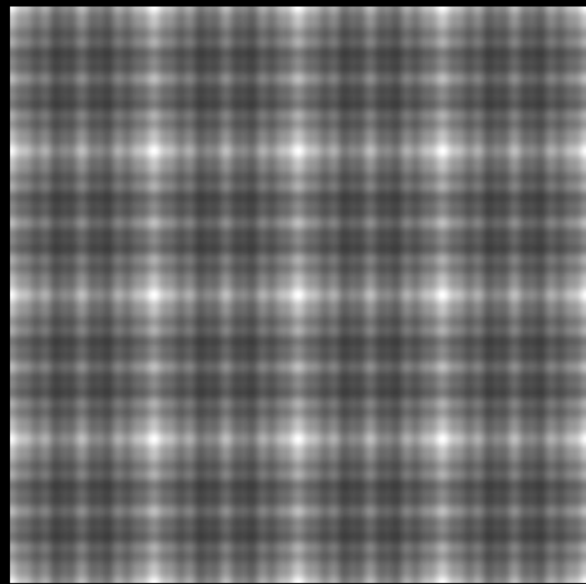
# Cosine Amplitude Ratio

- Examples show cosine amplitude ratios of 0.3535, 0.7070, and 1.414



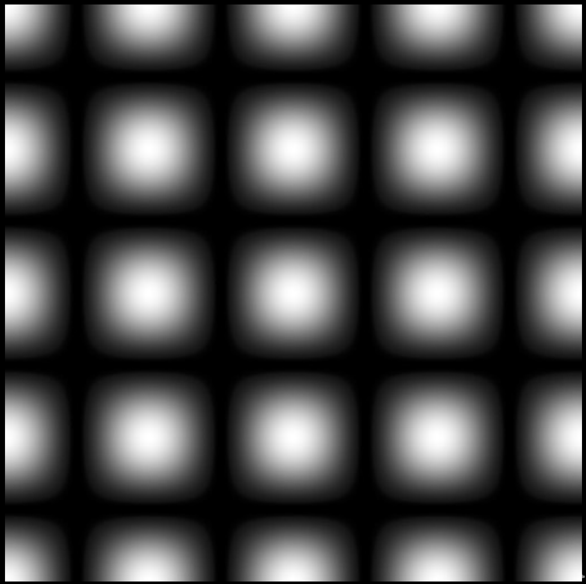
# Cosine Offset

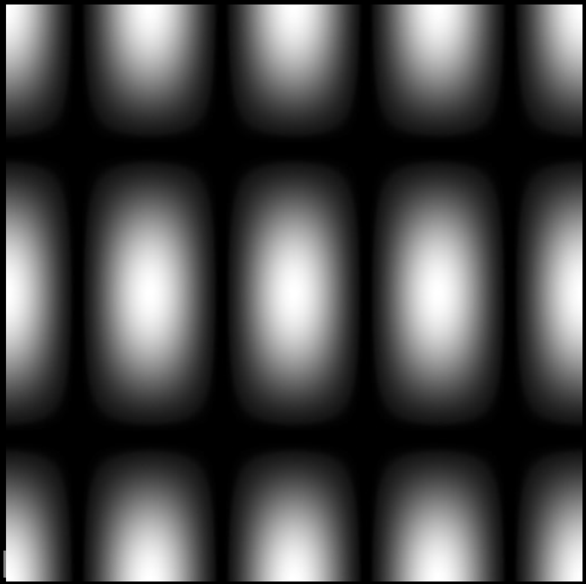
- Examples show cosine offset values of 0.5, 1.0, and 1.5

# Base Periods

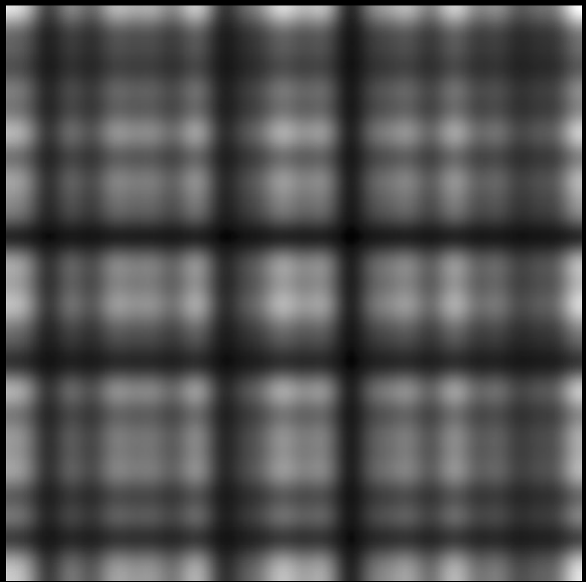
- 1st example : same base period (256) for x and y; 2nd : x base period of 256 & y base period of 512

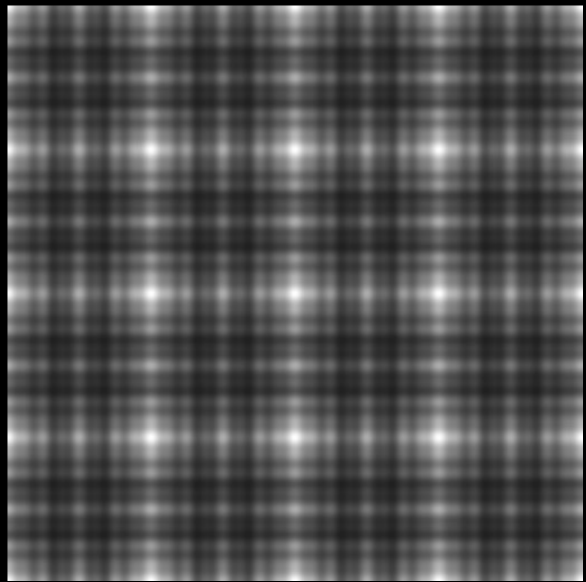


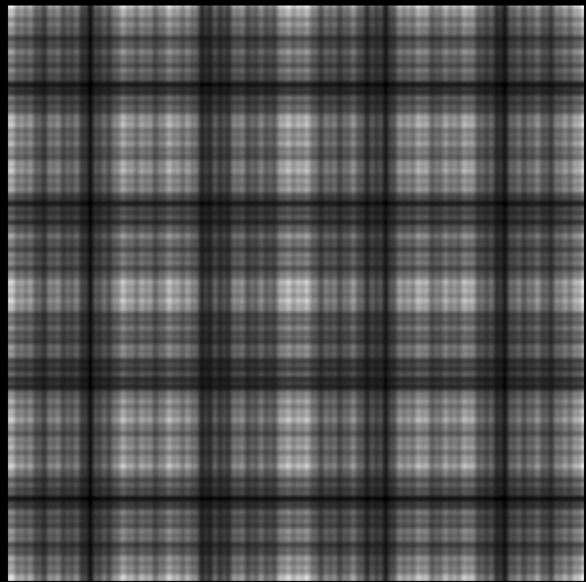


# Base Period Ratio

- Examples have base period ratios of 1.5, 2.0, and 2.5

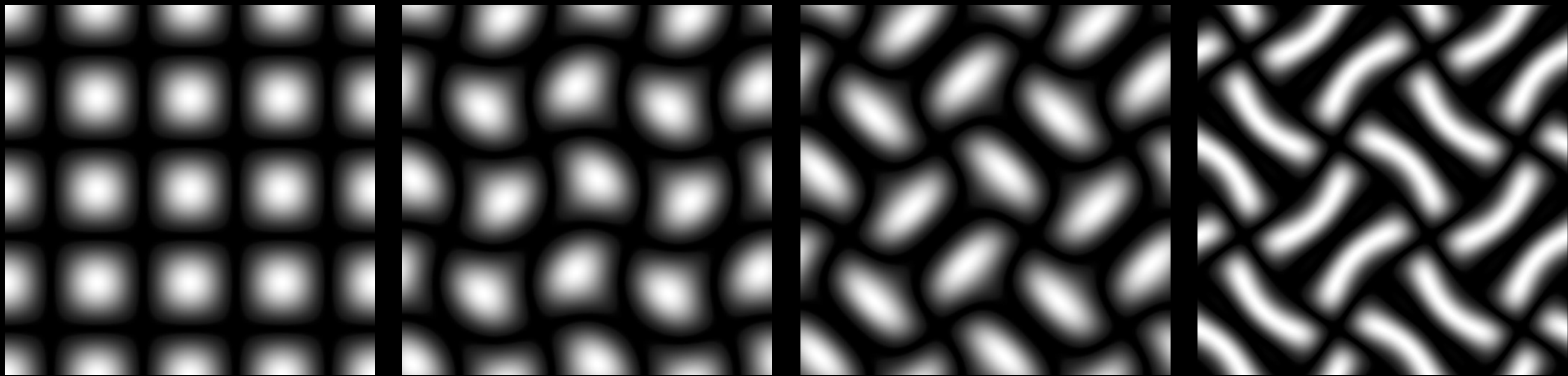






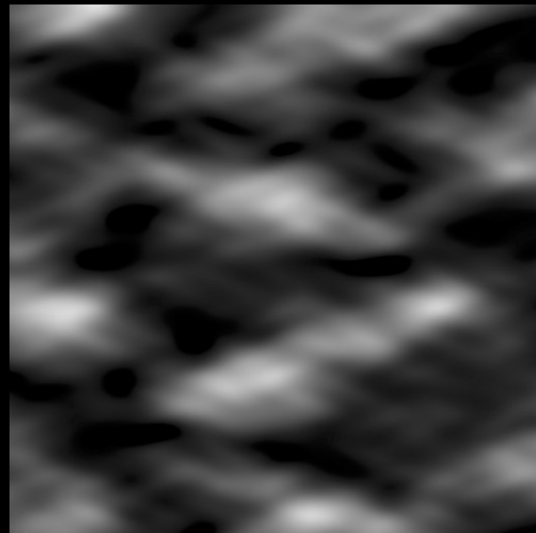
# Phase Shift Amplitude

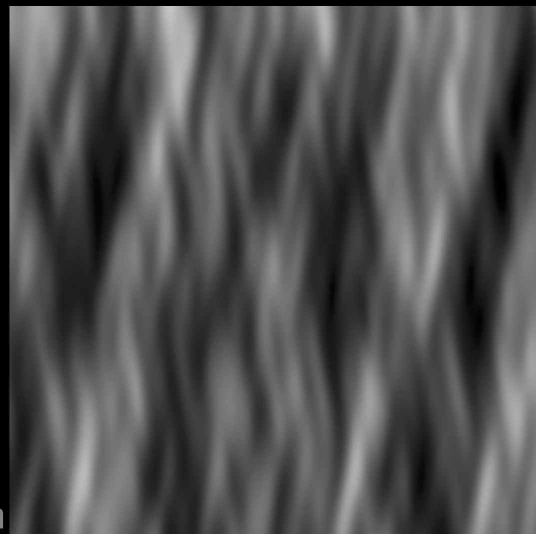
- Examples have phase shift amplitudes of 0, π/4, π/2, and π



# Cosine Texture Results

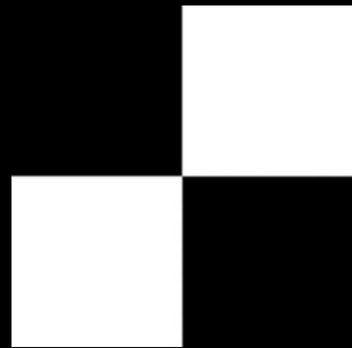
- Cloud-like texture possible with 4 cosine components, x global offset of 200, cosine amplitude ratio of 0.5, phase shift amplitude of $\pi/2$, cosine offset of 0.75, x base period of 655, y base period of 325, and base period ratio of 1.7
- Bark-like texture possible with 4 cosine components, cosine amplitude ratio of 0.707, cosine offset of 1.0, x base period of 256, y base period of 1216, and base period ratio of 1.7

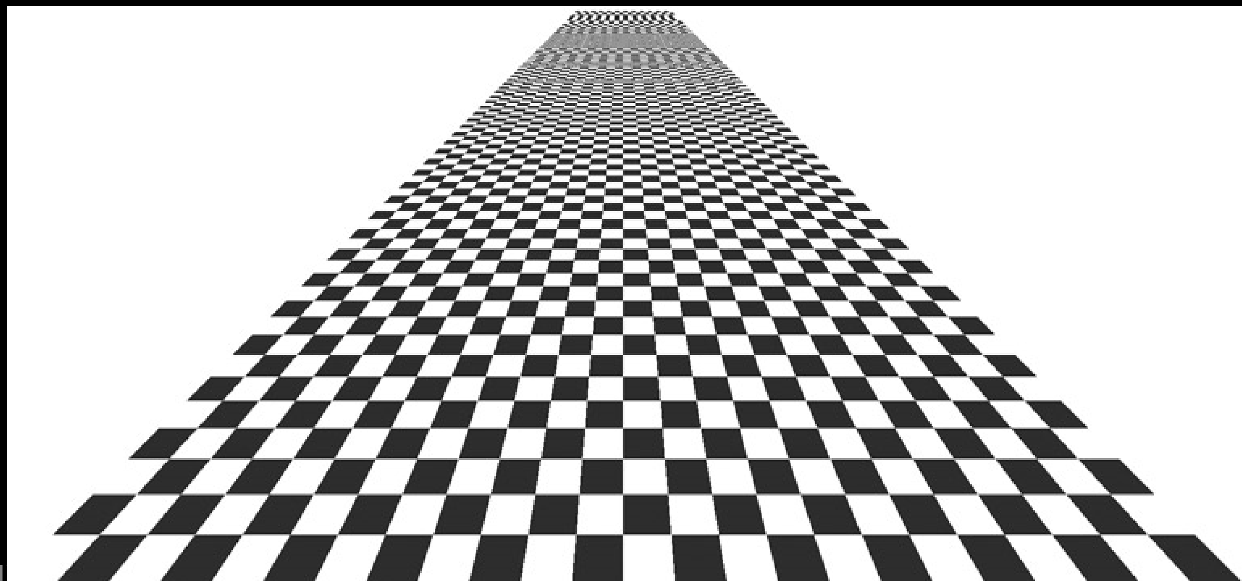




# Antialiasing Textures

- Frequency of entity & rate sampled can cause visible artifacts
- Example: appears like multiple textures used
- Aliasing causes visible artifacts





# Aliasing Corrections

- Aliasing artifacts: reduced by altering sampling method
- Common techniques include
  - Supersampling
  - Supersampling with jittering
  - Inverse mapping
- Techniques discussed re textures
  - Apply to other antialiasing applications

# Supersampling

- Multiple evenly spaced samples taken from texture for each location
- Average / weighted average taken of samples to produce final result
- One weighted average uses 9 samples and filter (weighting):

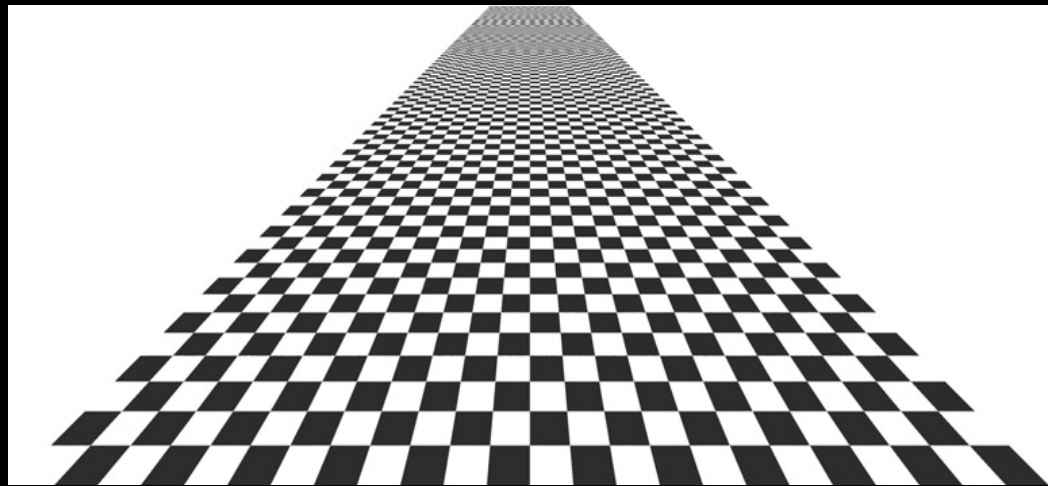
| | | |
|---|---|---|
| 1 | 2 | 1 |
| 2 | 4 | 2 |
| 1 | 2 | 1 |

# Supersampling With Jittering

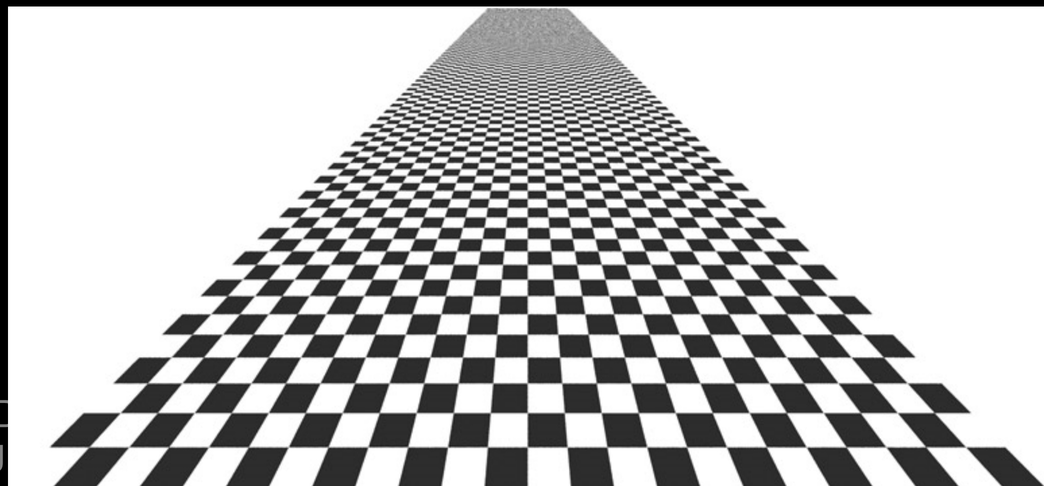
- Sample locations shifted by random amount
  - Instead of evenly-spaced samples
- ==> helps disturb regular sampling frequency

# Supersampling Examples

- Supersampling

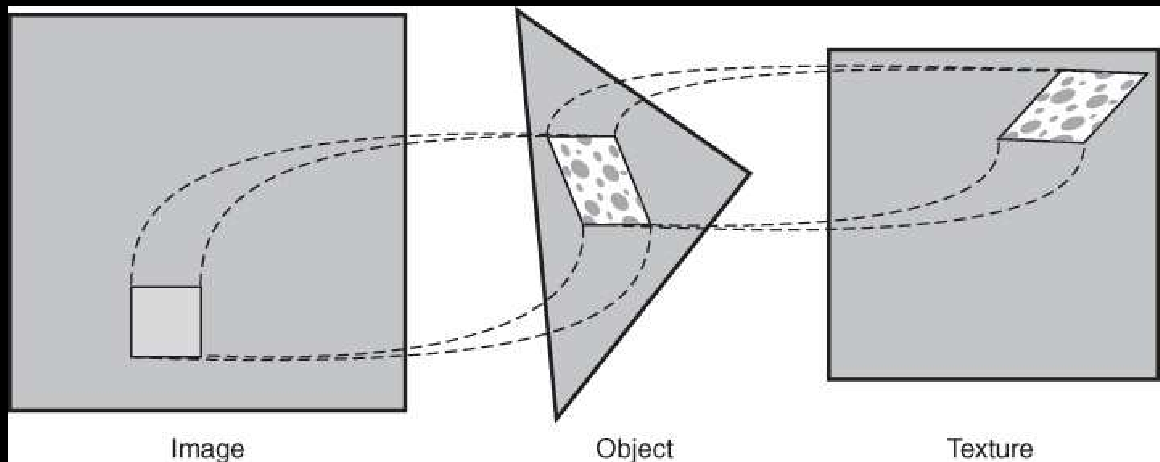


- Supersampling with jittering

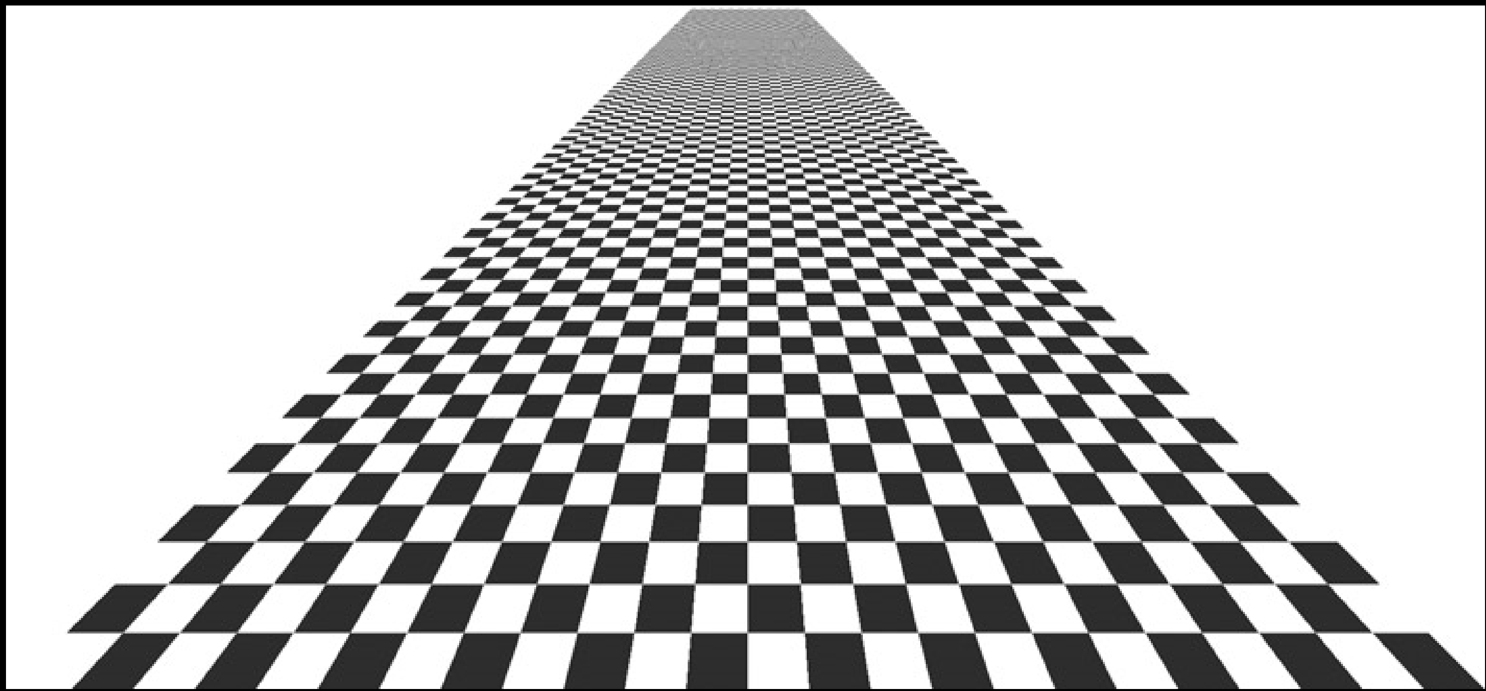


# Inverse Mapping

- Area of pixel projected back onto object
- That object area projected back into texture
- Integrating or averaging values in this area of texture ==> final texture result to be used



# Inverse Mapping Example



# OpenGL Texture

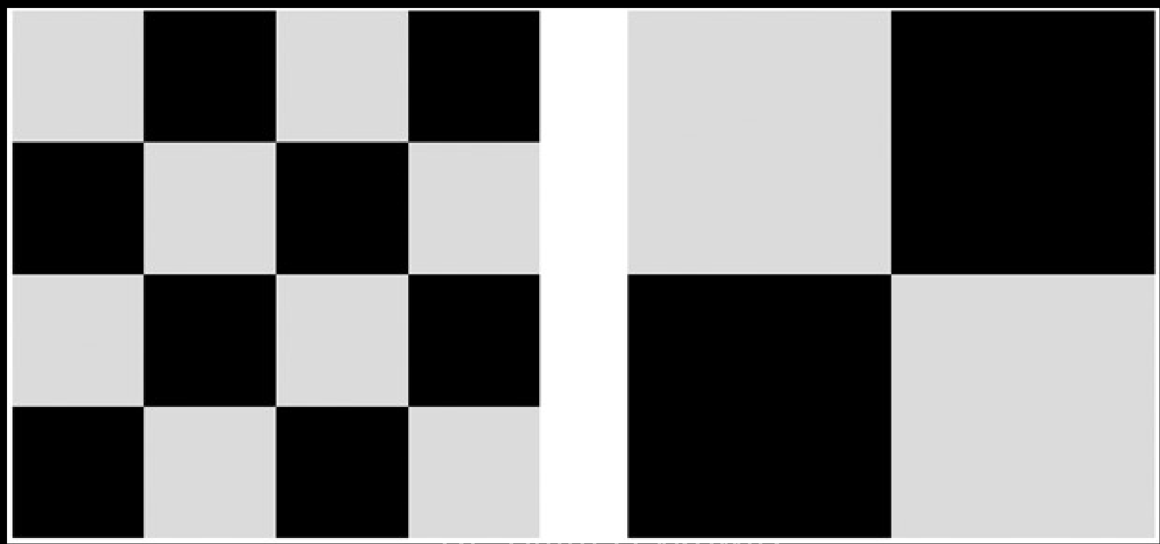
- Steps to using texture in OpenGL :
  - Create / load texture into OpenGL
  - Enable texturing
  - Specify how texture is to be used
  - Specify texture coordinates for polygon vertices or surface corners
- Allows programmer to specify if texture is to be added before or after specular highlight
- Allows textures of varying resolutions to deal with aliasing problems
- Also support for environment mapping

# Texture Coordinates

- Texture coordinates
  - influence how texture is applied
  - are in range [0.0, 1.0]
- If OpenGL told to repeat texture
  - ==> values outside of range cause multiple copies of texture used
- If OpenGL told to clamp coordinates
  - ==> any values
    - below range set to 0.0
    - above range set to 1.0

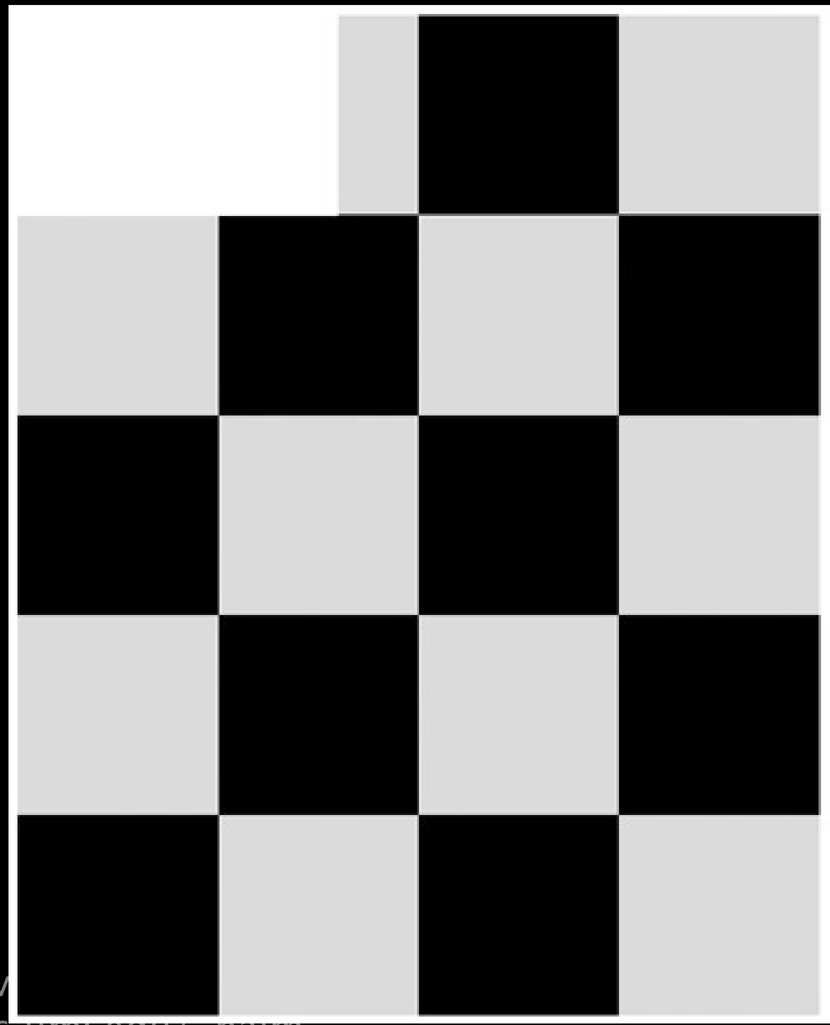
# Texture Coordinates

- In 1st example, texture coordinates in range [0.0, 1.0]
- In 2nd example, texture coordinates in range [0.0, 0.5]



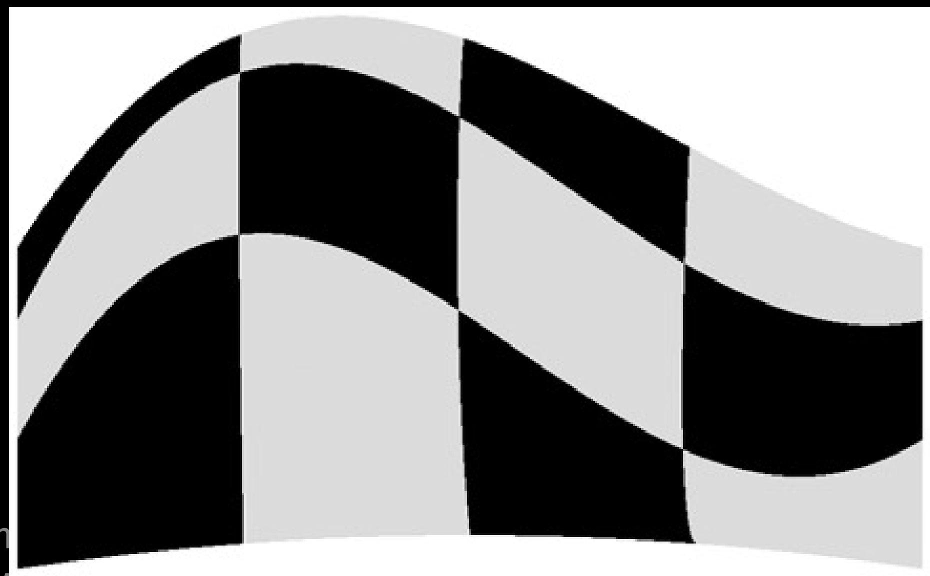
# Texture Coordinates

- By picking appropriate texture coordinates, two adjacent planes can be textured seamlessly



# Bézier Surface Texturing

- Bézier surfaces can be textured if points also supplied for four corner control points



# Specular Highlights

- OpenGL can include specular component
  - before texturing
  - after texturing