CSE 5542 - Real Time Rendering
Week 11, 12, 12, 13

Texture Mapping

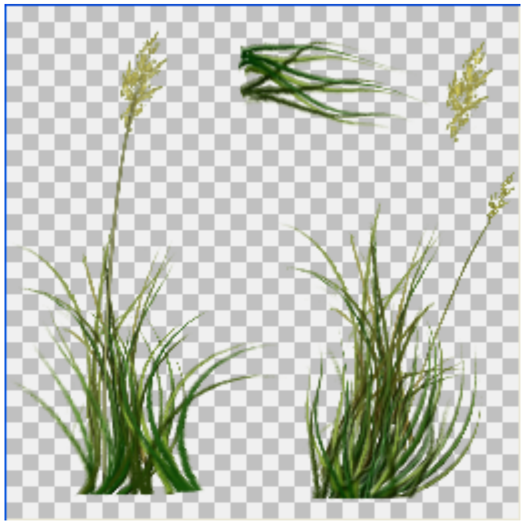
Courtesy: Ed Angel

Limits of Geometric Modeling

Millions of Polygons/Second

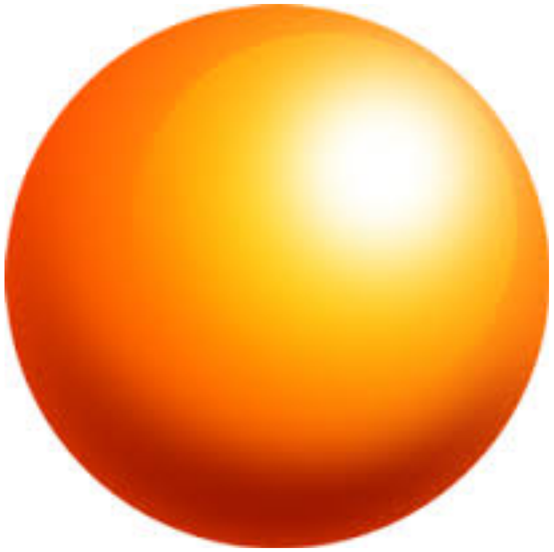
Cannot Do

Use Textures

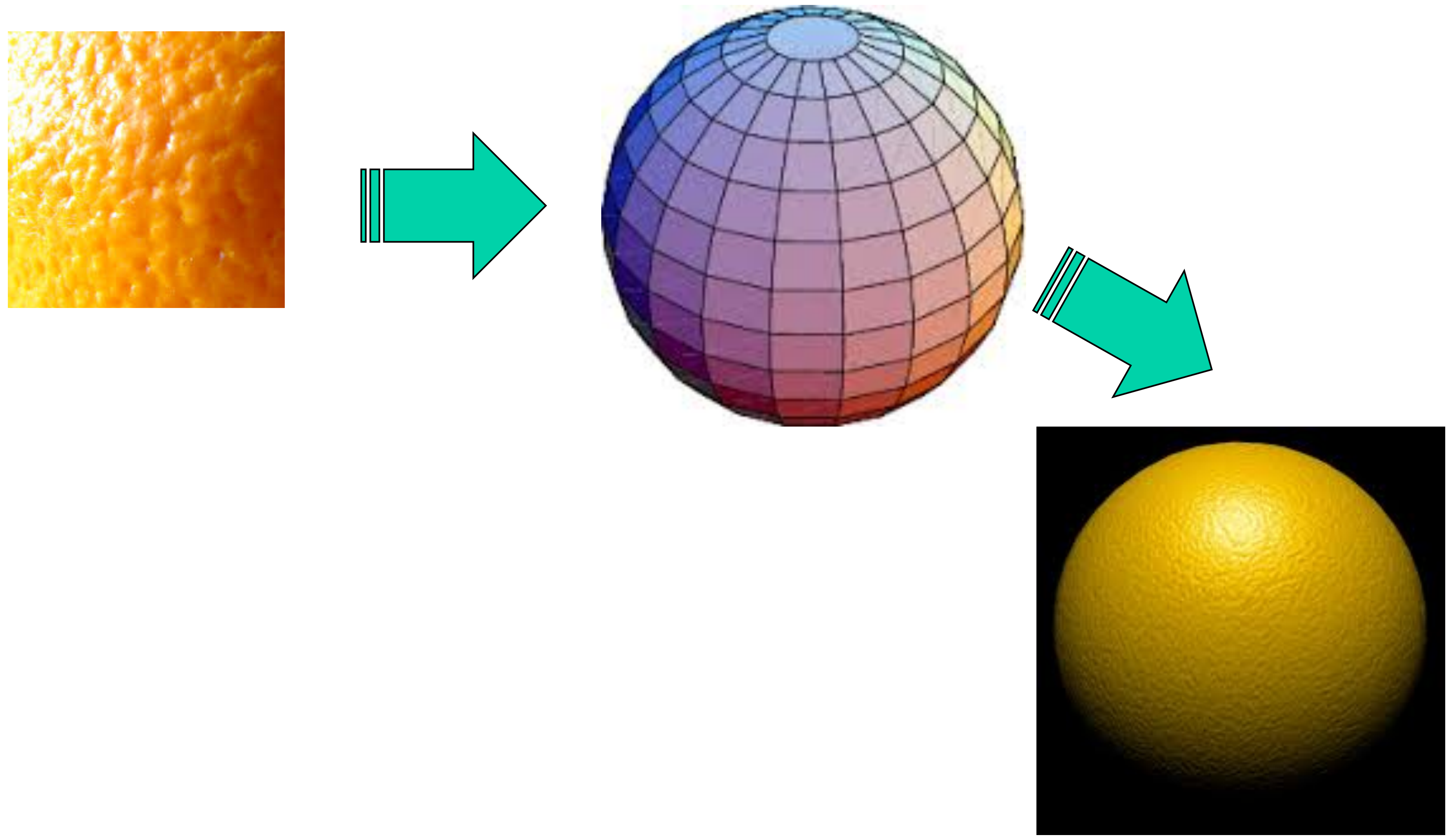


Orange

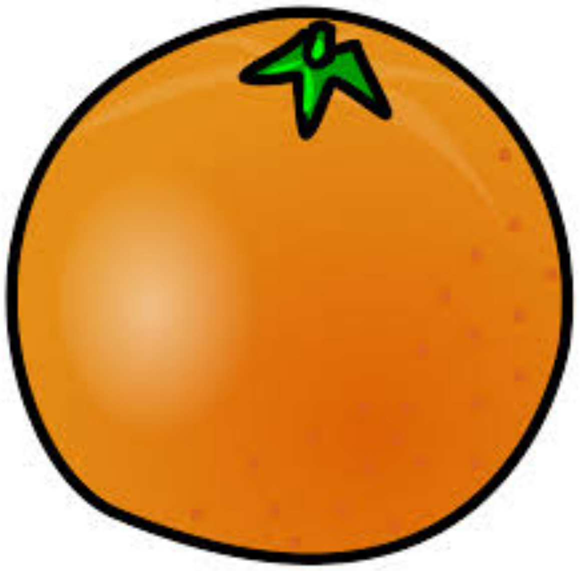
Orange Spheres



Texture Mapping

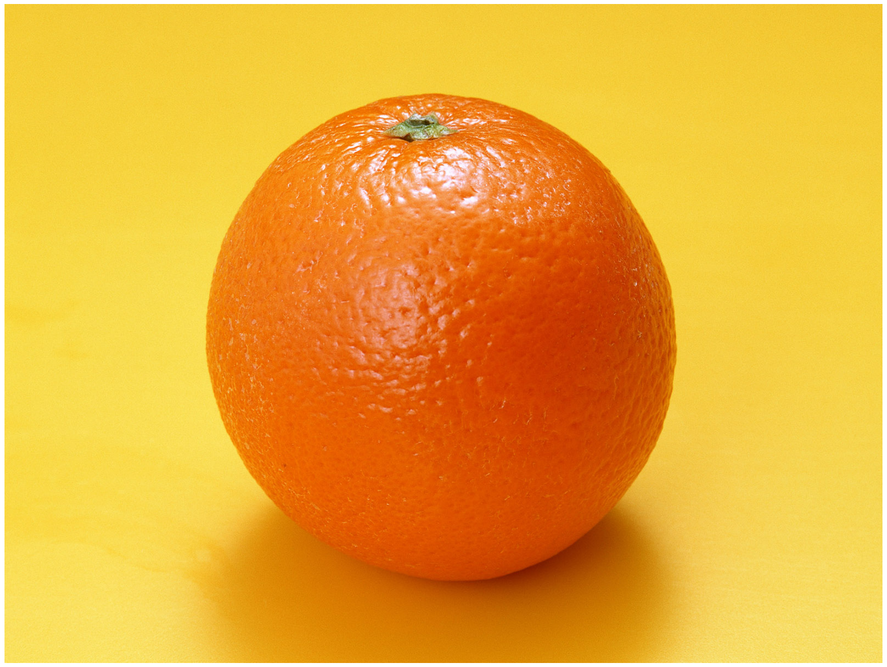


Looking Better

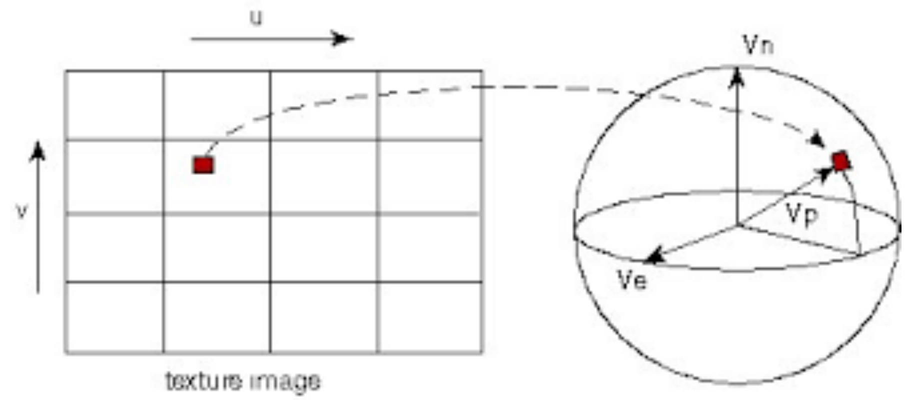


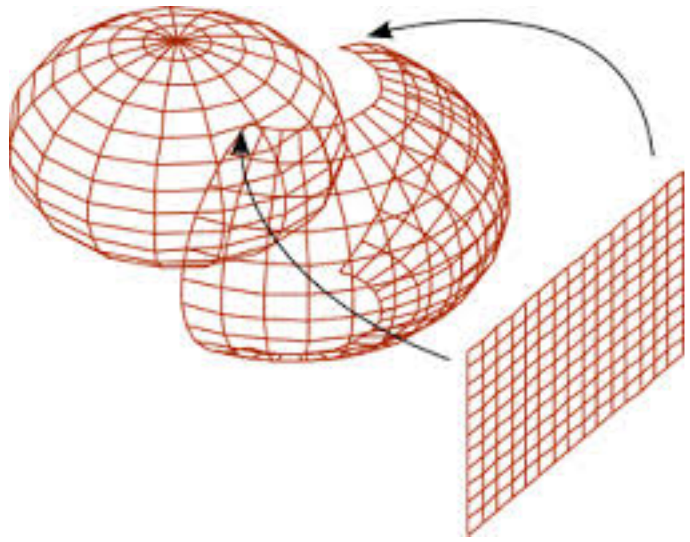
Still Not Enough

Local Variation

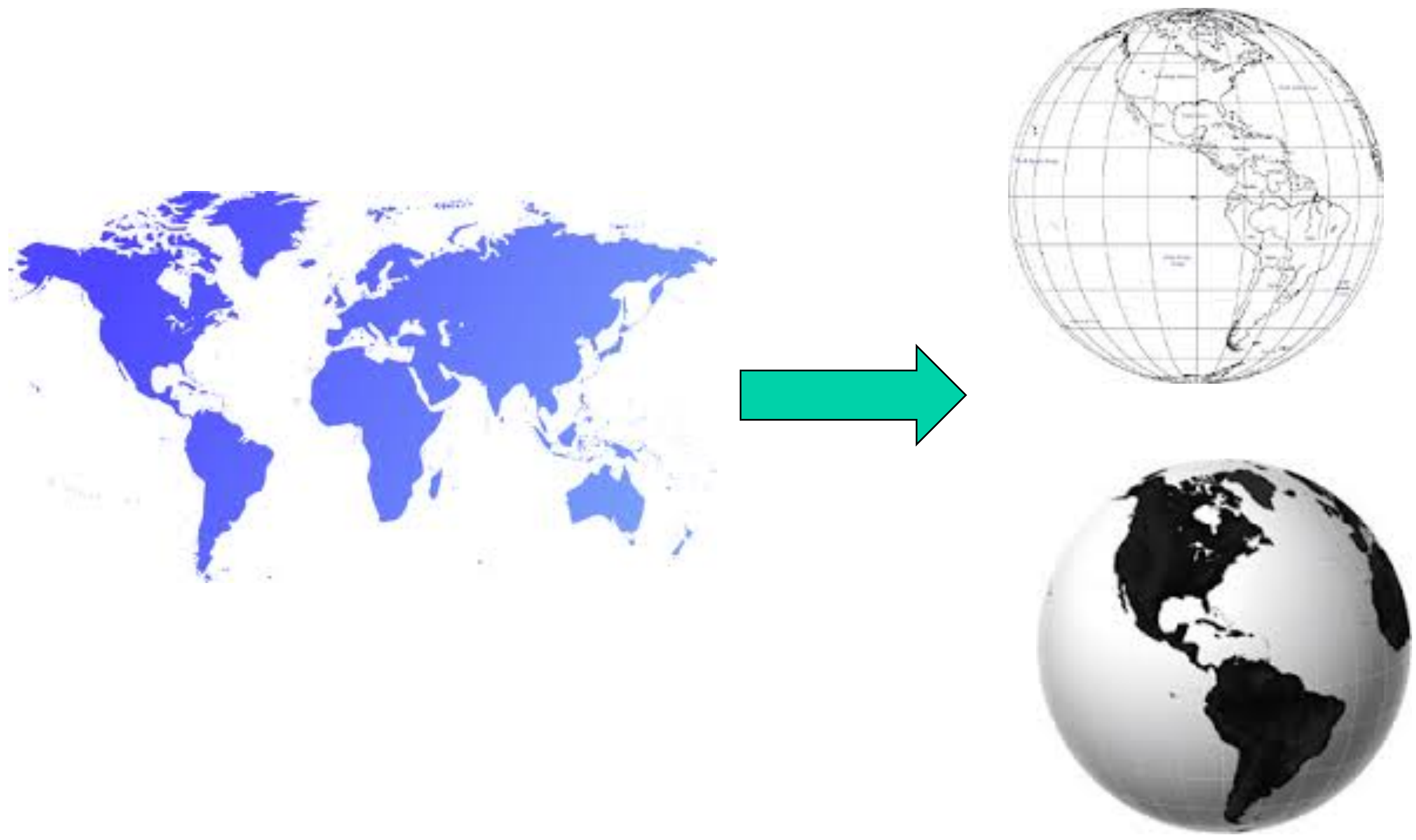


Texture Mapping

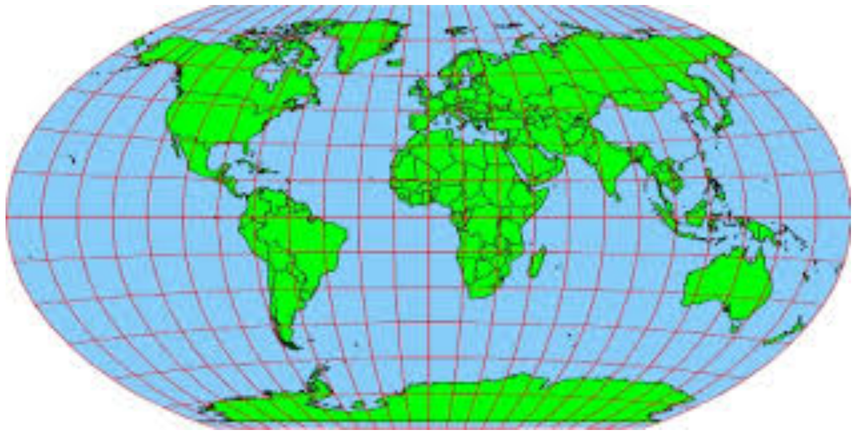




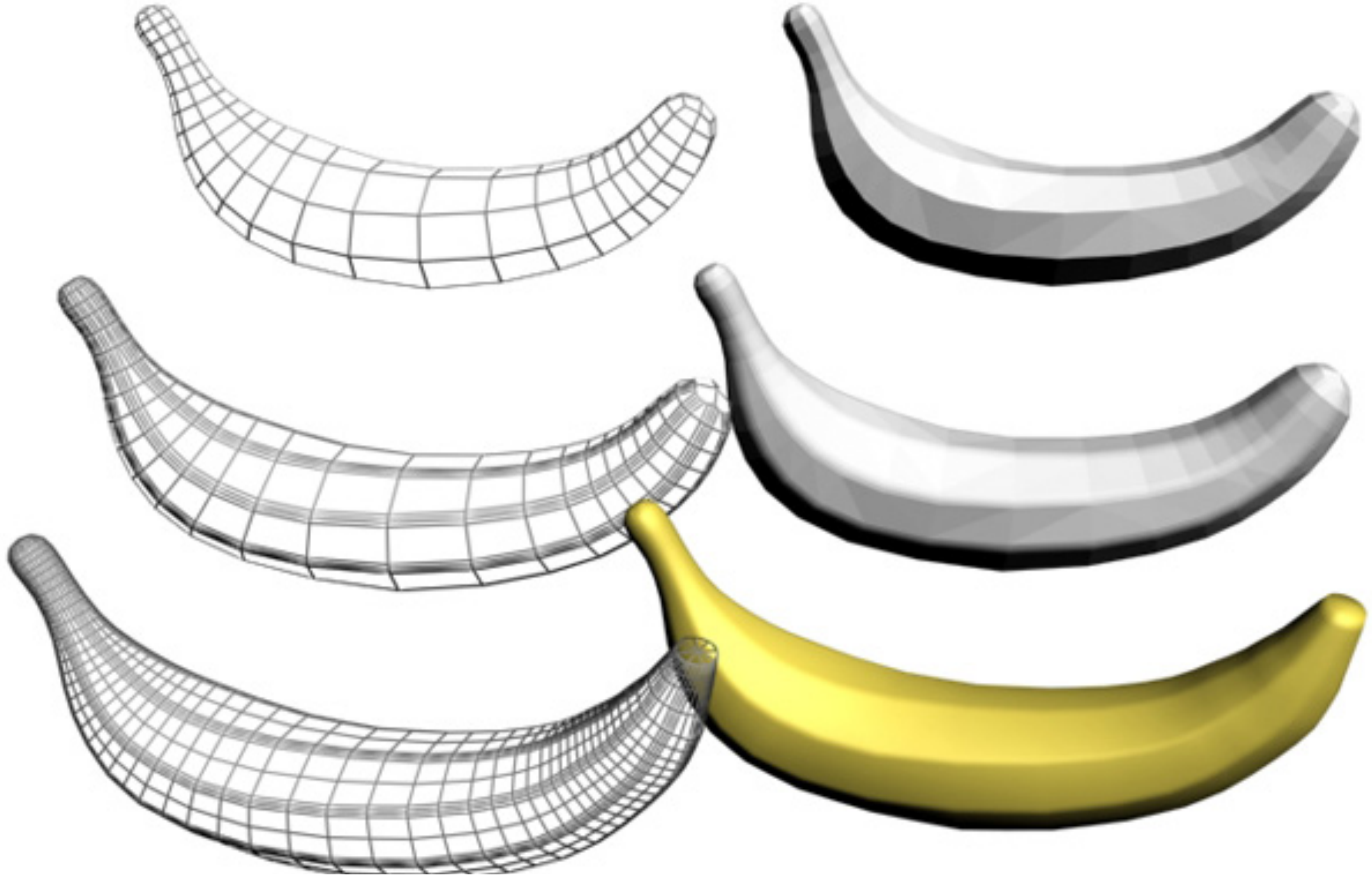
Globe



Not Mercator



Yet Another Fruit

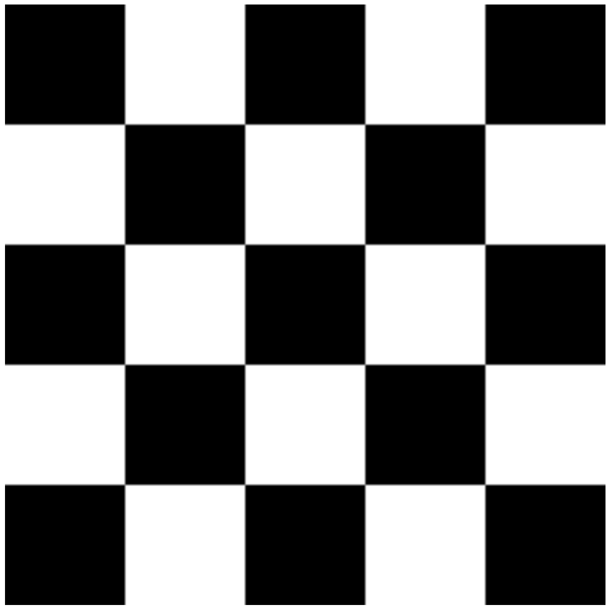


Three Types of Mapping

Generating Textures

Pictures

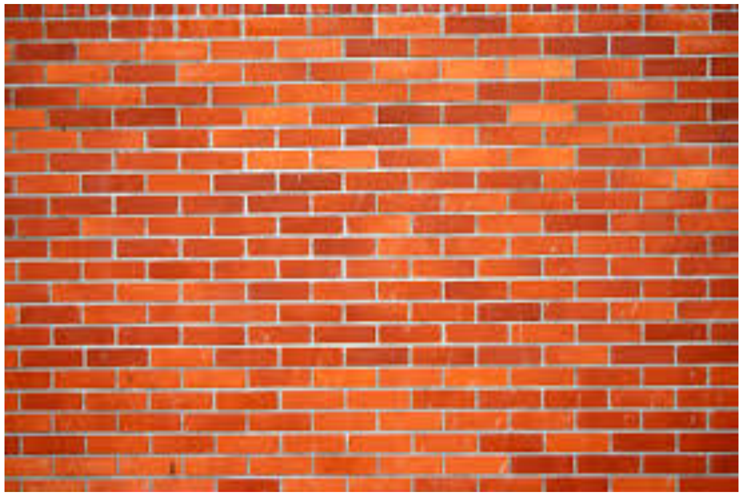
Algorithms



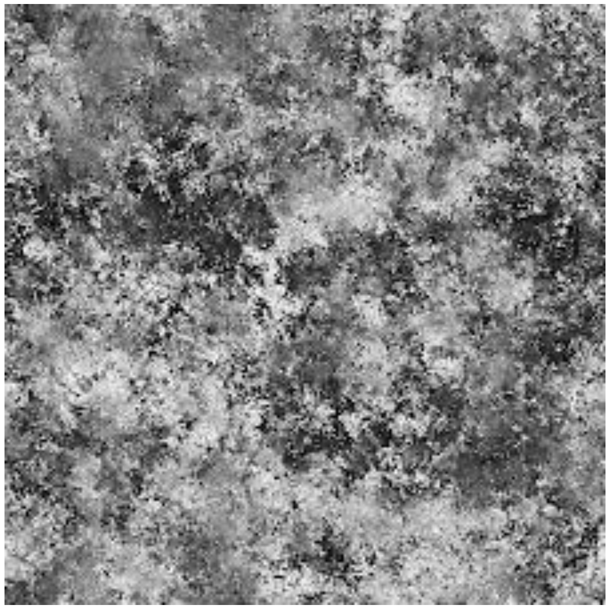
Checkerboard Texture

```
 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;
```

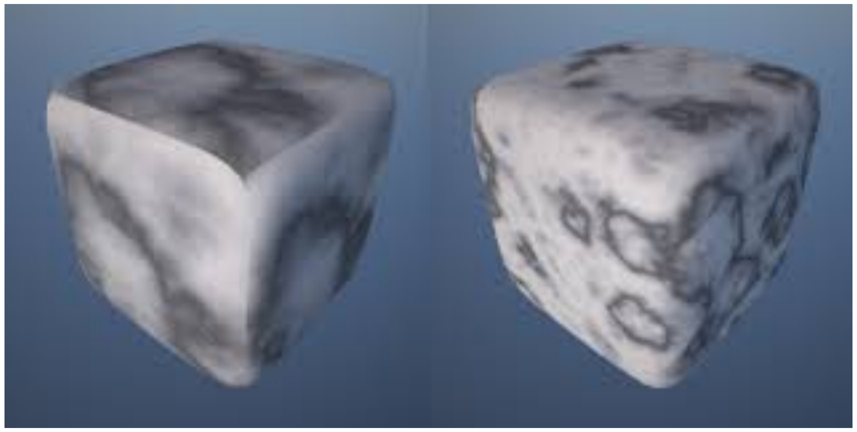
Brick Wall



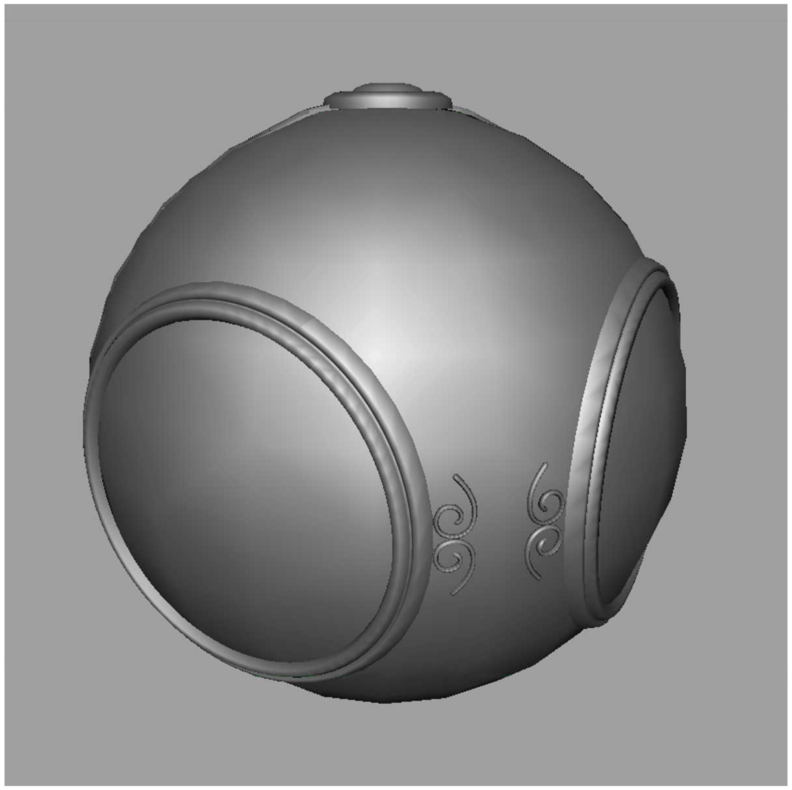
Noise

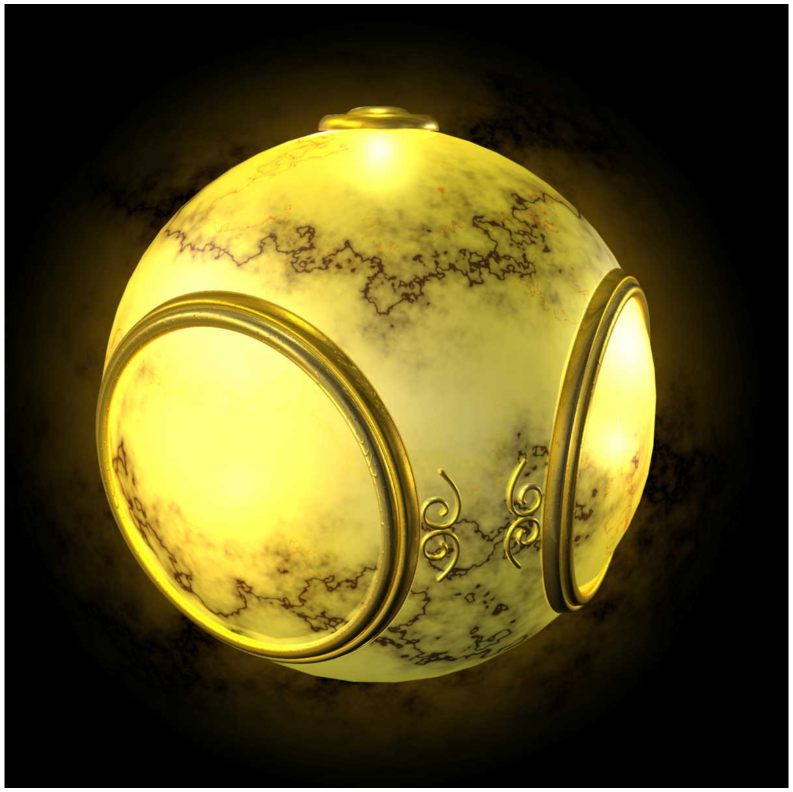


Marble



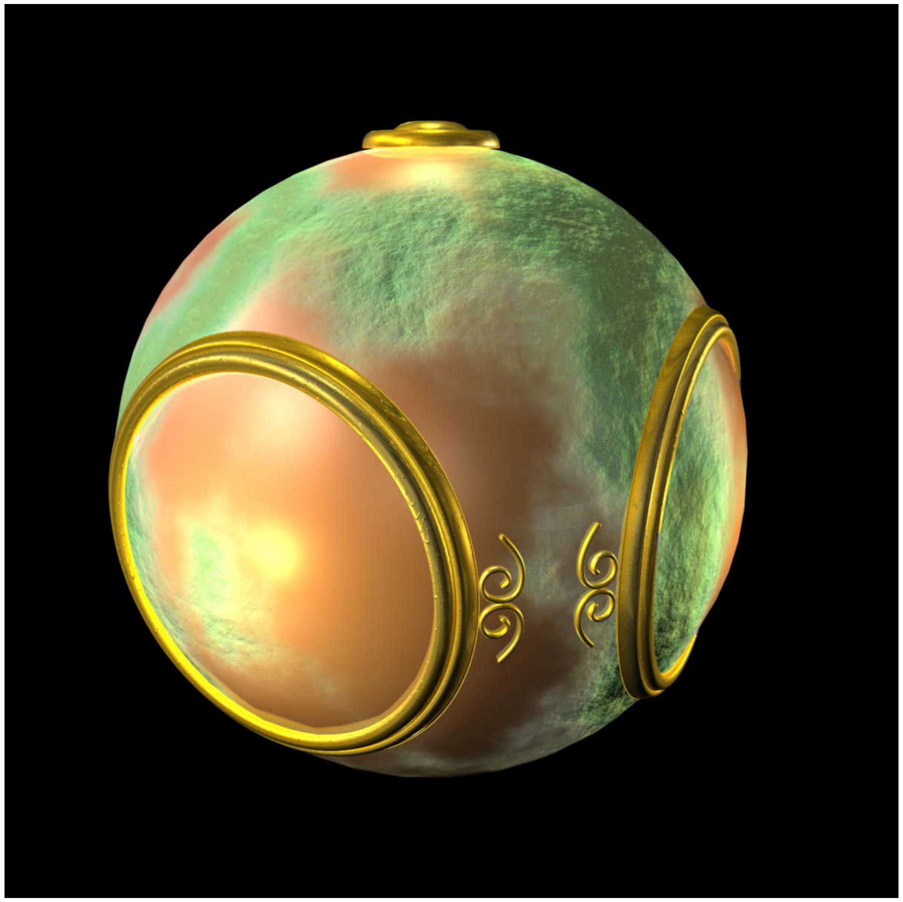
Texture Mapping


geometric model


texture mapped

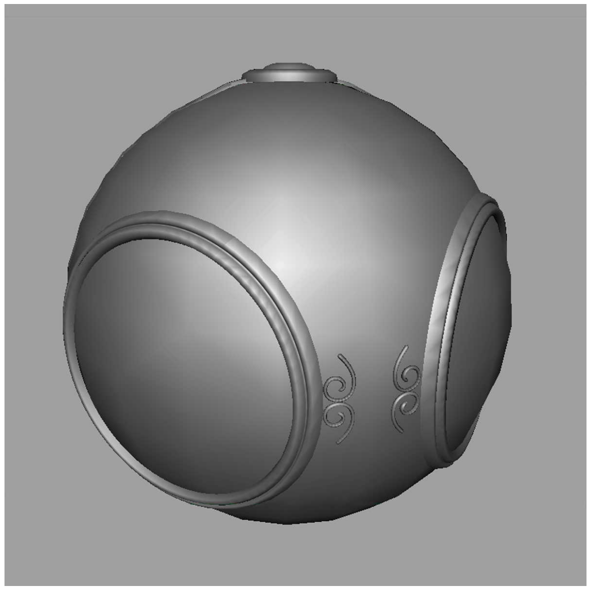
Environment Mapping

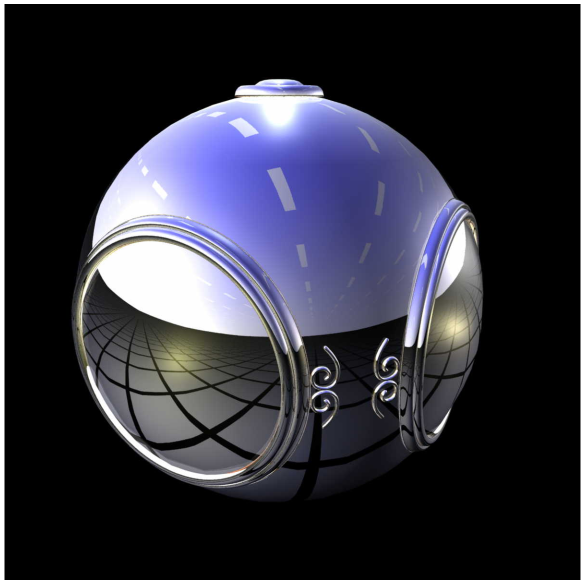
Bump Mapping

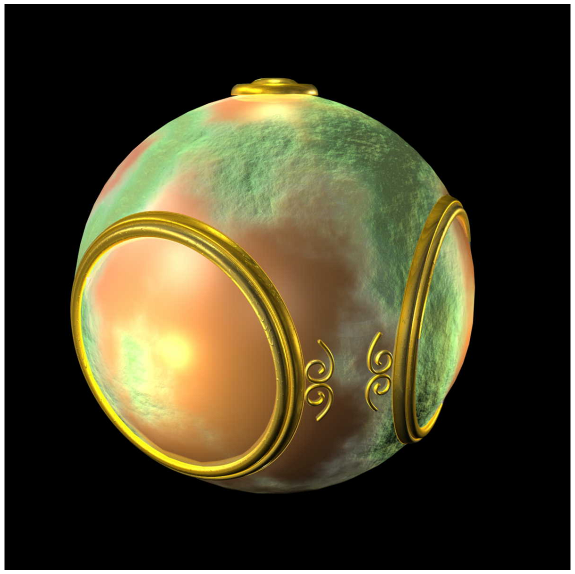


Three Types

Texture mapping


smooth shading

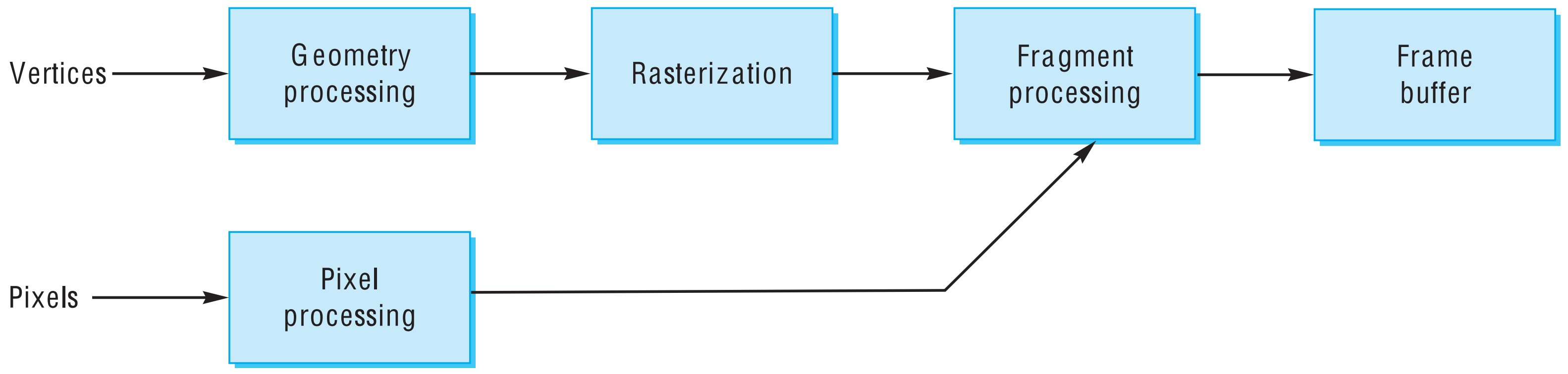

environment mapping


bump mapping

Texture Mapping - Pipeline

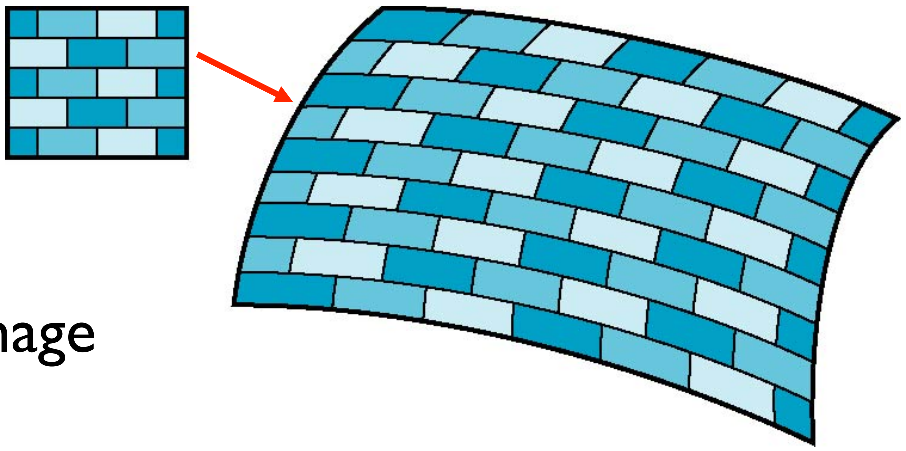
Mapping techniques are implemented at the end of the rendering pipeline

- Very efficient because few polygons make it past the clipper

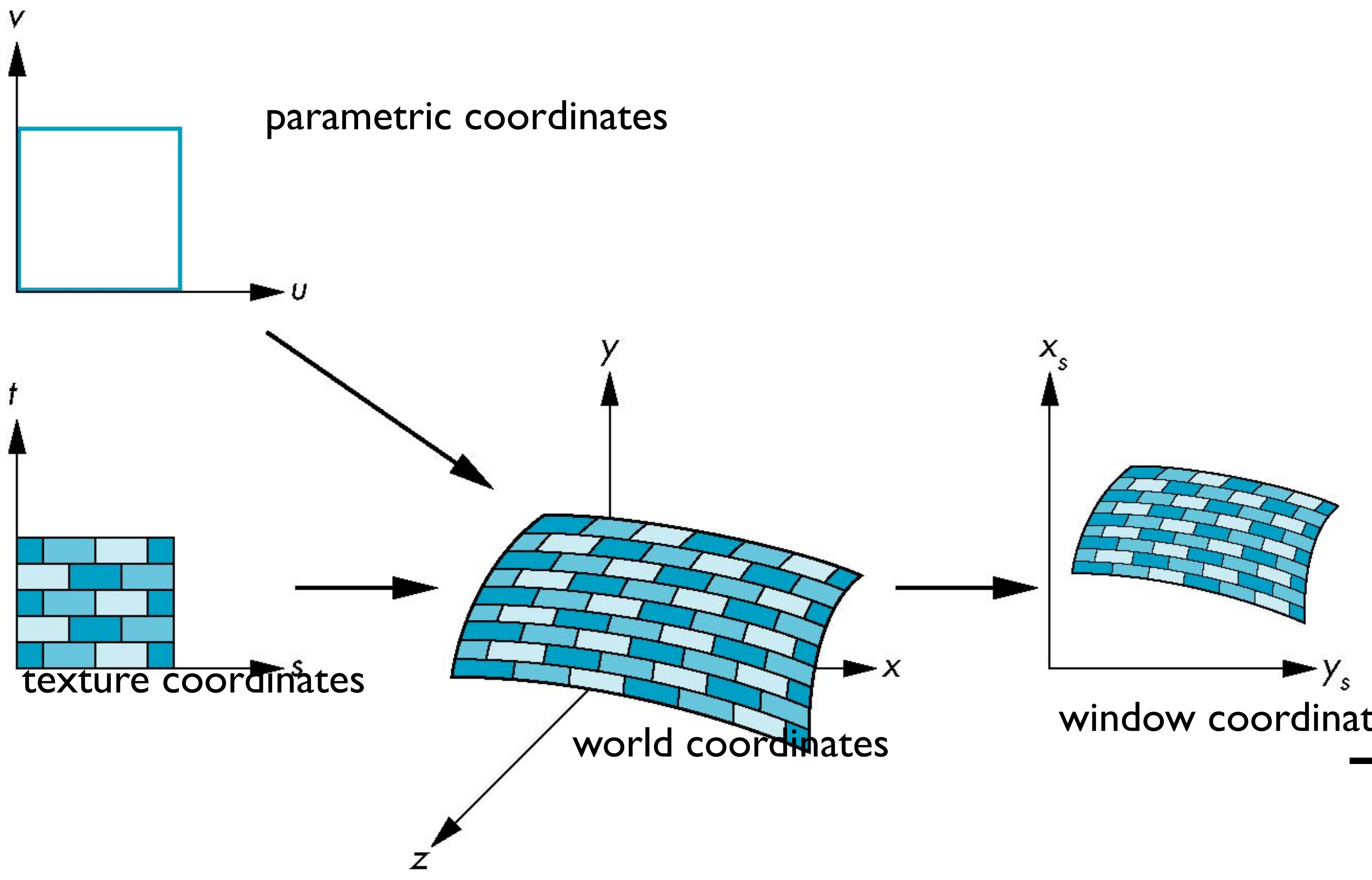


Mapping Mechanics

3 or 4 coordinate systems involved



Texture Mapping



Coordinate Systems

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

Mapping Functions

Mapping from texture coords to point on surface

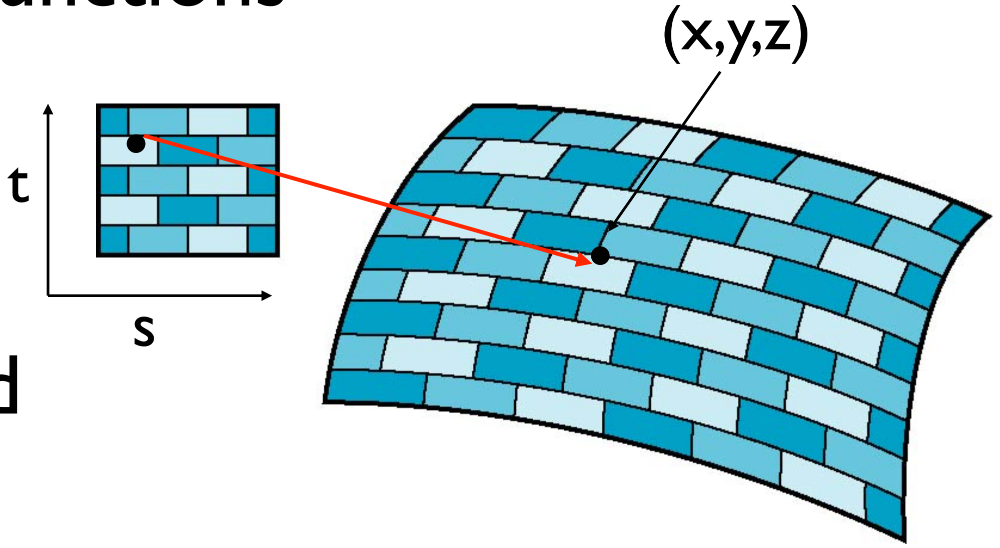
- Appear to need three functions

  $x = x(s,t)$

  $y = y(s,t)$

  $z = z(s,t)$

- Other direction needed



Backward Mapping

Mechanics

- Given a pixel want point on object it corresponds
- Given point on object want point in the texture it corresponds

Need a map of the form

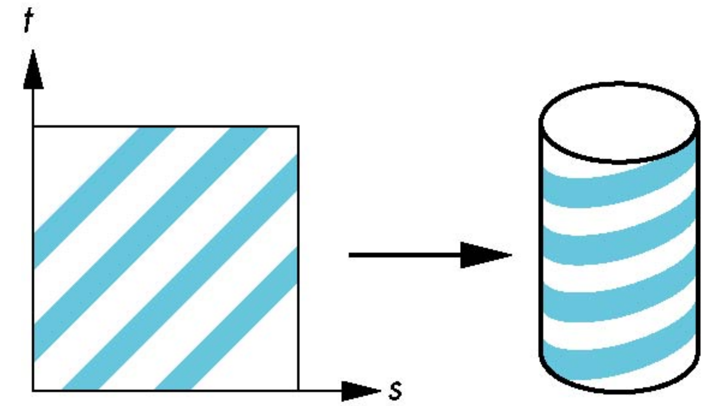
$s = s(x,y,z)$

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

<u>Such functions are difficult to find in general</u>

Two-part mapping

- First map texture to a simple intermediate surface
- 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

We can use a parametric sphere

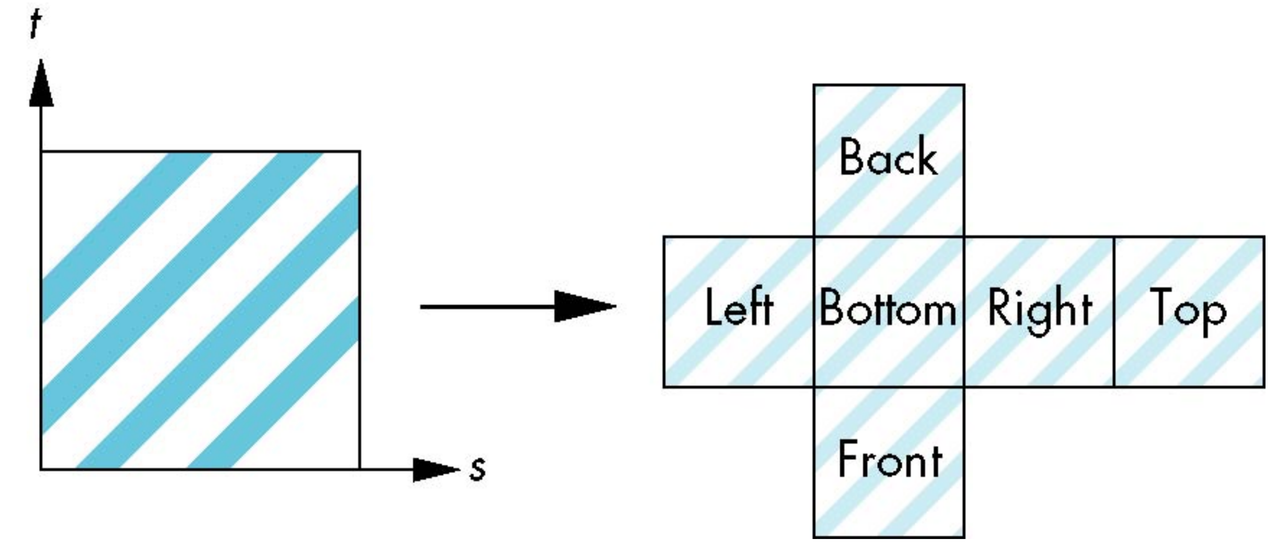
$x = r \cos 2\pi u$
$y = r \sin 2\pi u \cos 2\pi v$
$z = r \sin 2\pi u \sin 2\pi v$

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

Spheres are used in environmental maps

Box Mapping

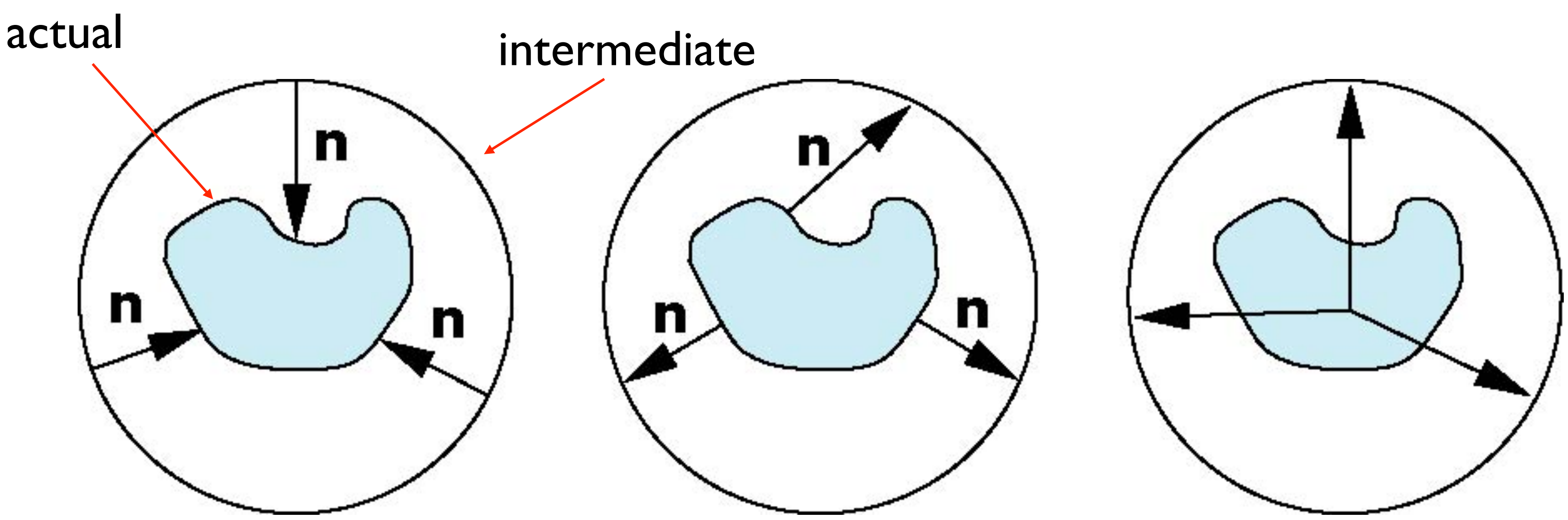
- Easy to use with simple orthographic projection
- Also used in environment maps



Second Mapping

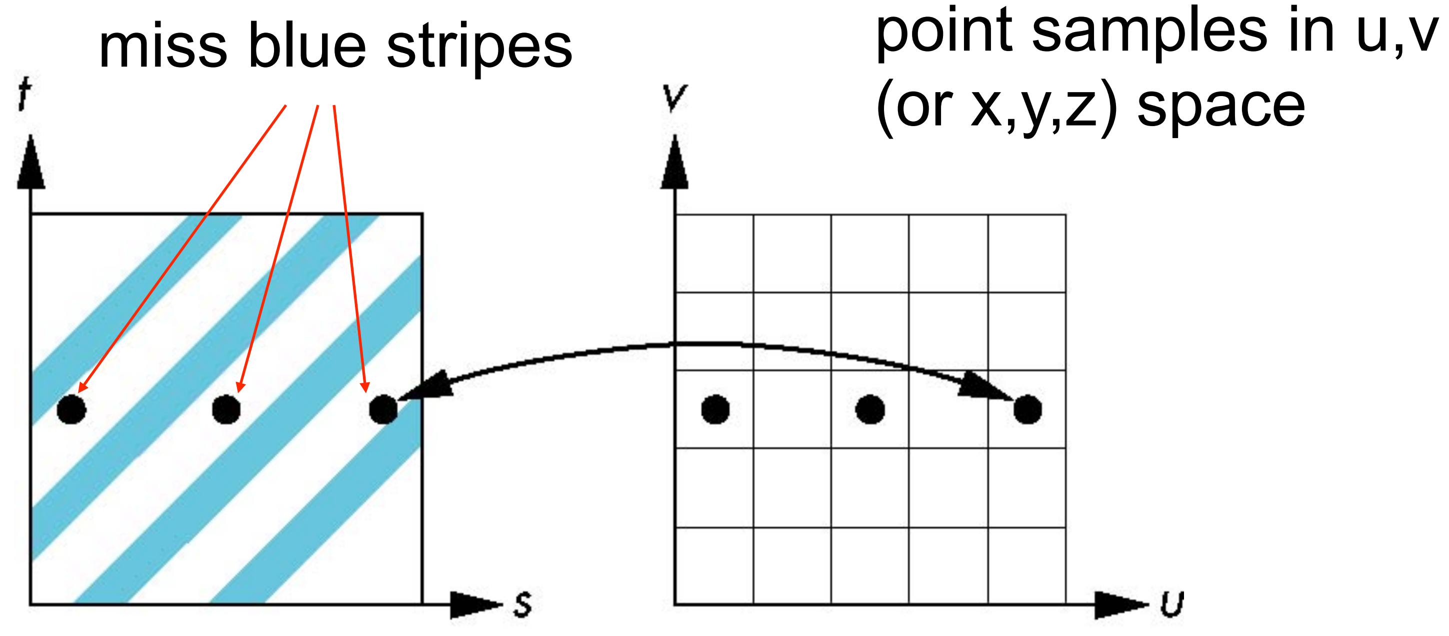
Map from intermediate object to actual object

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



Aliasing

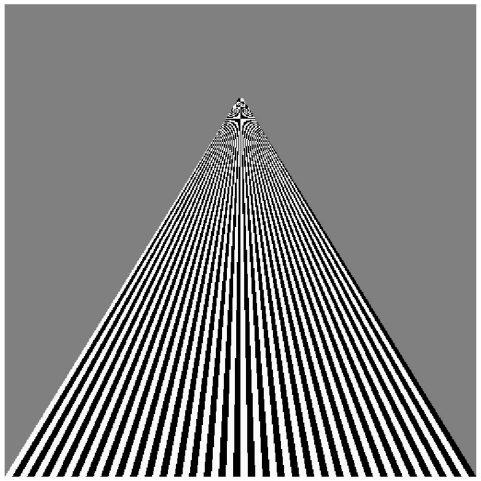
Point sampling of texture leads to aliasing errors

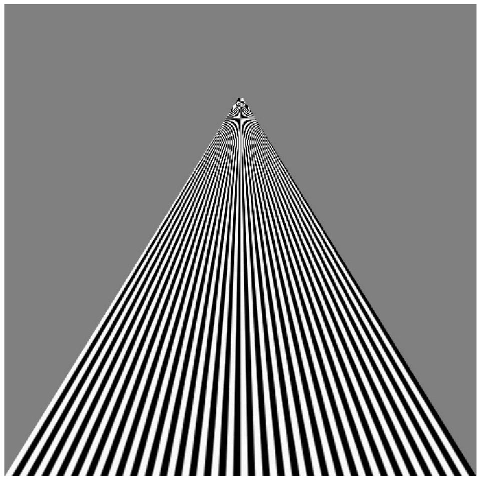


point samples in texture space

Anti-Aliasing in Textures

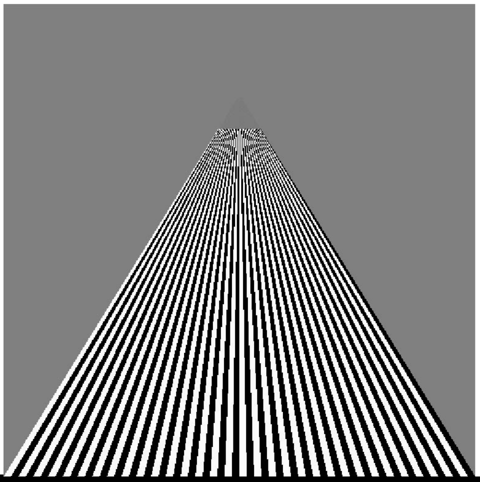
point sampling

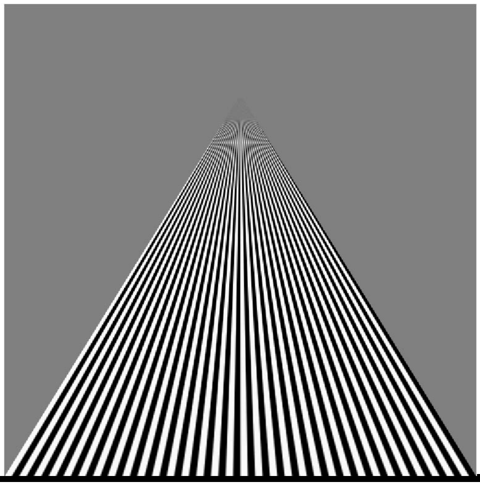




linear filtering

mipmapped point sampling

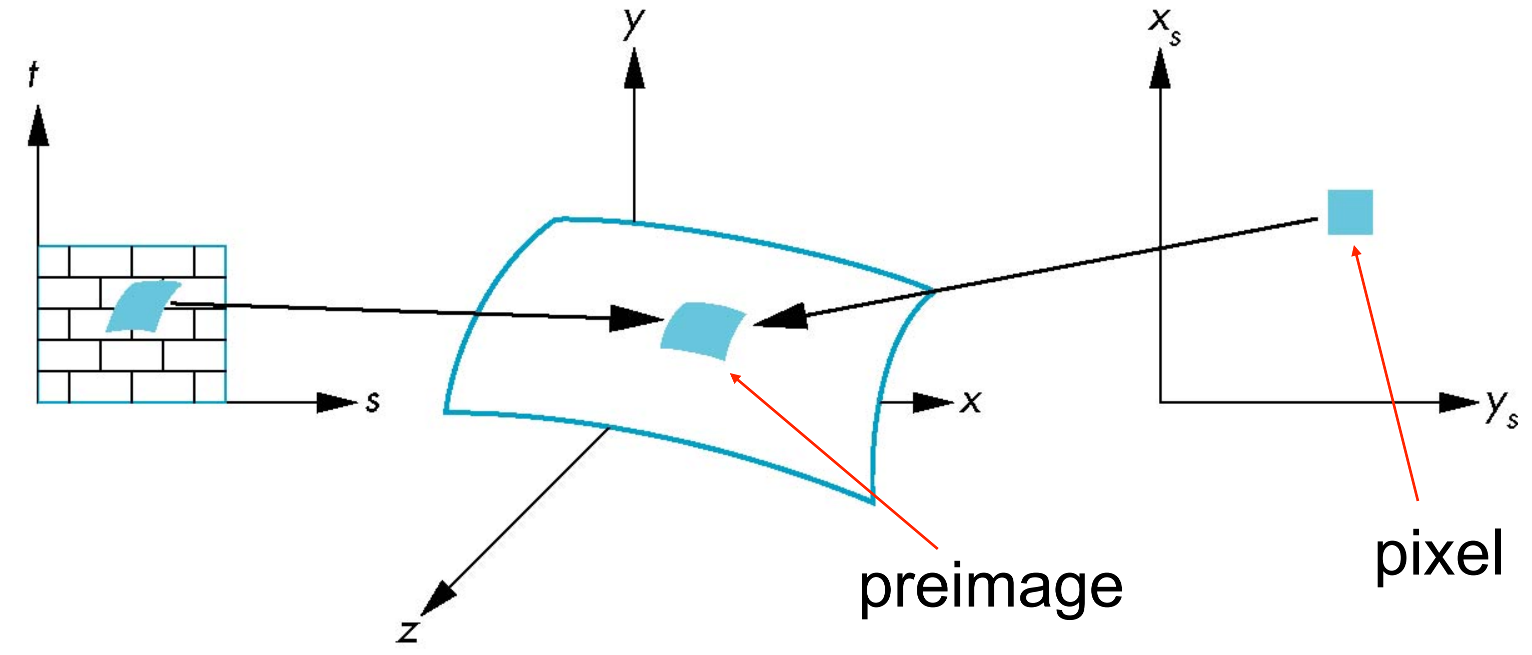




mipmapped linear filtering

Area Averaging

A better but slower option is to use *area averaging*



Note that *preimage* of pixel is curved

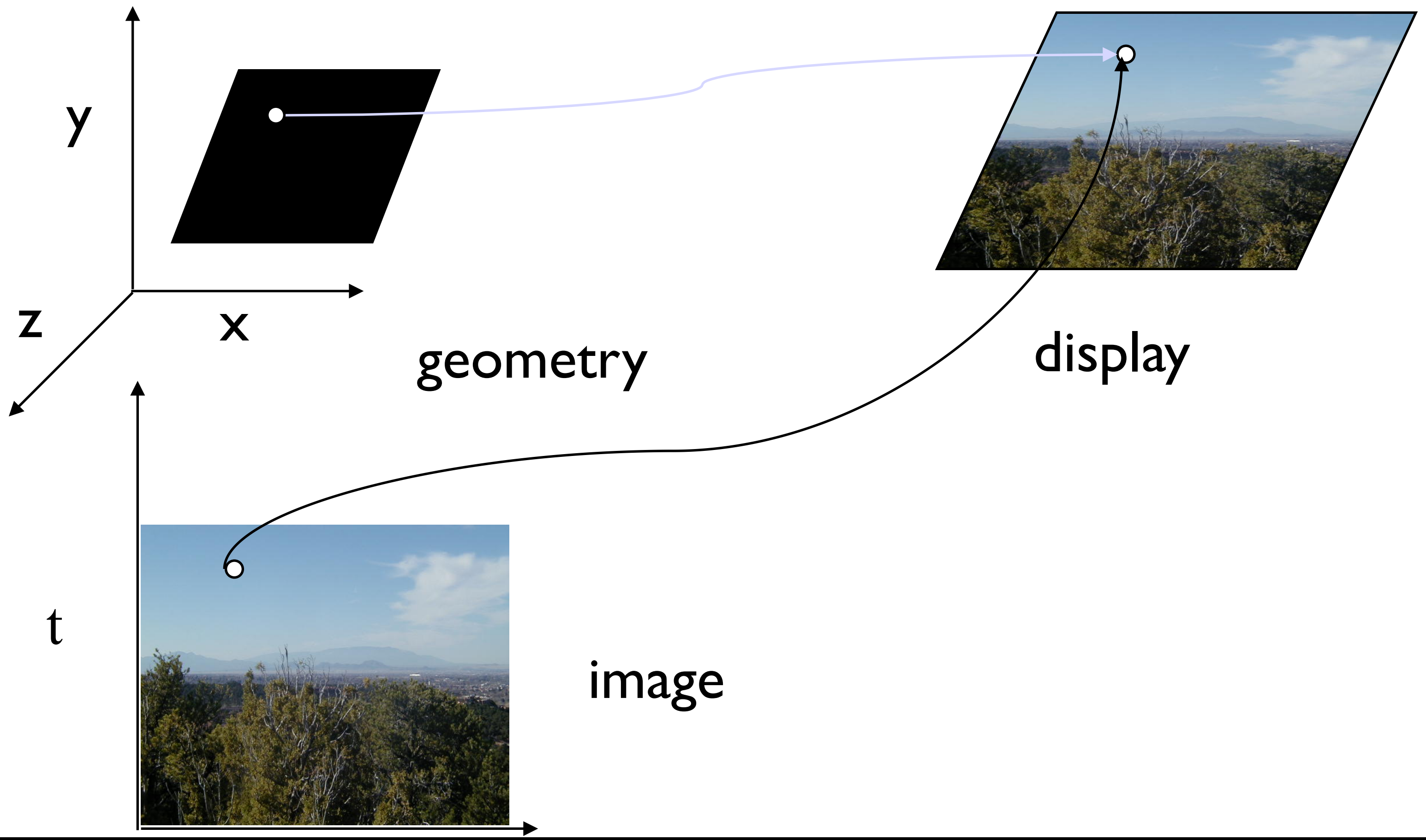
OpenGL Texture

Basic Stragegy

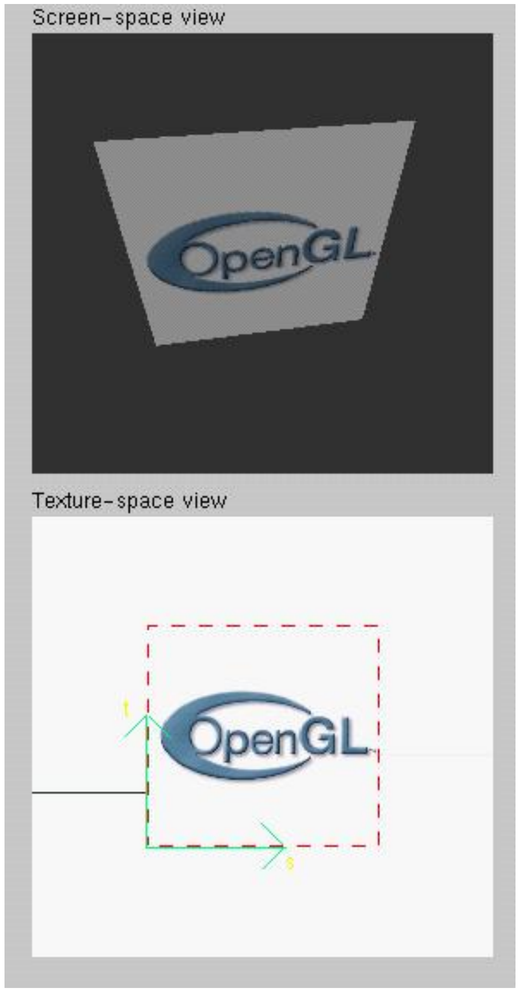
Three steps

1. Specify texture
    - read or generate image
    - assign to texture
    - enable texturing
2. Assign texture coordinates to vertices
    - Proper mapping function is left to application
3. Specify texture parameters
    - wrapping, filtering

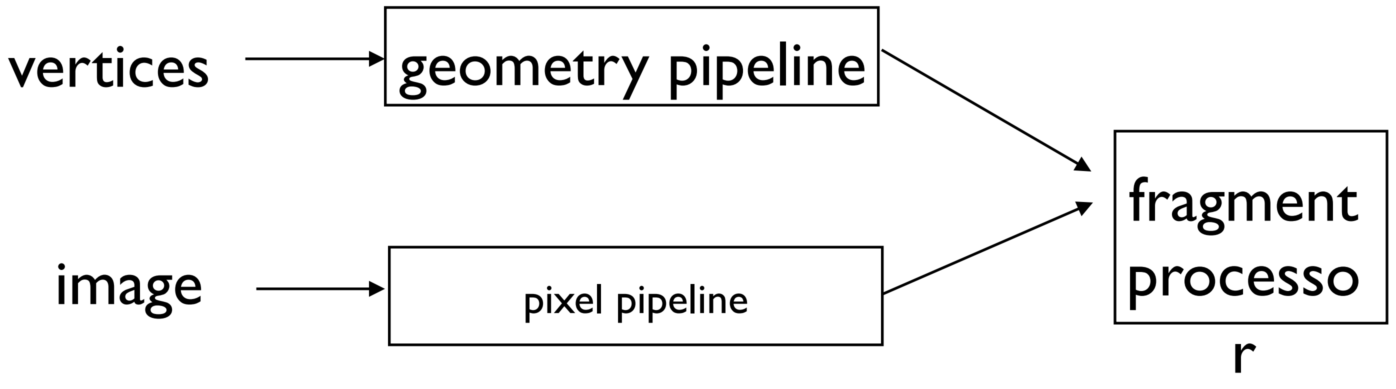
Texture Mapping



Texture Example



Texture Mapping in OpenGL



Specifying a Texture Image

- Define a texture image from an array of *texels* (texture elements) in CPU memory
  Glubyte my_texels[512][512];
- Define as any other pixel map
  - Scanned image
  - Generate by application code
- Enable texture mapping
  - **glEnable(GL_TEXTURE_2D)**
  - OpenGL supports 1-4 dimensional texture maps

Defining a Texture Image

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

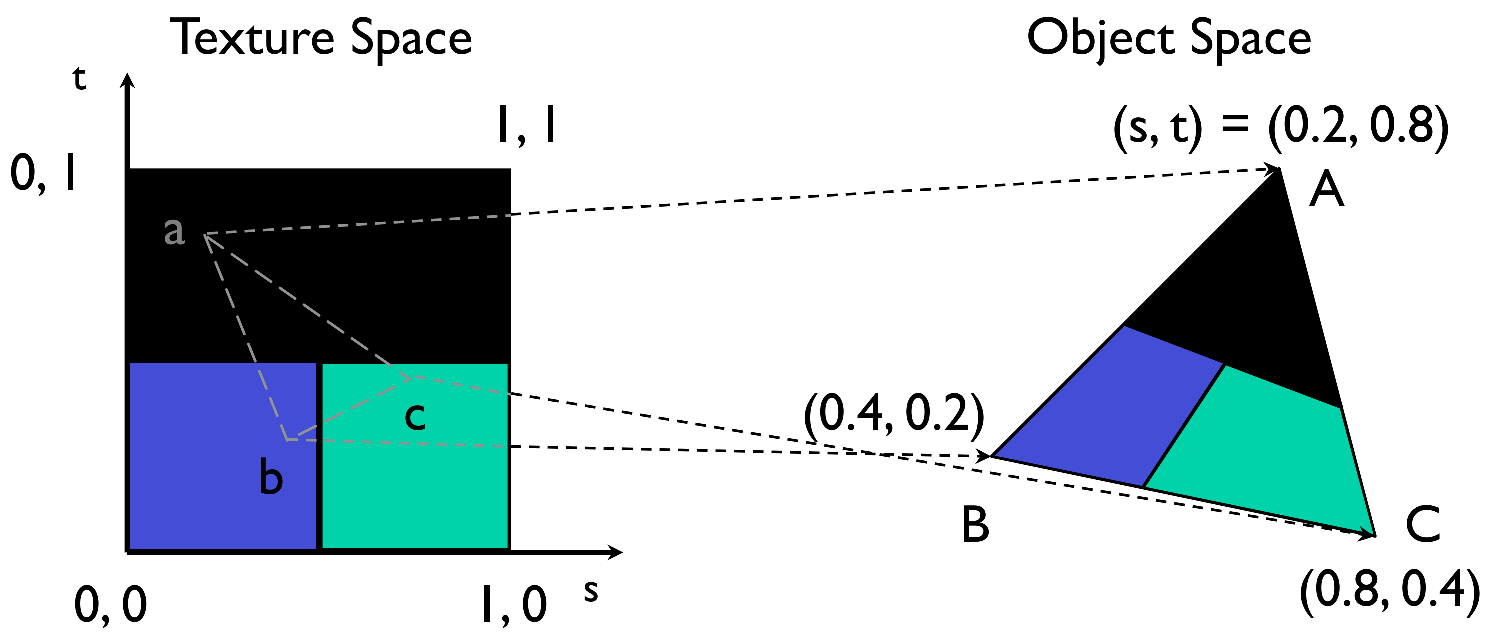
format and type: describe texels

texels: pointer to texel array

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

Mapping a Texture

- Based on parametric texture coordinates
- **glTexCoord*()** specified at each vertex



GLSL - Typical Code

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

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

Adding Texture Coordinates

```
void quad( int a, int b, int c, int d )
{
    quad_colors[Index] = colors[a];
    points[Index] = vertices[a];
    tex_coords[Index] = vec2( 0.0, 0.0 );
    index++;
   quad_colors[Index] = colors[a];
    points[Index] = vertices[b];
    tex_coords[Index] = vec2( 0.0, 1.0 );
    Index++;

// other vertices
}
```

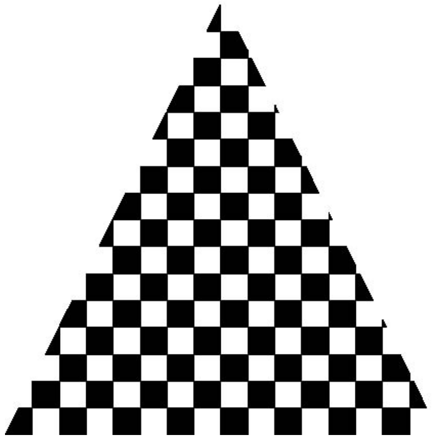
Role of Interpolation

Interpolation

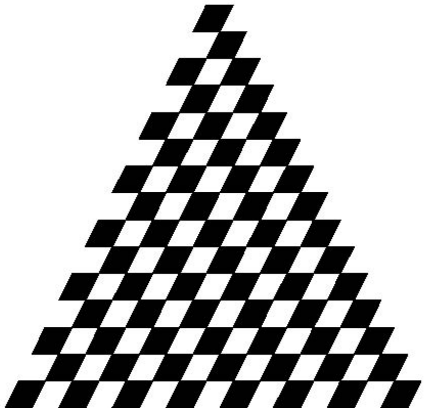
OpenGL uses interpolation to find proper texels from specified texture coordinates

Can be distorted

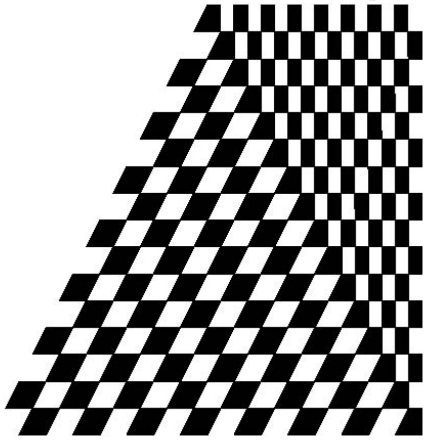
good selection of tex coordinates



poor selection of tex coordinates



texture stretched over trapezoid showing effects of bilinear interpolation



Interpolation

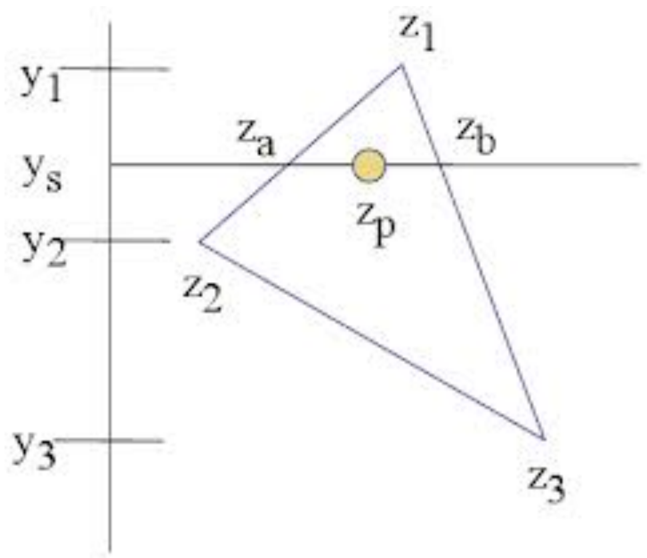
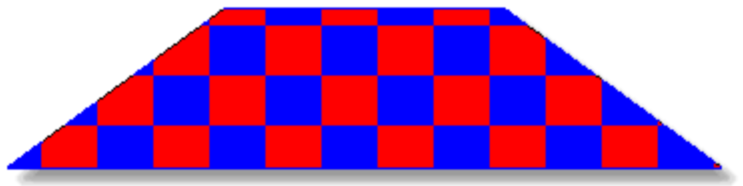
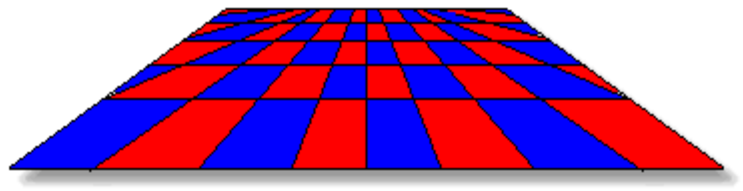


Figure 1.0 - Affine and perspective texture mapped polygons.


a. Affine texture mapping - notice no perspective cues.


b. Perspective texture mapping - notice 3D perspective both near and far.

Control of Texture Mapping

Texture Parameters

OpenGL has a variety of parameters that determine how texture is applied

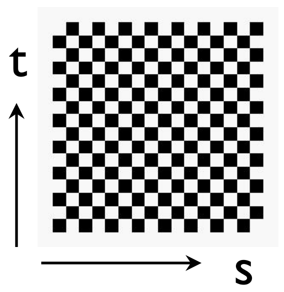
- Wrapping parameters determine what happens if s and t are outside the (0,1) range
- Filter modes allow us to use area averaging instead of point samples
- Mipmapping allows us to use textures at multiple resolutions
- Environment parameters determine how texture mapping interacts with shading

Wrapping Mode

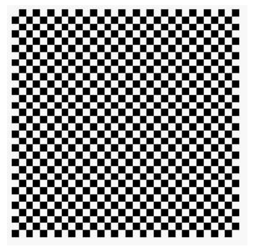
Clamping: if s,t > 1 use 1, if s,t <0 use 0

Wrapping: use s,t modulo 1

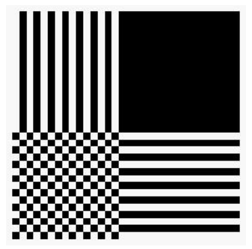
```
glTexParameteri( GL_TEXTURE_2D,
        GL_TEXTURE_WRAP_S, GL_CLAMP )
glTexParameteri( GL_TEXTURE_2D,
        GL_TEXTURE_WRAP_T, GL_REPEAT )
```



texture



GL_REPEAT wrapping

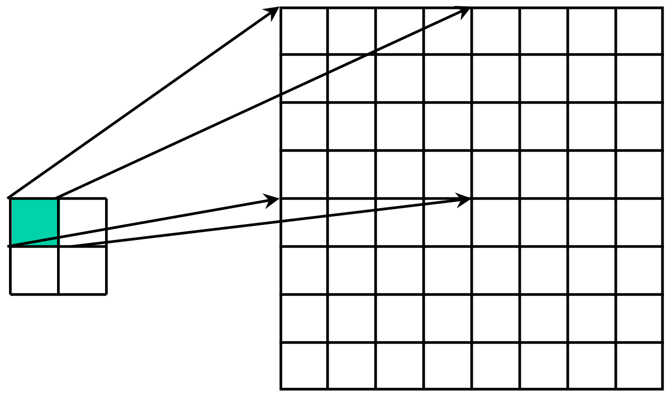


GL_CLAMP wrapping

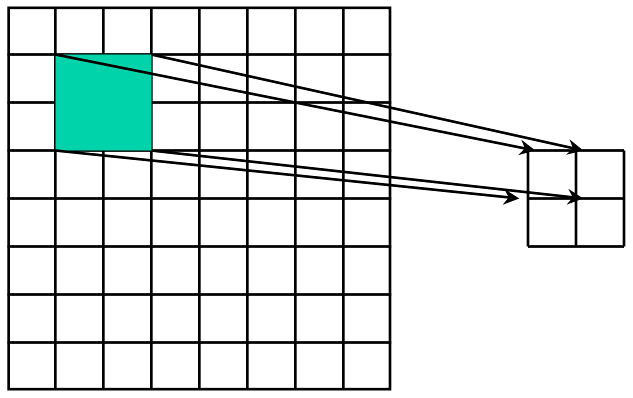
Magnification/Minification

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



Magnification



Minification

Filter Modes

Modes determined by

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

**glTexParameteri(GL_TEXTURE_2D, GL_TEXURE_MAG_FILTER,
GL_NEAREST);**

**glTexParameteri(GL_TEXTURE_2D, GL_TEXURE_MIN_FILTER,
GL_LINEAR);**

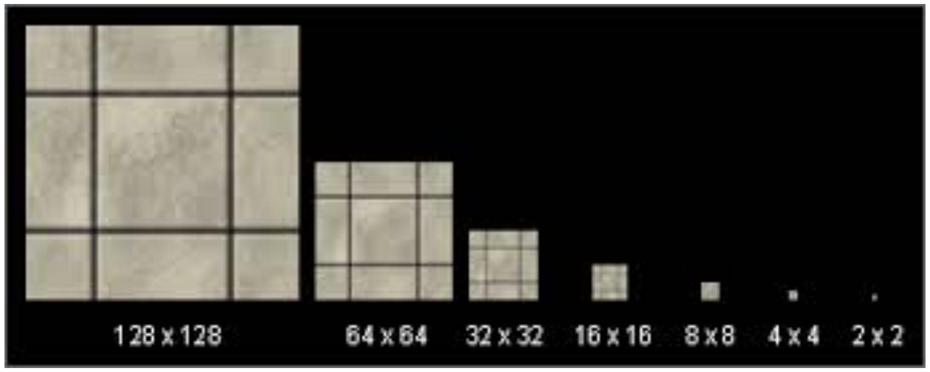
Note that linear filtering requires a border of an extra texel for filtering at edges (border = 1)

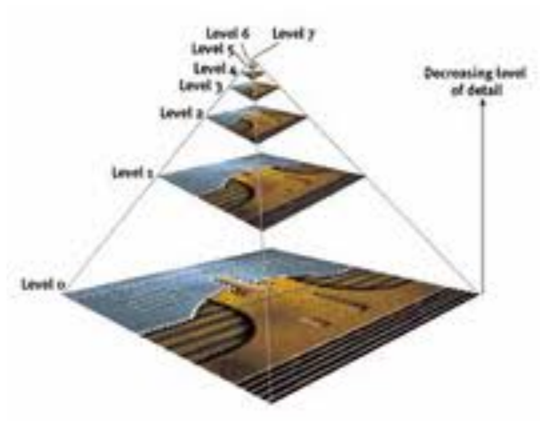
Mipmapped Textures

- *Mipmapping* allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects
- Declare mipmap level during texture definition

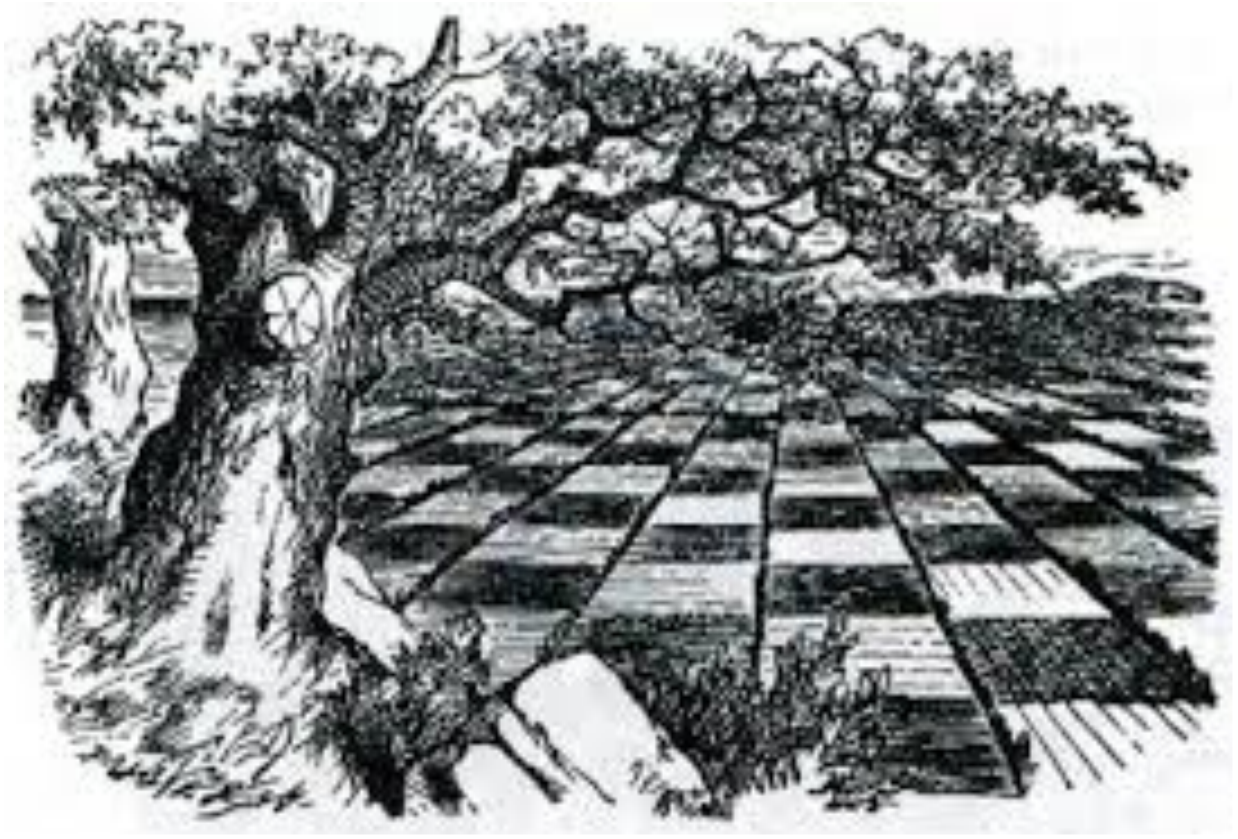
  glTexImage2D(GL_TEXTURE_*D, level, …)

MipMaps

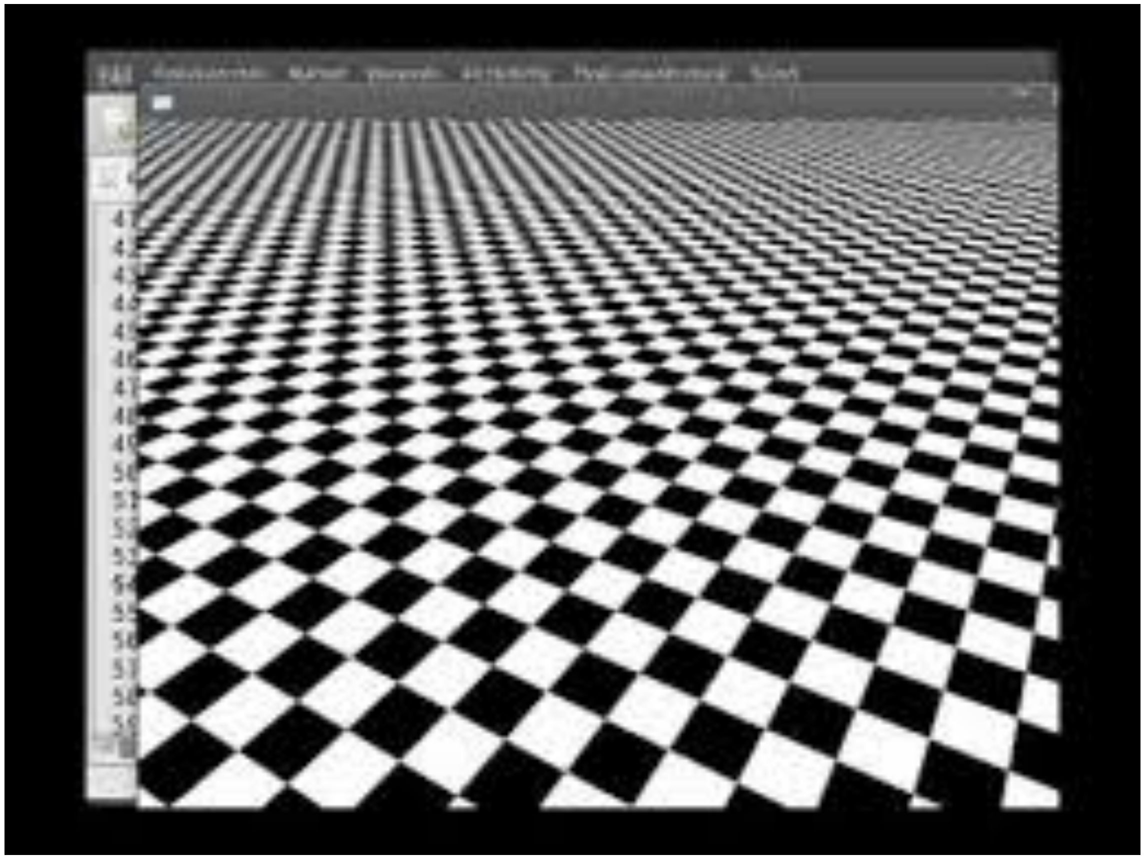

128 x 128
64 x 64
32 x 32
16 x 16
8 x 8
4 x 4
2 x 2


Level 6
Level 7
Level 5
Level 4
Level 3
Level 2
Level 1
Level 0
Decreasing level of detail

Mip-Mapping

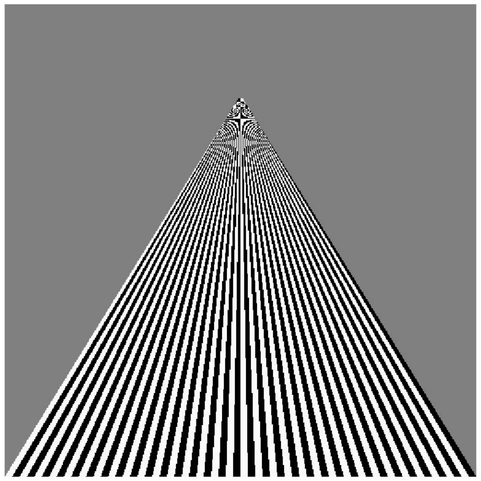


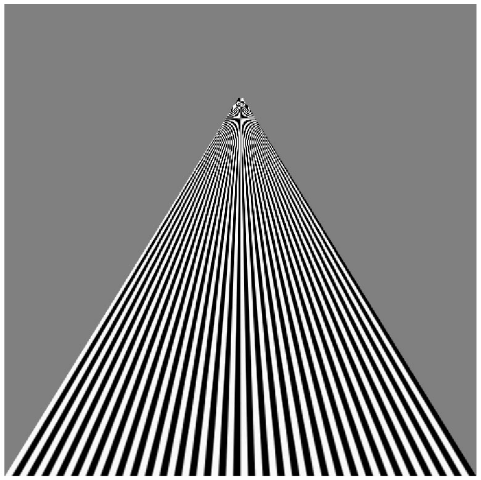
Mip-Mapping



Example

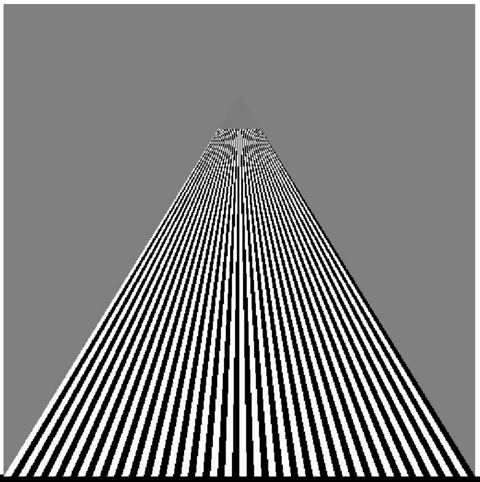
point sampling

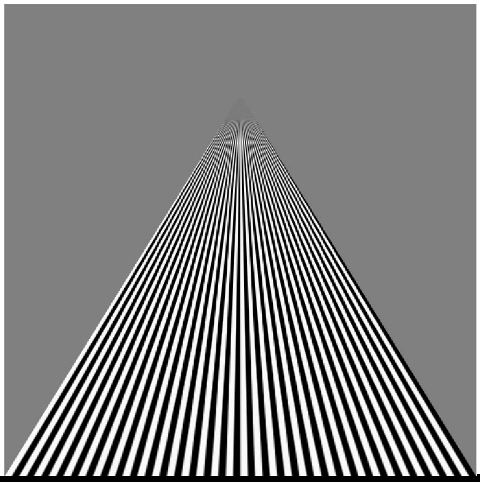




linear filtering

mipmapped point sampling





mipmapped linear filtering

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**

Using Texture Objects

1. specify textures in texture objects
2. set texture filter
3. set texture function
4. set texture wrap mode
5. set optional perspective correction hint
6. bind texture object
7. enable texturing
8. supply texture coordinates for vertex
   – coordinates can also be generated

Other Texture Features

- Environment Maps
  - Start with image of environment through a wide angle lens
    - Can be either a real scanned image or an image created in OpenGL
  - Use this texture to generate a spherical map
  - Alternative is to use a cube map
- Multitexturing
  - Apply a sequence of textures through cascaded texture units

GLSL

Samplers

https://www.opengl.org/wiki/Sampler_(GLSL)

Applying Textures

- Textures are applied during fragment shading by a **sampler**
- Samplers return a texture color from a texture object

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

void main()  {
    gl_FragColor = color * texture2D( texture, texCoord );
}
```

Vertex Shader

- Usually vertex shader will output texture coordinates to be rasterized
- Must do all other standard tasks too
  - Compute vertex position
  - Compute vertex color if needed

```
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
```

Adding Texture Coordinates

```
void quad( int a, int b, int c, int d )
{
    quad_colors[Index] = colors[a];
    points[Index] = vertices[a];
    tex_coords[Index] = vec2( 0.0, 0.0 );
    index++;
   quad_colors[Index] = colors[a];
    points[Index] = vertices[b];
    tex_coords[Index] = vec2( 0.0, 1.0 );
    Index++;

// other vertices
}
```

Texture Object

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

glBindTexture( GL_TEXTURE_2D, textures[0] );
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, TextureSize,
   TextureSize, 0, GL_RGB, GL_UNSIGNED_BYTE, image );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
    GL_REPEAT );
glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
     GL_REPEAT );
glTexParameterf( GL_TEXTURE_2D,
      GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameterf( GL_TEXTURE_2D,
      GL_TEXTURE_MIN_FILTER, GL_NEAREST );
glActiveTexture( GL_TEXTURE0 );
```

Linking with Shaders

```
GLuint vTexCoord = glGetAttribLocation( program, "vTexCoord" );
glEnableVertexAttribArray( vTexCoord );
glVertexAttribPointer( vTexCoord, 2, GL_FLOAT, GL_FALSE, 0,
                       BUFFER_OFFSET(offset) );

// Set the value of the fragment shader texture sampler variable
//   ("texture") to the the appropriate texture unit. In this case,
//   zero, for GL_TEXTURE0 which was previously set by calling
//   glActiveTexture().
glUniform1i( glGetUniformLocation(program, "texture"), 0 );
```

Vertex Shader Applications

- Moving vertices
  - Morphing
  - Wave motion
  - Fractals
- Lighting
  - More realistic models
  - Cartoon shaders

Wave Motion Vertex Shader

```
uniform float time;
uniform float xs, zs, // frequencies
uniform float h; // height scale
uniform mat4 ModelView, Projection;
in vec4 vPosition;

void main() {
  vec4 t =vPosition;
  t.y = vPosition.y
     + h*sin(time + xs*vPosition.x)
     + h*sin(time + zs*vPosition.z);
  gl_Position = Projection*ModelView*t;
}
```

Particle System

```
uniform vec3 init_vel;
uniform float g, m, t;
uniform mat4 Projection, ModelView;
in vPosition;
void main(){
vec3 object_pos;
object_pos.x = vPosition.x + vel.x*t;
object_pos.y = vPosition.y + vel.y*t
        + g/(2.0*m)*t*t;
object_pos.z = vPosition.z + vel.z*t;
gl_Position = Projection*
   ModelView*vec4(object_pos,1);
}
```

Example

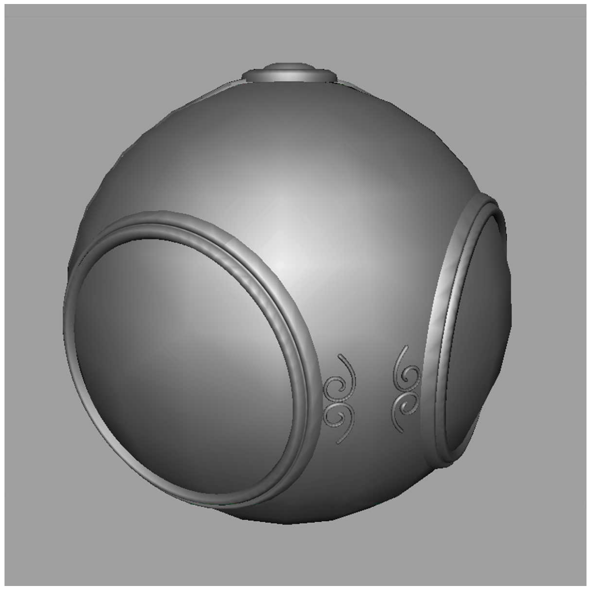
http://www.lighthouse3d.com/tutorials/glsl-core-tutorial/glsl-core-tutorial-texturing-with-images/

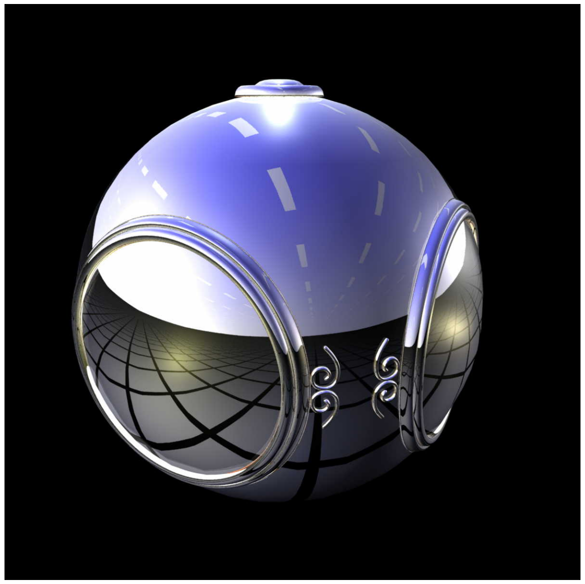
Example

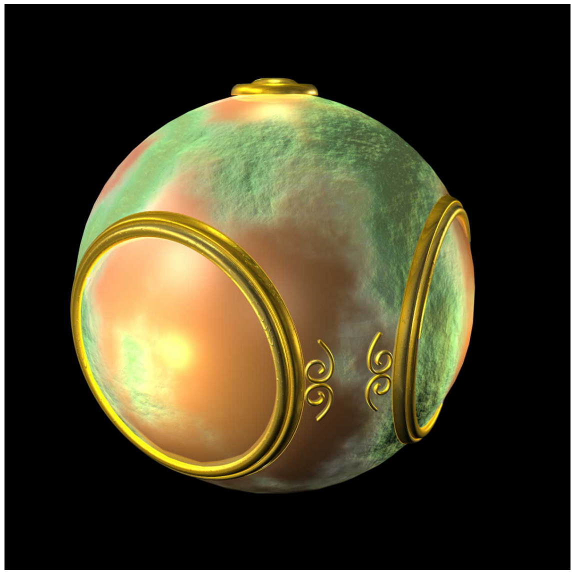
http://www.lighthouse3d.com/tutorials/glsl-tutorial/simple-texture/

Fragment Shader

Texture mapping


smooth shading


environment mapping


bump mapping

Cube Maps

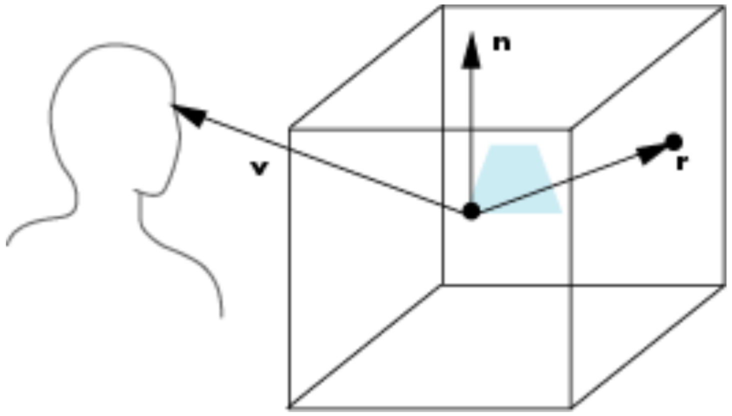
- We can form a cube map texture by defining six 2D texture maps that correspond to the sides of a box
- Supported by OpenGL
- Also supported in GLSL through cubemap sampler

```
vec4 texColor = textureCube(mycube, texcoord);
```

- Texture coordinates must be 3D

Environment Map

Use reflection vector to locate texture in cube map



Environment Maps with Shaders

- Computed in world coordinates
  - keep track of modeling matrix & pass as a uniform variable
- Use reflection map or refraction map
- Simulate water

Reflection Map Vertex Shader

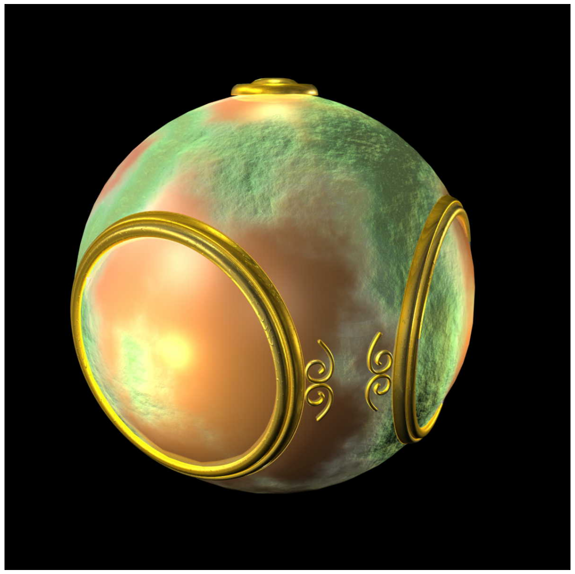
```
uniform mat4 Projection, ModelView, NormalMatrix;
in vec4 vPosition;
in vec4 normal;
out vec3 R;

void main(void)
{
   gl_Position = Projection*ModelView*vPosition;
   vec3 N = normalize(NormalMatrix*normal);
   vec4 eyePos = ModelView*gvPosition;
   R = reflect(-eyePos.xyz, N);
}
```

Reflection Map Fragment Shader

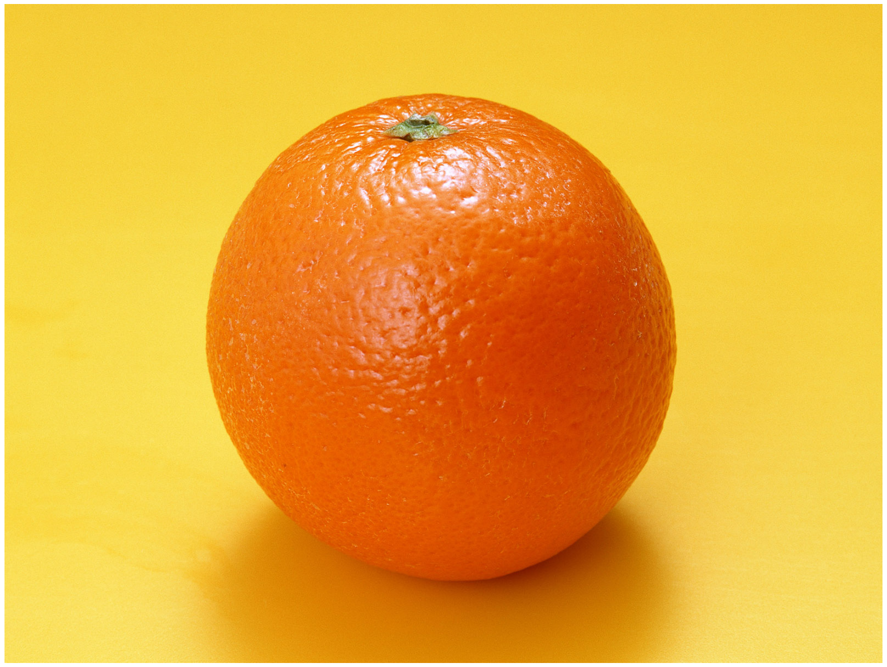
```
in vec3 R;
uniform samplerCube texMap;

void main(void)
{
    gl_FragColor = textureCube(texMap, R);
}
```

Bump Mapping

- Perturb normal for each fragment
- Store perturbation as textures



Back 2 Orange

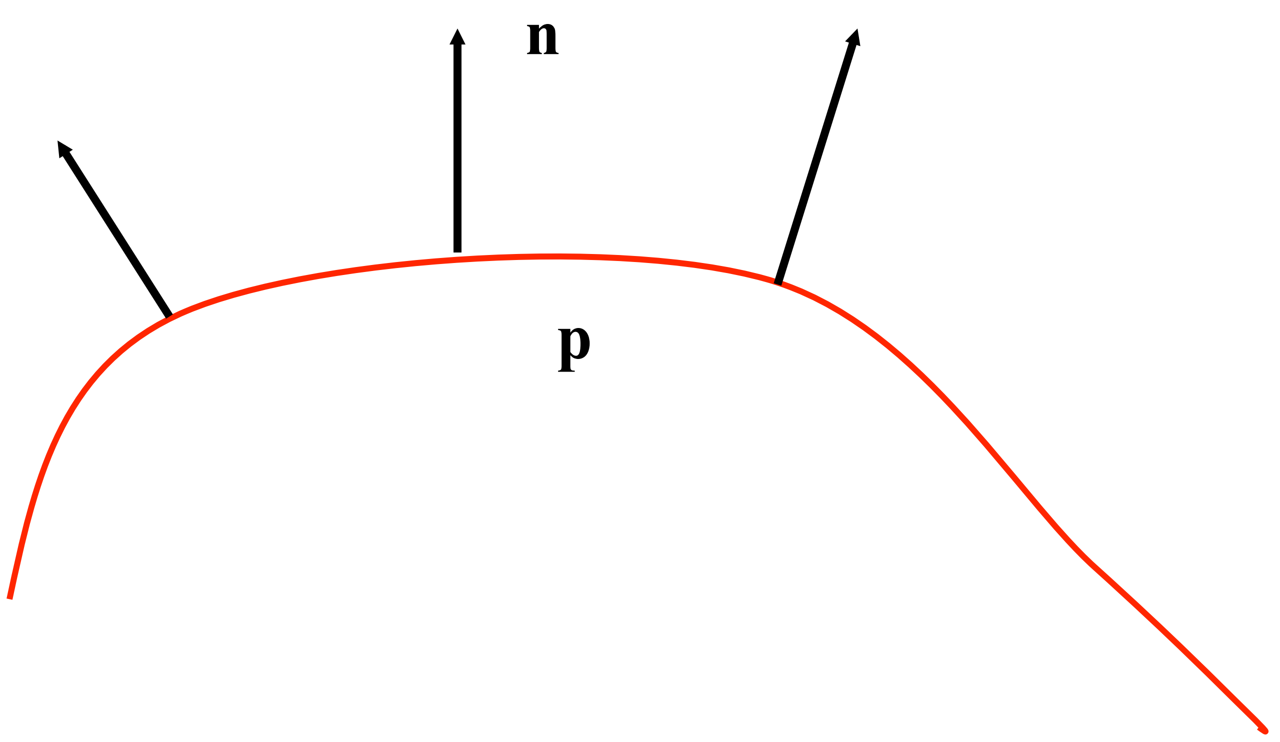


The Orange

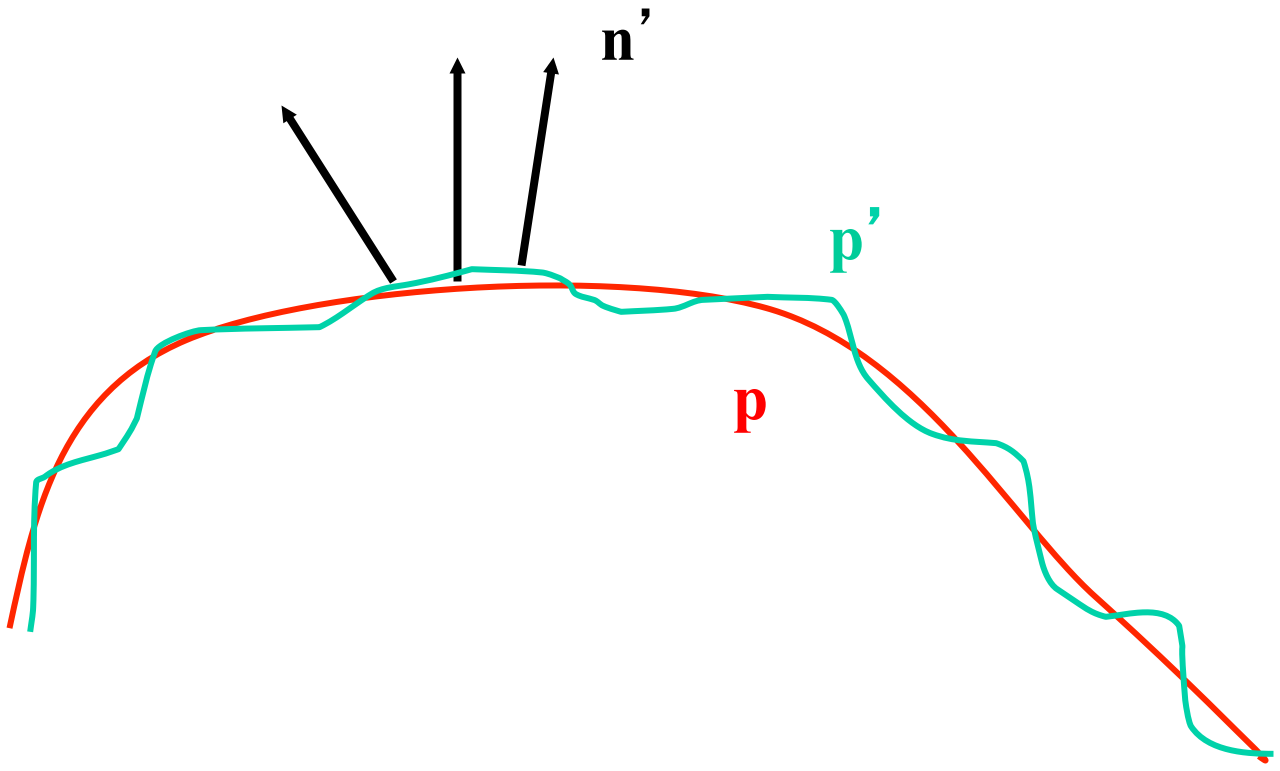
- Texture map a photo of an orange onto a surface
  - Captures dimples
  - Will not be correct if we move viewer or light
  - We have shades of dimples rather than their correct orientation
- Ideally perturb normal across surface of object and compute a new color at each interior point

Bump Mapping (Blinn)

Consider a smooth surface



Rougher Version



Equations

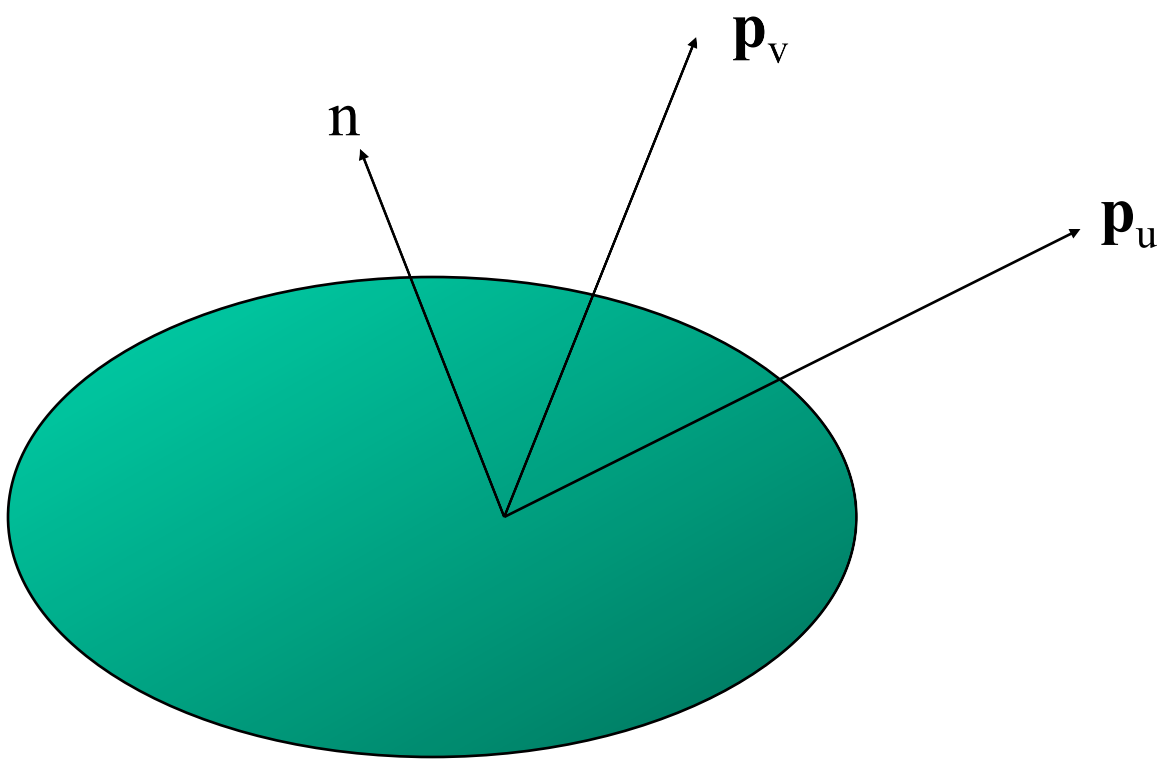
$$\mathbf{p}(u,v) = [x(u,v), y(u,v), z(u,v)]^T$$

$$\mathbf{p}_u=[\, \partial x/ \partial u, \partial y/ \partial u, \partial z/ \partial u]^T$$

$$\mathbf{p}_v=[\, \partial x/ \partial v, \partial y/ \partial v, \partial z/ \partial v]^T$$

$$\mathbf{n} = (\mathbf{p}_u \times \mathbf{p}_v) / |\mathbf{p}_u \times \mathbf{p}_v|$$

Tangent Plane



Displacement Function

$$\mathbf{p}' = \mathbf{p} + d(u,v)\,\mathbf{n}$$

$d(u,v)$ is the bump or displacement function

$|d(u,v)| \ll 1$

Perturbed Normal

$$\mathbf{n}' = \mathbf{p}'_u \times \mathbf{p}'_v$$

$$\mathbf{p}'_u = \mathbf{p}_u + (\partial d / \partial u)\mathbf{n} + d(u,v)\mathbf{n}_u$$

$$\mathbf{p}'_v = \mathbf{p}_v + (\partial d / \partial v)\mathbf{n} + d(u,v)\mathbf{n}_v$$

If d is small, we can neglect last term

Approximating the Normal

$$\mathbf{n}' = \mathbf{p}'_u \times \mathbf{p}'_v$$

$$\approx \mathbf{n} + (\partial d/\partial u)\mathbf{n} \times \mathbf{p}_v + (\partial d/\partial v)\mathbf{n} \times \mathbf{p}_u$$

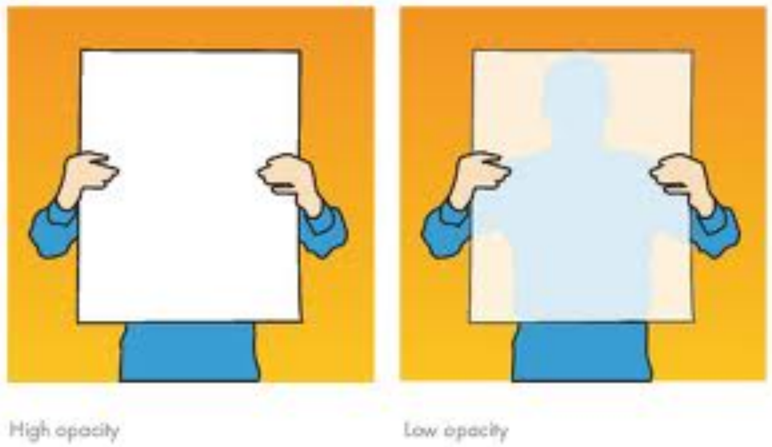
The vectors $n \times p_v$ and $n \times p_u$ lie
in the tangent plane
Hence the normal is displaced in the tangent plane
Must precompute the arrays $\partial d/\partial u$ and $\partial d/\partial v$
Finally,we perturb the normal during shading

Compositing & Blending

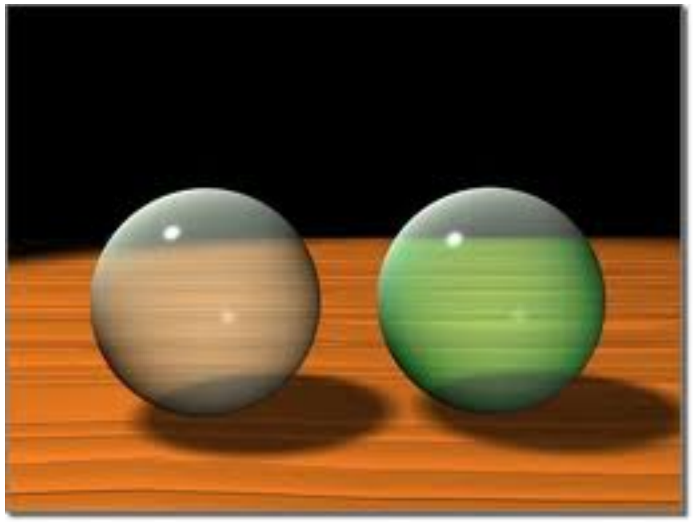
A

- Blending for translucent surfaces
- Compositing images
- Antialiasing

A



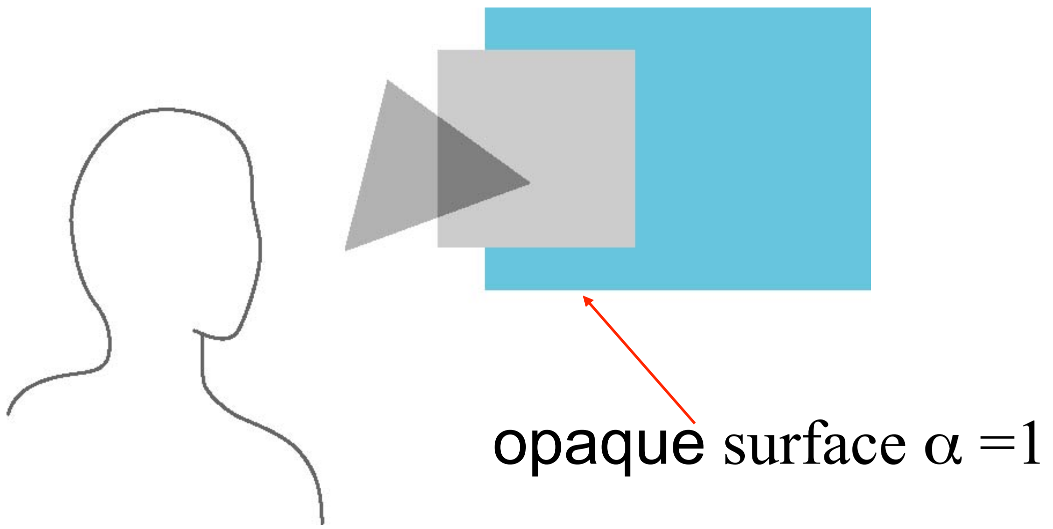
A



A

- Opaque surfaces permit no light to pass through
- Transparent surfaces permit all light to pass
- Translucent surfaces pass some light

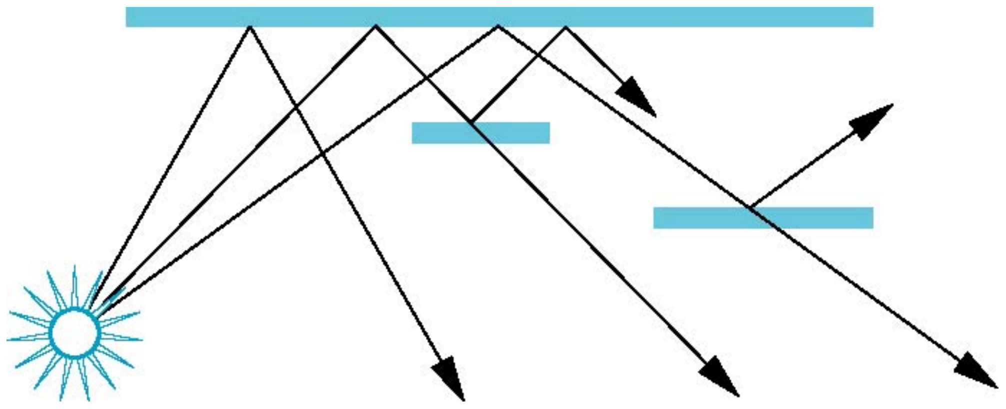
translucency = 1 – opacity (α)



Physical Models

Translucency in a physically correct manner is difficult

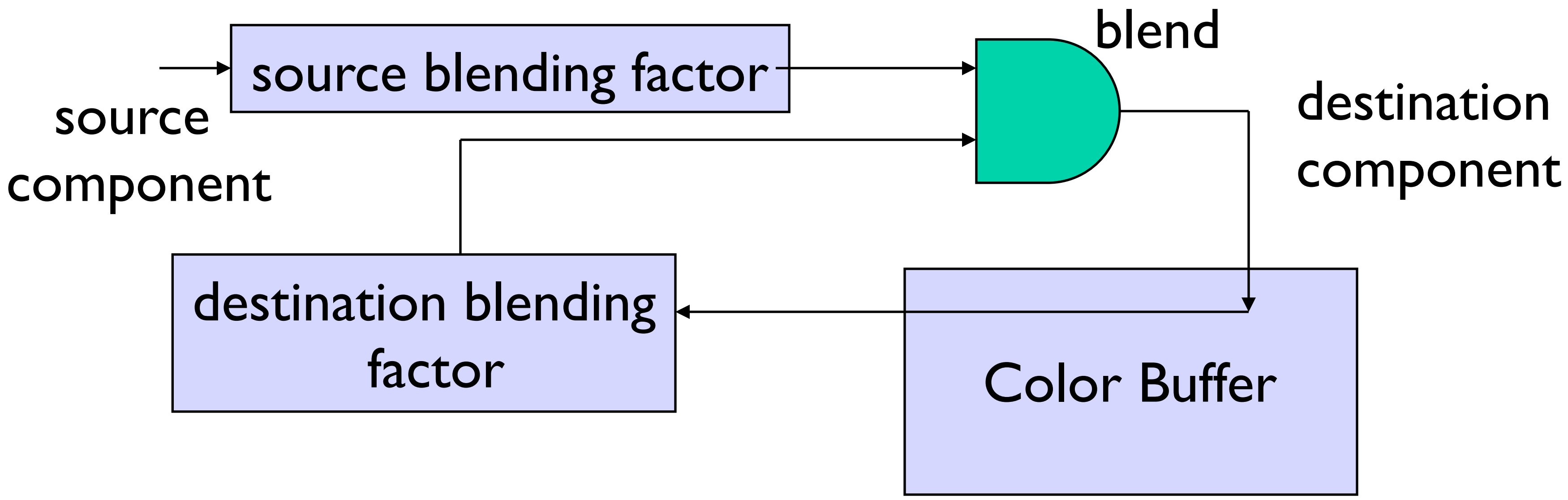
- the complexity of the internal interactions of light and matter
- Using a pipeline renderer



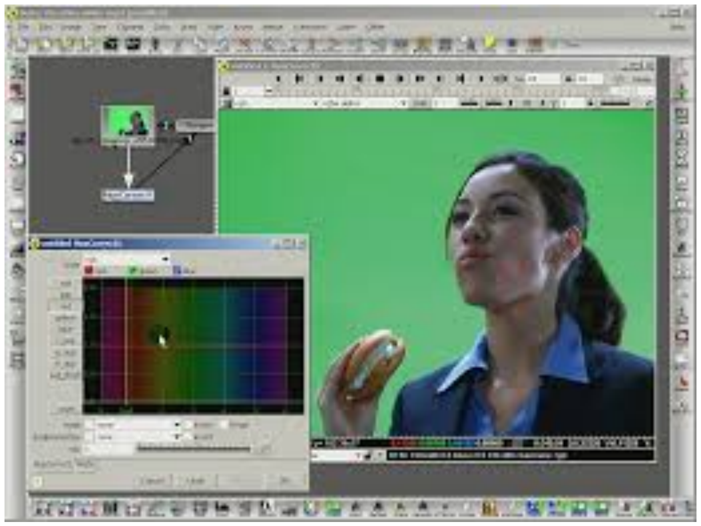
Compositing Operation

Rendering Model

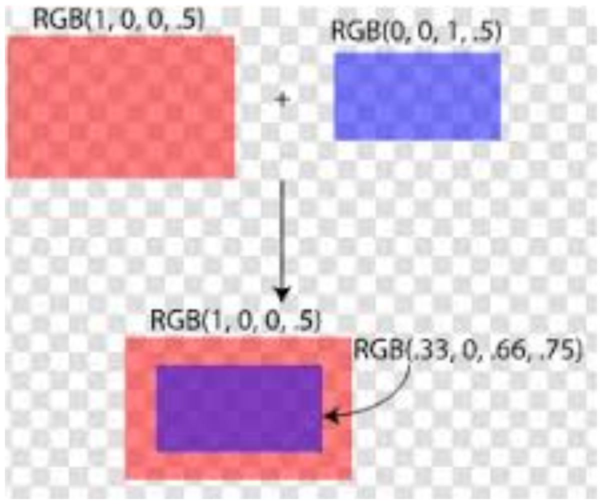
- Use A component of RGBA (or RGBa) color for opacity
- During rendering expand to use RGBA values



Examples



One Method



Blending Equation

We can define source and destination blending factors for each RGBA component

$\mathbf{s} = [s_r, s_g, s_b, s_a]$

$\mathbf{d} = [d_r, d_g, d_b, d_a]$

Suppose that the source and destination colors are

$\mathbf{b} = [b_r, b_g, b_b, b_a]$

$\mathbf{c} = [c_r, c_g, c_b, c_a]$

Blend as

$\mathbf{c}' = [b_r s_r + c_r d_r, b_g s_g + c_g d_g , b_b s_b + c_b d_b , b_a s_a + c_a d_a]$

OpenGL

Must enable blending and pick source and destination factors

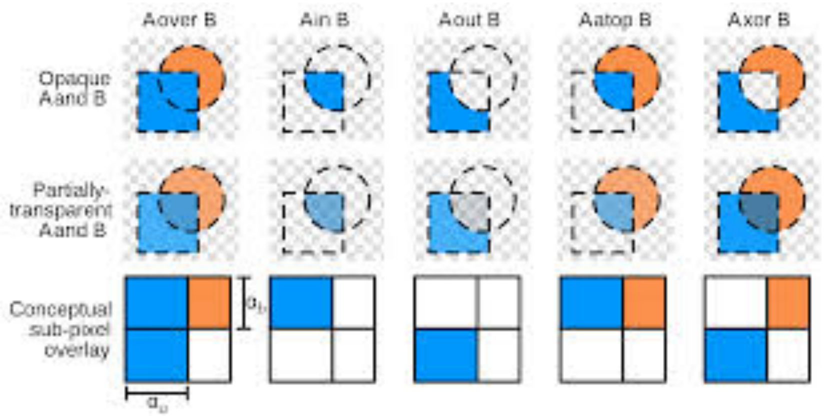
glEnable(GL_BLEND)

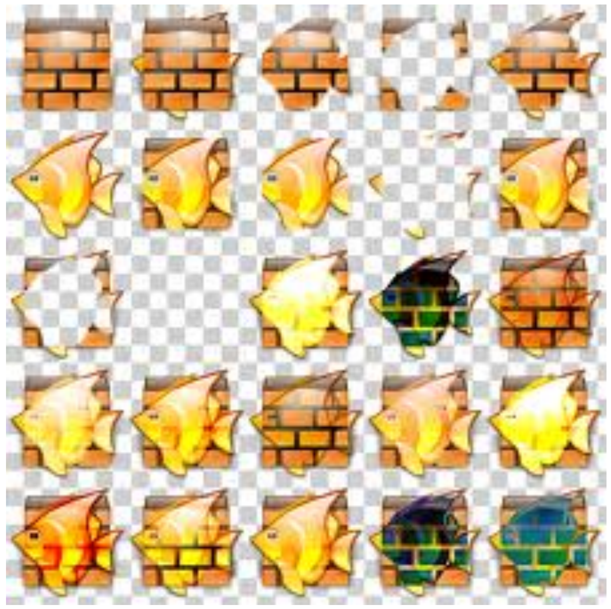
glBlendFunc(source_factor,
destination_factor)

Only certain factors supported

- **GL_ZERO, GL_ONE**
- **GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA**
- **GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA**
- See Redbook for complete list

Operator


Aover B
Ain B
Aout B
Aatop B
Axor B
Opaque A and B
Partially-transparent A and B
Conceptual sub-pixel overlay
α_b
α_a



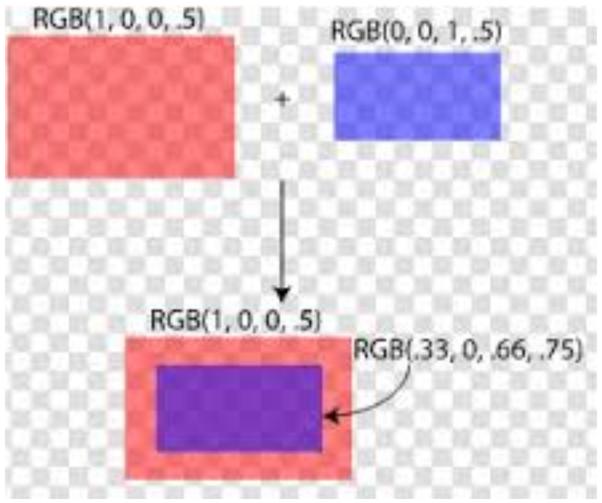
Example

- Start with the opaque background color $(R_0,G_0,B_0,1)$
  - Initial destination color
- Blend in a translucent polygon with color (R_1,G_1,B_1,a_1)
- Select **GL_SRC_ALPHA** and **GL_ONE_MINUS_SRC_ALPHA** as the source and destination blending factors

$$R'_1 = a_1 R_1 + (1 - a_1) R_{0,} \ldots\ldots$$

- Note this formula is correct if polygon is either opaque or transparent

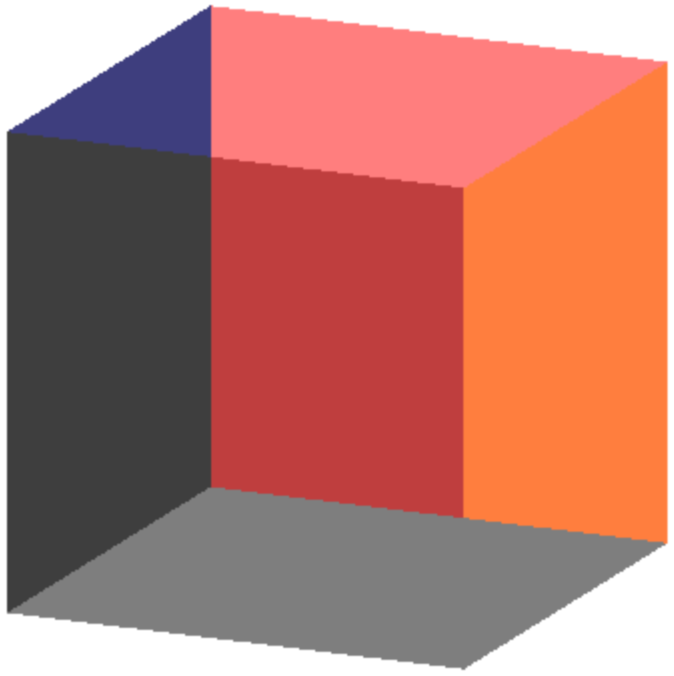
Works Here Too…



Clamping and Accuracy

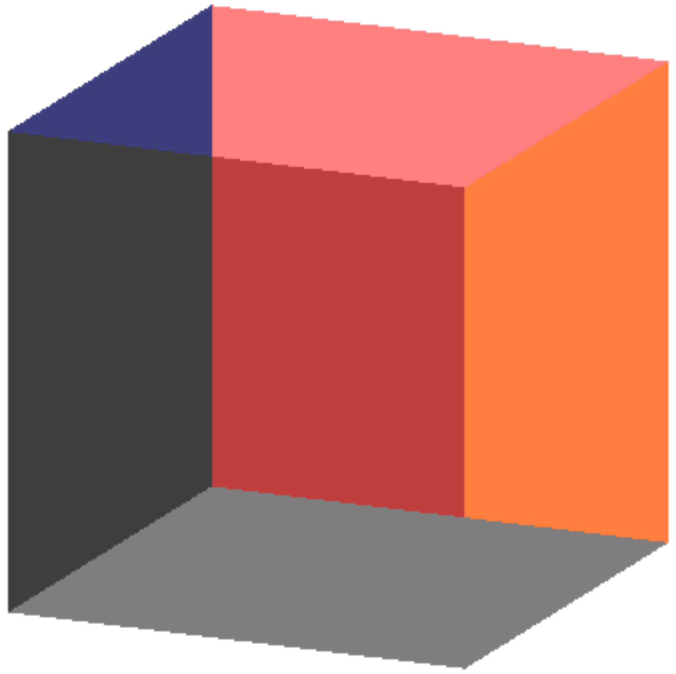
- All RGBA are clamped to the range (0,1)
- RGBA values 8 bits !
  - Loose accuracy after much components together
  - Example: add together n images
    - Divide all color components by n to avoid clamping
    - Blend with source factor = 1, destination factor = 1
    - But division by n loses bits

Order Dependency



Order Dependency

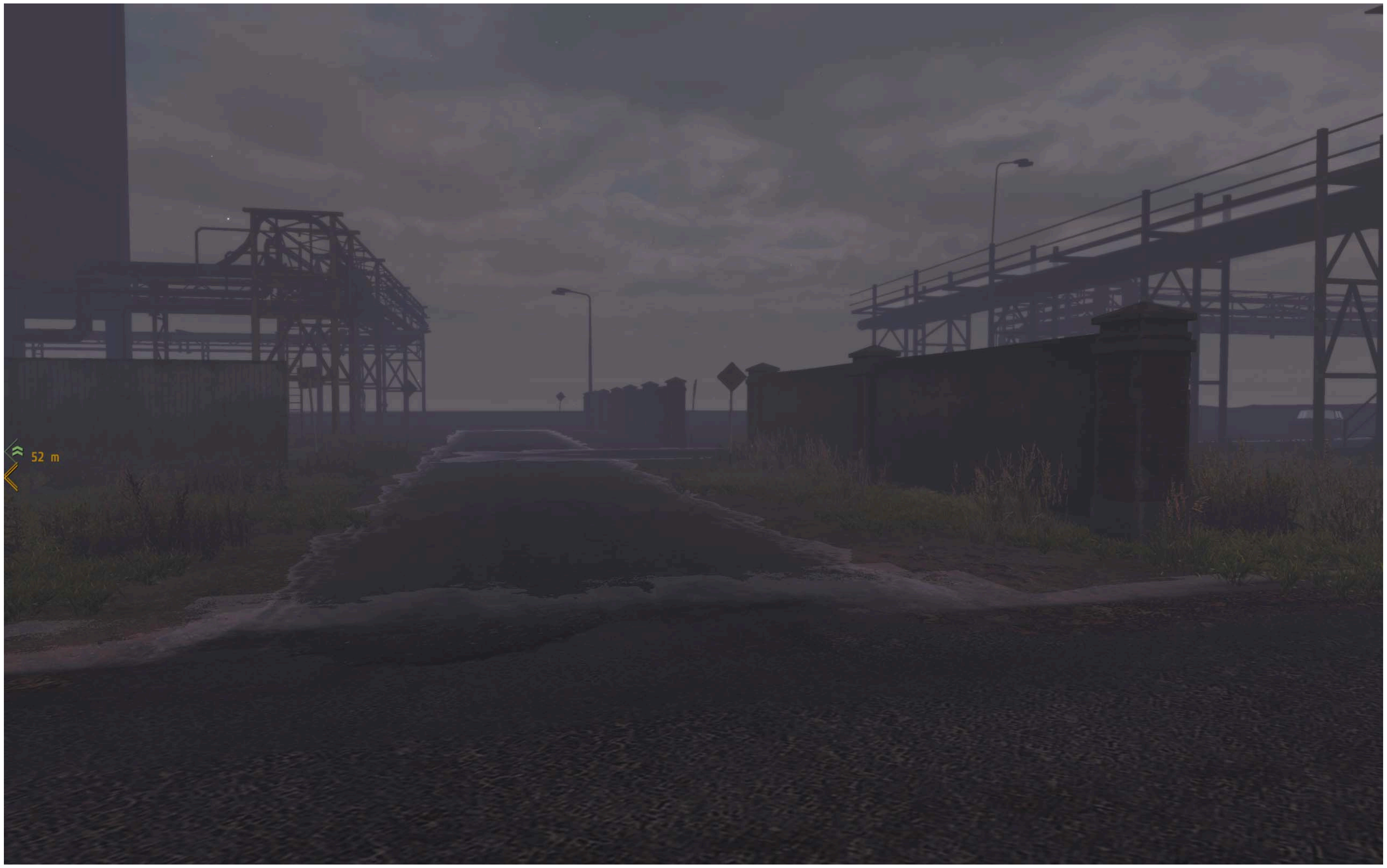
- Is this image correct?
  - Probably not
  - Polygons are rendered in the order they pass down the pipeline
  - Blending functions are order dependent



HSR with A

- Polygons which are opaque & translucent
- Opaque polygons block all polygons behind & affect depth buffer
- Translucent polygons should not affect depth buffer
  - Render with **glDepthMask(GL_FALSE)** which makes depth buffer read-only
- Sort polygons first to remove order dependency

Fog



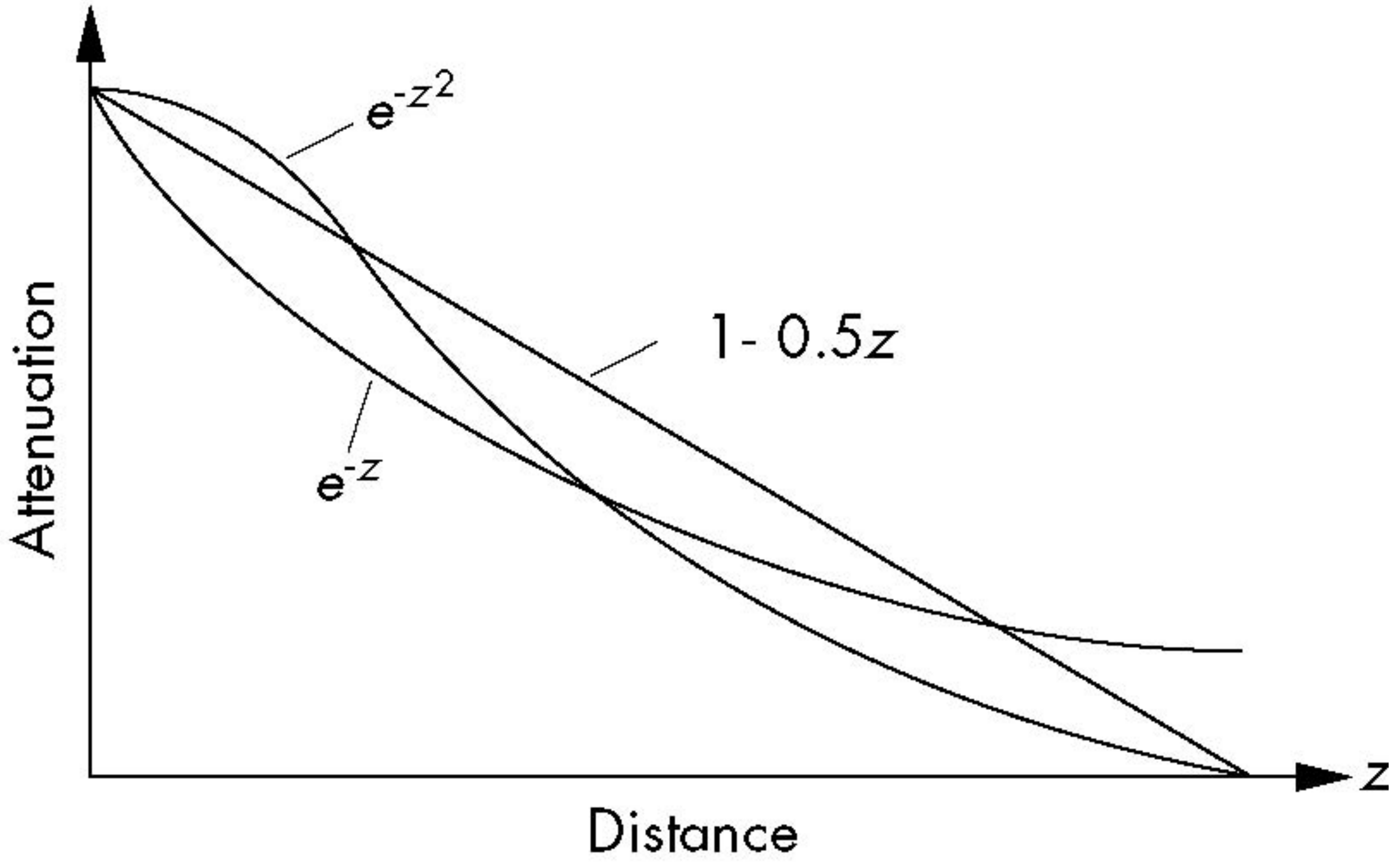
Simulate Fog

- Composite with fixed color and have blending factors depend on depth
  - Simulates a fog effect
- Blend source color C_s and fog color C_f by

$$C_s' = f\,C_s + (1-f)\,C_f$$

- f is the *fog factor*
  - Exponential
  - Gaussian
  - Linear (depth cueing)

Fog Functions

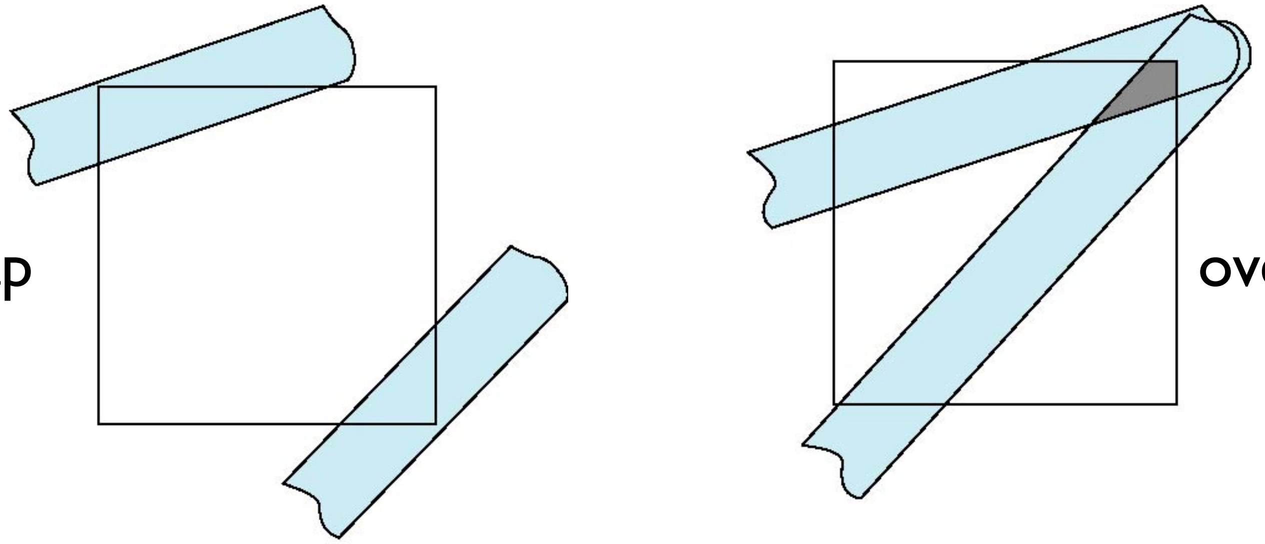


Antialiasing

Color a pixel by adding fraction of color to frame buffer

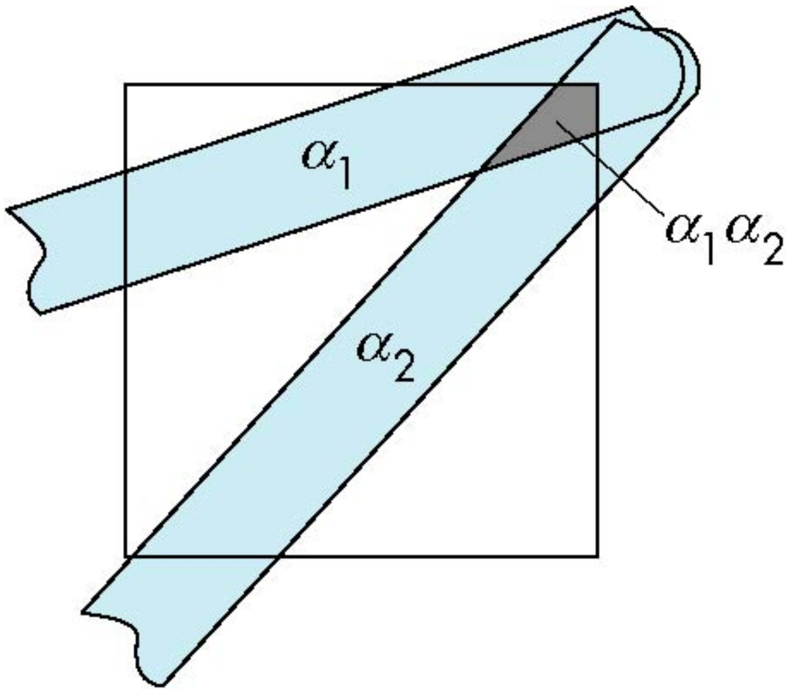
– Fraction depends on percentage of pixel covered by fragment

– Fraction depends on whether there is overlap



Area Averaging

Use average area $a_1+a_2-a_1a_2$ as blending factor



OpenGL Antialiasing

Enable separately for points, lines, or polygons

```
glEnable(GL_POINT_SMOOTH);
glEnable(GL_LINE_SMOOTH);
glEnable(GL_POLYGON_SMOOTH);

glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Accumulation

- Compositing/blending limited by resolution of frame buffer
  - Typically 8 bits per color component
- A*ccumulation buffer* was a high resolution buffer (16 or more bits per component) that avoided this problem
- Could write into it or read from it with a scale factor
- Slower than direct compositing into the frame buffer