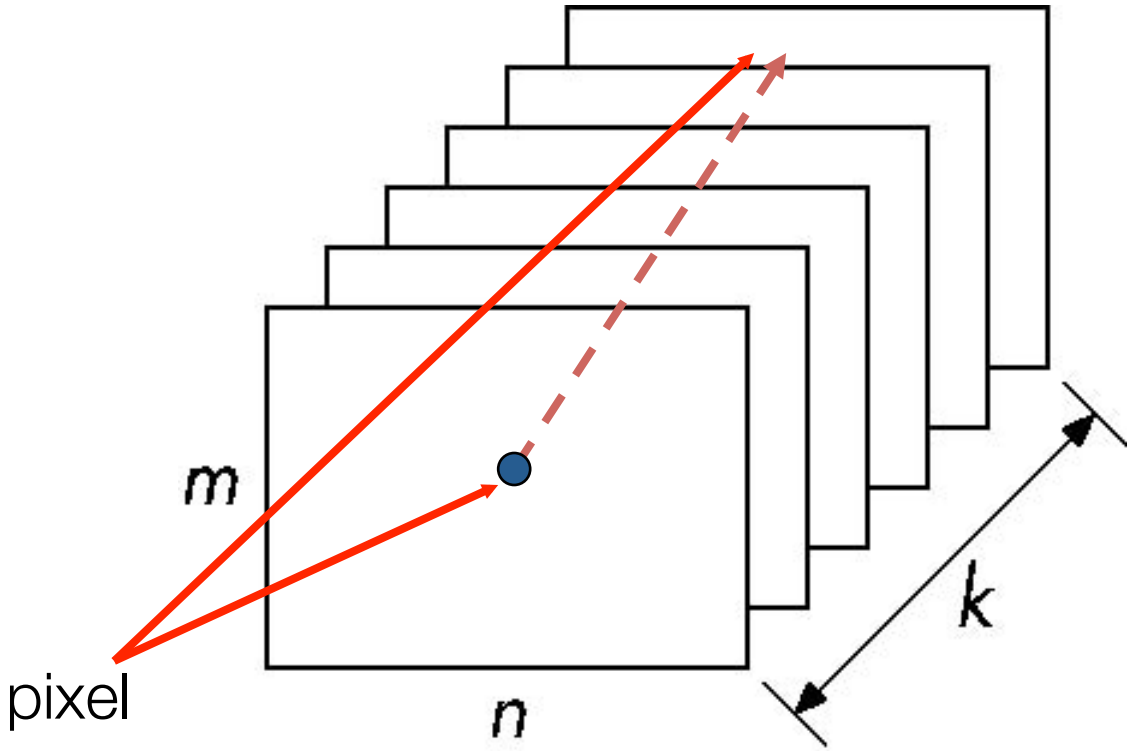
Learning Objectives

- Students completing this lecture will be able to
  - Explain the following terms: environment mapping, bump mapping, texture wrapping, texture mipmapping
  - Describe the benefits of backward texture mapping
  - Write WebGL code to perform texture mapping

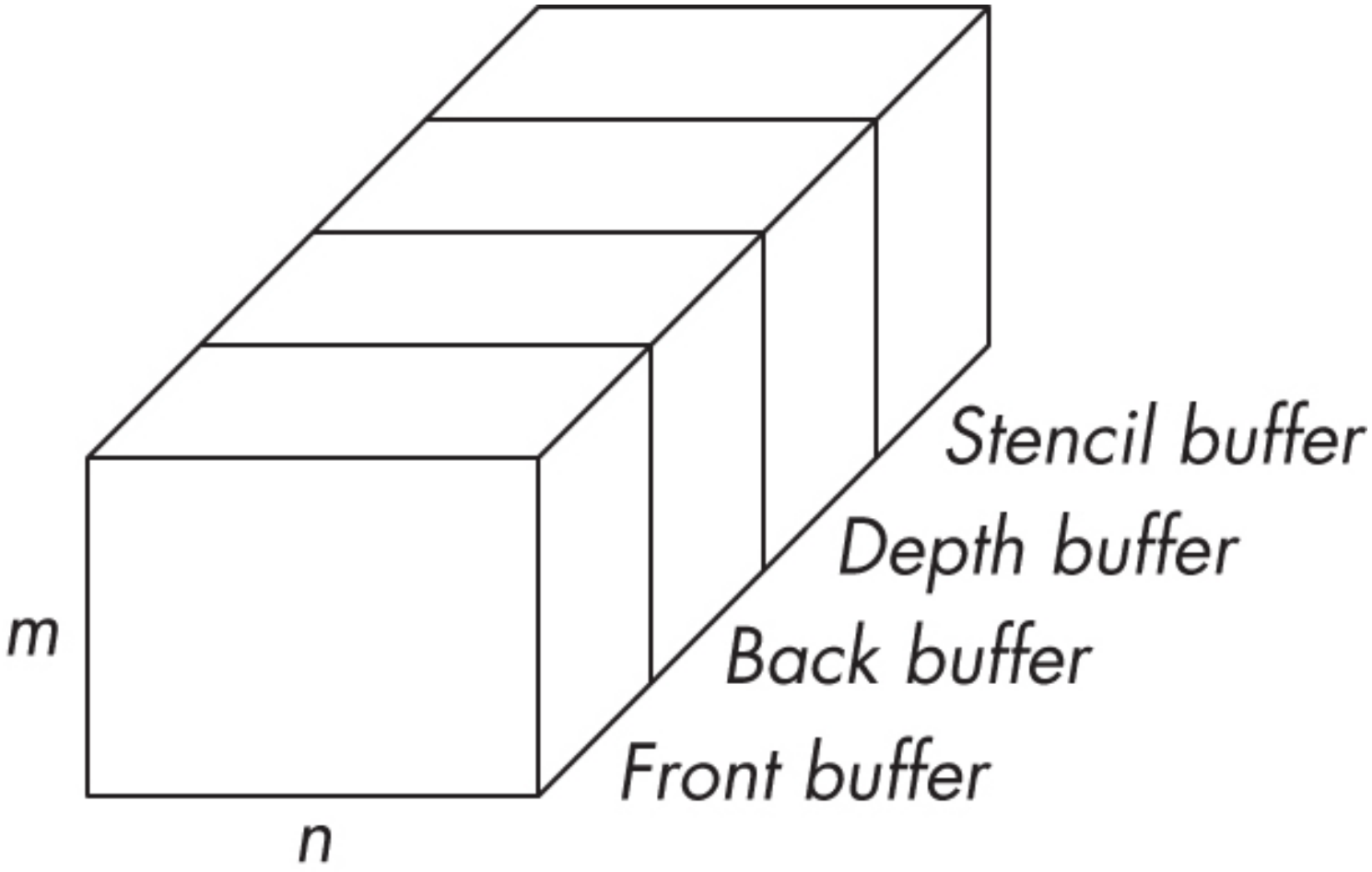
Buffers

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

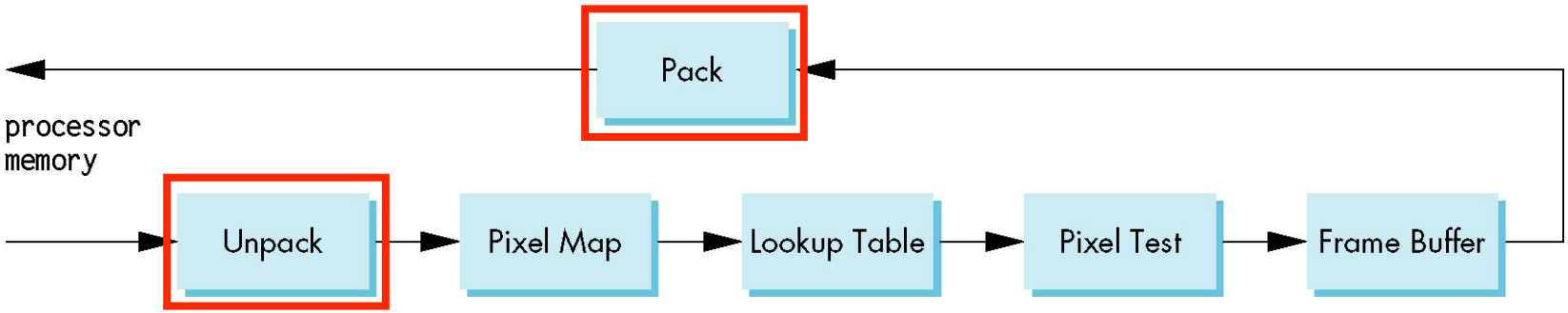
- Desktop OpenGL supported other buffers
  - Auxiliary color buffers
  - Accumulation buffer
  - These were on application side
  - Now deprecated
- GPUs have their own or attached memory
  - Texture buffers
  - Off-screen buffers
    - Not under control of window system
    - May be floating point

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: unpacking converts pixels from the format in the user program to the format used internally by OpenGL
    - Pixels can also be mapped into new values through user-defined lookup tables
    - A series of tests to determine whether the pixel should be written into the frame buffer, and how it should be written
  - 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

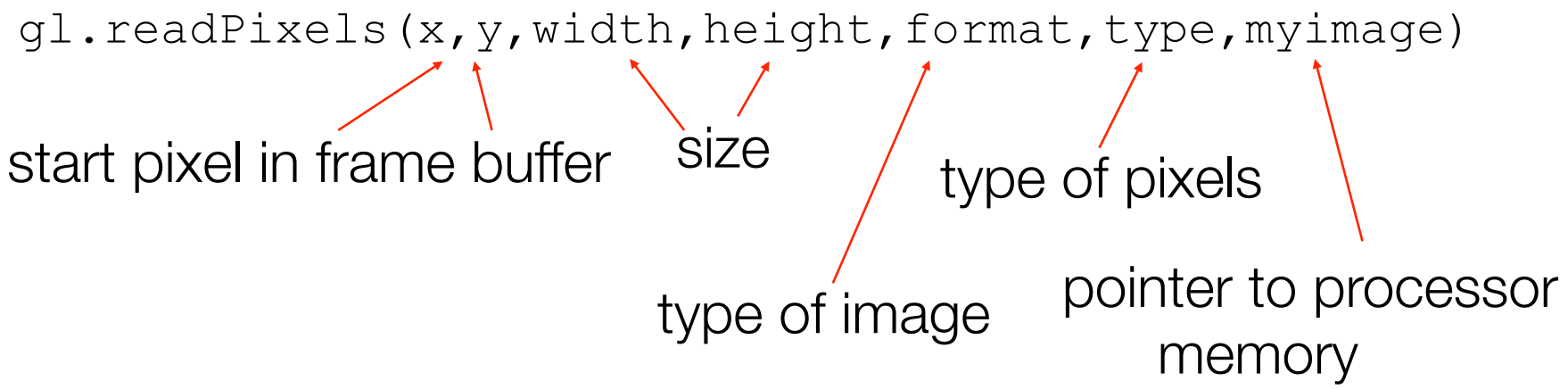
Deprecated OpenGL Functions

- `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




```
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
  - Do not display on screen
  - Image processing or GPGPU applications

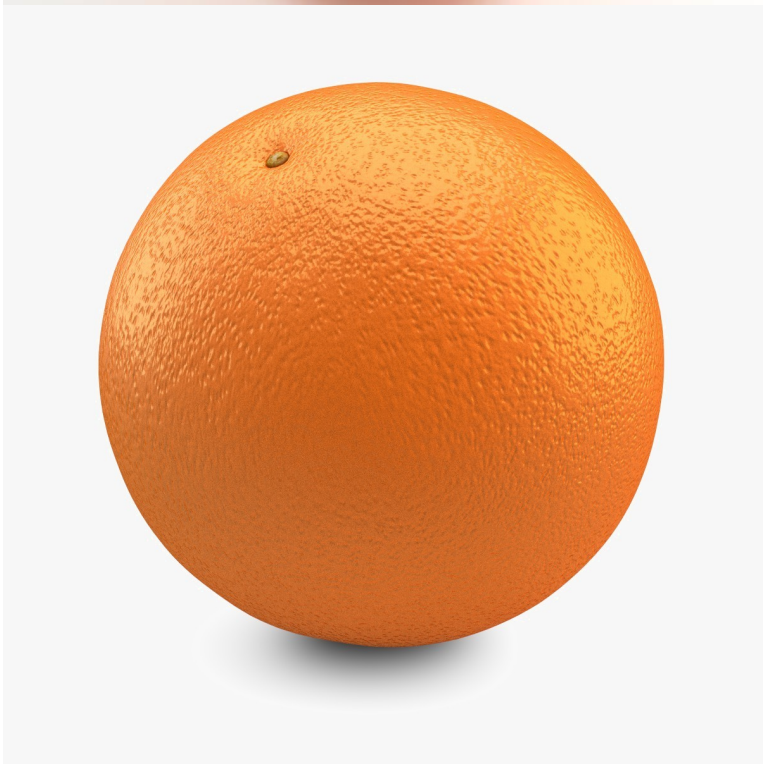
Texture Mapping

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

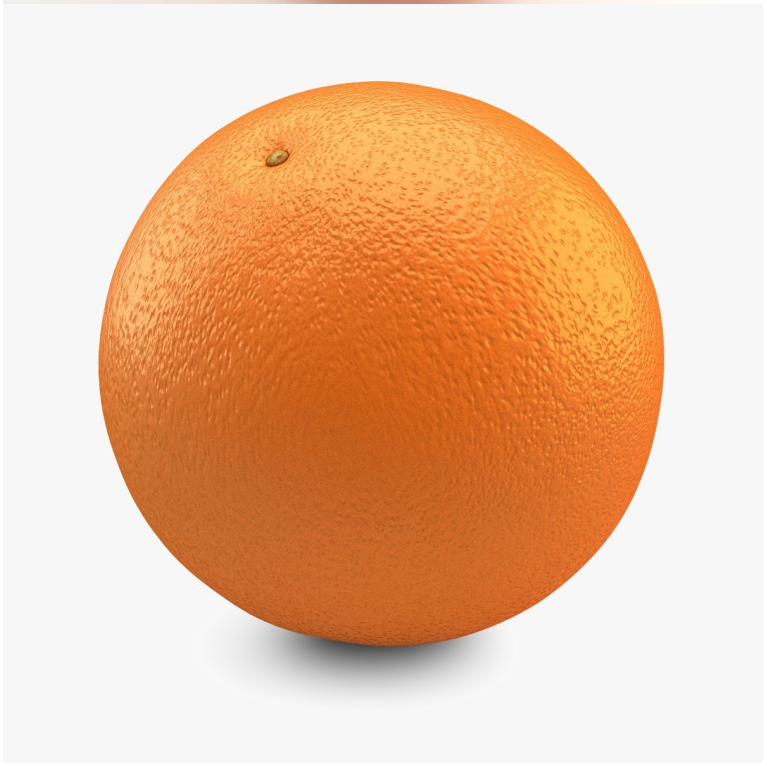
Modeling an Orange

- Consider the problem of modeling an orange
- 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

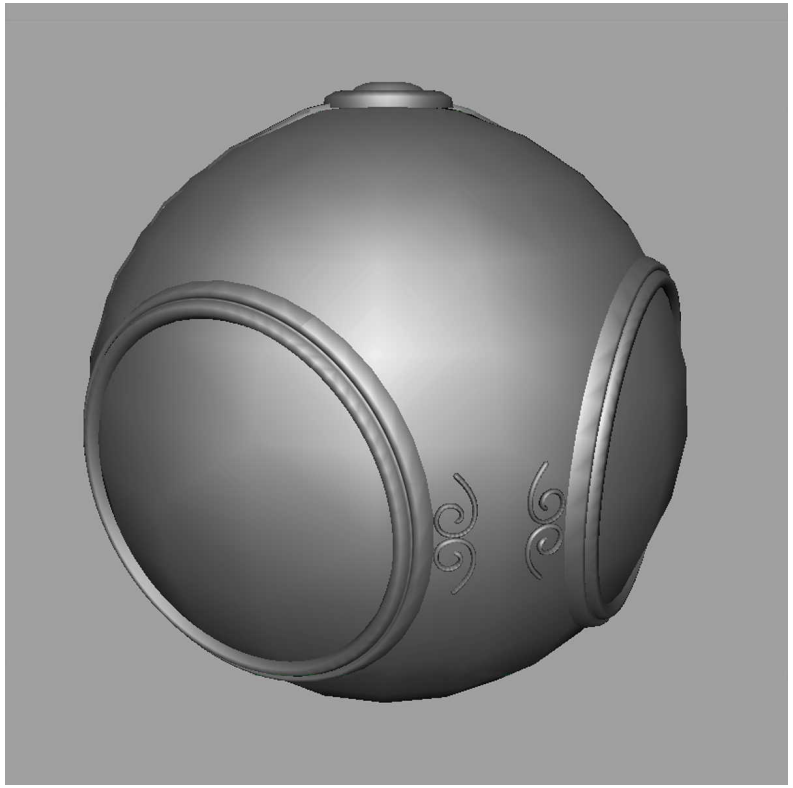
- Take a picture of a real orange, scan it, and “paste” onto simple geometric model
  - This process is known as texture mapping
- Still might not be sufficient because resulting surface will be smooth
  - Need to change local shape
  - Bump mapping



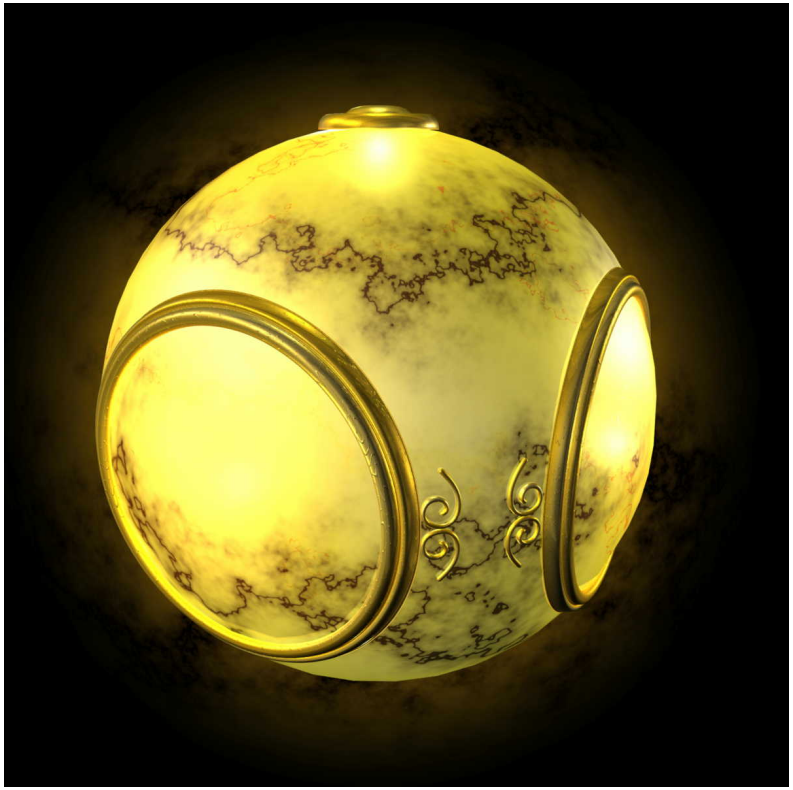
Three Types of Mapping

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

Texture Mapping



geometric model

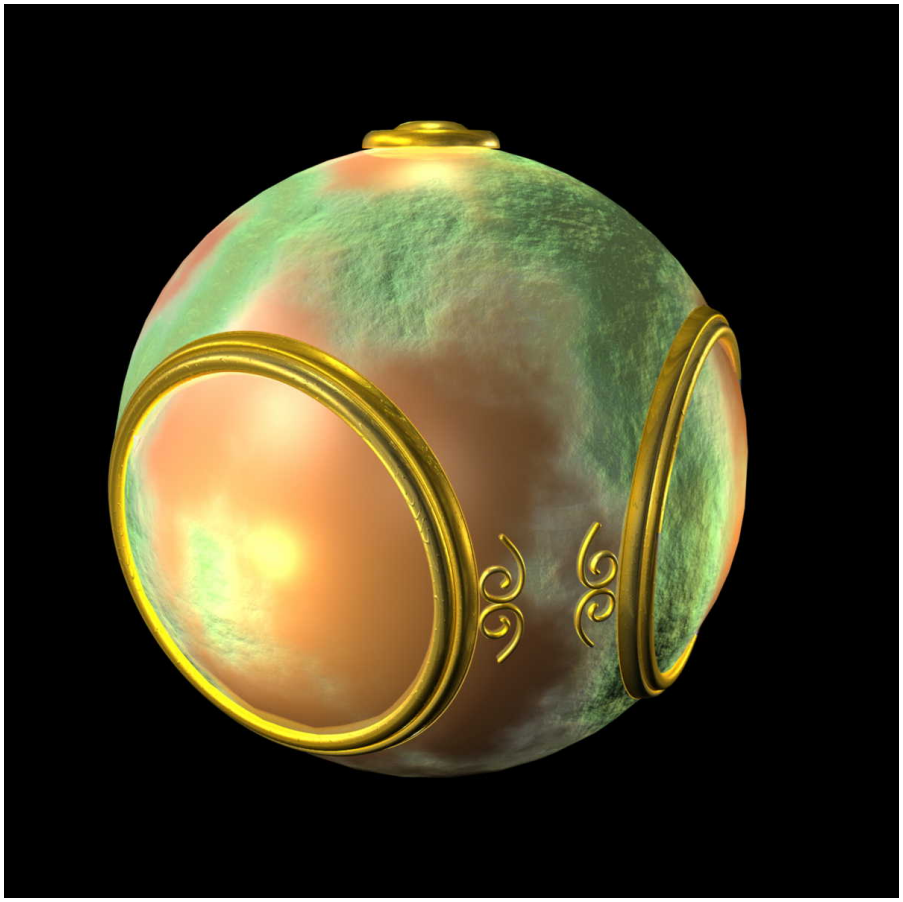


texture mapped

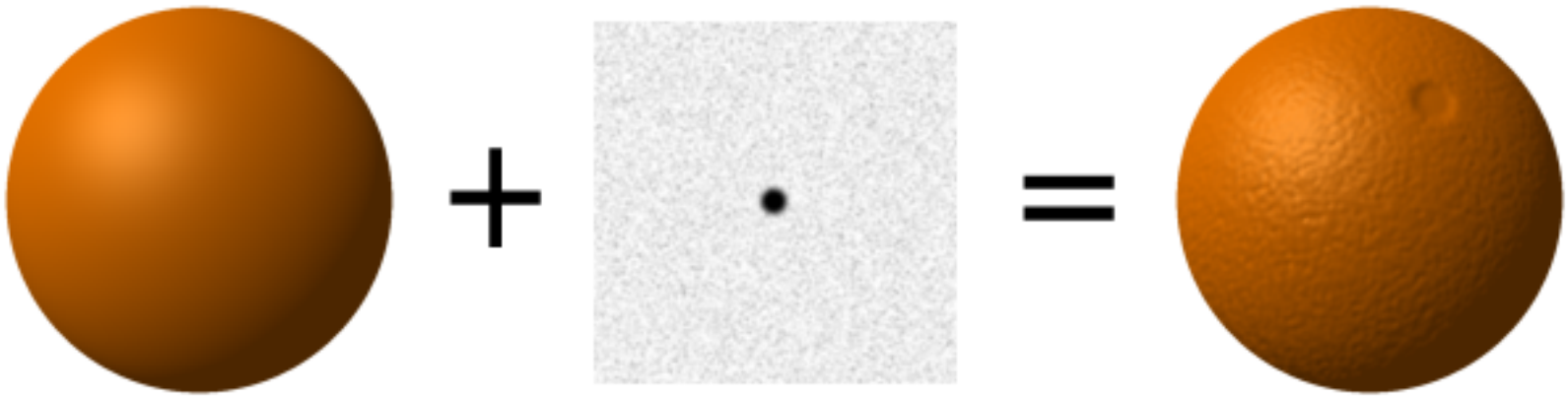
Environment Mapping


Reflective environment Mapping Demo
NVIDIA

Bump Mapping



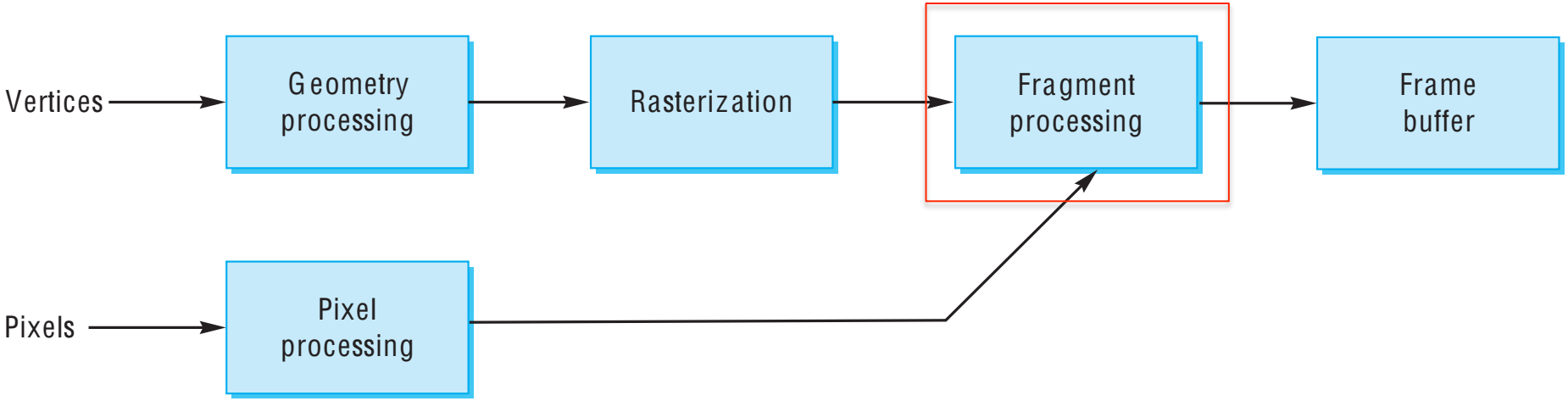
Bump Mapping



bump map

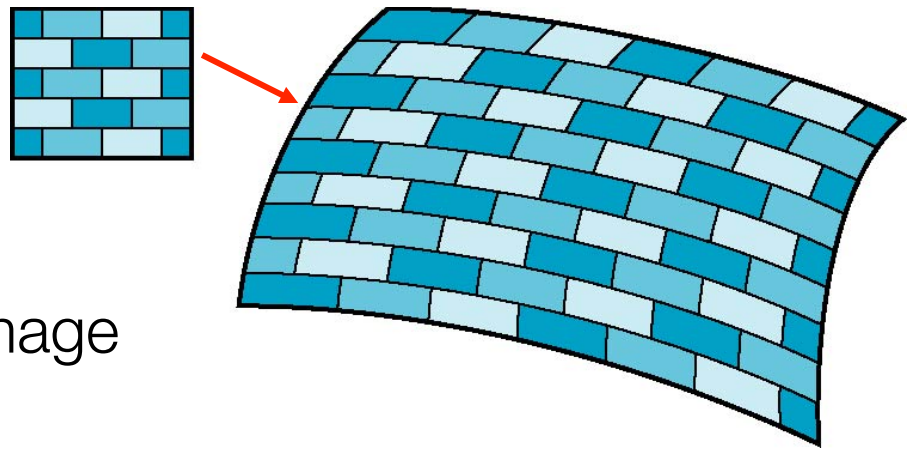
Where does mapping take place?

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



Is it simple?

- Although the idea is simple, mapping an image to a surface involves 3 or 4 coordinate systems



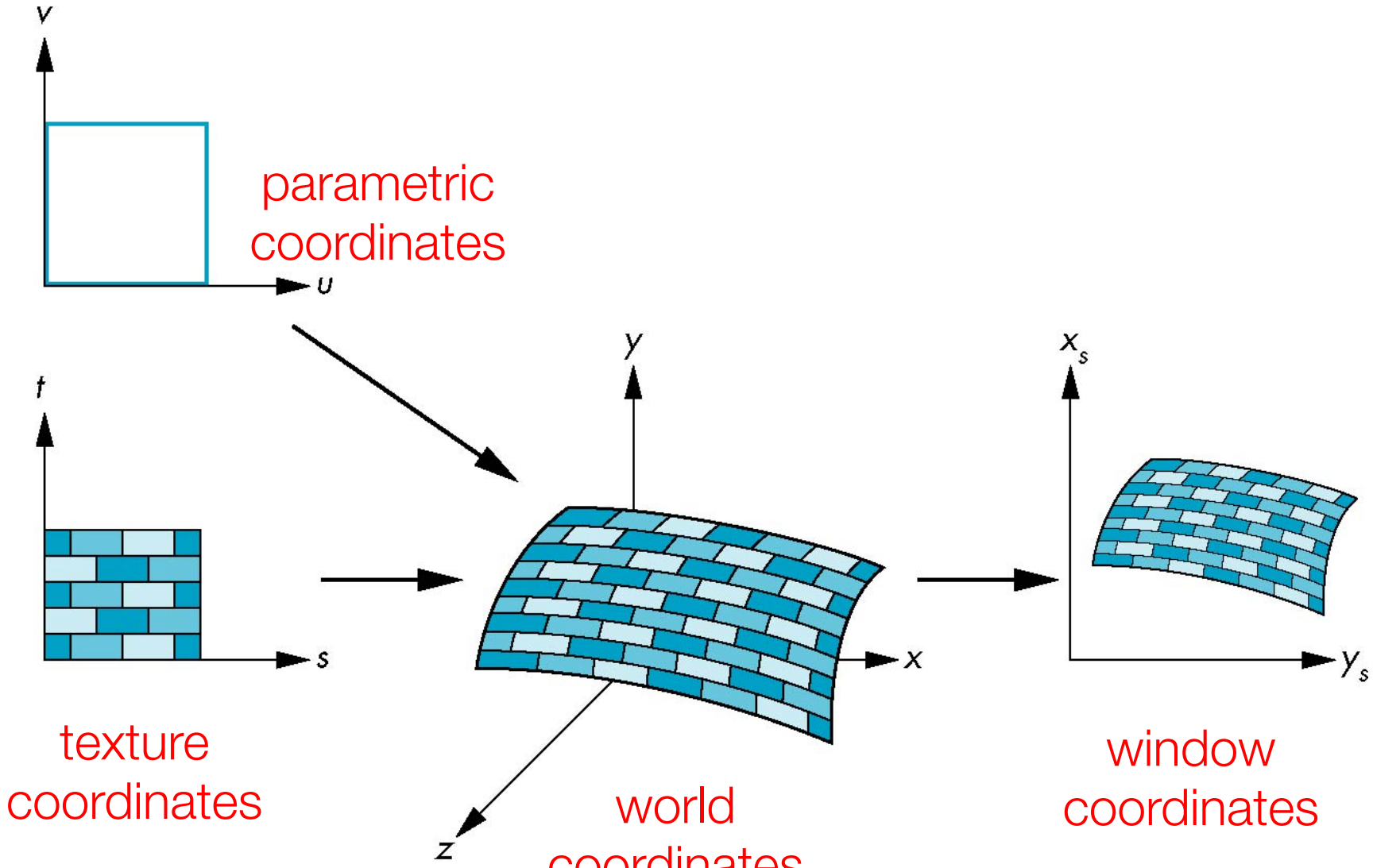
2D image

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

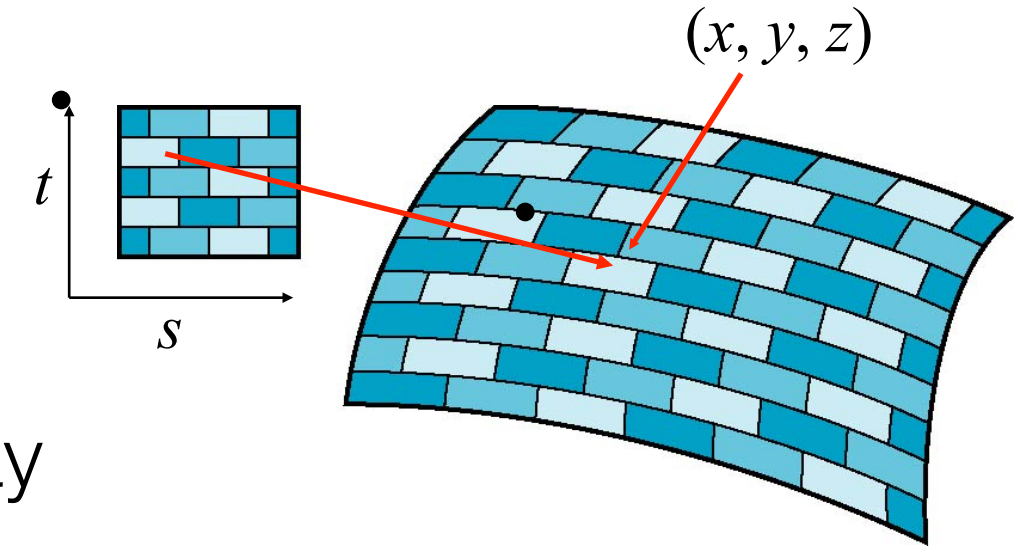


Mapping Functions

- Basic problem is how to find the maps
- Consider mapping from texture coordinates to a point on the 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
  - Forward: Given a point in the texture, we want to know to which point on an object it corresponds
  - Backward: Given a point on an object, we want to know to which point in the texture it corresponds
- Need a map of the form

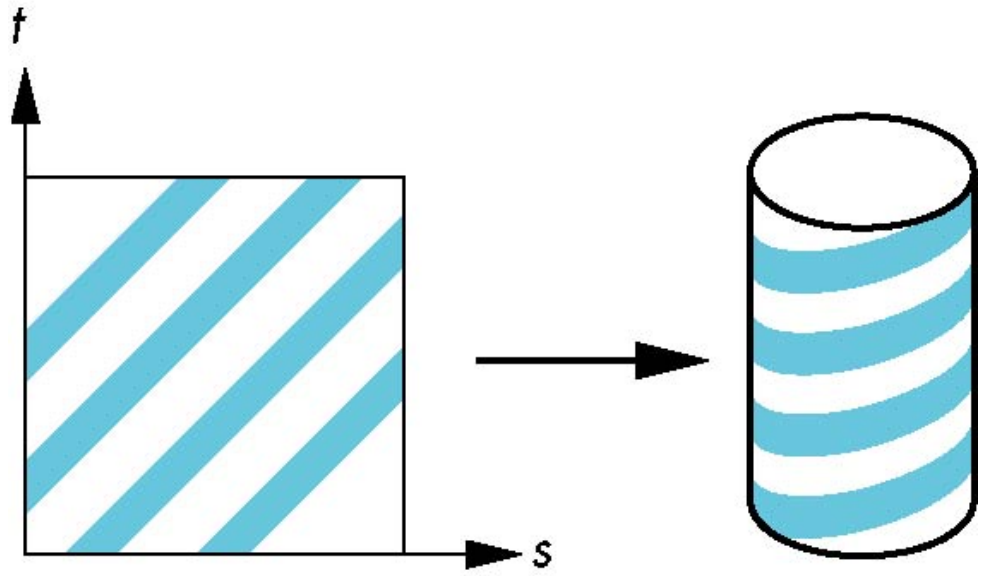
$$s = s(x, y, z)$$

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

- 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

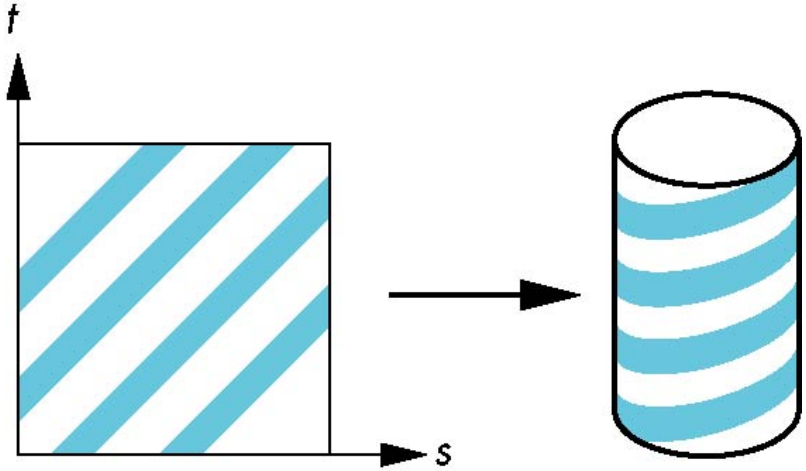
Parametric cylinder

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

Maps rectangle in u, v (u and v are in [0, 1]) space to cylinder of radius r and height h in world coordinates

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

Maps from texture space



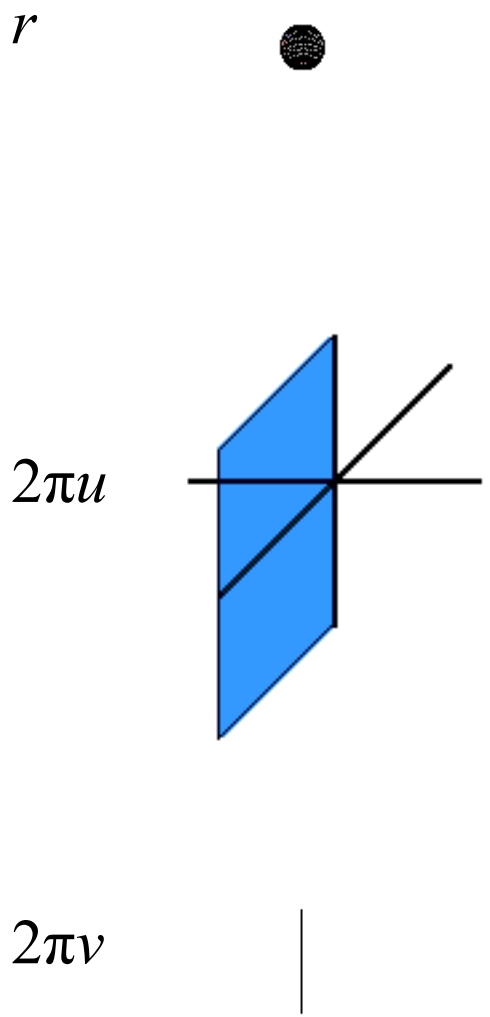
Spherical Map

We can use a parametric sphere

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

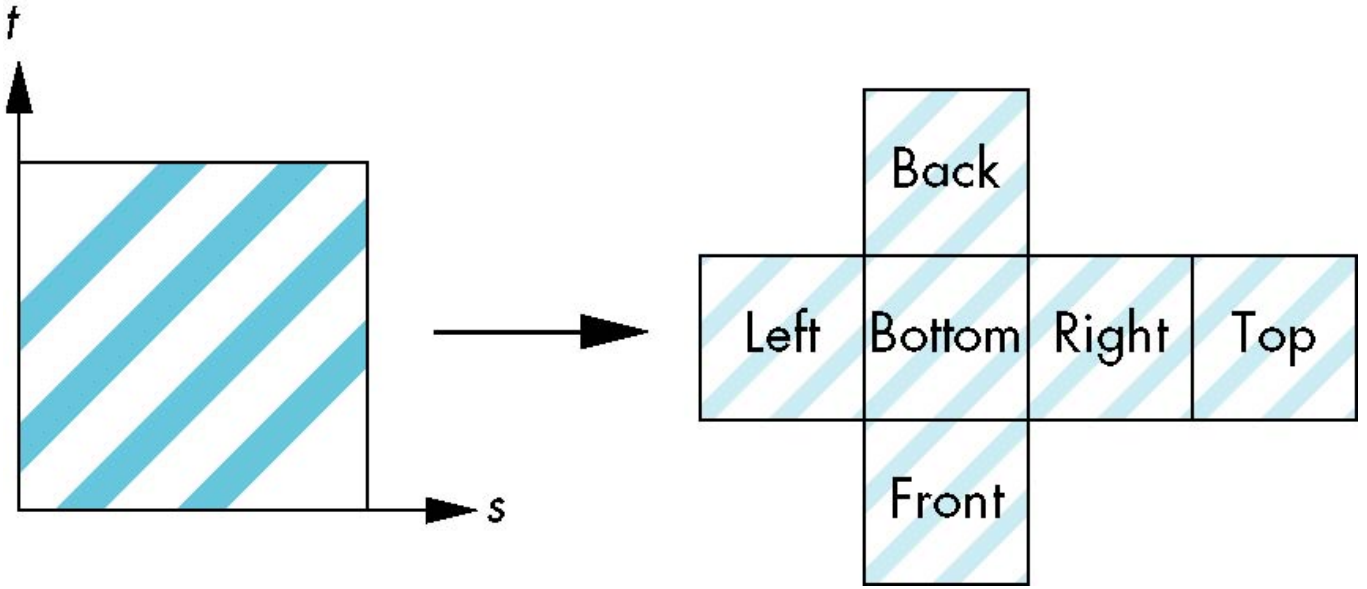
in a similar manner to the cylinder but have to decide where to put the distortion (Mercator projection puts the most distortion at the poles)

Spheres are used in environmental maps



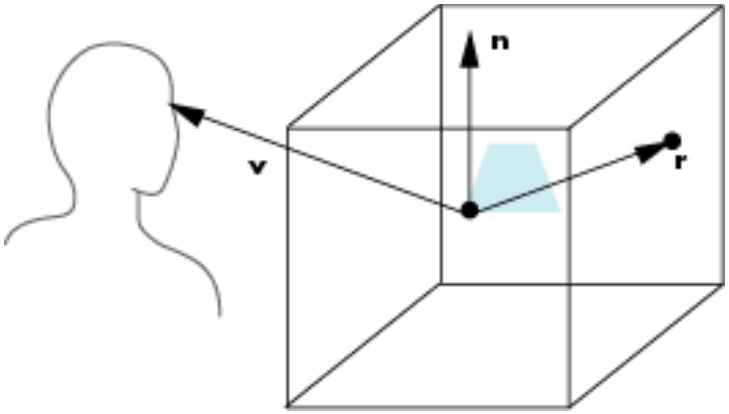
Box Mapping

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



Environment Map

- Use reflection vector to locate texture in cube map



Reflection vs. Refraction

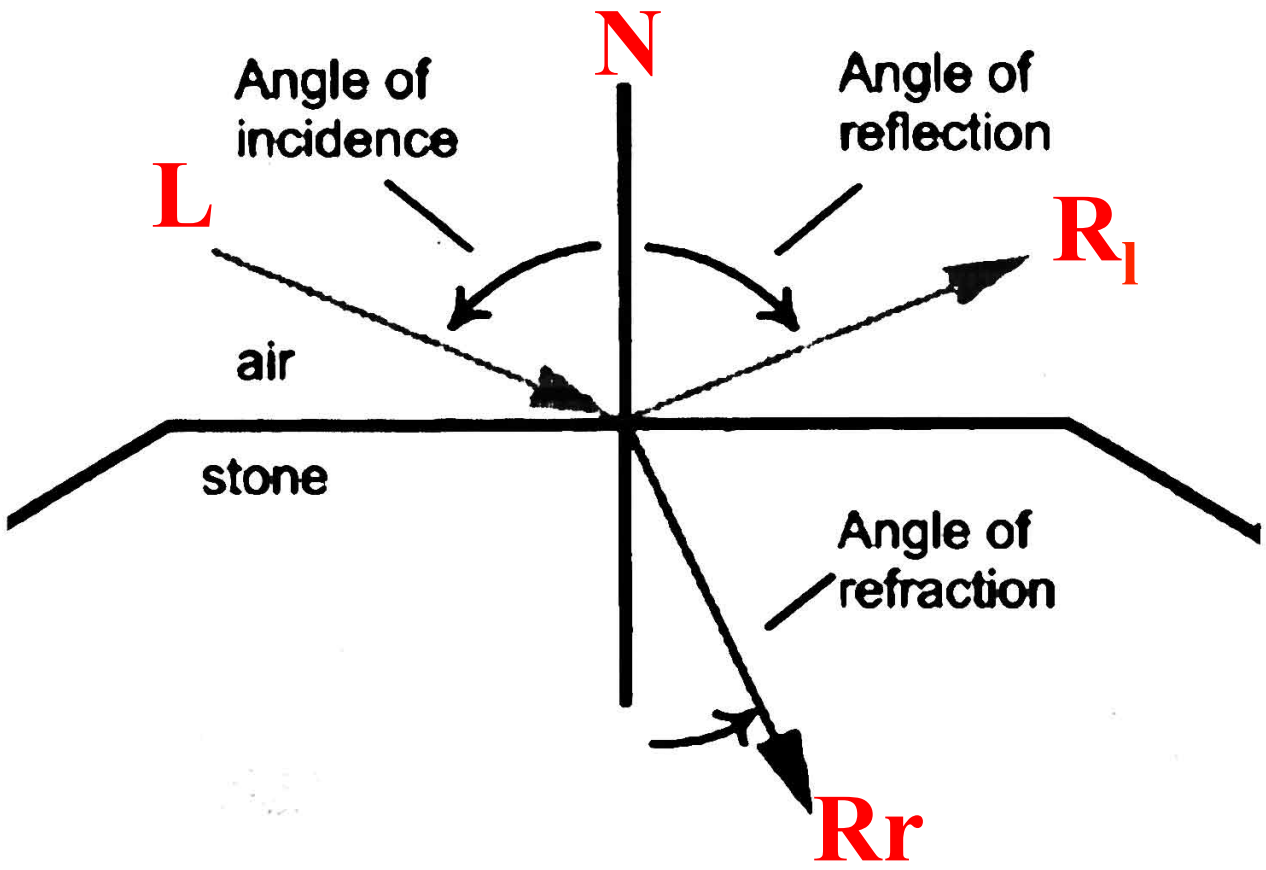
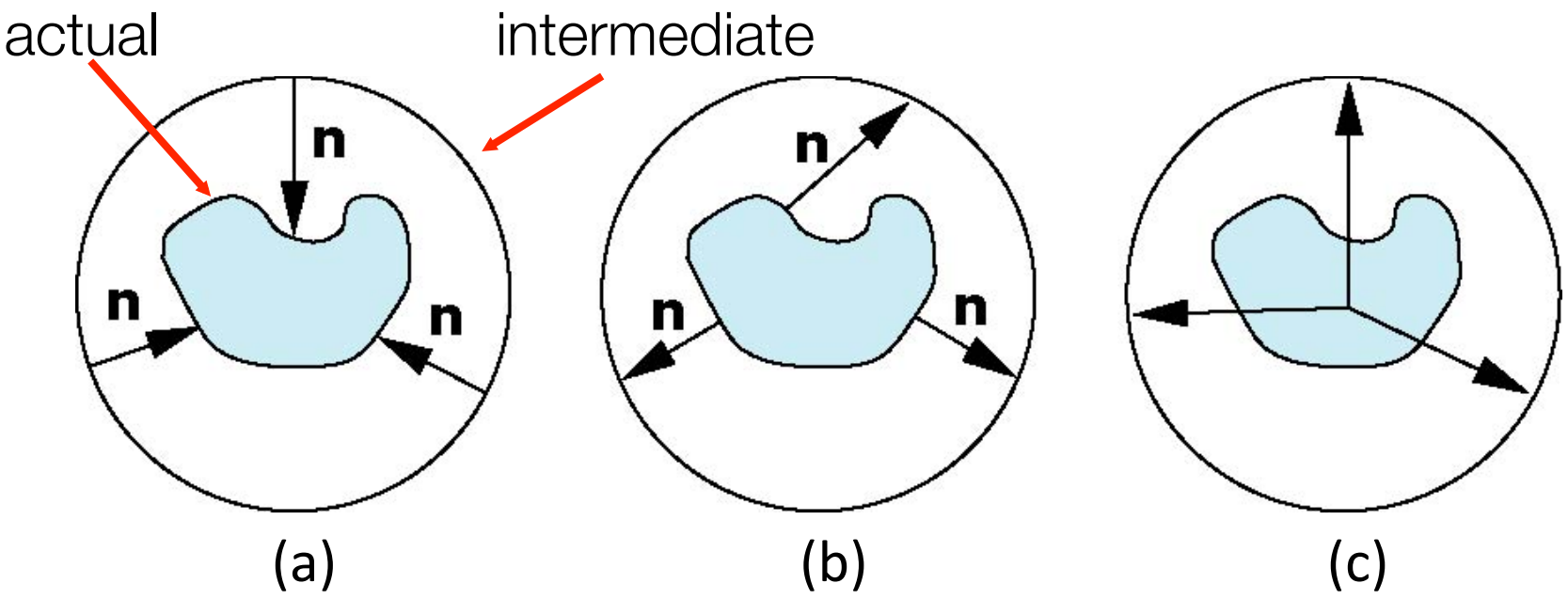


Figure 6.5 : Angles of Incidence, reflection and refraction

Second Mapping

- Map from intermediate object to actual object
  - (a) Normals from intermediate to actual
  - (b) Normals from actual to intermediate
  - (c) Vectors from center of intermediate

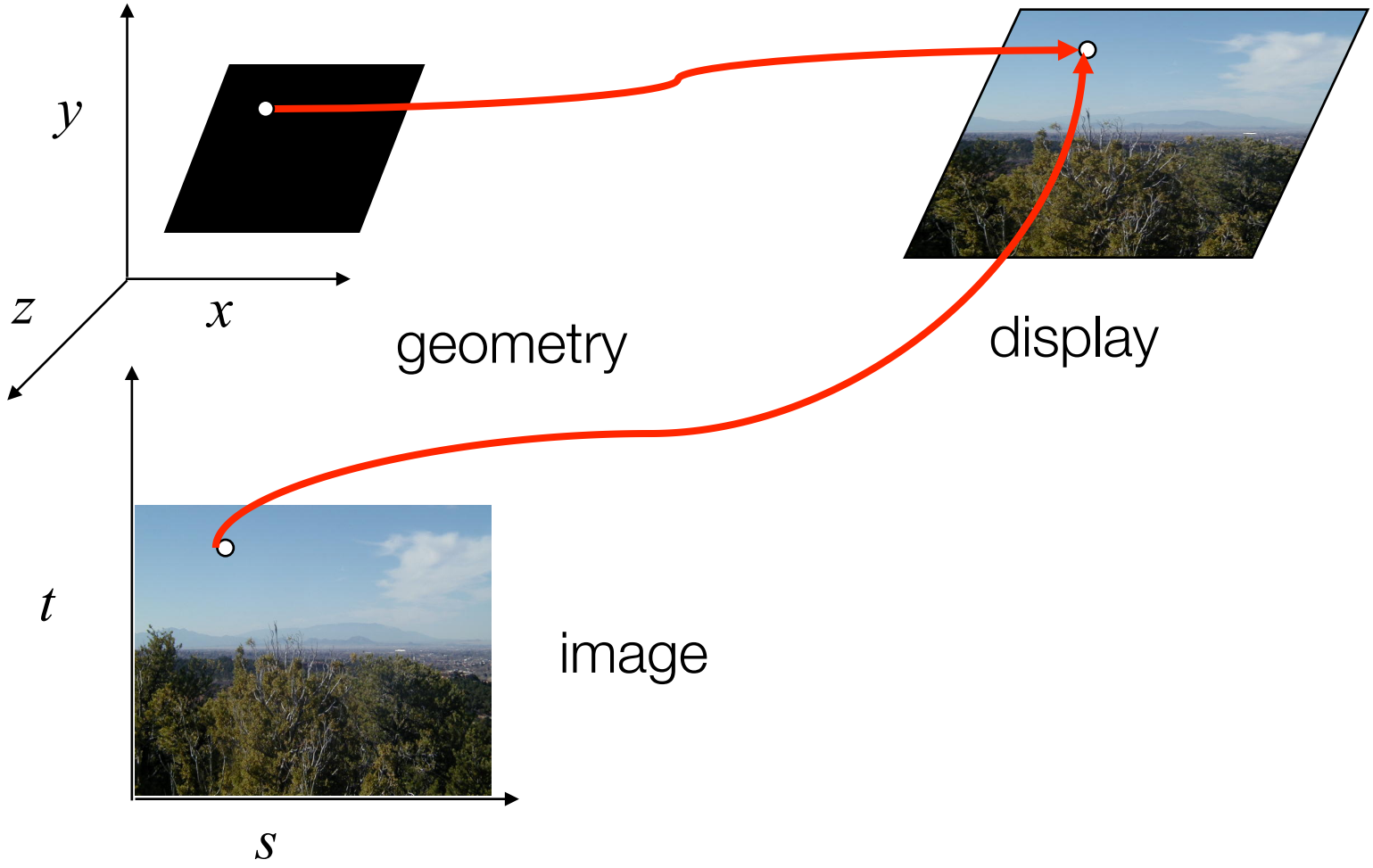


WebGL Texture Mapping

Basic Strategy

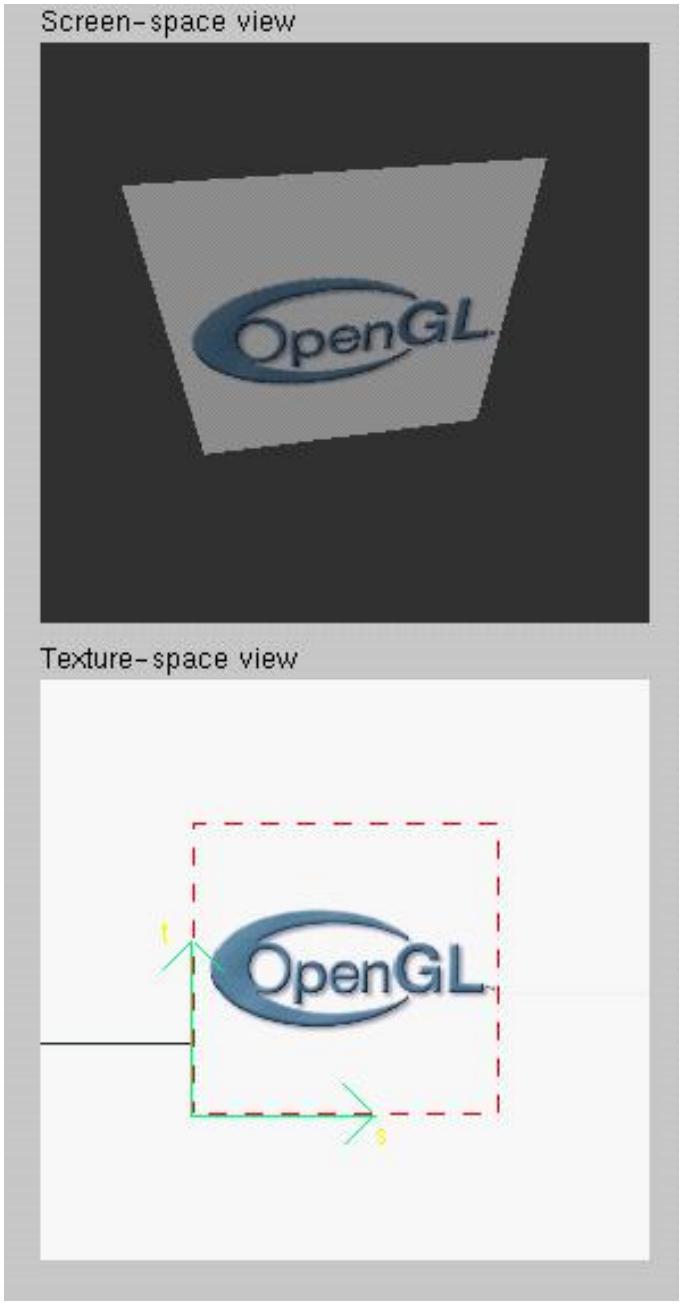
- Three steps to applying a texture
  1. Specify the texture
     - Read or generate image
     - Assign to texture
     - Enable texturing
  2. Assign texture coordinates to vertices
     - Proper mapping function is left to application
  3. Specify texture parameters
     - Wrapping, filtering

Texture Mapping



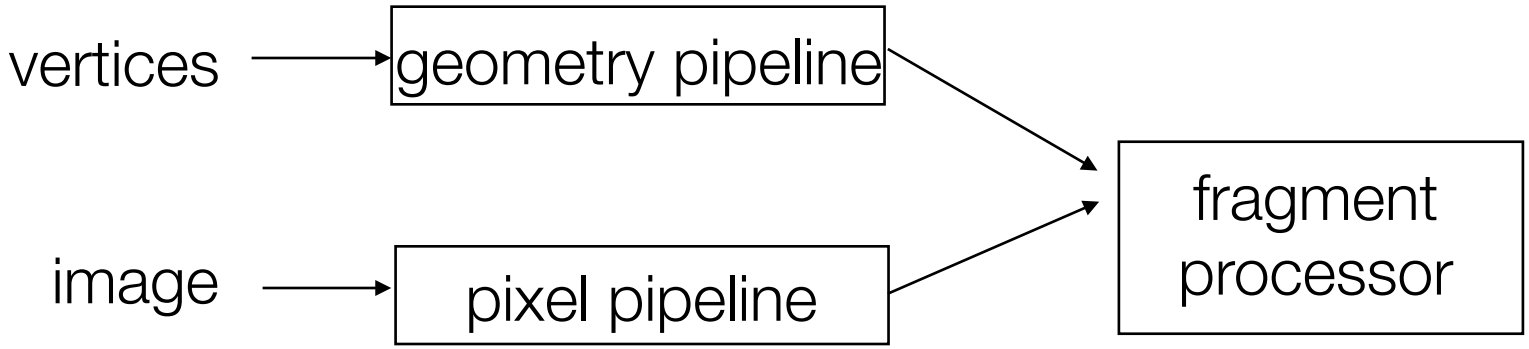
Texture Example

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



Texture Mapping and the OpenGL 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
- Define an image in a standard format such as JPEG
  - Scanned image
  - Generate by application code
- WebGL supports only two dimensional texture maps
  - No need to enable as in desktop OpenGL
  - Desktop OpenGL supports 1-4 dimensional texture maps

Define Image as a Texture

```
gl.texImage2D( target, level, internalformat,
      format, type, image );
```

`target`: type of texture, e.g., `gl.TEXTURE_2D`

`level`: used for mipmapping (discussed later)

`internalformat`: describe internal format of the image

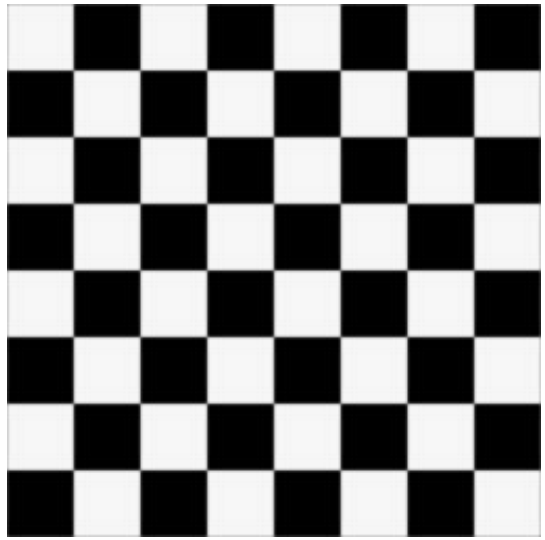
`format and type`: describe texel format and type

`image`: image object to be used as a texture

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGB, gl.RGB,
   gl.UNSIGNED_BYTE, my_image);
```

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; // ^ XOR
            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;
        }
    }
```

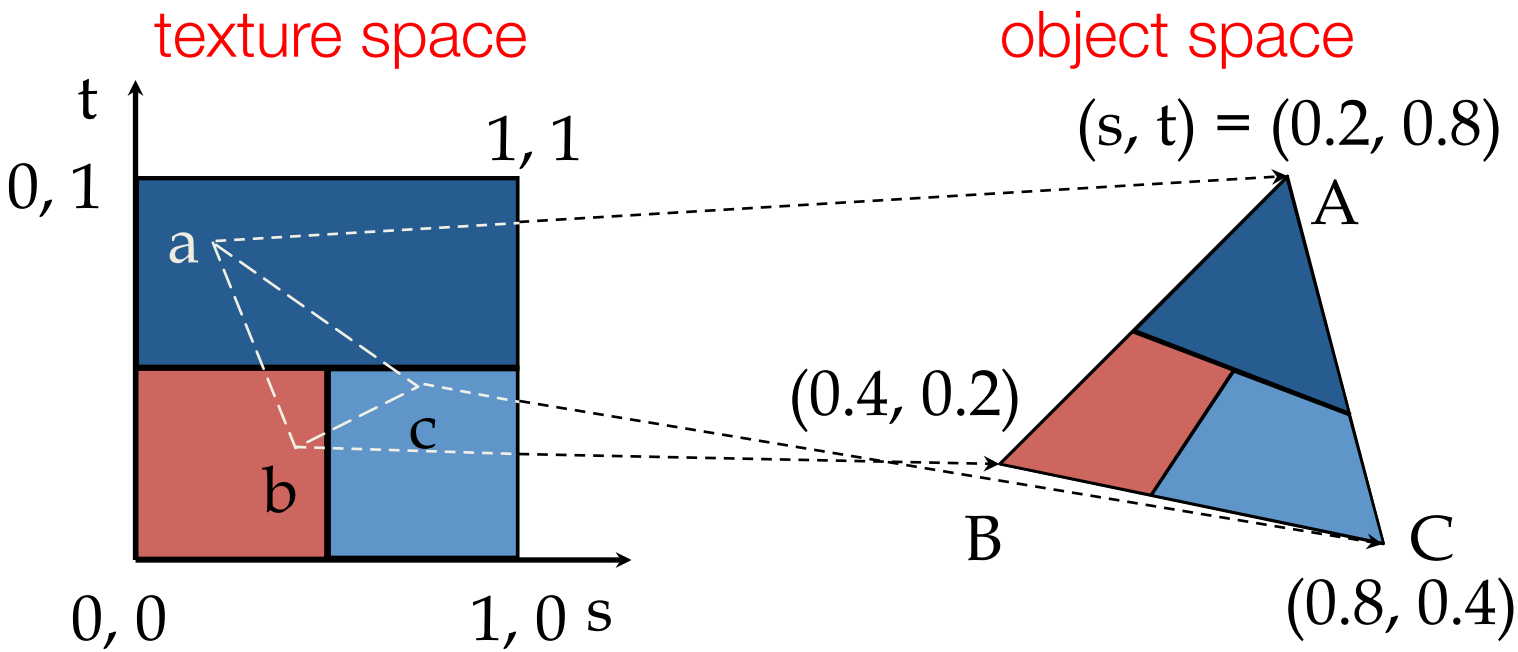


Using a GIF image

```
// 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”
hidden></img>

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

Mapping a Texture

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



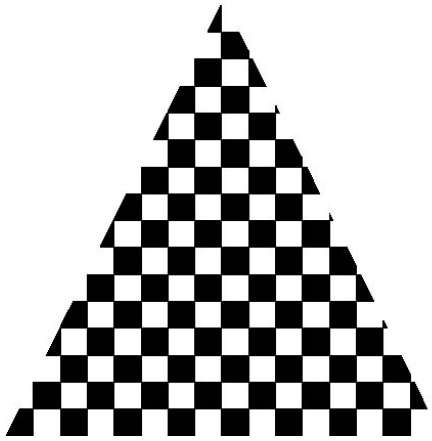
Cube Example

```
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[1]);

     pointsArray.push(vertices[b]);
     colorsArray.push(vertexColors[a]);
     texCoordsArray.push(texCoord[0]);
     // etc
```

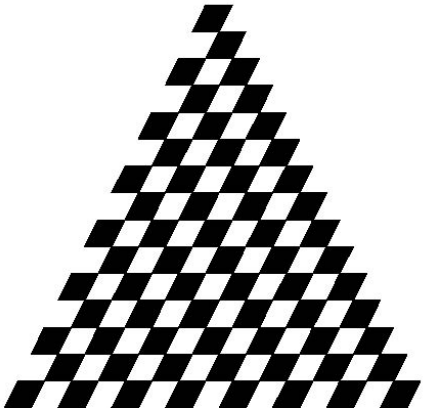
Interpolation

- WebGL uses interpolation to find proper texels from specified texture coordinates
- Can have distortions

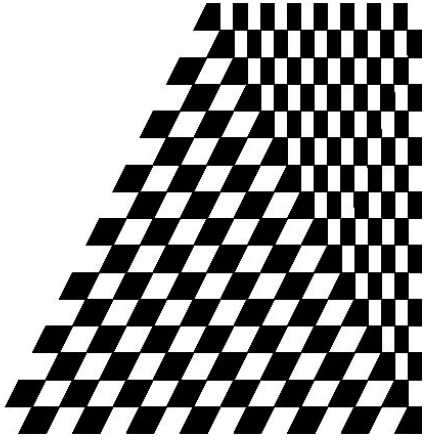
good selection of tex coordinates



poor selection of tex coordinates



quadrilateral is treated as two triangles



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

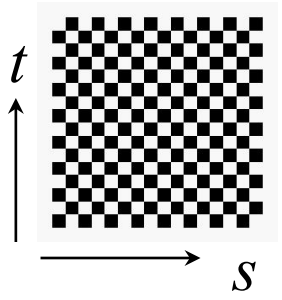
- WebGL has a variety of parameters that determine how texture is applied
  - Wrapping parameters determine what happens if s and t are outside the (0,1) range
  - Filter modes allow us to use area averaging instead of point samples
  - Mipmapping allows us to use textures at multiple resolutions
  -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

Wrapping Mode

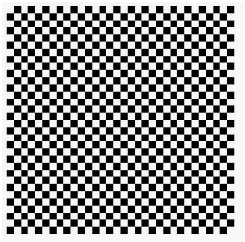
Clamping: if $s, t > 1$ use 1, if $s, t < 0$ use 0

Wrapping: use s, t modulo 1

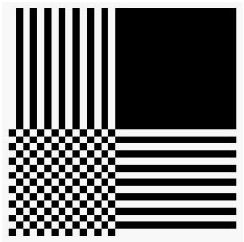
```
gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
   gl.CLAMP_TO_EDGE )
gl.TexParameteri( gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
   gl.REPEAT )
```



texture



gl.REPEAT wrapping

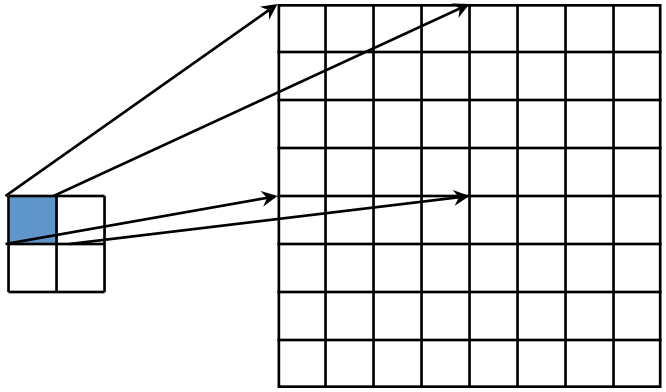


gl.CLAMP_TO_EDGE wrapping

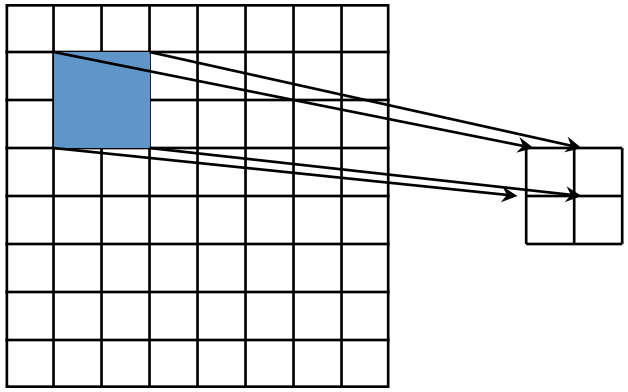
Magnification and Minification

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/Zoom in



Texture Polygon
Minification/Zoom out

Filter Modes

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

gl.texParameteri(gl.TEXTURE_2D, gl.TEXURE_MAG_FILTER,
                 gl.NEAREST); // image bigger than texture

gl.texParameteri(gl.TEXTURE_2D, gl.TEXURE_MIN_FILTER,
                 gl.LINEAR); // image smaller than texture
```

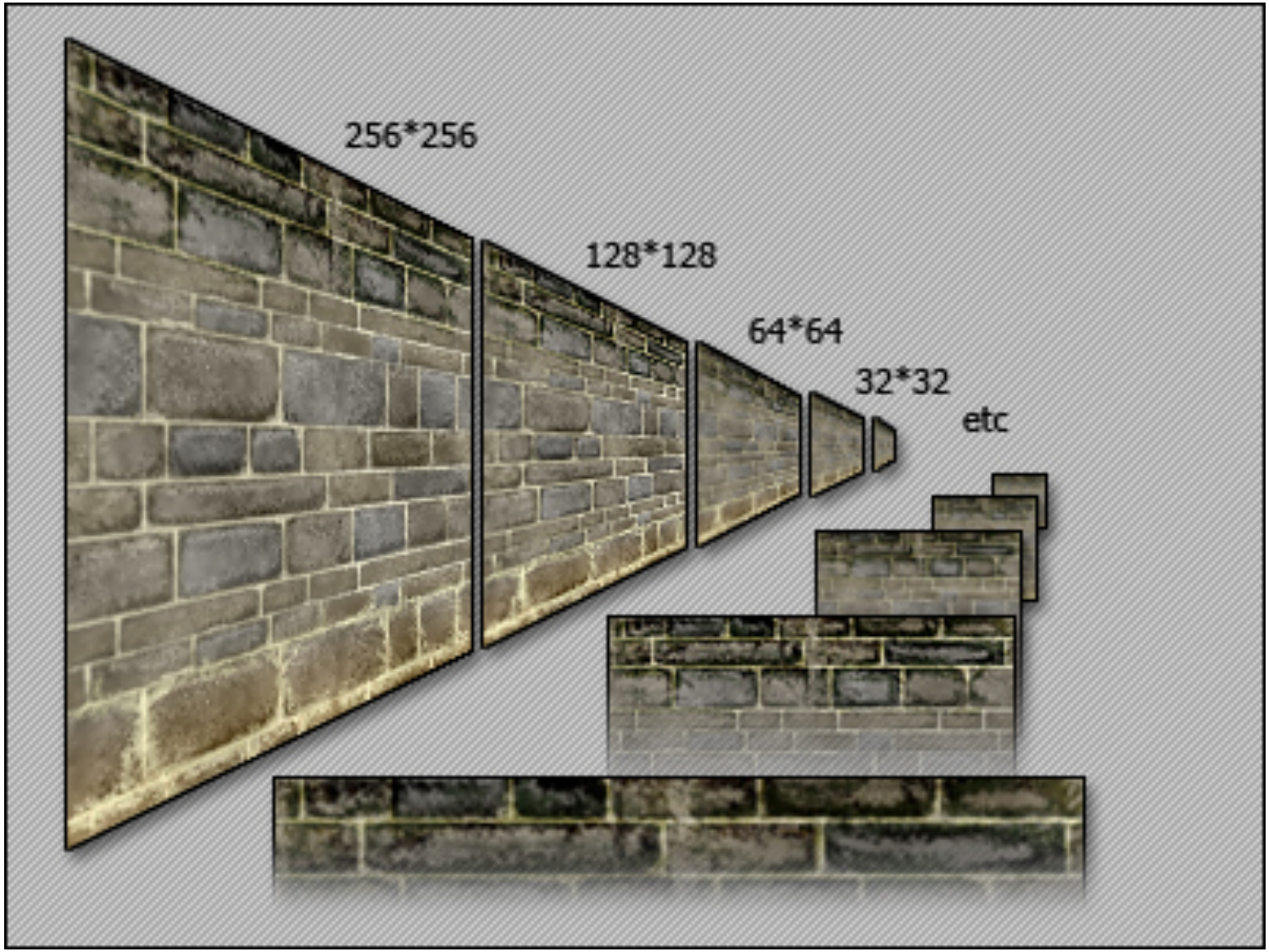
pname	Return type	Description	Possible return values
Available in a WebGL 1 context			
`gl.TEXTURE_MAG_FILTER`	GLenum	Texture magnification filter	`gl.LINEAR` (default value), `gl.NEAREST`.
`gl.TEXTURE_MIN_FILTER`	GLenum	Texture minification filter	`gl.LINEAR`, `gl.NEAREST`, `gl.NEAREST_MIPMAP_NEAREST`, `gl.LINEAR_MIPMAP_NEAREST`, `gl.NEAREST_MIPMAP_LINEAR` (default value), `gl.LINEAR_MIPMAP_LINEAR`.

Mipmapped Textures

- Allow for prefiltered texture maps of decreasing resolutions
- Lessen interpolation errors for smaller textured objects
- Declare mipmap level during texture definition
  - Typically set to 0 (the base image level)
  - Level n is the nth mipmap reduction level

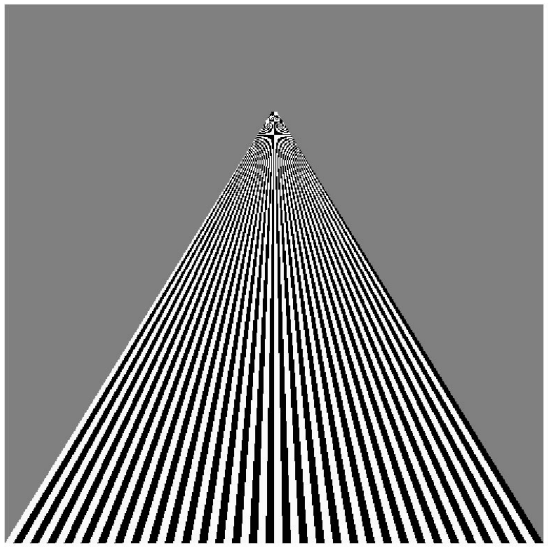
```
gl.texImage2D( gl.TEXTURE_*D, level, … );
gl.generateMipmap( gl.TEXTURE_2D );
```

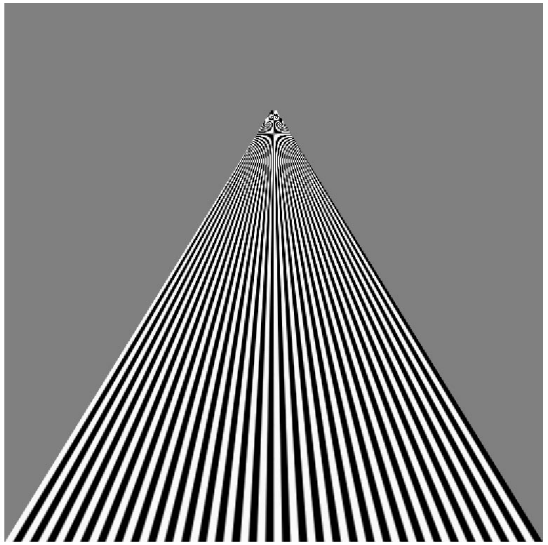
Mipmapped Textures



Example

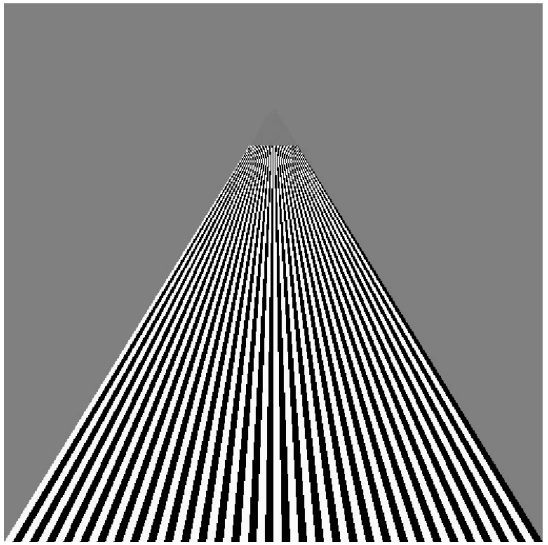
point sampling

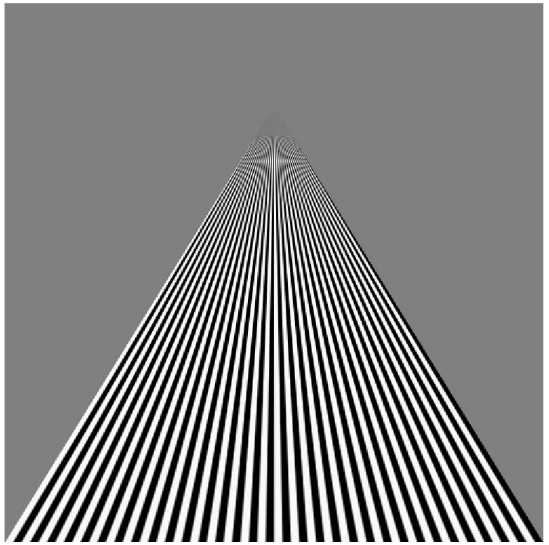




linear filtering

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
- Multi-texturing
  - 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

```
varying vec4 fColor;  //color from rasterizer
varying vec2 fTexCoord; //texture coordinate from rasterizer
//texture object from application
uniform sampler2D textureSampler;

void main()  {
    gl_FragColor = fColor*texture2D(textureSampler,fTexCoord);
}
```

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

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

varying vec4 fColor; //output color to be interpolated
varying vec2 fTexCoord; //output tex coordinate to be interpolated
```

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 );
```

Linking with Shaders

```
    gl.activeTexture( gl.TEXTURE0 );
    var textureSamplerLoc = gl.getUniformLocation(program,
                        "textureSampler");
    // Set the value of the fragment shader texture sampler
    // variable ("textureSampler") to the the appropriate
    // texture unit. In this case, zero for GL_TEXTURE0
    // which was previously set by calling
    // gl.activeTexture().
    gl.uniform1i(textureSamplerLoc, 0 );
}
   // in init() function
   var vTexCoordLoc = gl.getAttribLocation( program,
                          "vTexCoord" );
    gl.enableVertexAttribArray( vTexCoordLoc );
    gl.vertexAttribPointer( vTexCoordLoc, 2, gl.FLOAT,
                              false, 0, 0);
```