The University of New Mexico

Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

The University of New Mexico

Buffers

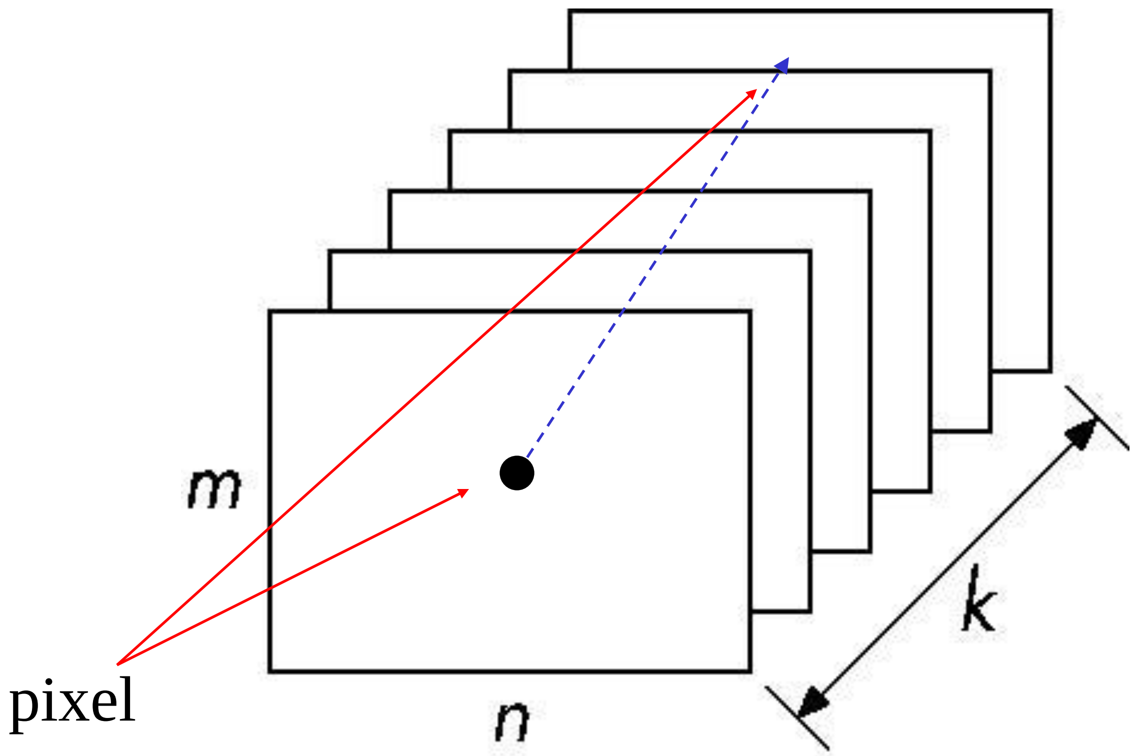
Ed Angel
Professor Emeritus of Computer Science
University of New Mexico

Objectives

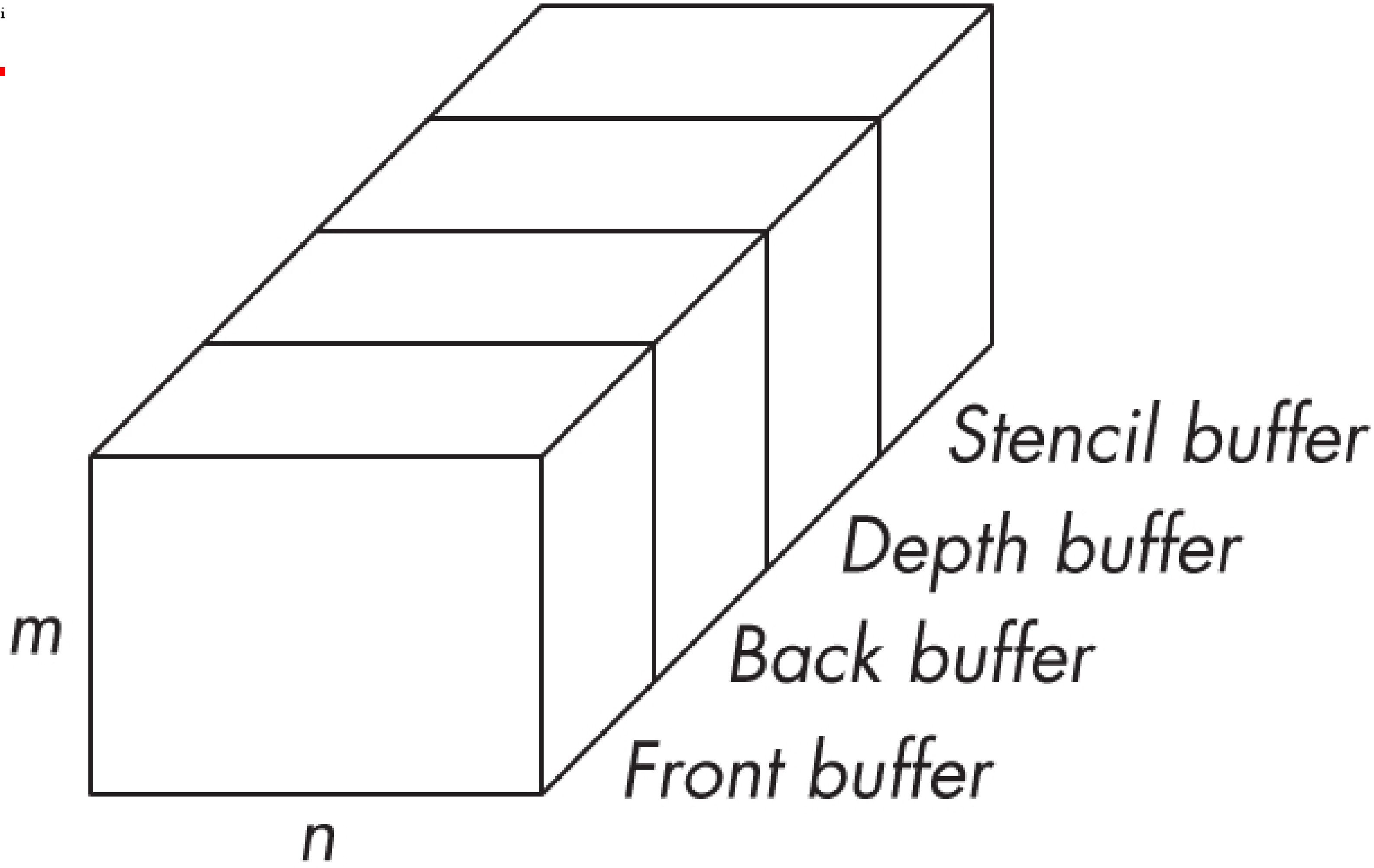
- Introduce additional WebGL buffers
- Reading and writing buffers
- Buffers and Images

Buffer

Define a buffer by its spatial resolution (n x m) and its depth (or precision) k, the number of bits/pixel



WebGL Frame Buffer



Where are the Buffers?

- HTML5 Canvas
  - Default front and back color buffers
  - Under control of local window system
  - Physically on graphics card
- Depth buffer also on graphics card
- Stencil buffer
  - Holds masks
- Most RGBA buffers 8 bits per component
- Latest are floating point (IEEE)

Other Buffers

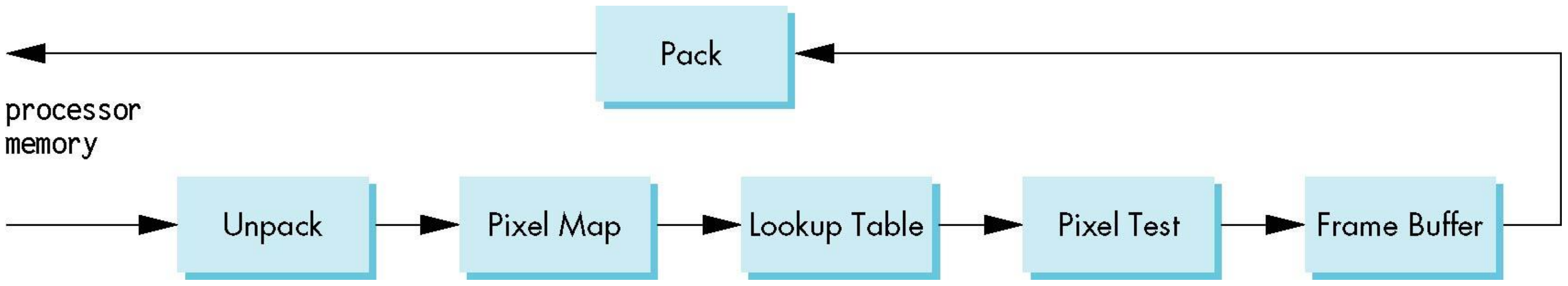
- desktop OpenGL supported other buffers
  - auxiliary color buffers
  - accumulation buffer
  - these were on application side
  - now deprecated
- GPUs have their own or attached memory
  - texture buffers
  - off-screen buffers
    - not under control of window system
    - may be floating point

Images

- Framebuffer contents are unformatted
  - usually RGB or RGBA
  - one byte per component
  - no compression
- Standard Web Image Formats
  - jpeg, gif, png
- WebGL has no conversion functions
  - Understands standard Web formats for texture images

The (Old) Pixel Pipeline

- OpenGL has a separate pipeline for pixels
  - Writing pixels involves
    - Moving pixels from processor memory to the frame buffer
    - Format conversions
    - Mapping, Lookups, Tests
  - Reading pixels
    - Format conversion



Packing and Unpacking

- Compressed or uncompressed
- Indexed or RGB
- Bit Format
  - little or big endian
- WebGL (and shader-based OpenGL) lacks most functions for packing and unpacking
  - use texture functions instead
  - can implement desired functionality in fragment shaders

Deprecated Functionality

- glDrawPixels
- glCopyPixels
- glBitMap

Buffer Reading

- WebGL can read pixels from the framebuffer with gl.readPixels
- Returns only 8 bit RGBA values
- In general, the format of pixels in the frame buffer is different from that of processor memory and these two types of memory reside in different places
  - Need packing and unpacking
  - Reading can be slow
- Drawing through texture functions and off-screen memory (frame buffer objects)

WebGL Pixel Function

```
gl.readPixels(x,y,width,height,format,type,myimage)
```

start pixel in frame buffer

size

type of pixels

type of image

pointer to processor memory

```
var myimage[512*512*4];

gl.readPixels(0,0, 512, 512, gl.RGBA,
      gl.UNSIGNED_BYTE, myimage);
```

Render to Texture

- GPUs now include a large amount of texture memory that we can write into
- Advantage: fast (not under control of window system)
- Using frame buffer objects (FBOs) we can render into texture memory instead of the frame buffer and then read from this memory
  - Image processing
  - GPGPU

Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

The University of New Mexico

BitBlt

Ed Angel

Professor Emeritus of Computer Science

University of New Mexico

Objectives

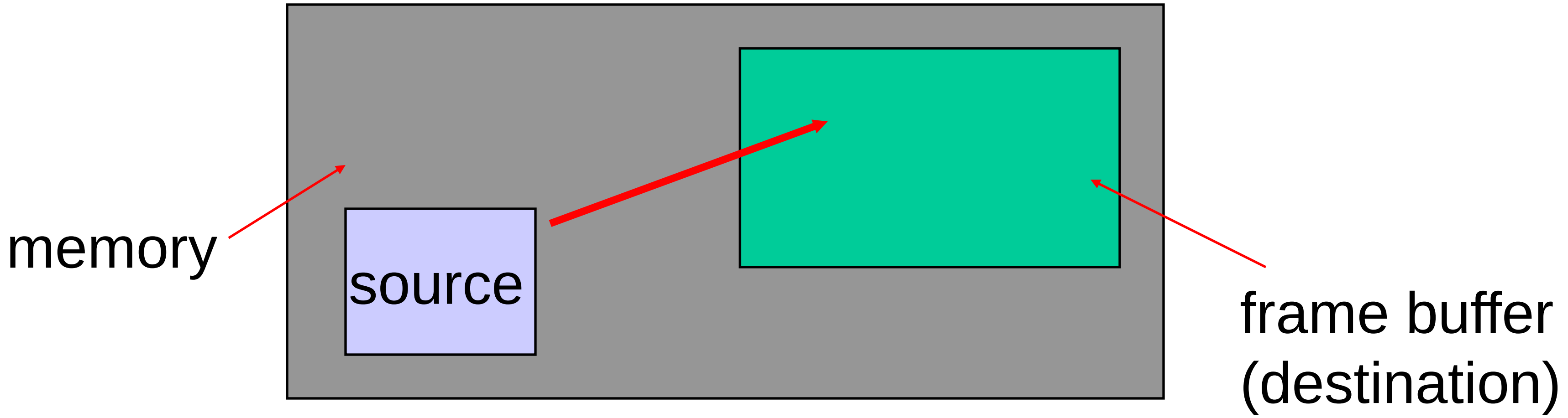
- Introduce reading and writing of blocks of bits or bytes
- Prepare for later discussion compositing and blending

Writing into Buffers

- WebGL does not contain a function for writing bits into frame buffer
  - Use texture functions instead
- We can use the fragment shader to do bit level operations on graphics memory
- Bit Block Transfer (BitBlt) operations act on blocks of bits with a single instruction

BitBlt

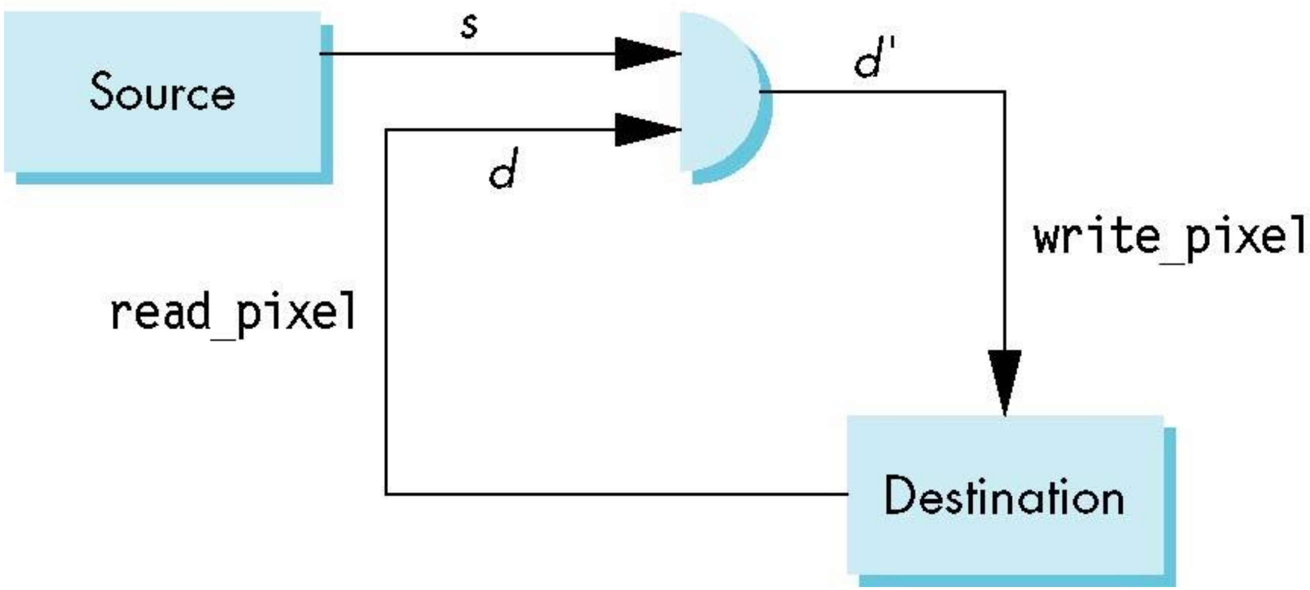
- Conceptually, we can consider all of memory as a large two-dimensional array of pixels
- We read and write rectangular block of pixels
- The frame buffer is part of this memory



writing into the frame buffer

Writing Model

Read destination pixel before writing source



The University of New Mexico

Bit Writing Modes

- Source and destination bits are combined bitwise
- 16 possible functions (one per column in table)

replace

XOR

OR

s	*d*	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

XOR mode

- XOR is especially useful for swapping blocks of memory such as menus that are stored off screen

If S represents screen and M represents a menu the sequence

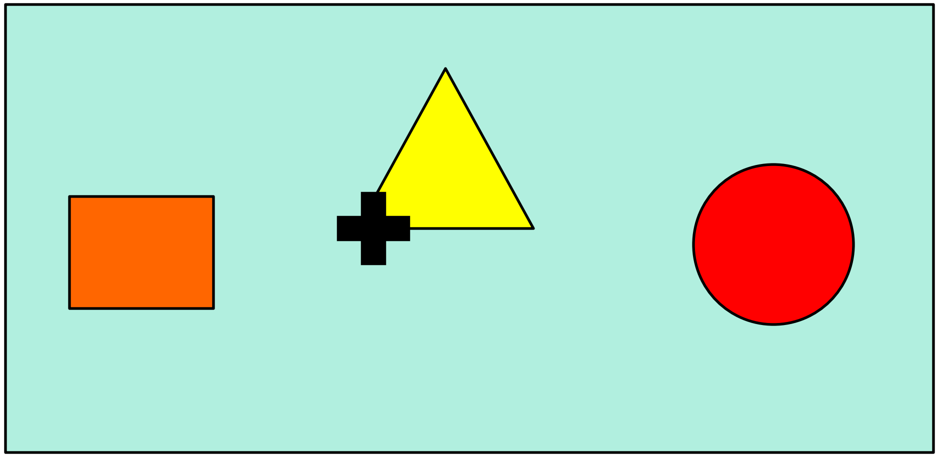
$$S \leftarrow S \oplus M$$
$$M \leftarrow S \oplus M$$
$$S \leftarrow S \oplus M$$

swaps S and M

- Same strategy used for rubber band lines and cursors

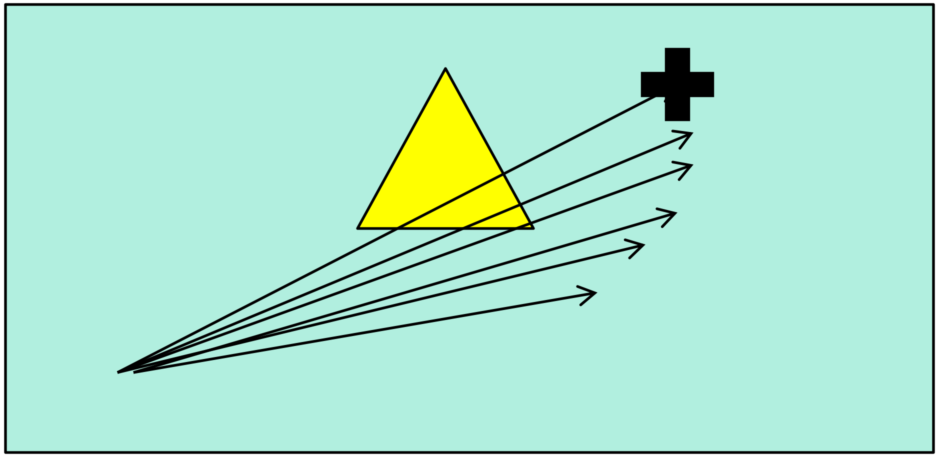
Cursor Movement

- Consider what happens as we move a cursor across the display
- We cover parts of objects
- Must return to original colors when cursor moves away



Rubber Band Line

- Fix one point
- Draw line to location of cursor
- Must return state of crossed objects when line moves



Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

Texture Mapping

Ed Angel
Professor Emeritus of Computer Science
University of New Mexico

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

- 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

The University of New Mexico

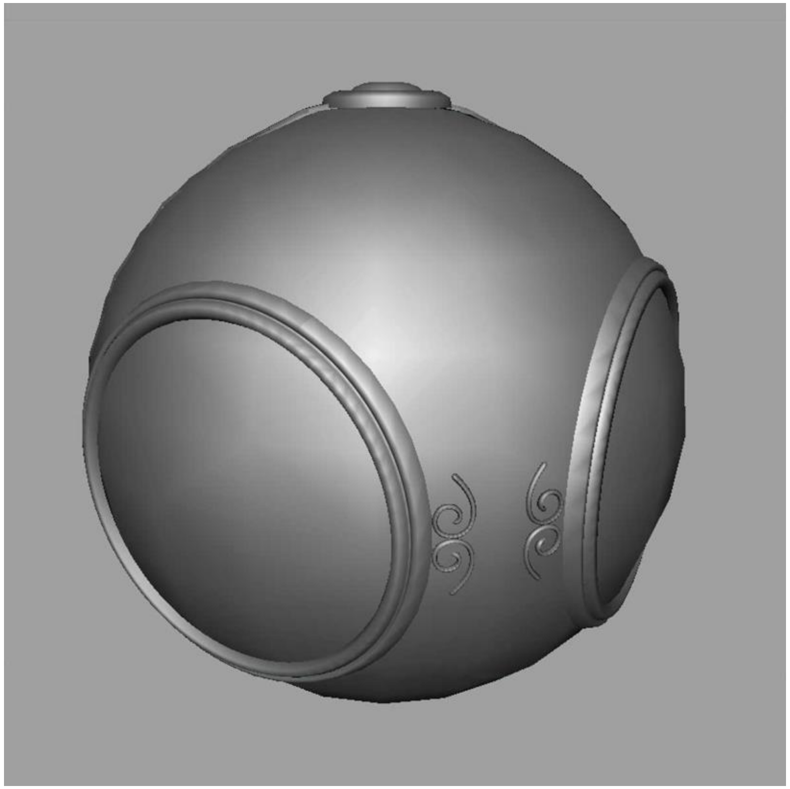
Modeling an Orange (2)

- 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

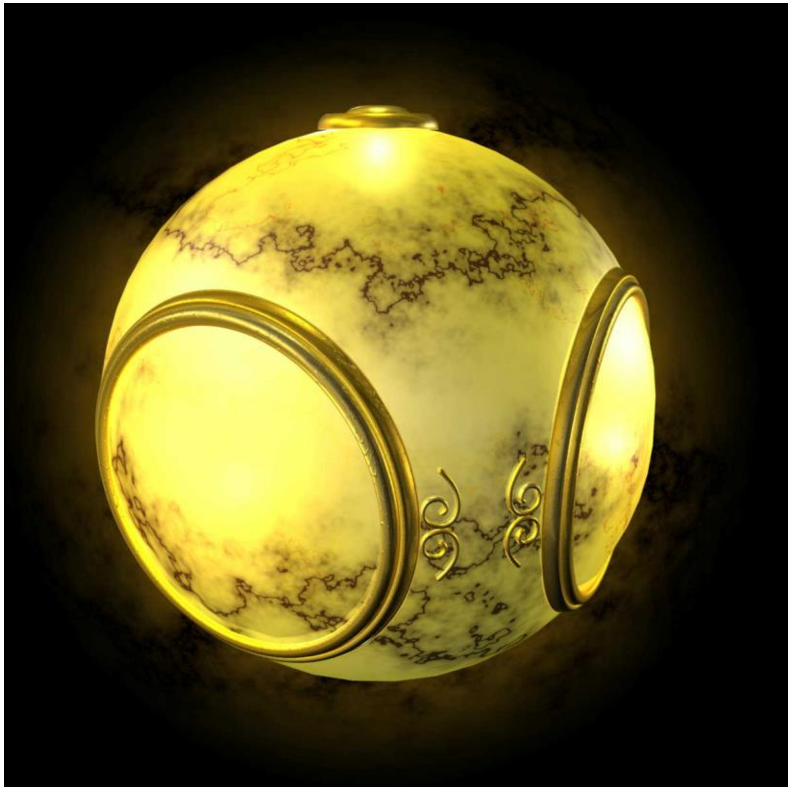
Three Types of Mapping

- 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



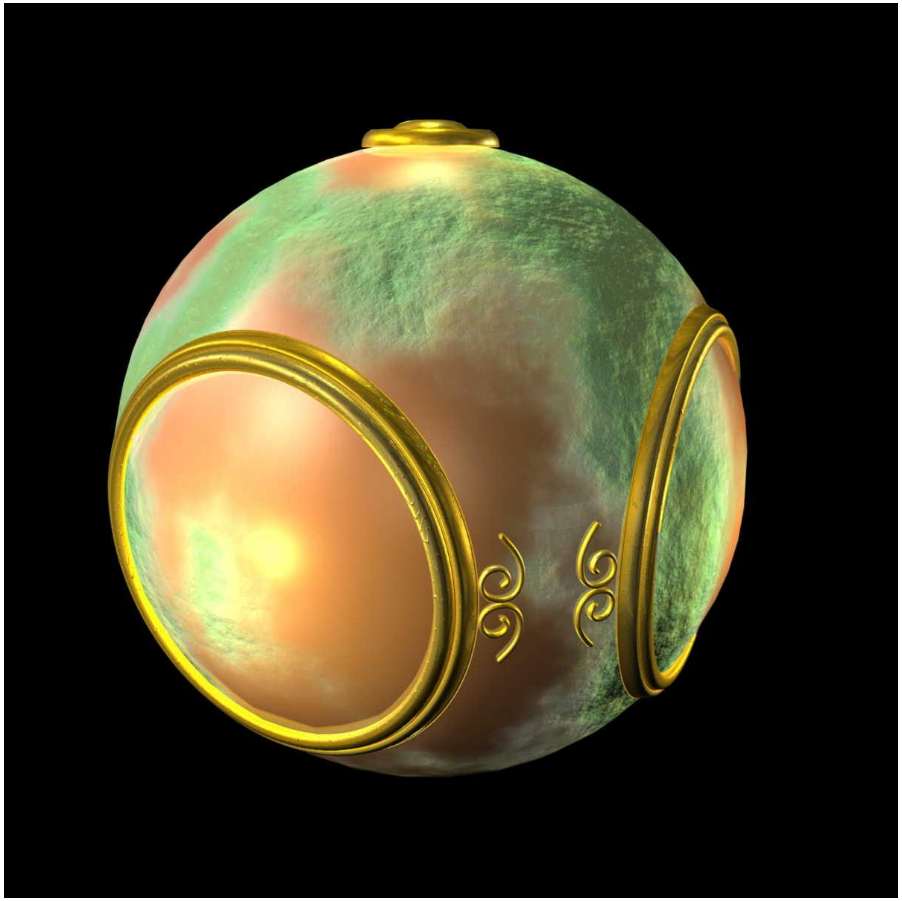
geometric model



texture mapped

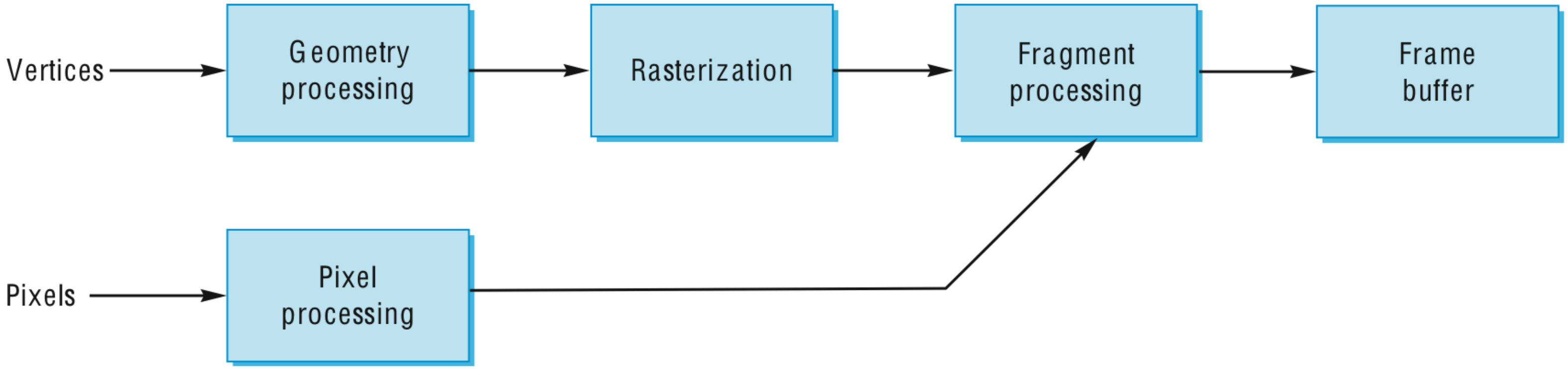
Environment Mapping

Bump Mapping



Where does mapping take place?

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



Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

Texture Mapping

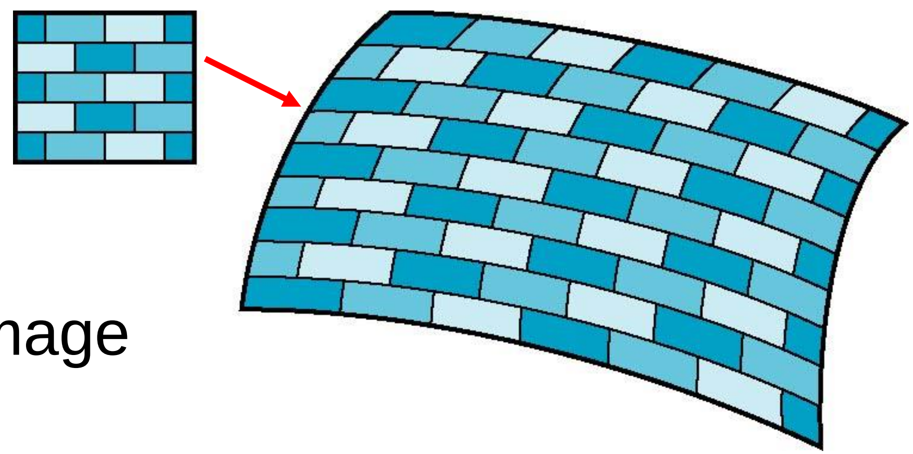
Ed Angel
Professor Emeritus of Computer Science
University of New Mexico

Objectives

- Basic mapping strategies
  - Forward vs backward mapping
  - Point sampling vs area averaging

Is it simple?

- Although the idea is simple---map an image to a surface---there are 3 or 4 coordinate systems involved



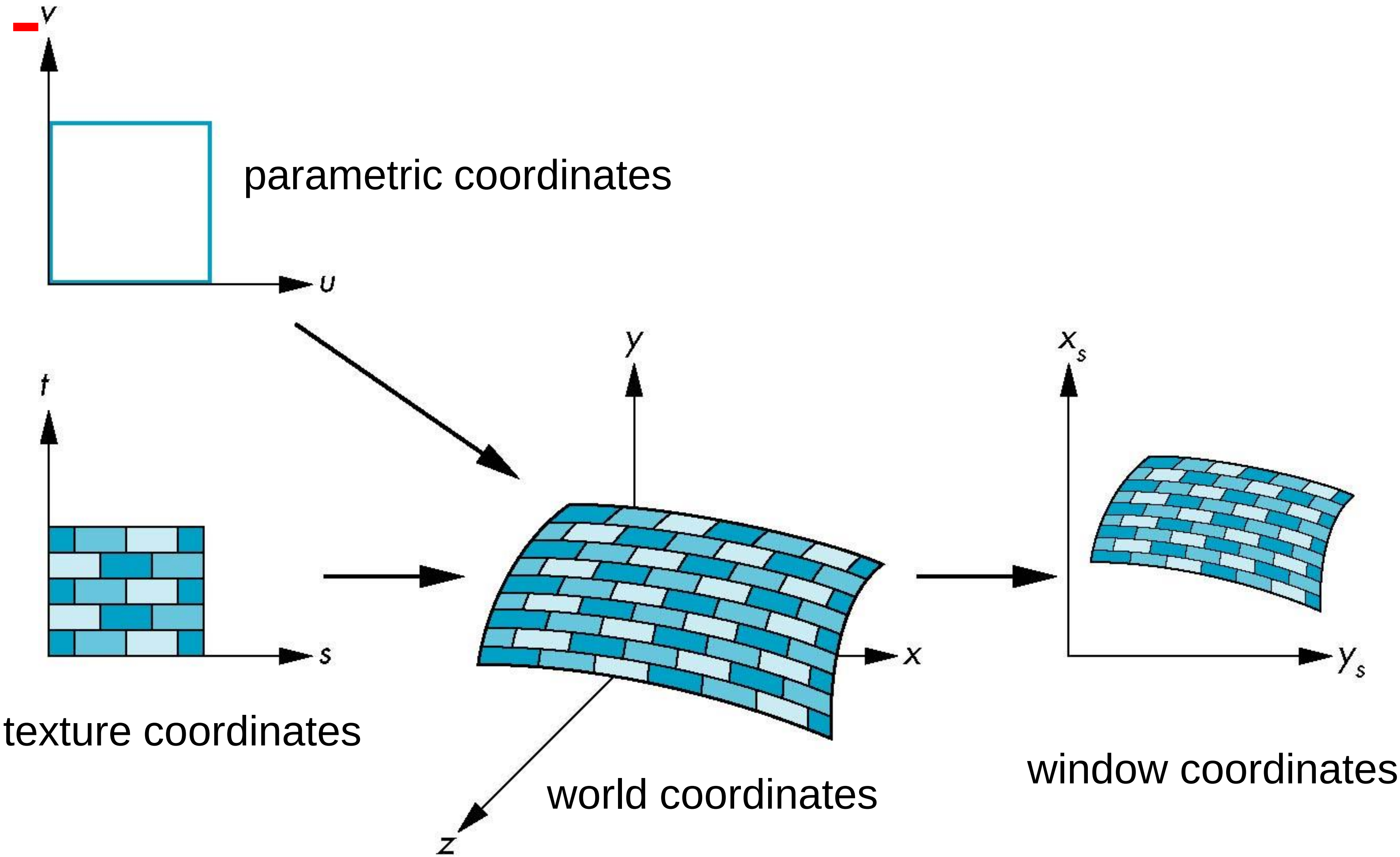
2D image

3D surface

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

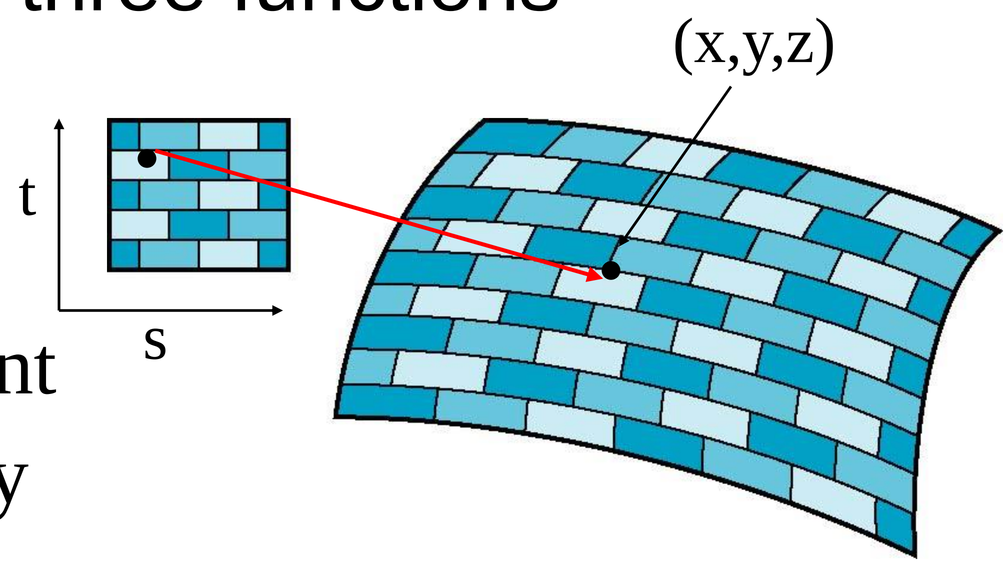
Texture Mapping



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

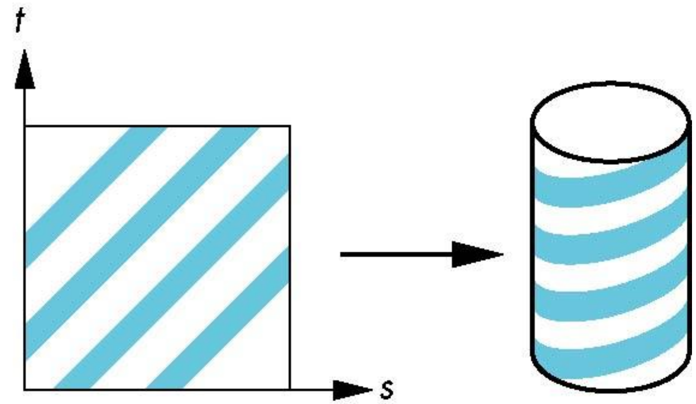
- We really want to go backwards
  - Given a pixel, we want to know to which point on an object it corresponds
  - Given a point on an object, we want to know to which point in the texture it corresponds
- Need a map of the form

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

  $t = t(x,y,z)$
- Such functions are difficult to find in general

Two-part mapping

- One solution to the mapping problem is to first map the texture to a 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

We can use a parametric sphere

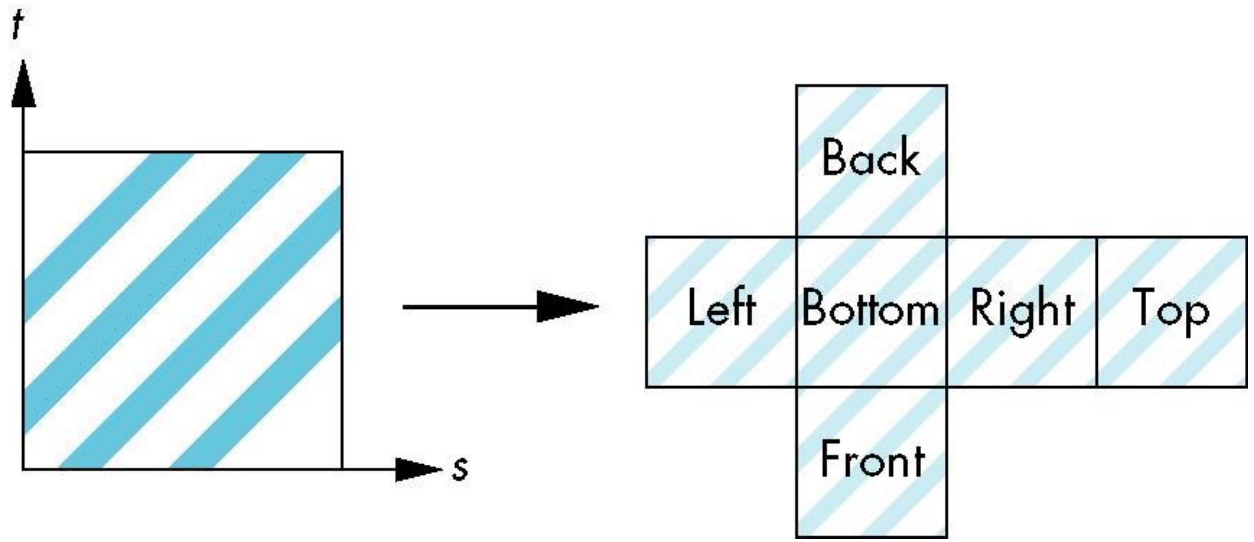
$$x = r \cos 2\pi u$$
$$y = r \sin 2\pi u \cos 2\pi v$$
$$z = r \sin 2\pi u \sin 2\pi v$$

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

Spheres are used in environmental maps

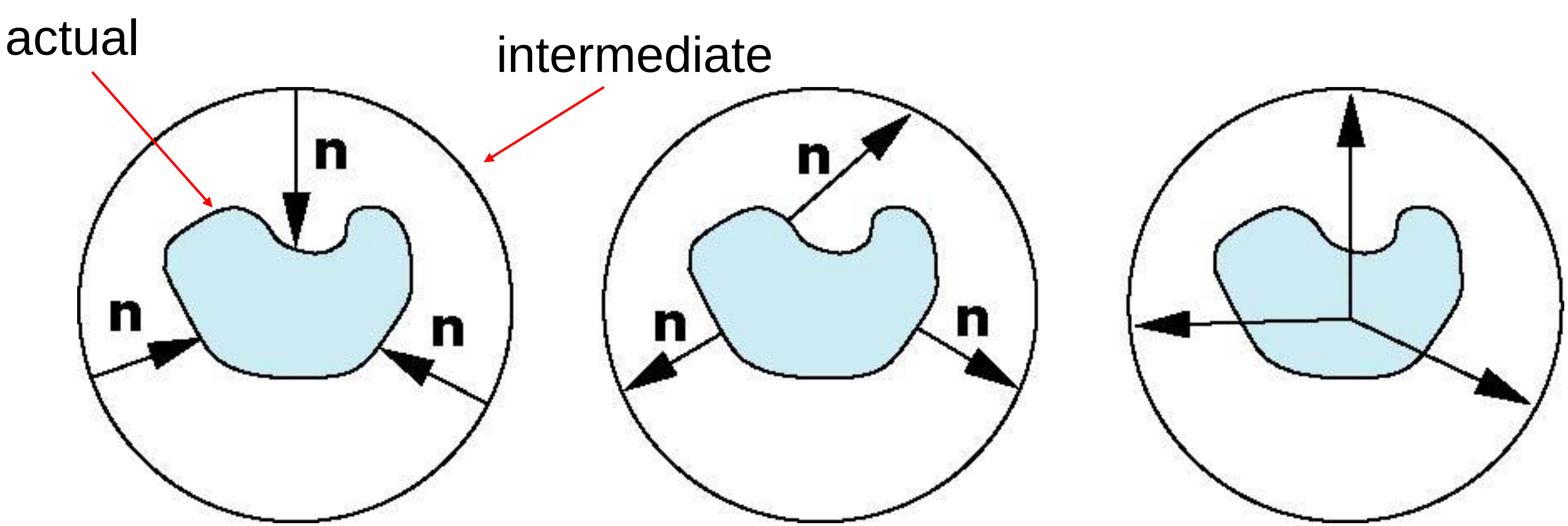
Box Mapping

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



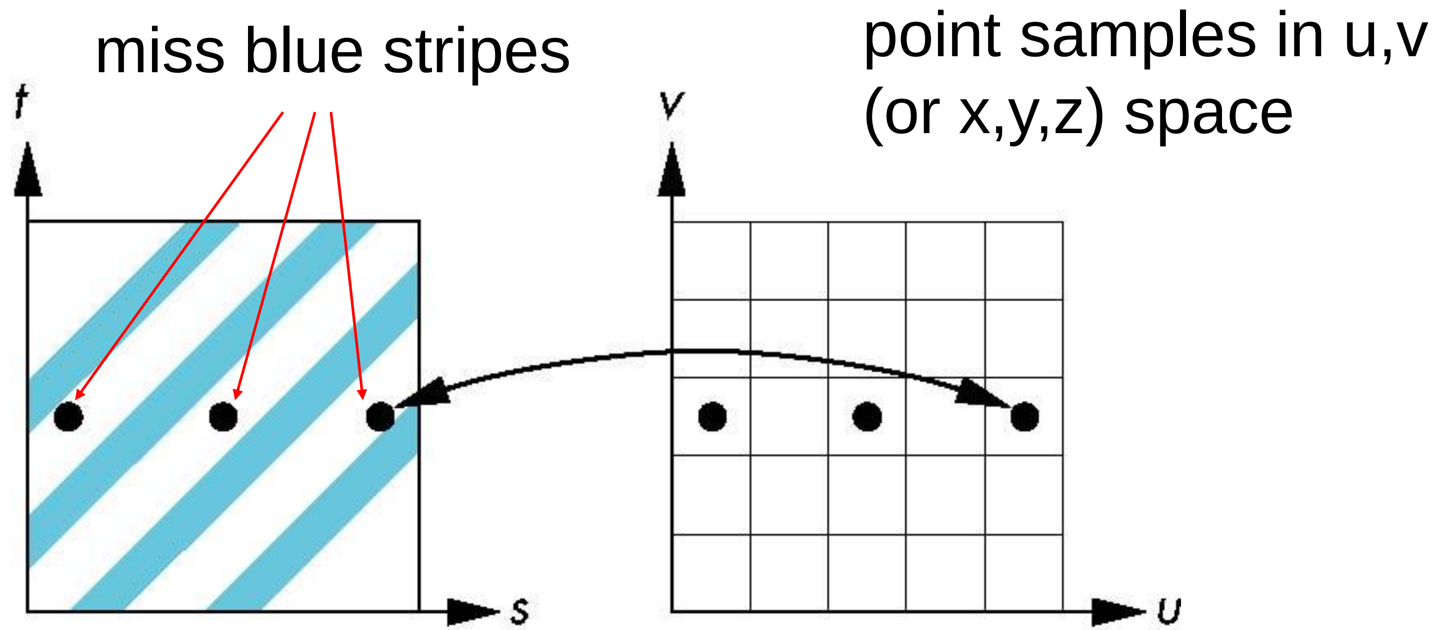
Second Mapping

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



Aliasing

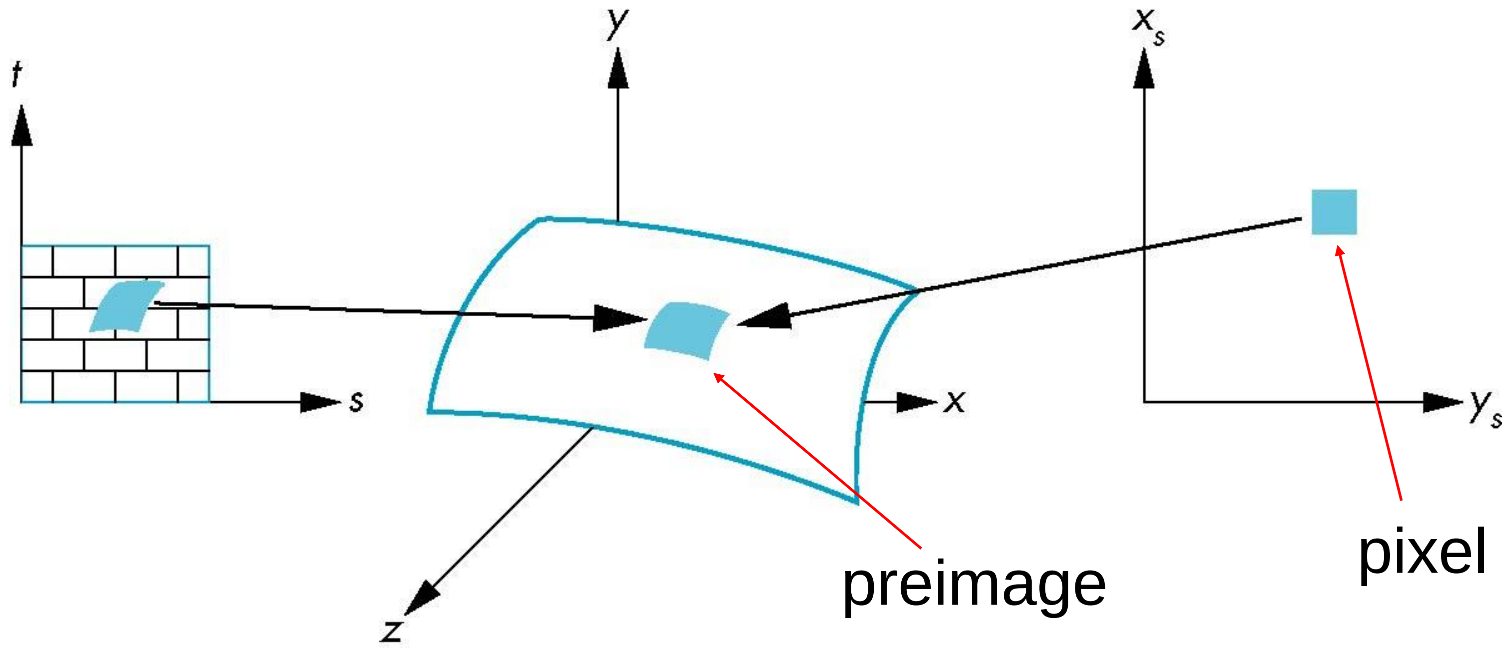
- Point sampling of the texture can lead to aliasing errors



point samples in texture space

Area Averaging

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



Note that *preimage* of pixel is curved

Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

WebGL Texture Mapping I

Ed Angel
Professor Emeritus of Computer Science
University of New Mexico

Objectives

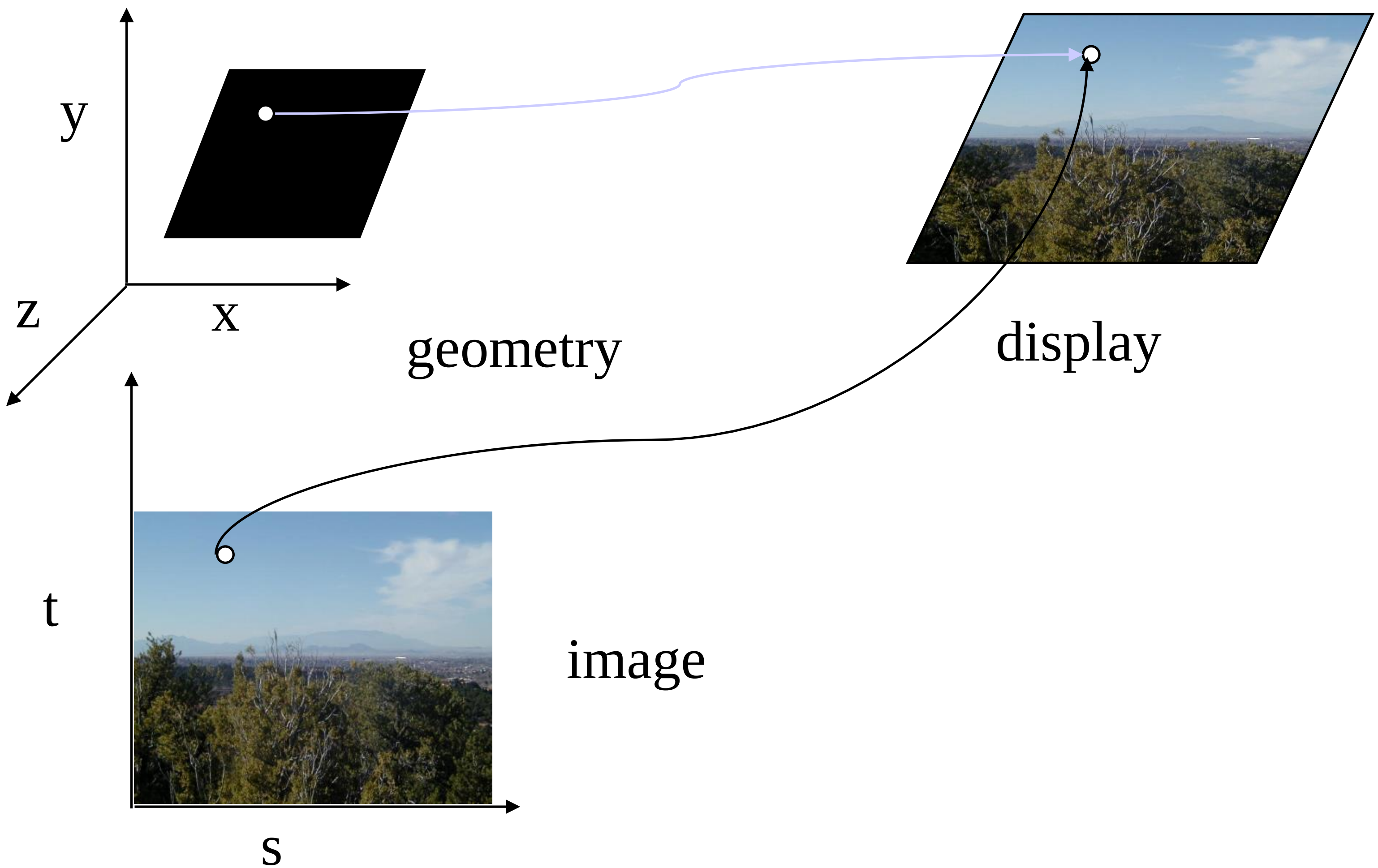
- Introduce WebGL texture mapping
  - two-dimensional texture maps
  - assigning texture coordinates
  - forming texture images

Basic Stragegy

Three steps to applying a texture

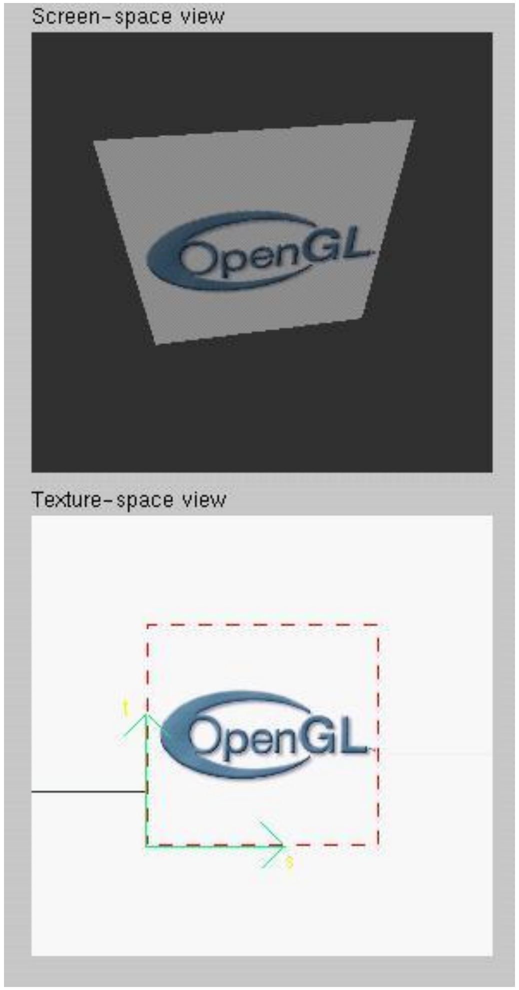
1. specify the texture
   - read or generate image
   - assign to texture
   - enable texturing
2. assign texture coordinates to vertices
   - Proper mapping function is left to application
3. specify texture parameters
   - wrapping, filtering

Texture Mapping



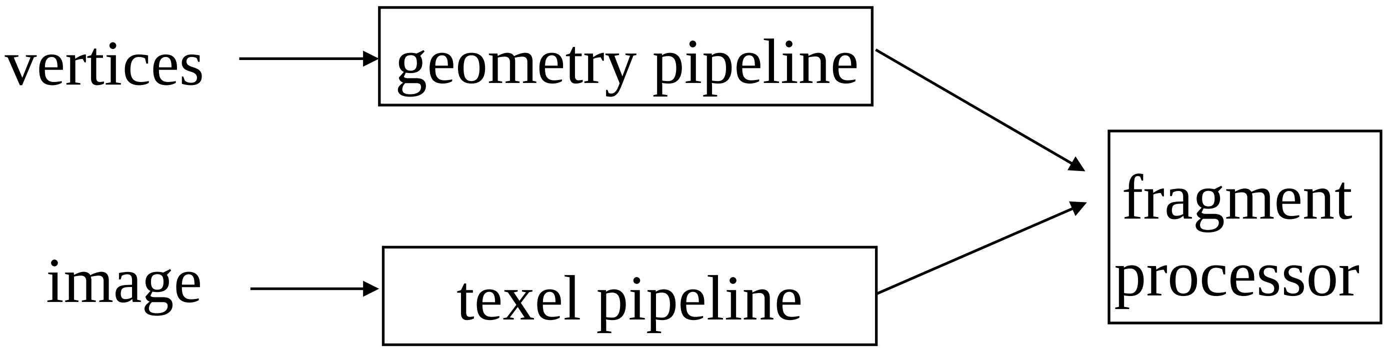
Texture Example

- The texture (below) is a 256 x 256 image that has been mapped to a rectangular polygon which is viewed in perspective



Texture Mapping and the WebGL Pipeline

- Images and geometry flow through separate pipelines that join during fragment processing
  - "complex" textures do not affect geometric complexity



Specifying a Texture Image

- Define a texture image from an array of *texels* (texture elements) in CPU memory
- Use an image in a standard format such as JPEG
  - Scanned image
  - Generate by application code
- WebGL supports only 2 dimensional texture maps
  - no need to enable as in desktop OpenGL
  - desktop OpenGL supports 1-4 dimensional texture maps

Define Image as a Texture

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

`target:` type of texture, e.g. `GL_TEXTURE_2D`

`level:` used for mipmapping (discussed later)

`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);
```

A Checkerboard Image

```
var image1 = new Uint8Array(4*texSize*texSize);
  for ( var i = 0; i < texSize; i++ ) {
    for ( var j = 0; j <texSize; j++ ) {
      var patchx = Math.floor(i/(texSize/numChecks));
      var patchy = Math.floor(j/(texSize/numChecks));
      if(patchx%2 ^ patchy%2) c = 255;
      else c = 0;
      //c = 255*(((i & 0x8) == 0) ^ ((j & 0x8)  == 0))
      image1[4*i*texSize+4*j] = c;
      image1[4*i*texSize+4*j+1] = c;
      image1[4*i*texSize+4*j+2] = c;
      image1[4*i*texSize+4*j+3] = 255;
    }
  }
```

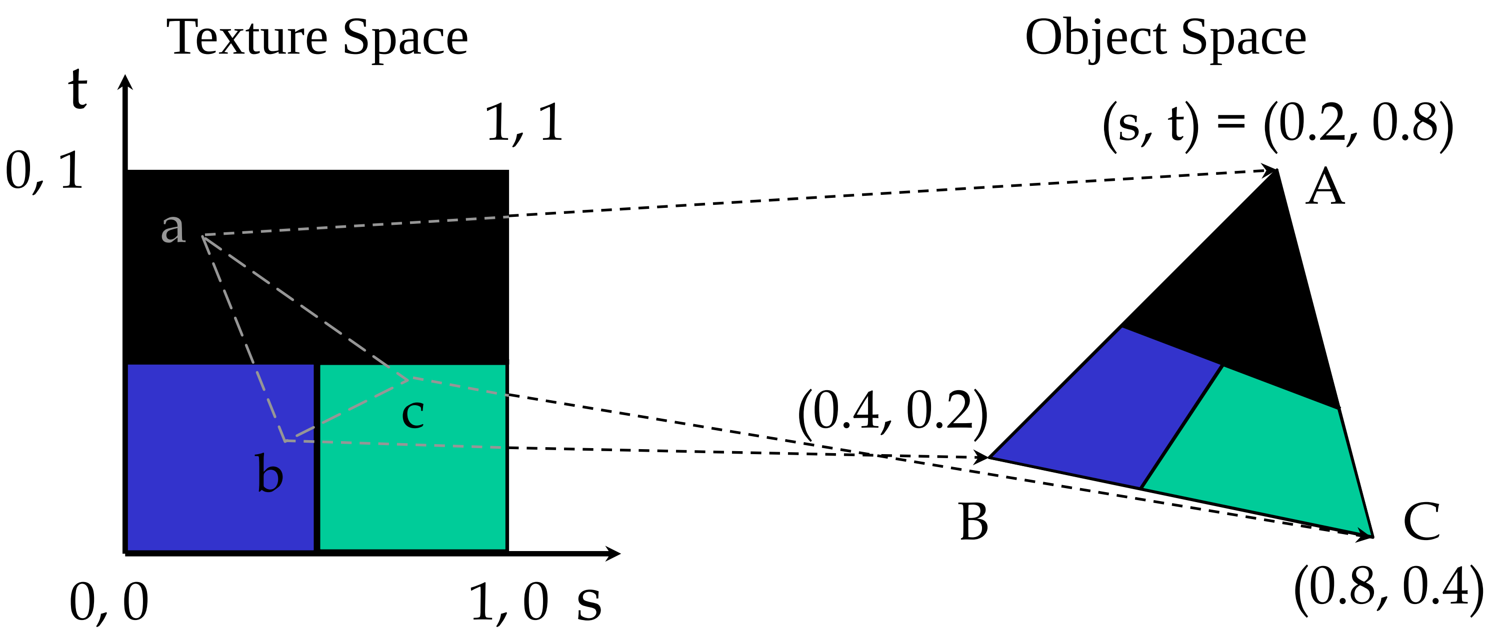
The University of New Mexico

Using a GIF image

```
// specify image in JS file

var image = new Image();
    image.onload = function() {
        configureTexture( image );
    }
    image.src = "SA2011_black.gif”

// or specify image in HTML file with <img> tag
// <img id = "texImage" src = "SA2011_black.gif"></img>

var image = document.getElementById("texImage”)
window.onload = configureTexture( image );
```

Mapping a Texture

- Based on parametric texture coordinates
- Specify as a 2D vertex attribute



Cube Example

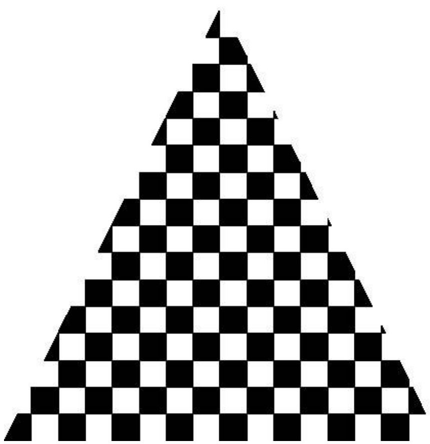
```
var texCoord = [
   vec2(0, 0),
   vec2(0, 1),
   vec2(1, 1),
   vec2(1, 0)
];

function quad(a, b, c, d) {
    pointsArray.push(vertices[a]);
    colorsArray.push(vertexColors[a]);
    texCoordsArray.push(texCoord[0]);

    pointsArray.push(vertices[b]);
     colorsArray.push(vertexColors[a]);
    texCoordsArray.push(texCoord[1]);
// etc
```

Interpolation

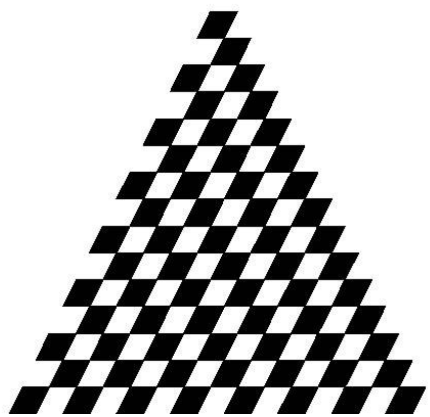
WebGL uses interpolation to find proper texels from specified texture coordinates

Can be distortions

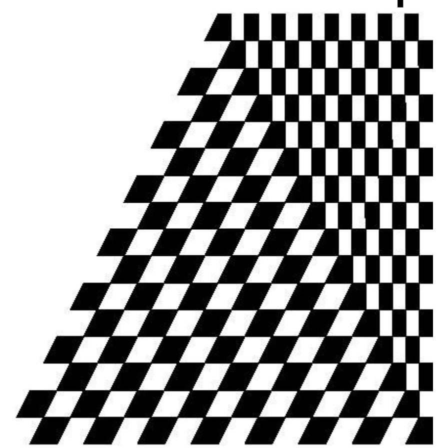
good selection
of tex coordinates



poor selection
of tex coordinates



texture stretched
over trapezoid
showing effects of
bilinear interpolation



Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

WebGL Texture Mapping II

Ed Angel
Professor Emeritus of Computer Science
University of New Mexico

Objectives

- Introduce the WebGL texture functions and options
    - texture objects
    - texture parameters
    - example code

The University of New Mexico

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

Texture Parameters

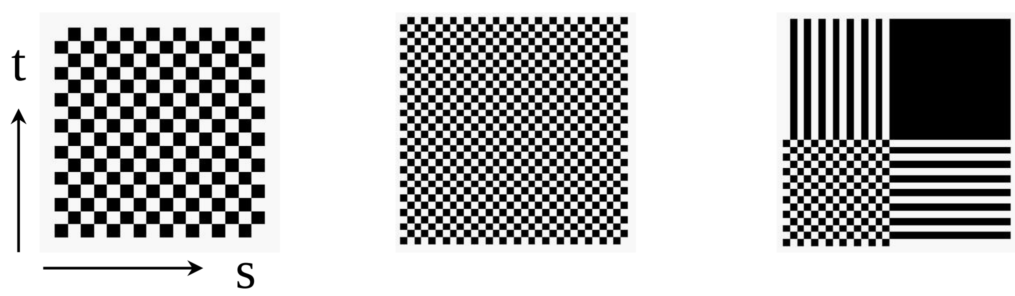
- WebGL has a variety of parameters that determine how texture is applied
  - Wrapping parameters determine what happens if s and t are outside the (0,1) range
  - Filter modes allow us to use area averaging instead of point samples
  - Mipmapping allows us to use textures at multiple resolutions
  -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

```
gl.texParameteri(gl.TEXTURE_2D,
     gl.TEXTURE_WRAP_S, gl.CLAMP )
gl.texParameteri( gl.TEXTURE_2D,
     gl.TEXTURE_WRAP_T, gl.REPEAT )
```



texture

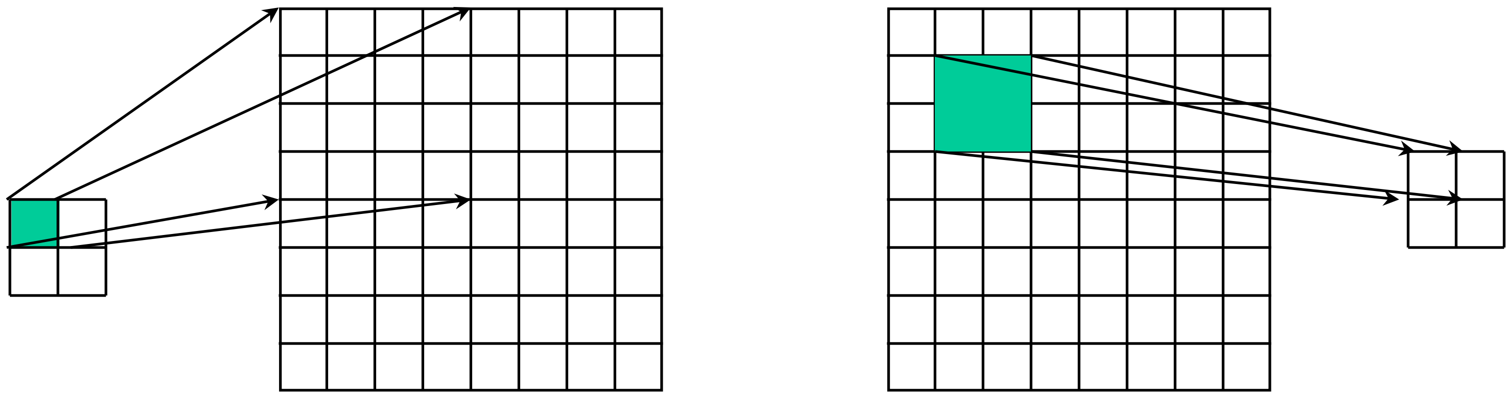
gl.REPEAT wrapping

gl.CLAMP wrapping

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

The University of New Mexico

Filter Modes

Modes determined by

```
gl.texParameteri( target, type, mode )
```

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXURE_MAG_FILTER,
                 GL_NEAREST);

gl.texParameteri(gl.TEXTURE_2D, gl.TEXURE_MIN_FILTER,
                gl.LINEAR);
```

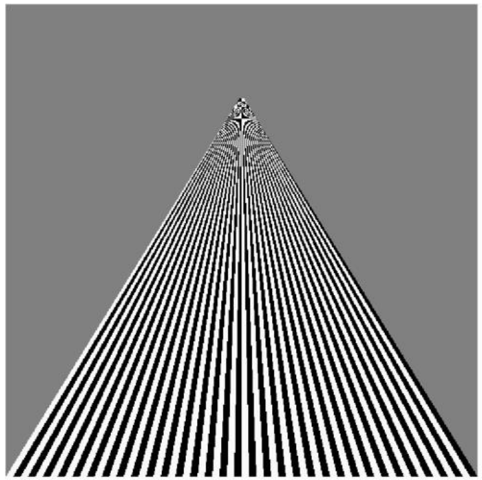
Mipmapped Textures

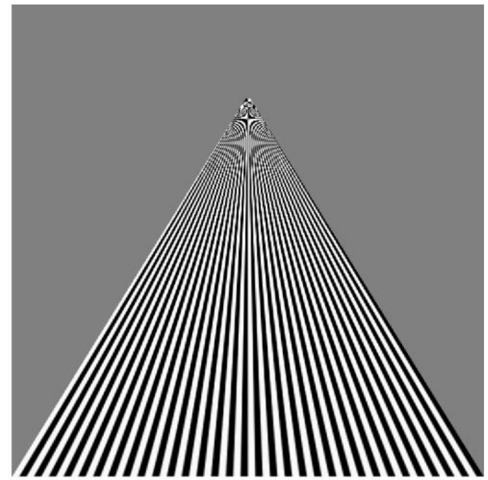
- *Mipmapping* allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects
- Declare mipmap level during texture definition

```
gl.texImage2D(gl.TEXTURE_*D, level, … )
```

Example

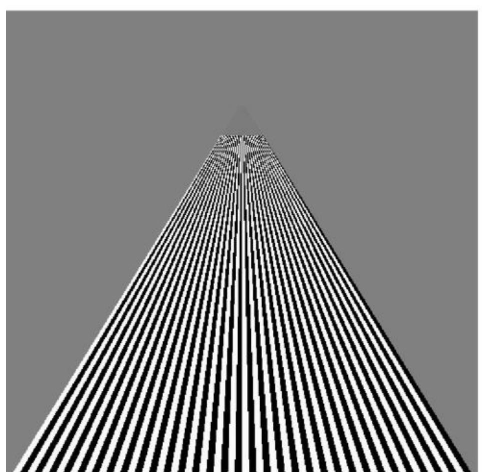
point sampling

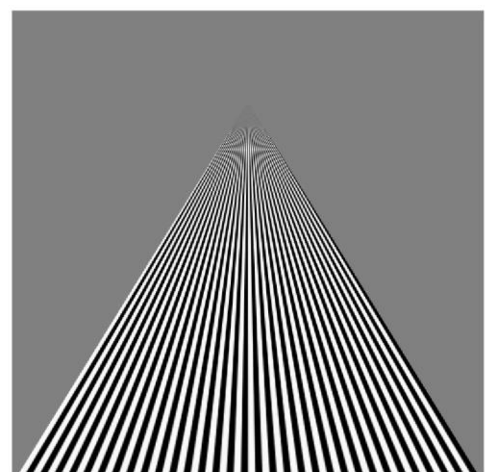




linear filtering

mipmapped point sampling





mipmapped linear filtering

Applying Textures

- Texture can be applied in many ways
  - texture fully determines color
  - modulated with a computed color
  - blended with and environmental color
- Fixed function pipeline has a function glTexEnv to set mode
  - deprecated
  - can get all desired functionality via fragment shader
- Can also use multiple texture units

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

Applying Textures

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

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

void main()  {
        fColor = 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
```

A Checkerboard Image

```
var image1 = new Uint8Array(4*texSize*texSize);
    for ( var i = 0; i < texSize; i++ ) {
        for ( var j = 0; j <texSize; j++ ) {
            var patchx = Math.floor(i/(texSize/numChecks));
            var patchy = Math.floor(j/(texSize/numChecks));
            if(patchx%2 ^ patchy%2) c = 255;
            else c = 0;
            //c = 255*(((i & 0x8) == 0) ^ ((j & 0x8)  == 0))
            image1[4*i*texSize+4*j] = c;
            image1[4*i*texSize+4*j+1] = c;
            image1[4*i*texSize+4*j+2] = c;
            image1[4*i*texSize+4*j+3] = 255;
        }
    }
```

Cube Example

```
var texCoord = [
   vec2(0, 0),
   vec2(0, 1),
   vec2(1, 1),
   vec2(1, 0)
];

function quad(a, b, c, d) {
    pointsArray.push(vertices[a]);
    colorsArray.push(vertexColors[a]);
    texCoordsArray.push(texCoord[0]);

    pointsArray.push(vertices[b]);
     colorsArray.push(vertexColors[a]);
    texCoordsArray.push(texCoord[1]);
// etc
```

Texture Object

```
function configureTexture( image ) {
     var texture = gl.createTexture();
     gl.bindTexture( gl.TEXTURE_2D, texture );
     gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
     gl.texImage2D( gl.TEXTURE_2D, 0, gl.RGB,
          gl.RGB, gl.UNSIGNED_BYTE, image );
     gl.generateMipmap( gl.TEXTURE_2D );
     gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
                  gl.NEAREST_MIPMAP_LINEAR );
     gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
                  gl.NEAREST );
      gl.activeTexture(gl.TEXTURE0);
      gl.uniform1i(gl.getUniformLocation(program, "texture"), 0);
}
```

Linking with Shaders

```
var vTexCoord = gl.getAttribLocation( program, "vTexCoord" );
gl.enableVertexAttribArray( vTexCoord );
gl.vertexAttribPointer( vTexCoord, 2, gl.FLOAT, false, 0, 0);

// 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
//   gl.activeTexture().

gl.uniform1i( glGetUniformLocation(program, "texture"), 0 );
```