Texture Mapping

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

- Introduce Mapping Methods
  - Texture Mapping
  - Environment Mapping
  - Bump Mapping
- Consider basic strategies
  - Forward vs backward mapping
  - Point sampling vs area averaging

The Limits of Geometric Modeling

- Although graphics cards can render over 10 million polygons per second, that number is insufficient for many phenomena
  - Clouds
  - Grass
  - Terrain
  - Skin

Modeling an Orange (1)

- Consider the problem of modeling an orange (the fruit)
- Start with an orange-colored sphere
  - Too simple
- Replace sphere with a more complex shape
  - Does not capture surface characteristics (small dimples)
    Takes too many polygons to model all the dimples

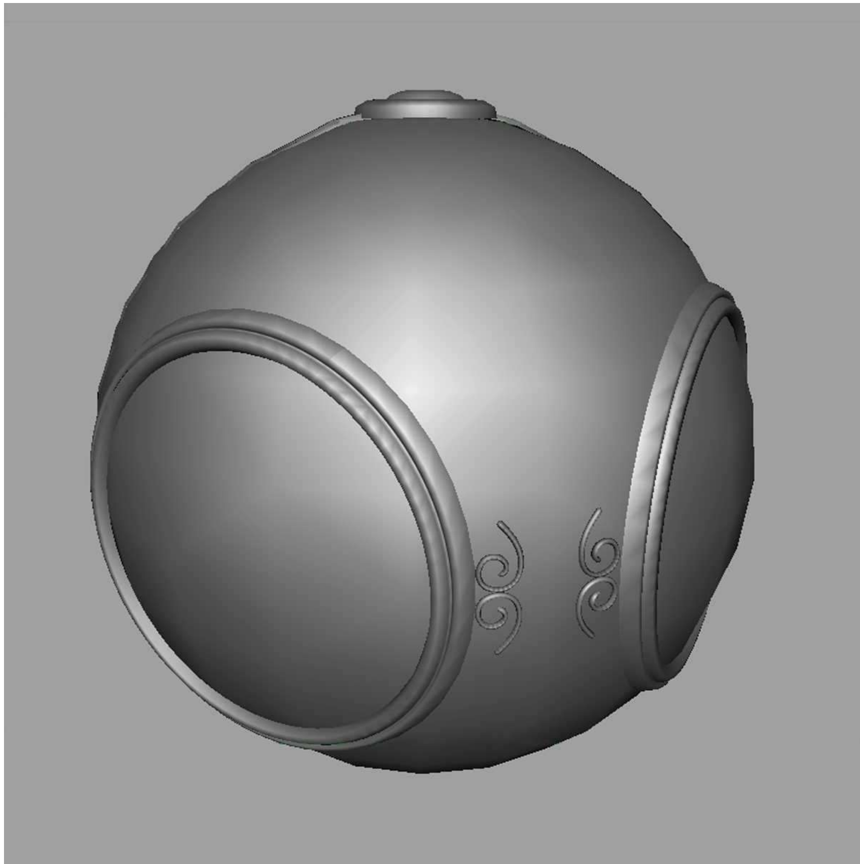
Modeling an Orange (2)

- Take a picture of a real orange, scan it, and “paste” onto simple geometric model
  - This process is known as texture mapping
- Still might not be sufficient because resulting surface will be smooth
  - Need to change local shape
  - Bump mapping

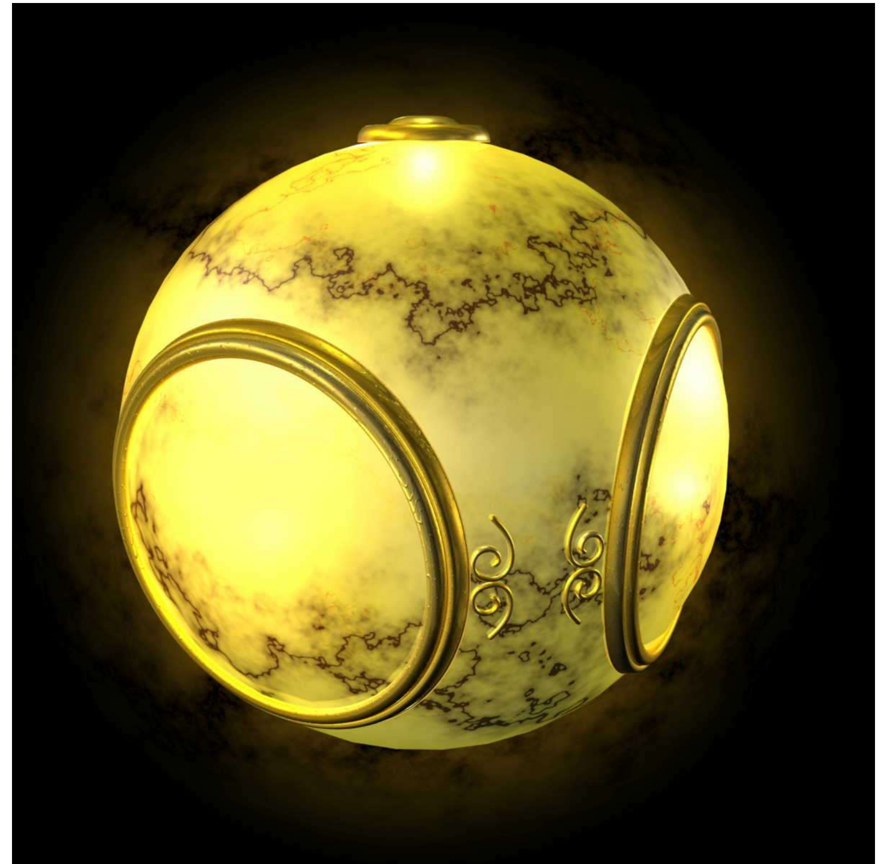
Three Types of Mapping

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

Texture Mapping



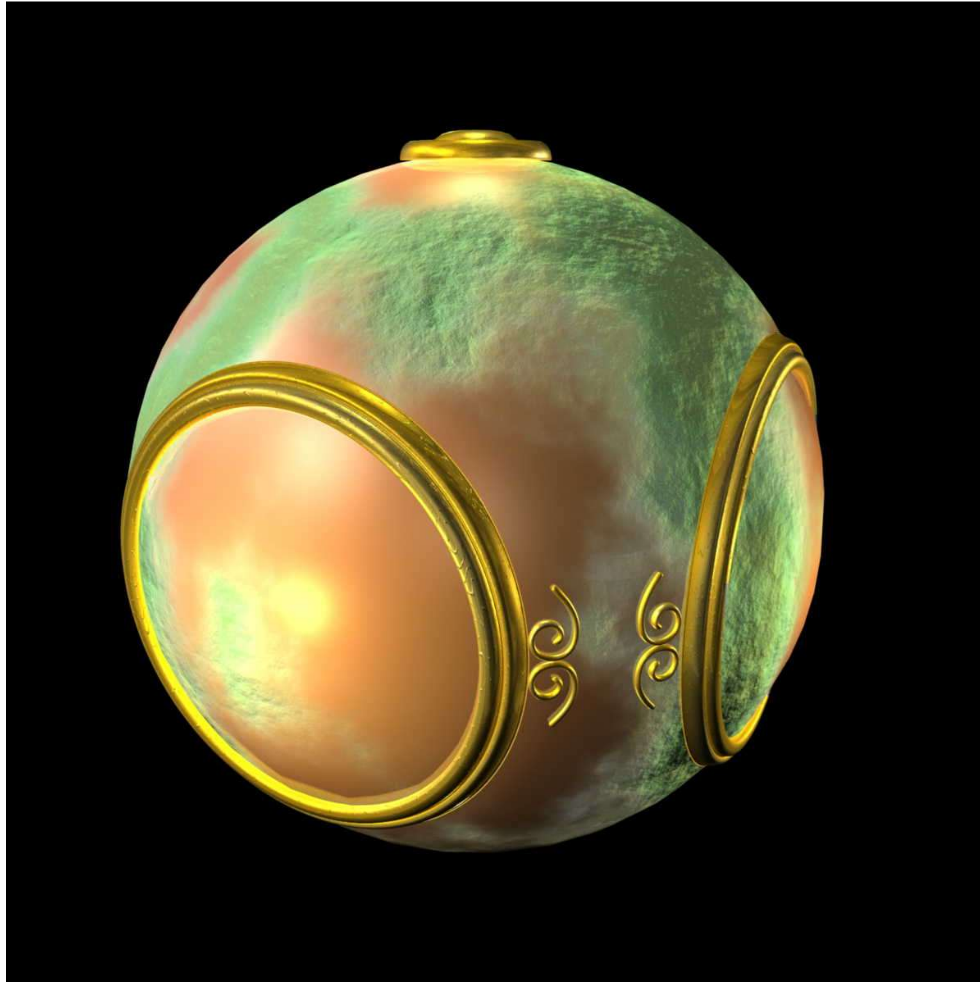
geometric model



texture mapped

Environment Mapping

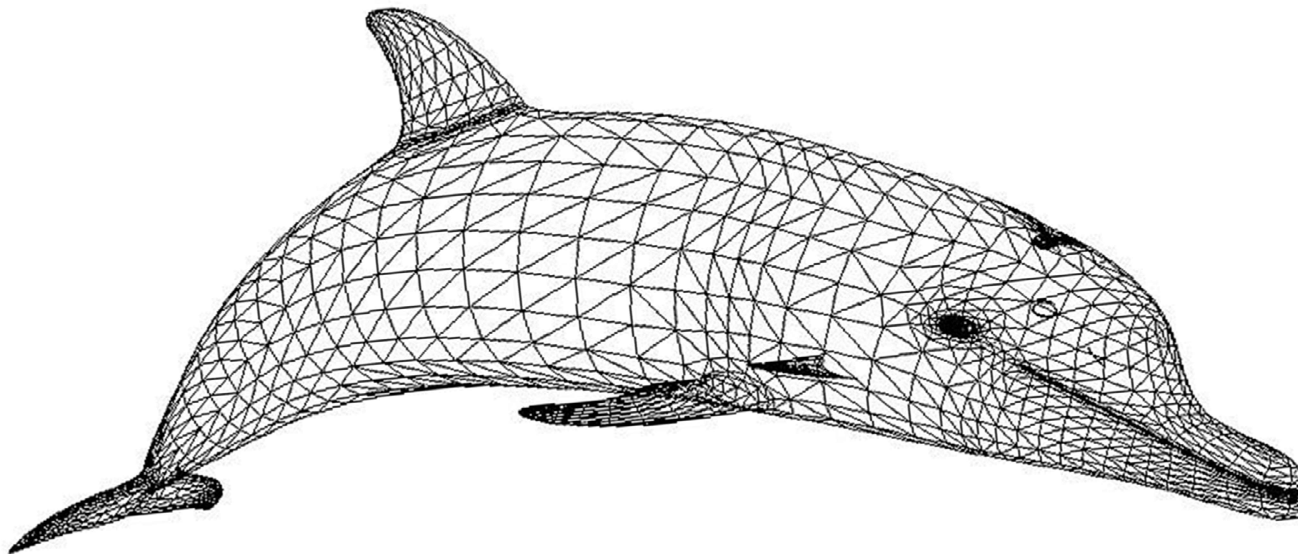
Bump Mapping



Texture Mapping

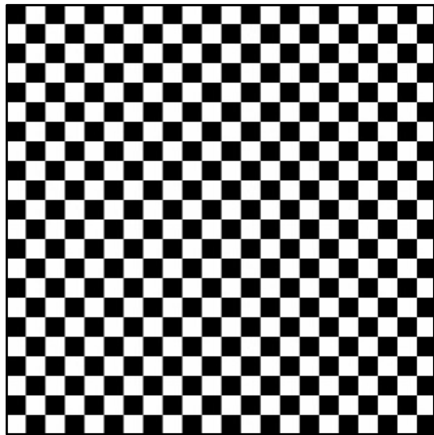
- Add image data to an object surface

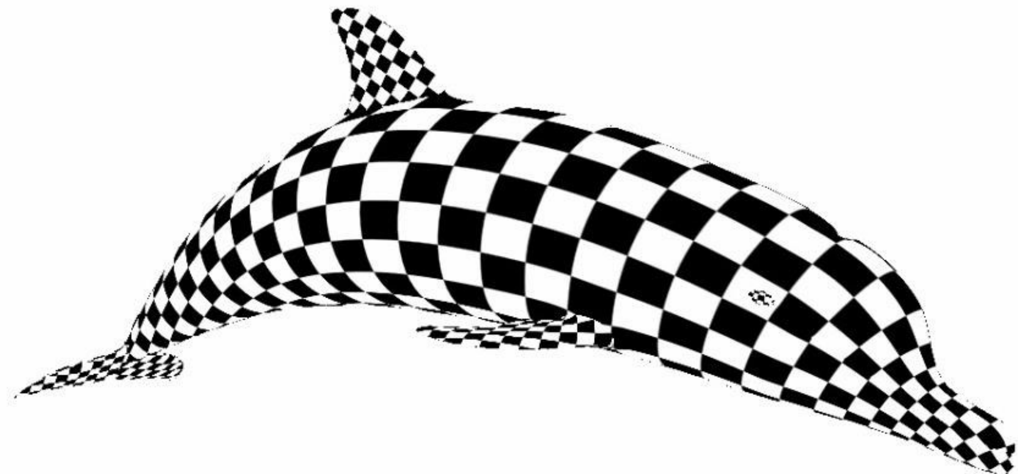
Example:
A model of a dolphin, made of triangle primitives

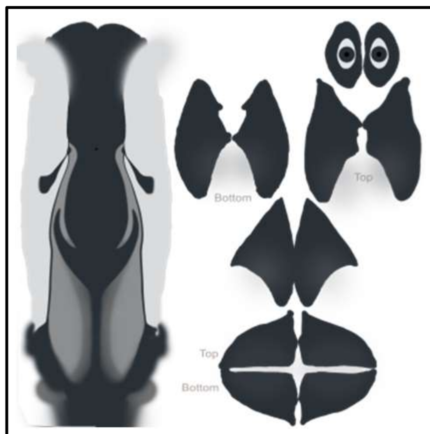


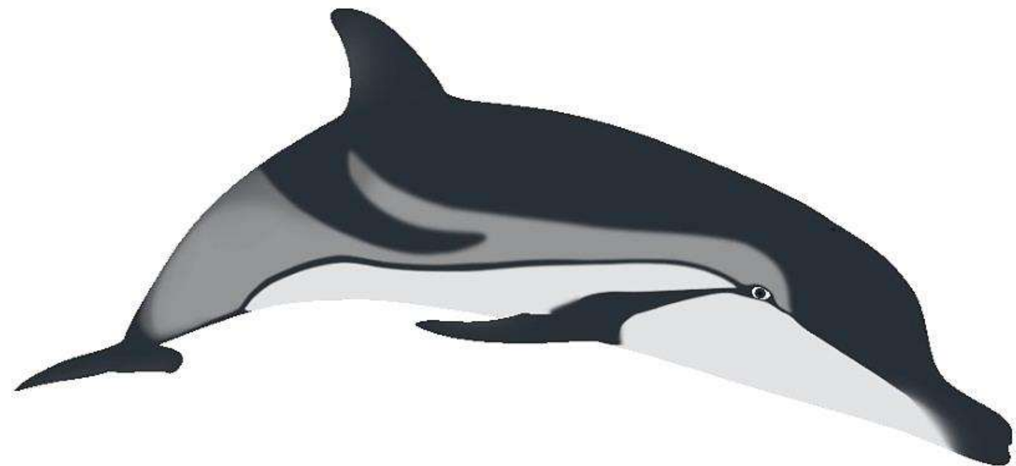
Texture Mapped Versions of Model

texture images ***textured objects***

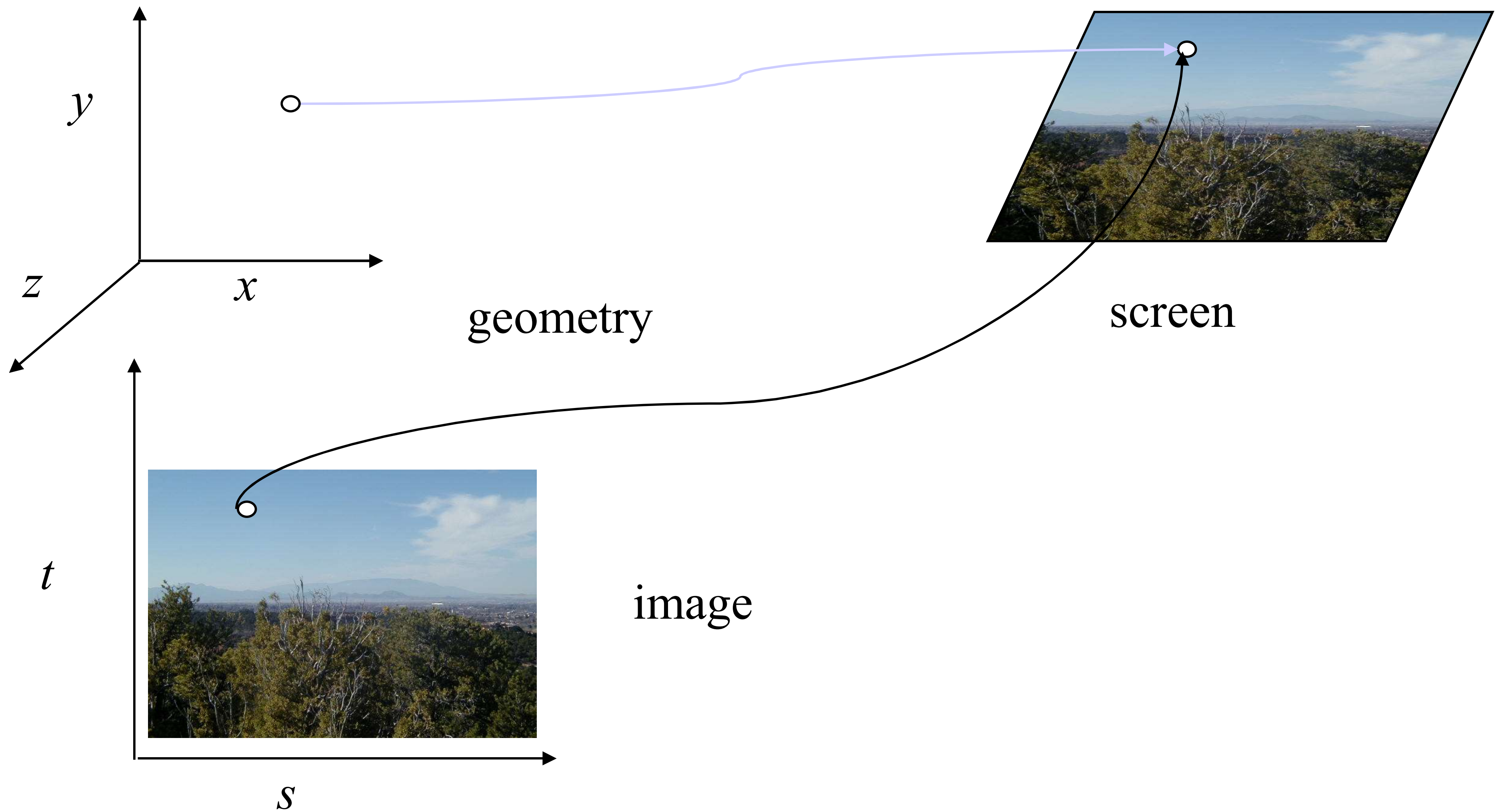






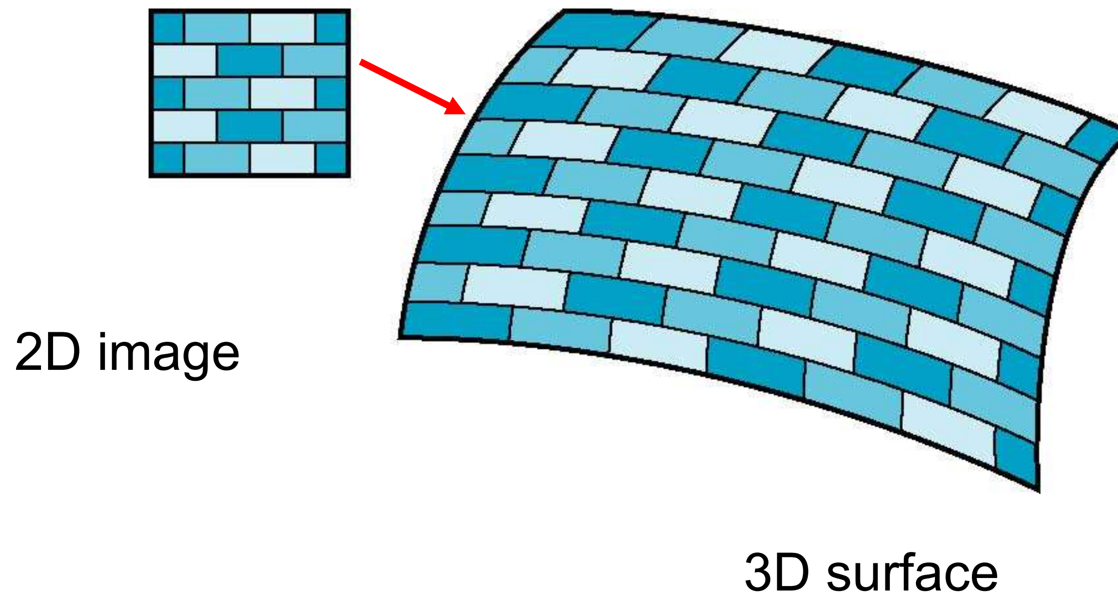


Texture Mapping



Basic Idea

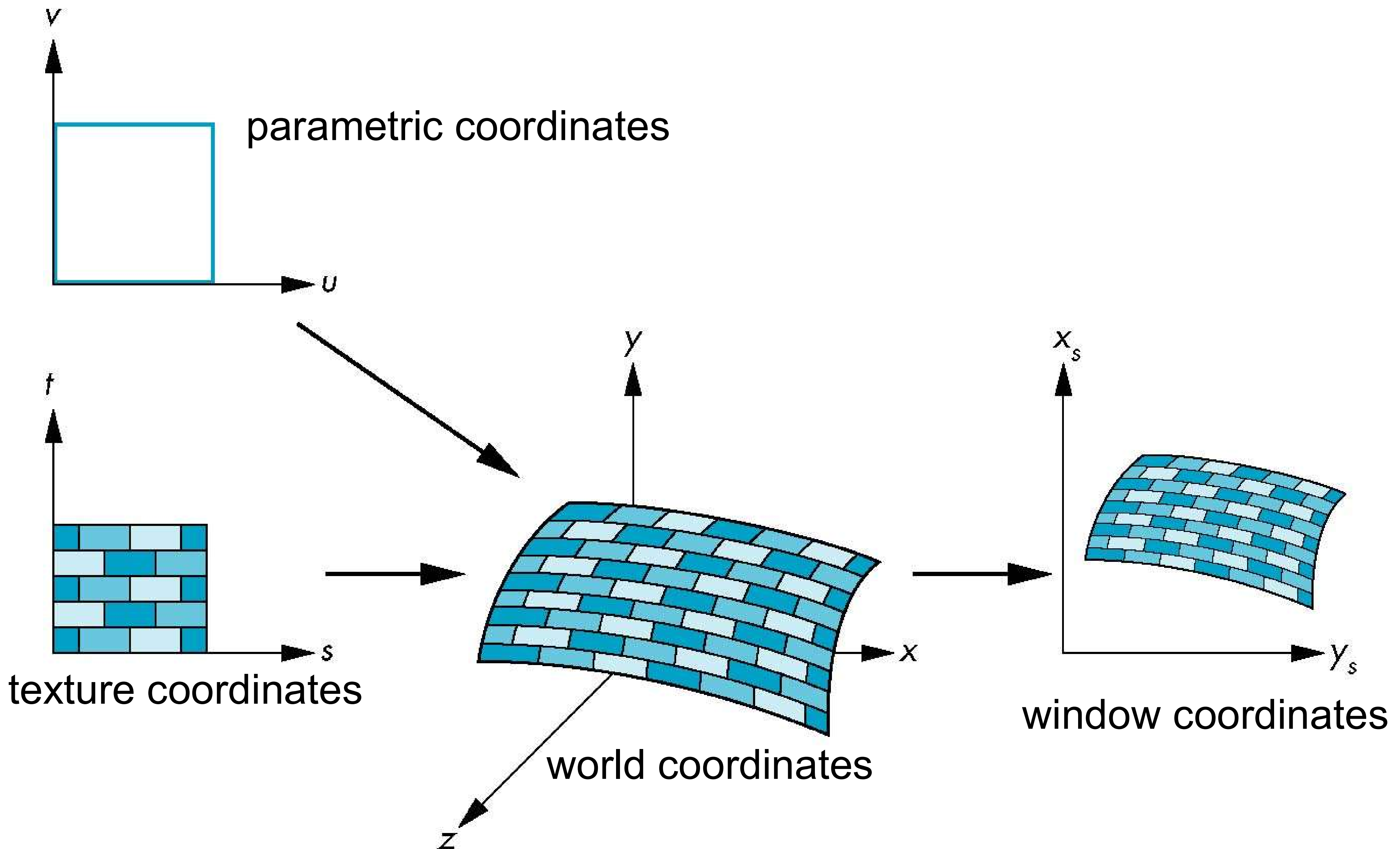
- Map an image to a surface
- There are 3 or 4 coordinate systems involved



Coordinate Systems

- Parametric coordinates
  - May be used to model curves and surfaces
- Texture coordinates
  - Used to identify points in the image to be mapped
- Object or World Coordinates
  - Conceptually, where the mapping takes place
- Window Coordinates
  - Where the final image is really produced

Texture Mapping



Mapping Functions

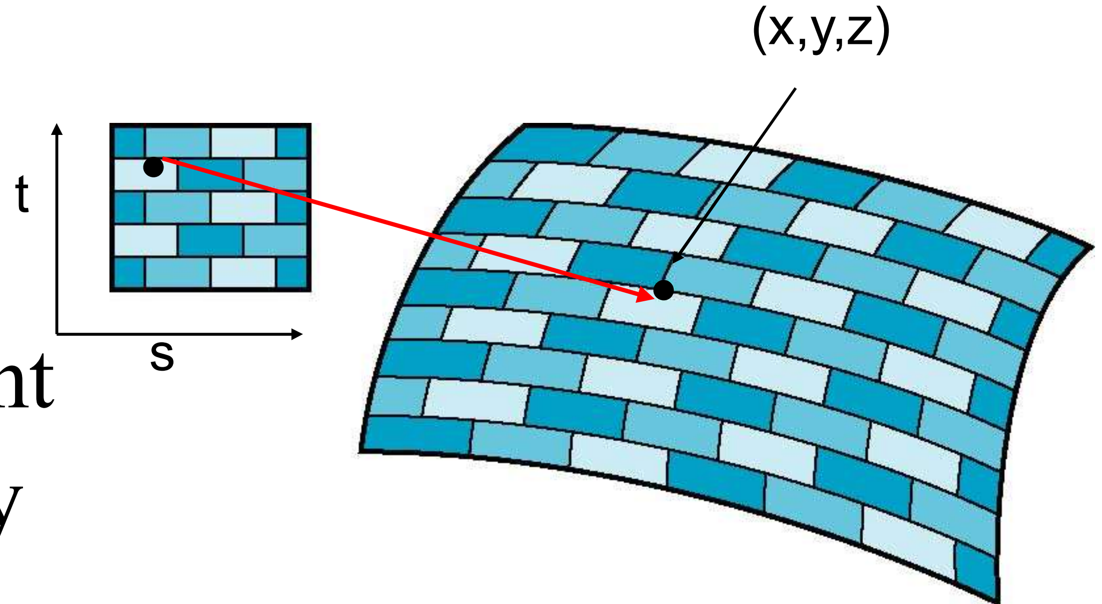
- Basic problem is how to find the maps
- Consider mapping from texture coordinates to a point a surface
- Appear to need three functions

$x = x(s,t)$

$y = y(s,t)$

$z = z(s,t)$

- But we really want to go the other way



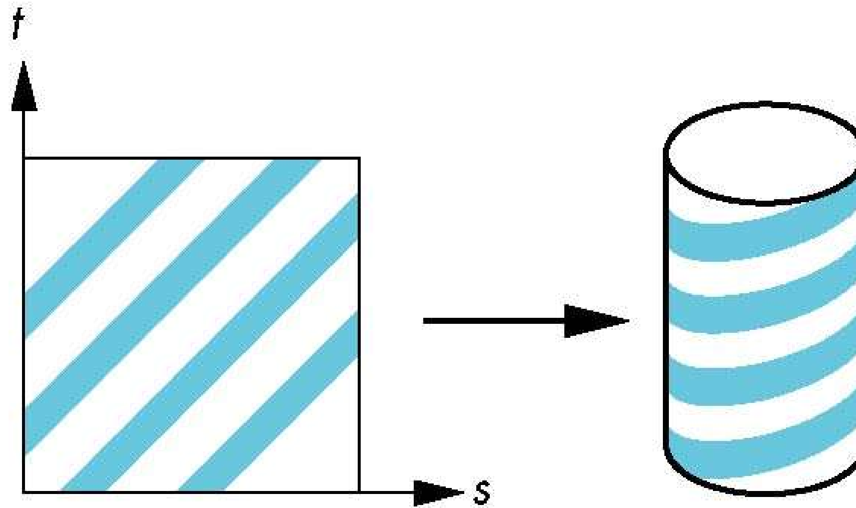
Backward Mapping

- We really want to go backwards
  - Given a pixel, we want to know to which point on an object it corresponds
  - Given a point on an object, we want to know to which point in the texture it corresponds
- Need a map of the form

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

- Such functions are difficult to find in general
- Simple examples: cylinder, sphere, box

Cylindrical Mapping



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

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 Mapping

We can use a parametric sphere

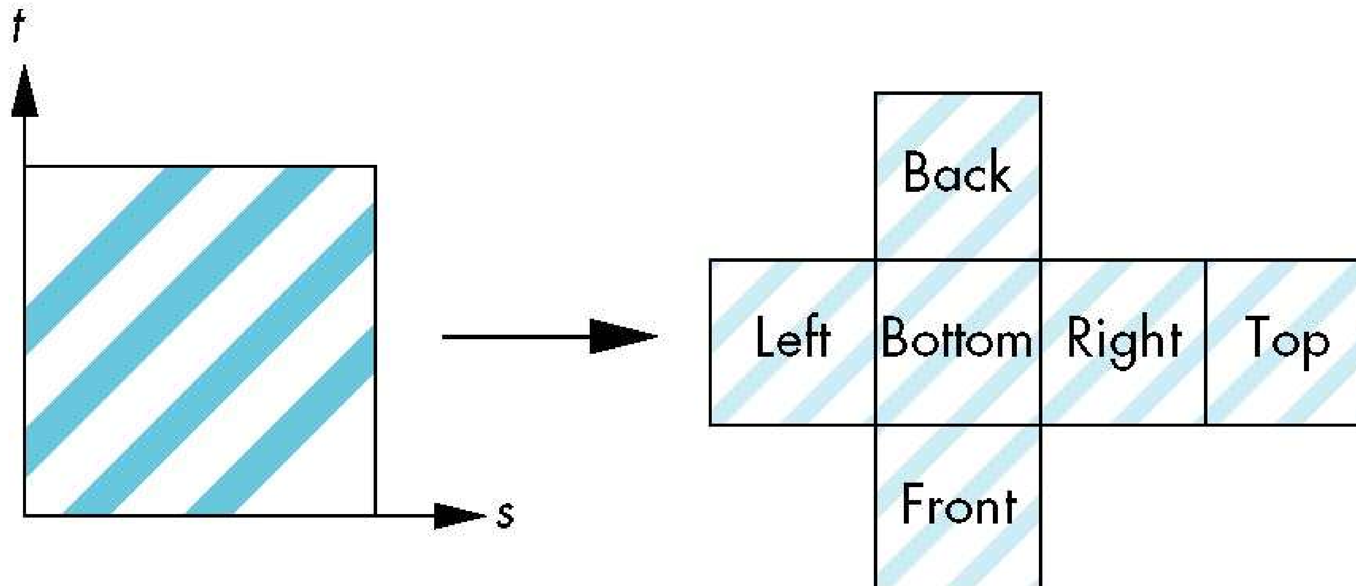
$$x = r \cos 2\pi u$$
$$y = r \sin 2\pi u \cos 2\pi v$$
$$z = r \sin 2\pi u \sin 2\pi v$$

in a similar manner to the cylinder but have to decide where to put the distortion

Spheres are used in environmental maps

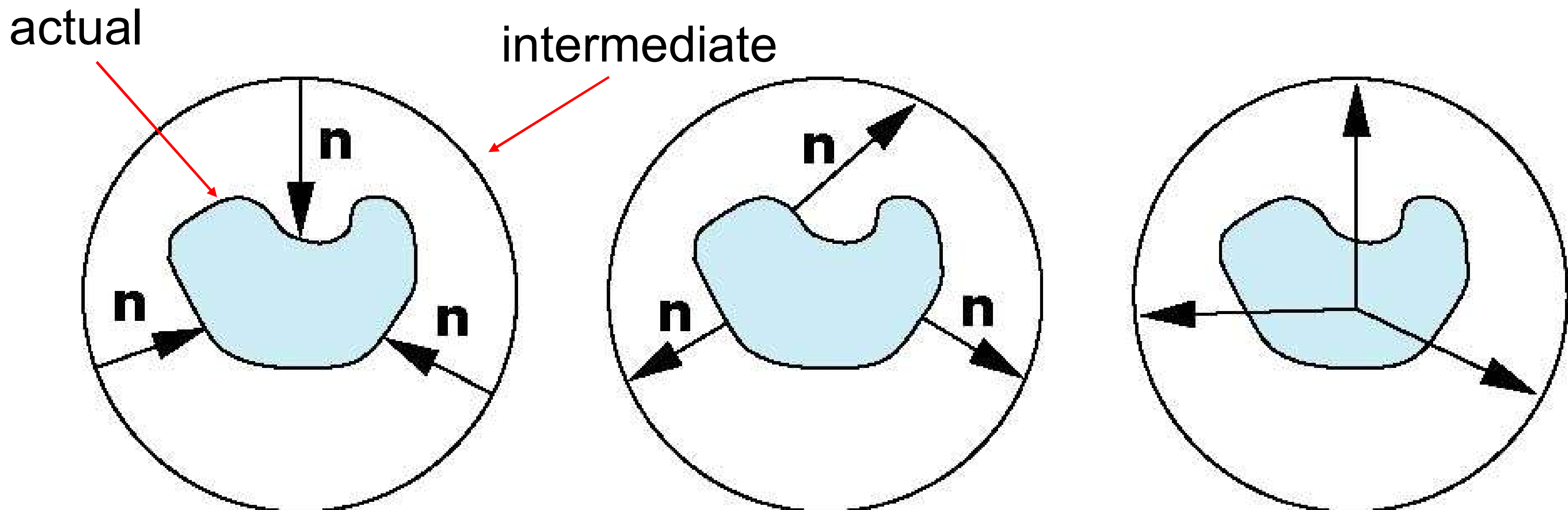
Box Mapping

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



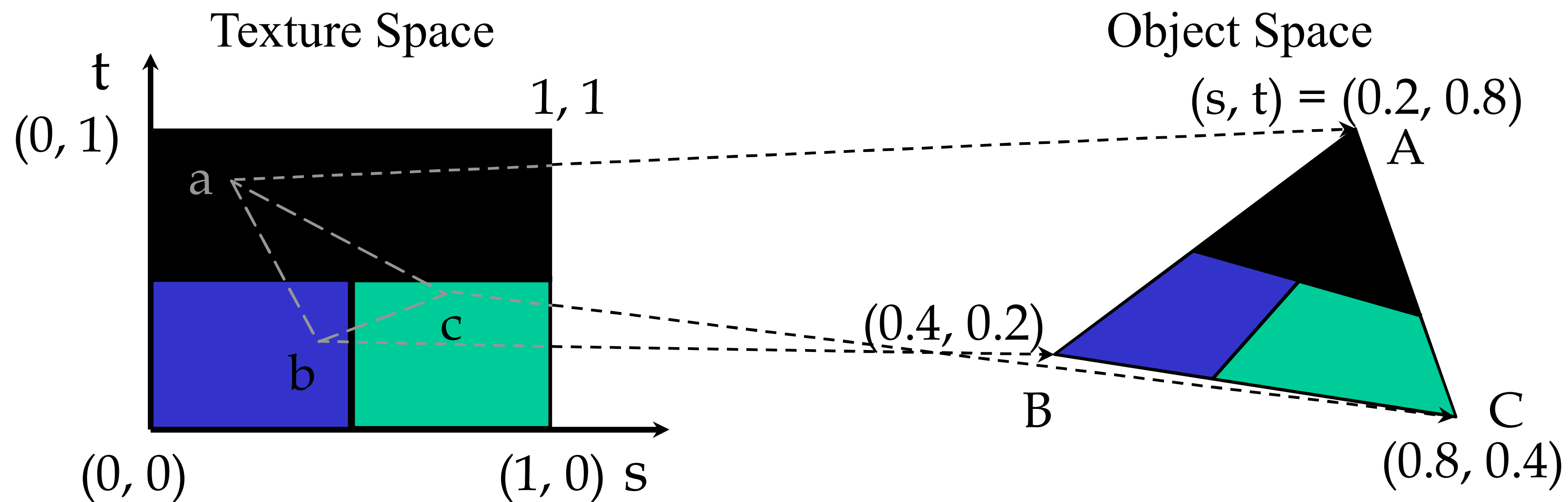
Second Mapping

- Map from intermediate object (e.g., sphere) to actual object
- Three variations:
  - Normals from intermediate to actual
  - Normals from actual to intermediate
  - Vectors from center of intermediate



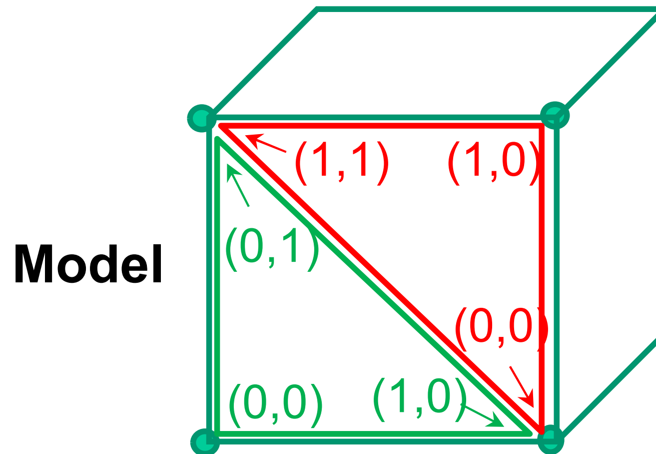
Texture Mapping Example

- Based on parametric texture coordinates
- Coordinates need to be specified at each vertex



(s,t) are texture coordinates.
They index into the texture space.

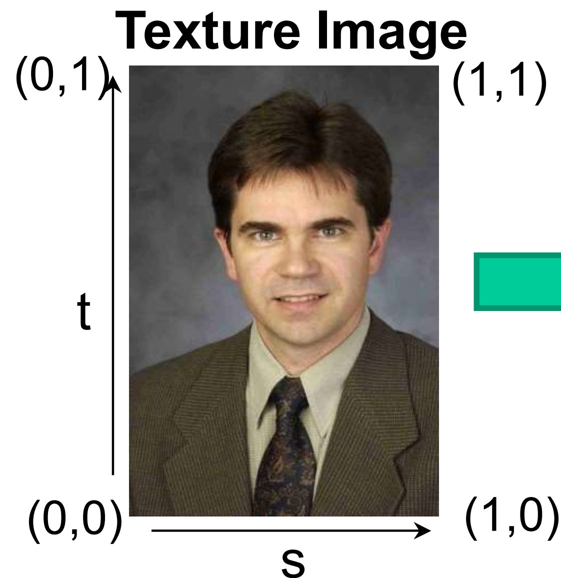
Texture Coordinates



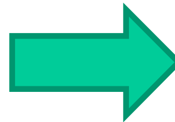
Model

Texture coordinates typically range from (0,0) to (1,1)

Texture Image

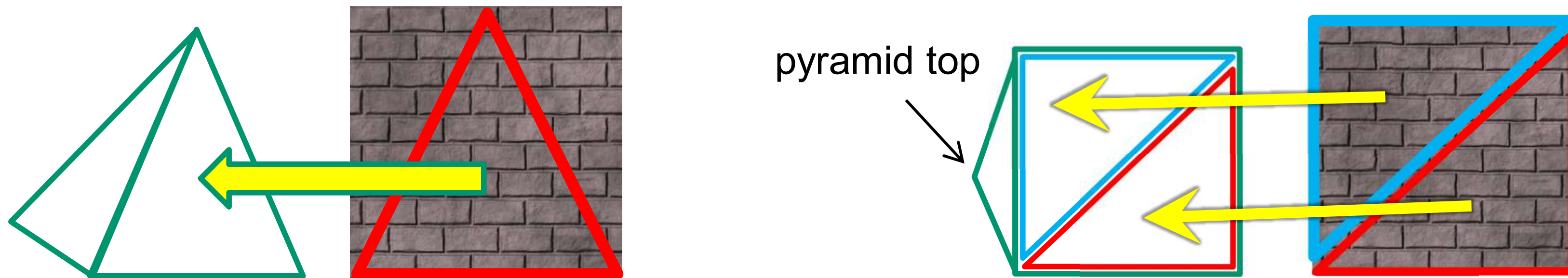


axes often labeled s, t



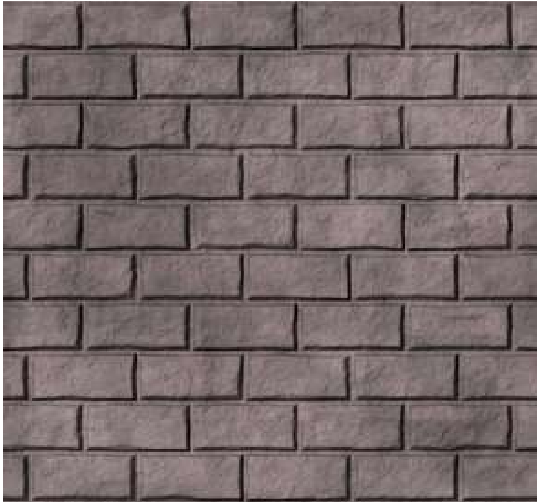
Selected pixels often called "texels"

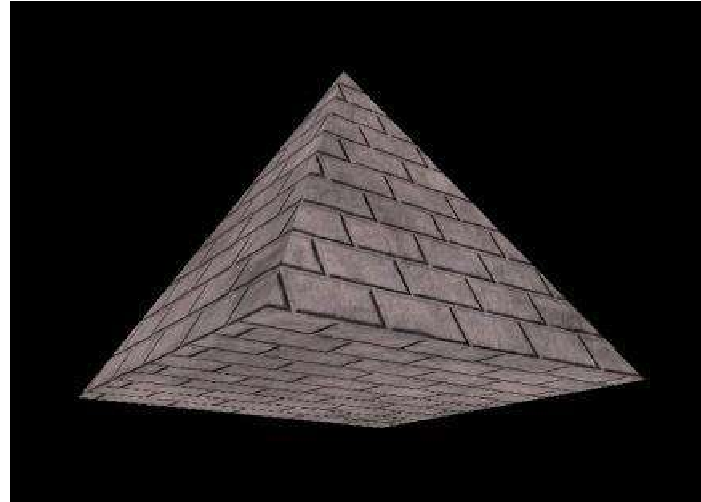
Texture Coordinates for Pyramid

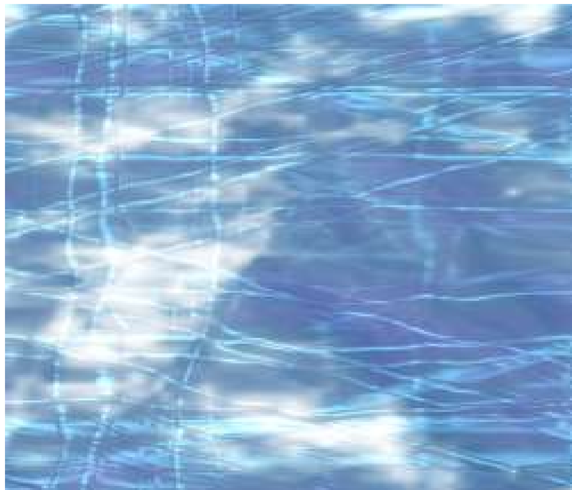


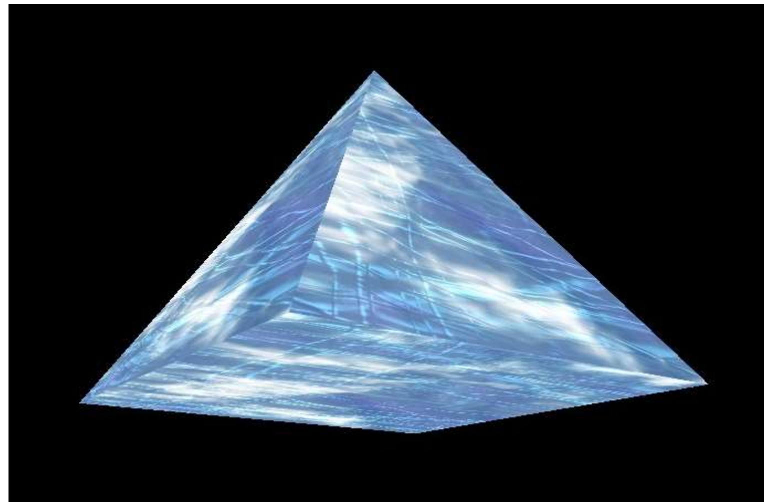

```
vertices          texture coordinates
(-1.0,-1.0, 1.0)    (0, 0)     // front face
( 1.0,-1.0, 1.0)    (1, 0)
( 0.0, 1.0, 0.0)   (.5, 1)
( 1.0,-1.0, 1.0)    (0, 0)     // right face
( 1.0,-1.0,-1.0)    (1, 0)
( 0.0, 1.0, 0.0)   (.5, 1)
( 1.0,-1.0,-1.0)    (0, 0)     // back face
(-1.0,-1.0,-1.0)    (1, 0)
( 0.0, 1.0, 0.0)   (.5, 1)
etc.
```

A Textured Pyramid





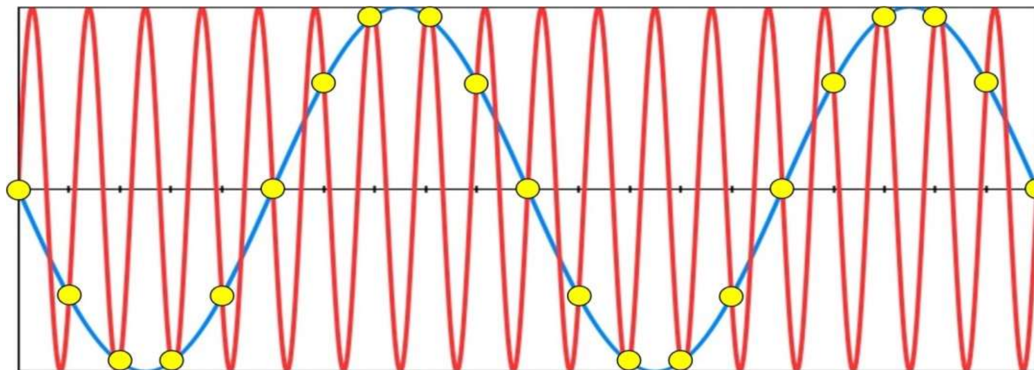




Texture Artifacts

Can occur when:
- Texture image resolution is too low (blurry, stretched)
- Texture image resolution is too high (!)

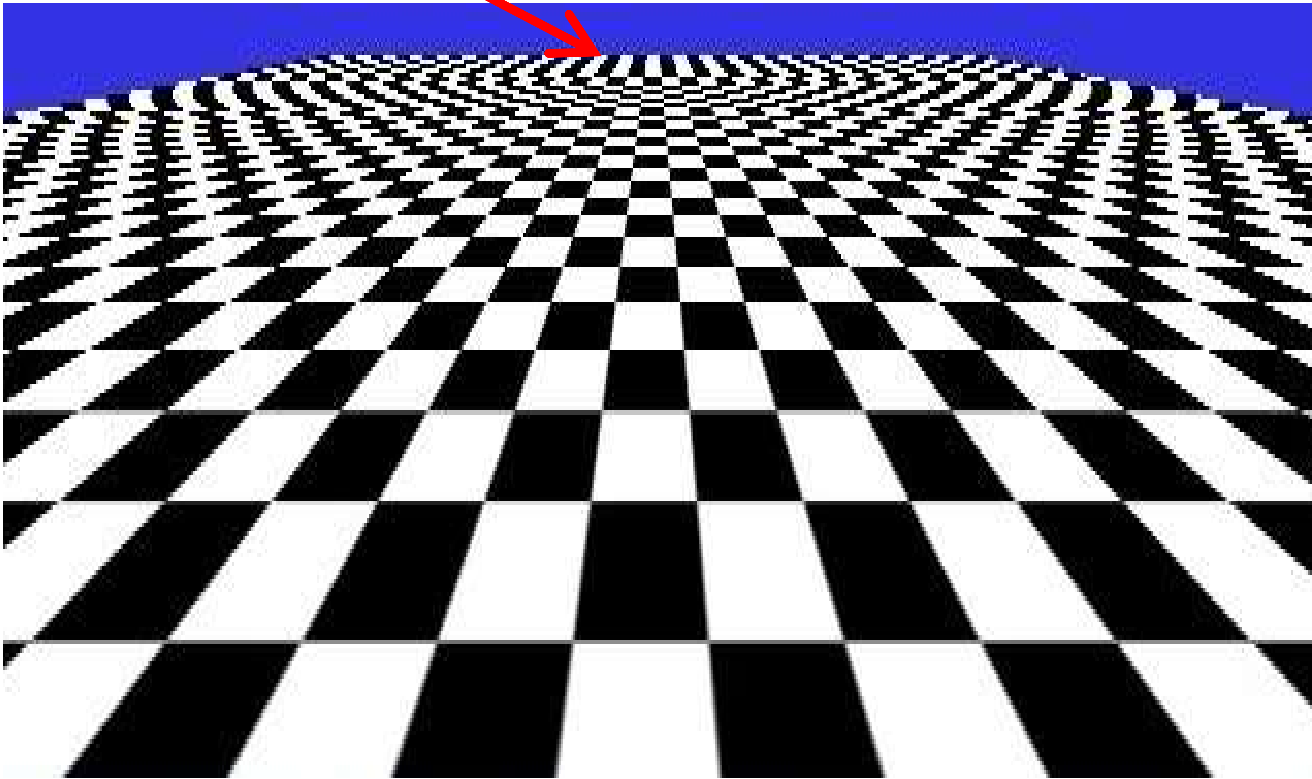
Why would this be a problem?
Because of sampling errors or aliasing (undersampling).



Example of aliasing. The original waveform is red, the reproduced incorrect waveform (due to inadequate sampling) is blue.

Aliasing

Example



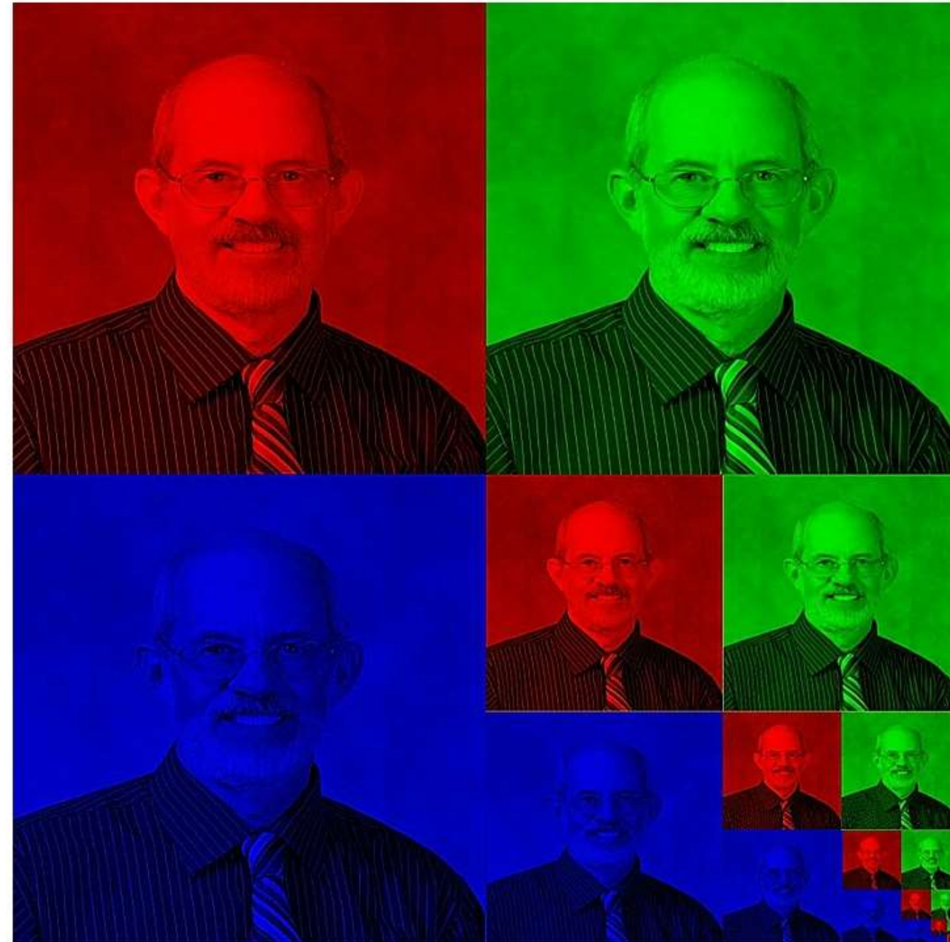
Can also cause “sparkling” effects when objects are animated.

Anti-Aliasing

- Use mipmap to store multiple resolutions of the texture image
- Requires 33% more memory

Input image

Mipmap pyramid

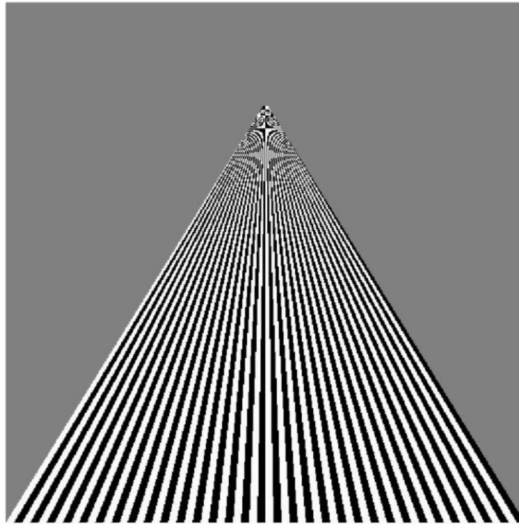


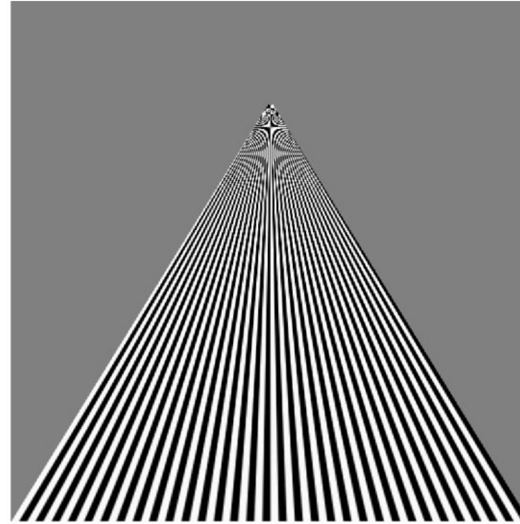
Mipmapped Textures

- *Mipmapping* allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects

Example

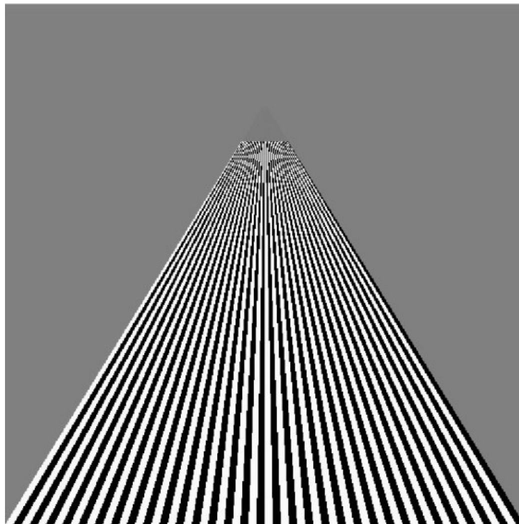
point sampling

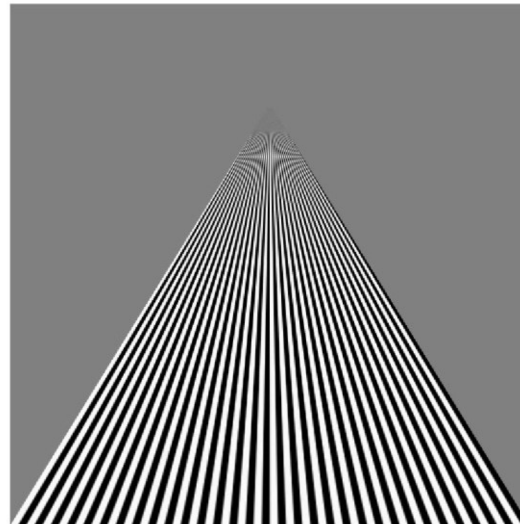




linear filtering

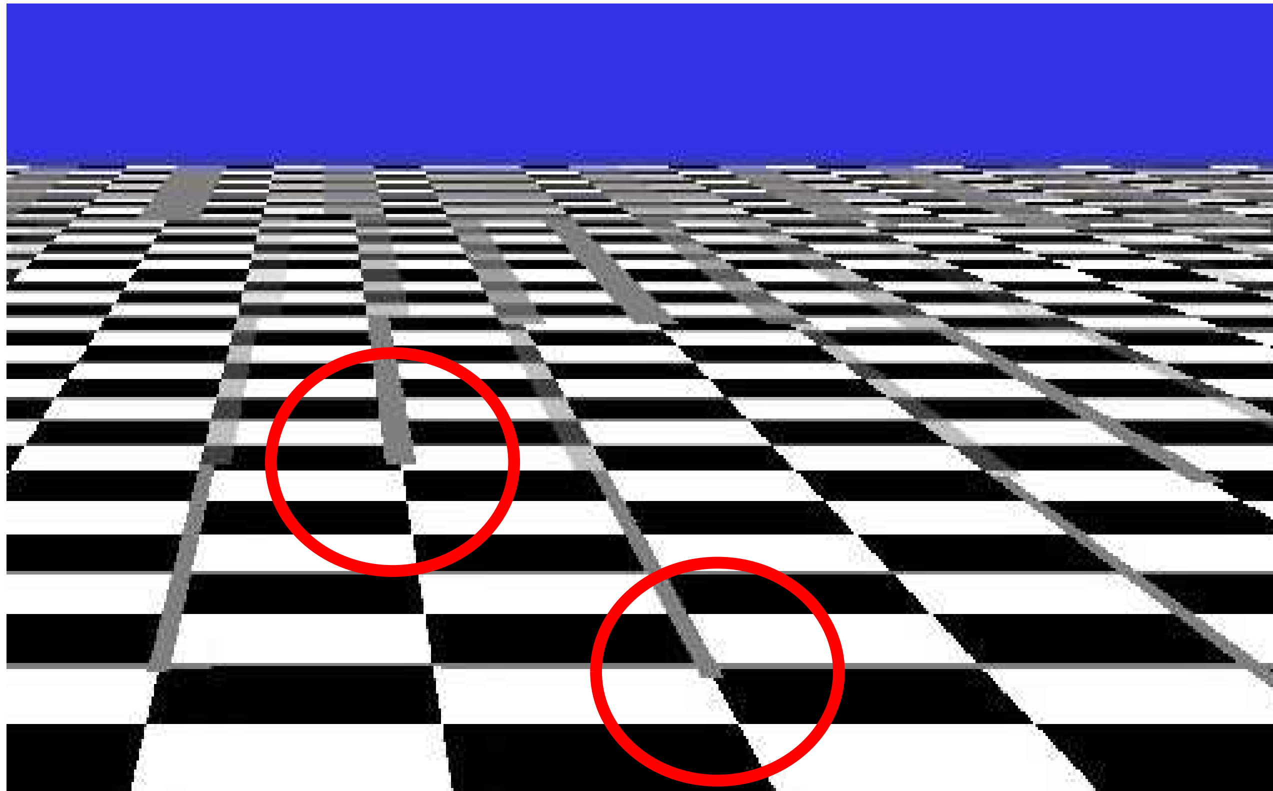
mipmapped point sampling





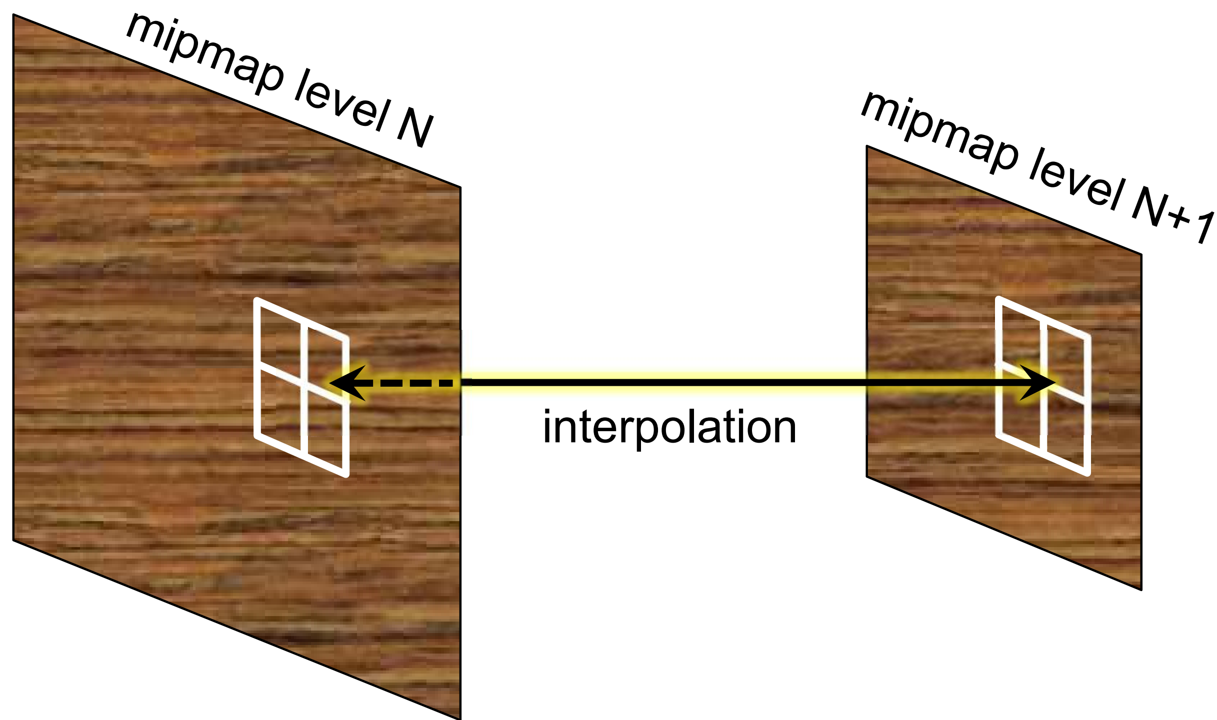
mipmapped linear filtering

Mipmapping Can Produce Artifacts



In this case, the vertical lines suddenly change from thick to thin at a mipmap boundary when applying linear filtering, whereby the four texels nearest the texture coordinate are interpolated. Trilinear filtering will correct these results.

Tri-Linear Filtering




```
glBindTexture(GL_TEXTURE_2D, brickTexture);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR );
glGenerateMipmap(GL_TEXTURE_2D);
```

Anisotropic Filtering (AF)

- Mipmapping can result in loss of detail when an object is tilted, because its primitives appear smaller along one axis (i.e., width vs. height) than the other.
- AF mitigates loss of detail (while still reducing aliasing and "sparkling") by sampling the texture at various *rectangular* resolutions, such as 256x128, 64x128)
- AF is more computationally expensive, and not all graphics cards support it. It is now commonly supported though.

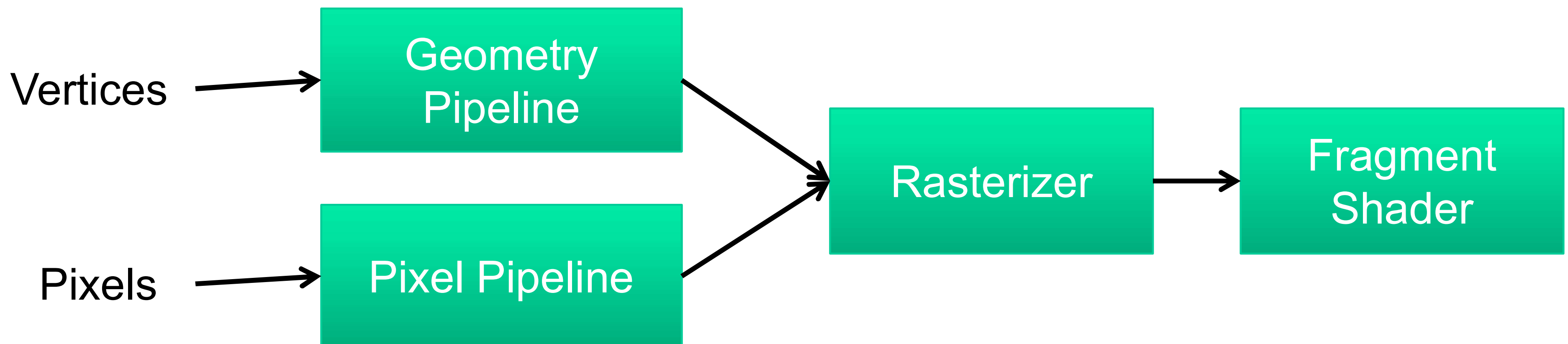
OpenGL supports AF via an OpenGL Extension…

Texture Mapping in OpenGL

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Texture Mapping in OpenGL Pipeline

- Images and geometry flow through separate pipelines that join at the rasterizer
  - "complex" textures do not affect geometric complexity



Applying Textures

Three steps to applying a texture

1. specify the texture
   - read or generate image
   - assign to texture buffer
   - enable texturing
2. assign texture coordinates to vertices
   - Used to index into texture to paint primitive
   - Proper mapping function is left to application
3. specify texture parameters
   - wrapping, filtering

Texture Mapping Mechanisms

- Several mechanisms need to be coordinated to support texture mapping
  - a texture image with the desired appearance
  - a texture object to hold the texture image (in this chapter we consider only 2D images),
  - a special uniform sampler variable so that the fragment shader can access the texture,
  - a buffer to hold the texture coordinates,
  - a vertex attribute for passing the texture coordinates down the pipeline, and
  - a texture unit on the graphics card.

Creating a Checkerboard Texture

```
GLubyte image[64][64][3];

// Create a  64 x 64 checkerboard pattern
for (int i = 0; i < 64; i++) {
        for (int j = 0; j < 64; j++) {
              GLubyte c = (((i & 0x8)==0)^((j & 0x8)==0)) * 255;
              image[i][j][0]  = c;
              image[i][j][1]  = c;
              image[i][j][2]  = c;
        }
}
```

Texture Objects

- Have OpenGL store your images
  - one image per texture object
  - may be shared by several graphics contexts

```
// generate texture object
Gluint texturedID;
glGenTextures(1, &textureID);

// associate image with texture object
glBindTexture(GL_TEXTURE_2D, textureID);
glTexImage2D (GL_TEXTURE_2D, 0, 3, w, h, 0,
              GL_RGB, GL_UNSIGNED_BYTE, data);
```

glTexImage2D Usage

```
glTexImage2D( target, level, components,
    w, h, border, format, type, texels );
```

- `target:` type of texture, e.g. `GL_TEXTURE_2D`
- `level:` used for mipmapping
- `components:` elements per texel
- `w, h:` width and height of `texels` in pixels
- `border:` used for smoothing (discussed later)
- `format and type:` describe texels
- `texels:` pointer to texel array

```
glTexImage2D(GL_TEXTURE_2D, 0, 3, 512, 512, 0,
 GL_RGB, GL_UNSIGNED_BYTE, my_texels);
```

Applying Texture to Cube

```
// add texture coordinate attribute to quad function
quad( int a, int b, int c, int d )
{
    vColors    [Index] = colors[a];
    vPositions[Index] = positions[a];
    vTexCoords[Index] = vec2( 0.0, 0.0 );
    Index++;

    vColors    [Index] = colors[b];
    vPositions[Index] = positions[b];
    vTexCoords[Index] = vec2( 1.0, 0.0 );
    Index++;
    … // rest of vertices
}
```

Load Vertex Positions and Texture Coordinates into VBO

```
float pyrTexCoords[36] =
{  0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,   // top and right faces
   0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,0.0f, 0.0f, 1.0f, 0.0f, 0.5f, 1.0f,   // back and left faces
   0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f,1.0f, 1.0f, 0.0f, 0.0f, 1.0f, 0.0f    // base triangles
}

// create two VBOs (one for vertices and one for texture coordinates
Gluint vbo[2];
glGenBuffers(2, vbo);

// load the two VBOs
glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(pyrPositions), pyrPositions, GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
glBufferData(GL_ARRAY_BUFFER, sizeof(pyrTexCoords), pyrTexCoords, GL_STATIC_DRAW);
```

The texture positions and texture coordinates are then sent to the shaders in vertex attributes. Note, therefore, that they will be interpolated!

Attribute Variables

```
offset = 0;
GLuint vPosition = glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,(void*)
  offset );

offset += sizeof(points);
GLuint vTexCoord = glGetAttribLocation( program, "vTexCoord" );
glEnableVertexAttribArray( vTexCoord );
glVertexAttribPointer( vTexCoord, 2,GL_FLOAT, GL_FALSE, 0,
                       (void *) offset );
```

Accessing Texture Object in a Shader

```
layout (binding=0) uniform  sampler2D  samp;
```

The “layout” qualifier states that this variable accesses texture unit #0.

- In the C++ application, associate the texture object with a texture unit (in this case, unit #0):

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, brickTexture);
```

- Send the texture coordinates in a vertex attribute.

```
in vec2 tc;   // texture coordinates
```

- Finally, in the shader, use the texture coordinates to look up the correct texel from the texture:

```
color = texture(samp, tc);
```

Vertex Shader

```
in vec4 vPosition; // vertex position in object coordinates
in vec4 vColor;    // vertex color from application
in vec2 vTexCoord; // texture coordinate from application

out vec4 color;    // output color to be interpolated
out vec2 texCoord; // output tex coordinate to be interpolated

  void main()
  {
      color       = vColor;
      texCoord    = vTexCoord;
      gl_Position = vPosition;
  }
```

Fragment Shader

```
in vec4 color;              // color from rasterizer
in vec2 texCoord;           // texture coordinate from rasterizer
uniform sampler2D texture; // texture object from application

void main()  {
    gl_FragColor = color * texture(texture, texCoord);
}
```

Textures are applied in the fragment shader by a **sampler.** Samplers return a texture color from a texture object

Fragment Shader for Modulating Intensity with Texture

```
in vec4 texCoord;

// Declare the sampler
uniform float     intensity;
uniform sampler2D diffuseMaterialTexture;

// Apply the material color
vec3 diffuse = intensity *
           texture(diffuseMaterialTexture, texCoord).rgb;
```

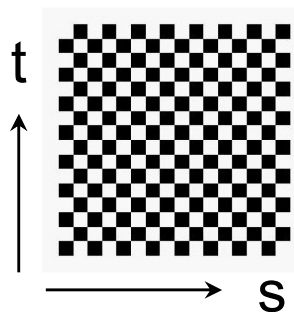
Texture Parameters

- OpenGL has a variety of parameters that determine how texture is applied
  - Wrapping parameters determine what happens if s and t are outside the (0,1) range
  - Filter modes allow us to use area averaging instead of point samples
  - Mipmapping allows us to use textures at multiple resolutions
  - Environment parameters determine how texture mapping interacts with shading

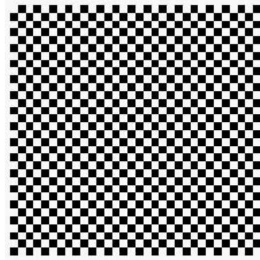
Wrapping Mode

Clamping: if $s,t > 1$ use 1, if $s,t < 0$ use 0
Wrapping: use s,t modulo 1

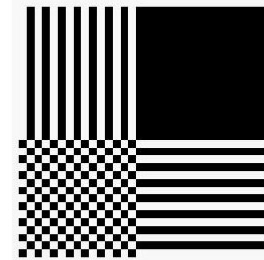
```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
```



texture



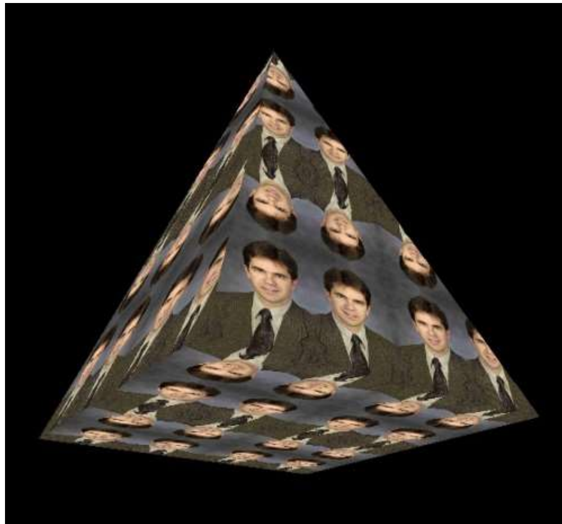
GL_REPEAT wrapping



GL_CLAMP wrapping

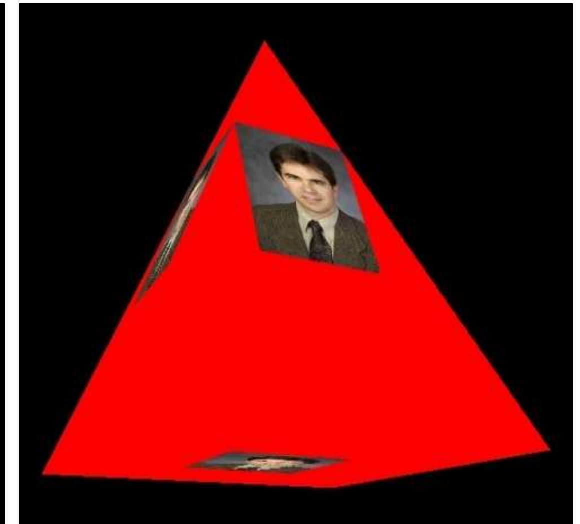
Wrapping Examples

OpenGL has several options when texture coordinates lie outside the (0,1) range.



GL_REPEAT
(and GL_MIRRORED_REPEAT)

GL_CLAMP_TO_EDGE



GL_CLAMP_TO_BORDER

for example:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
float redColor[4] = {1.0f, 0.0f, 0.0f, 1.0f };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, redColor);
```

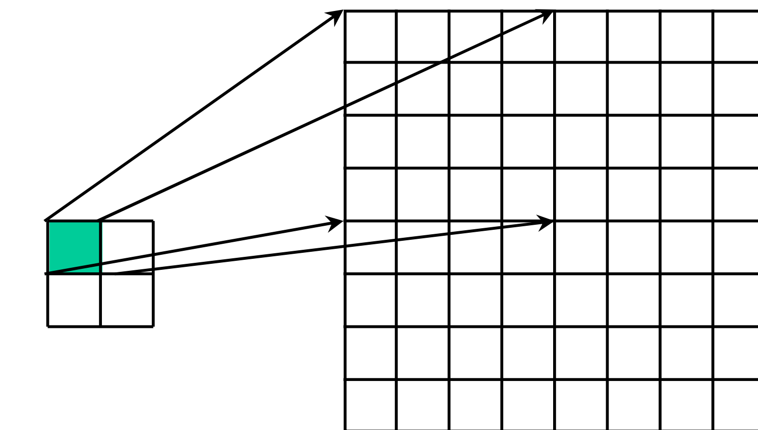
Example

```
GLuint textures[1];
glGenTextures( 1, textures );

glActiveTexture( GL_TEXTURE0 );
glBindTexture  ( GL_TEXTURE_2D, textures[0] );

glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, TextureSize,
              TextureSize, GL_RGB, GL_UNSIGNED_BYTE, image );

glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,     GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,     GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Magnification and Minification

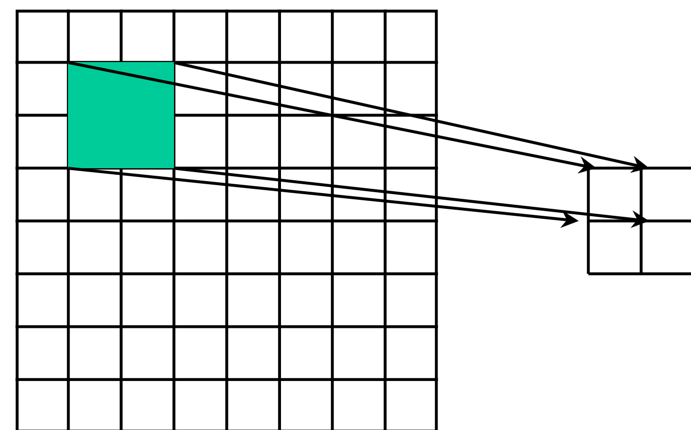
More than one texel can cover a pixel (*minification*) or more than one pixel can cover a texel (*magnification*)

Can use point sampling (nearest texel) or linear filtering (2 x 2 filter) to obtain texture values



Texture Polygon

Magnification



Texture Polygon

Minification

Filter Modes

Modes determined by

- **`glTexParameteri(target, type, mode)`**

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

Note that linear filtering requires an extra border texel for filtering at edges (border = 1)

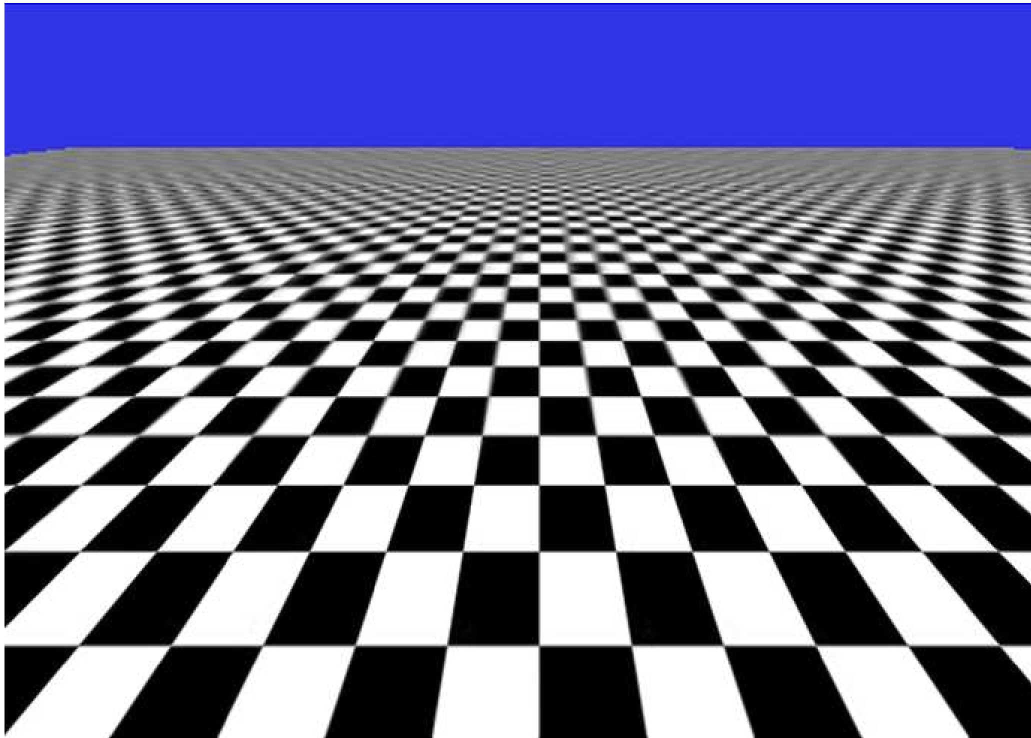
Texture Functions

- Controls how texture is applied
  - **`glTexEnv{fi}[v]( GL_TEXTURE_ENV, prop, param )`**
- **`GL_TEXTURE_ENV_MODE`** modes
  - **`GL_MODULATE`**: modulates with computed shade
  - **`GL_BLEND`**: blends with an environmental color
  - **`GL_REPLACE`**: use only texture color
  - **`GL(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);`**
- Set blend color with **`GL_TEXTURE_ENV_COLOR`**

Mipmap Generation

- OpenGL can generate mipmaps

```
glBindTexture(GL_TEXTURE_2D, brickTexture);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST );
glGenerateMipmap(GL_TEXTURE_2D);
```



OpenGL uses the resolution in the mipmap closest to the region being drawn.

Minification Techniques

Reducing mipmap artifacts with minification filter.

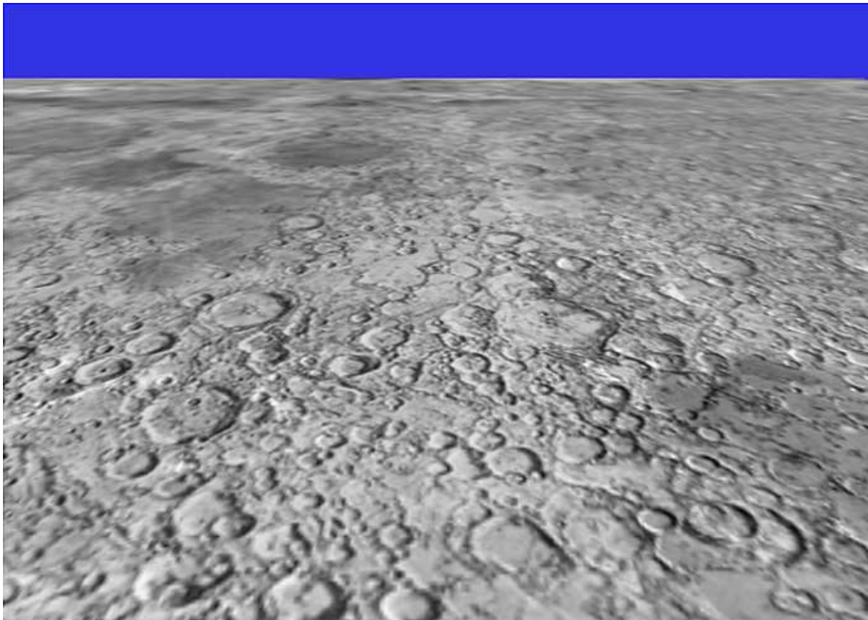
OpenGL supports four types of minification techniques:

- **GL_NEAREST_MIPMAP_NEAREST**
  chooses mipmap with resolution most similar to the region of pixels being textured. Obtains the nearest texel to the desired texture coordinates.
- **GL_LINEAR_MIPMAP_NEAREST** ("linear filtering")
  chooses mipmap with resolution most similar to the region of pixels being textured. Then interpolates the four texels nearest the texture coordinates.
- **GL_NEAREST_MIPMAP_LINEAR** ("bi-linear filtering")
  chooses the two mipmaps with resolutions nearest the region of pixels being textured. Interpolates the nearest texels from each mipmap.
- **GL_LINEAR_MIPMAP_LINEAR** ("tri-linear filtering")
  chooses the two mipmaps with resolutions nearest the region of pixels being textured. Interpolates the four nearest texels from each mipmap.

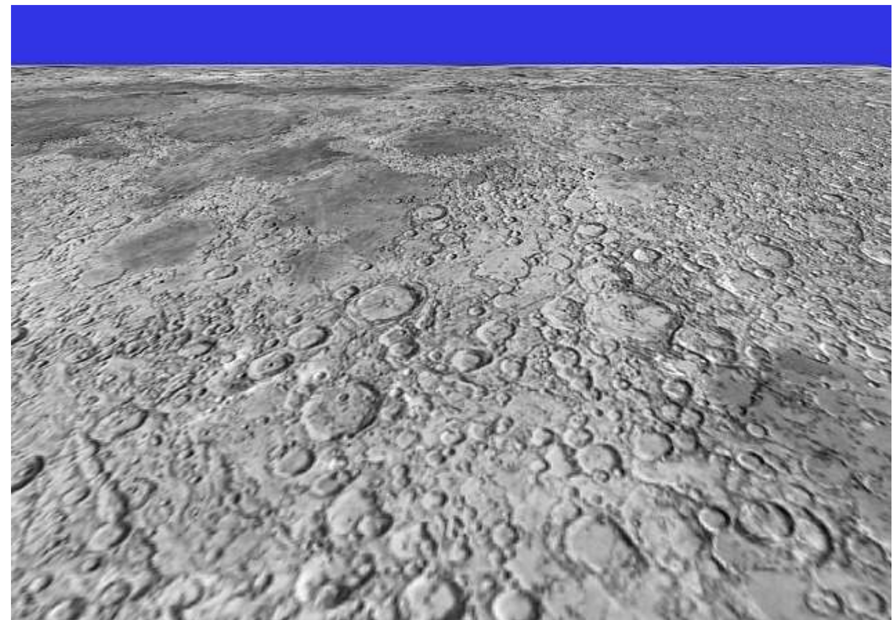
Anisotropic Filtering (AF)

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glGenerateMipmap(GL_TEXTURE_2D);
```

```
//  if also anisotropic filtering
if (glewIsSupported("GL_EXT_texture_filter_anisotropic"))  {
    GLfloat anisoSetting = 0.0f;
    glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &anisoSetting);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, anisoSetting);
}
```



mipmapping without AF



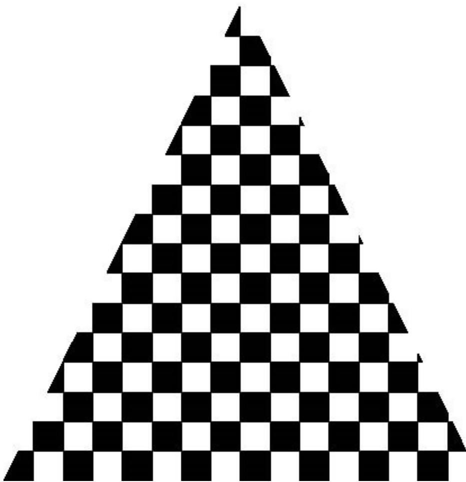
mipmapping with AF

Interpolation

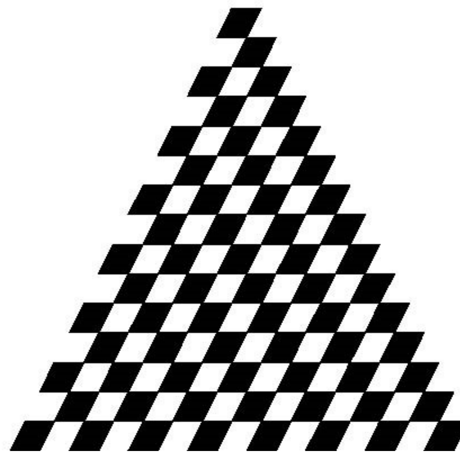
OpenGL uses interpolation (in rasterizer) to find proper texels from specified texture coordinates

There can be distortions:

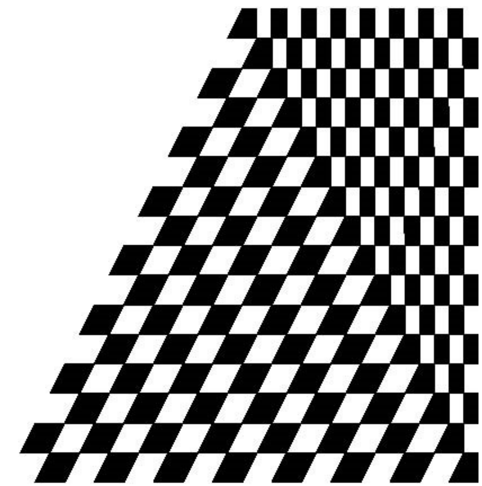
good selection of tex coordinates



poor selection of tex coordinates

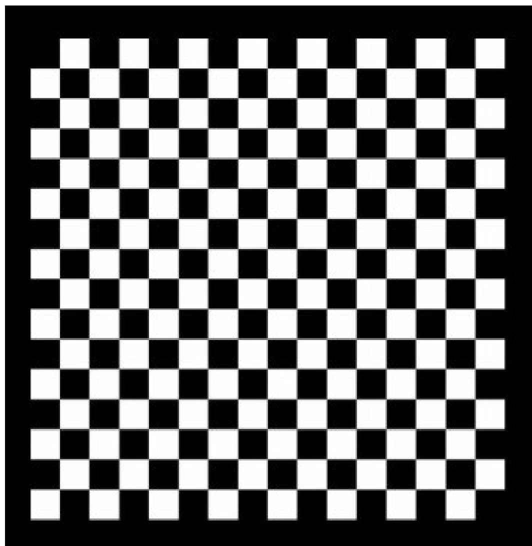


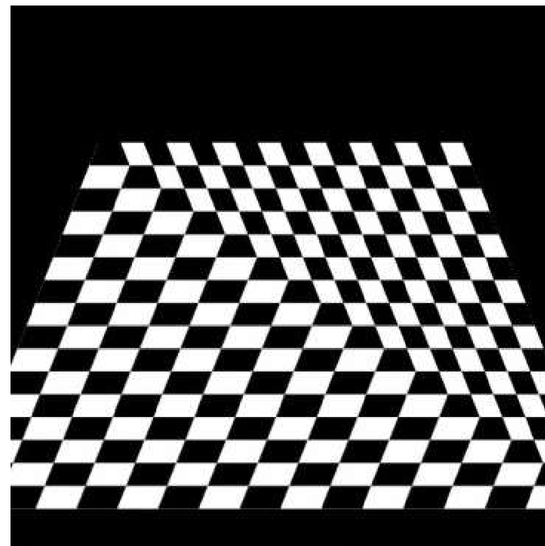
texture stretched over trapezoid showing effects of bilinear interpolation



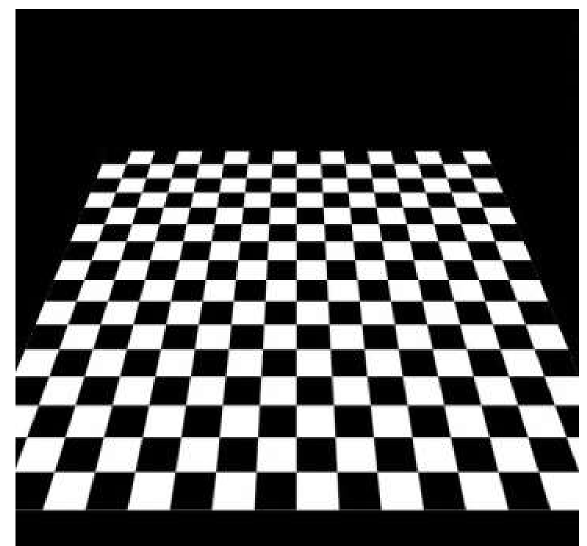
Perspective Distortion

OpenGL automatically corrects perspective distortion as "tilted" objects are textured in 3D space.





perspective distortion



OpenGL correction

This correction can be disabled in the shaders. e.g.:

`noperspective out vec2 texCoord;` (in the vertex shader)

`noperspective in vec2 texCoord;` (in the fragment shader)