CS 418: Interactive Computer Graphics

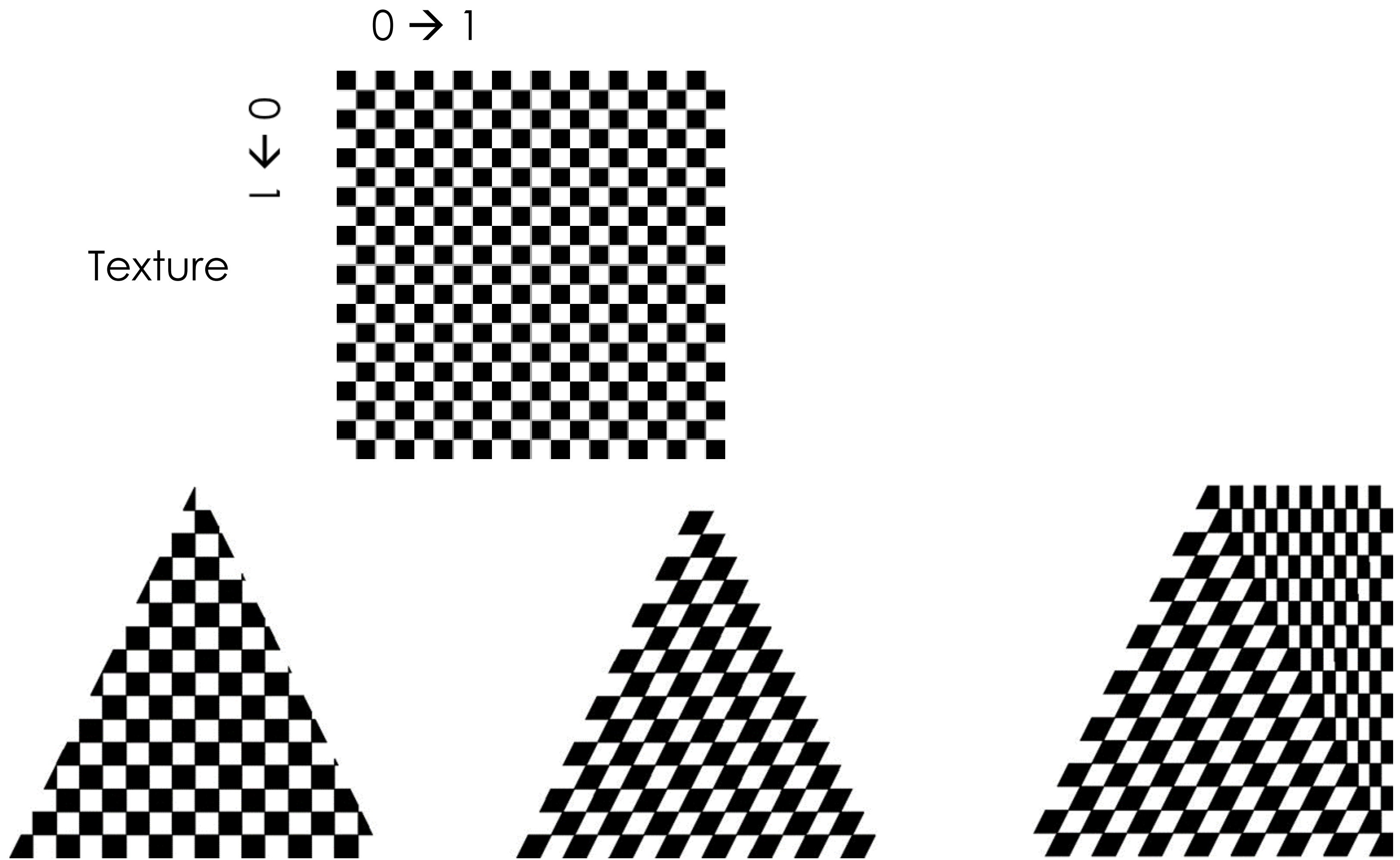
Texture Coordinates
Texture Filtering: Mipmapping

Eric Shaffer

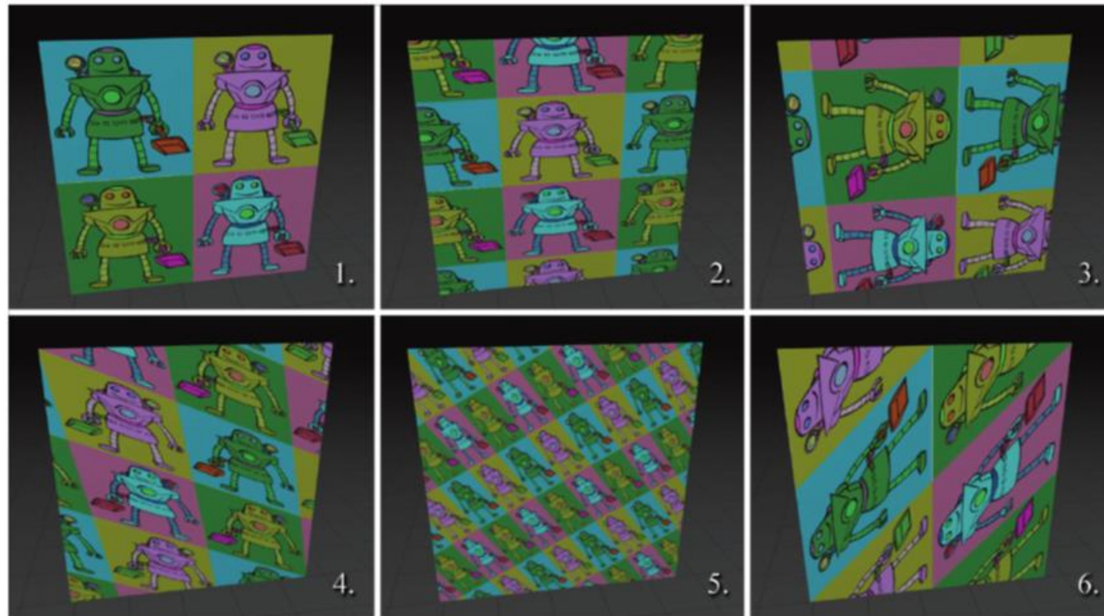
Reviewing Texture Coordinates

- We're using the following convention:
- (u,v) are the texture coordinates assigned in the parametric space with u and v in [0,1]
- (s,t) are the texel coordinates in a texture
- ….some people use (s,t) to denote the parametric coordinates…

Guess the Texture Coordinates



Match the Coordinates to the Image



(a)	$0:(0.20,-0.30)$	$1:(1.30,-0.30)$	$2:(1.30,1.20)$	$3:(0.20,1.20)$
(b)	$0:(5.00,-1.00)$	$1:(6.00,-1.00)$	$2:(6.00,0.00)$	$3:(5.00,0.00)$
(c)	$0:(1.00,0.00)$	$1:(-0.23,-0.77)$	$2:(0.00,1.00)$	$3:(1.24,1.77)$
(d)	$0:(2.00,0.00)$	$1:(1.00,1.00)$	$2:(0.00,1.00)$	$3:(1.00,0.00)$
(e)	$0:(-0.10,1.10)$	$1:(-0.10,0.10)$	$2:(0.90,0.10)$	$3:(0.90,1.10)$
(f)	$0:(0.00,-1.00)$	$1:(3.35,0.06)$	$2:(1.00,2.00)$	$3:(-2.36,0.94)$

Match each textured quad with the set of texture coordinates used to generate it given in the list below.

For the quad, the upper left vertex is number 0 and the vertices are enumerated clockwise around the quad.

For example

- The first image is axis aligned and doesn't repeat

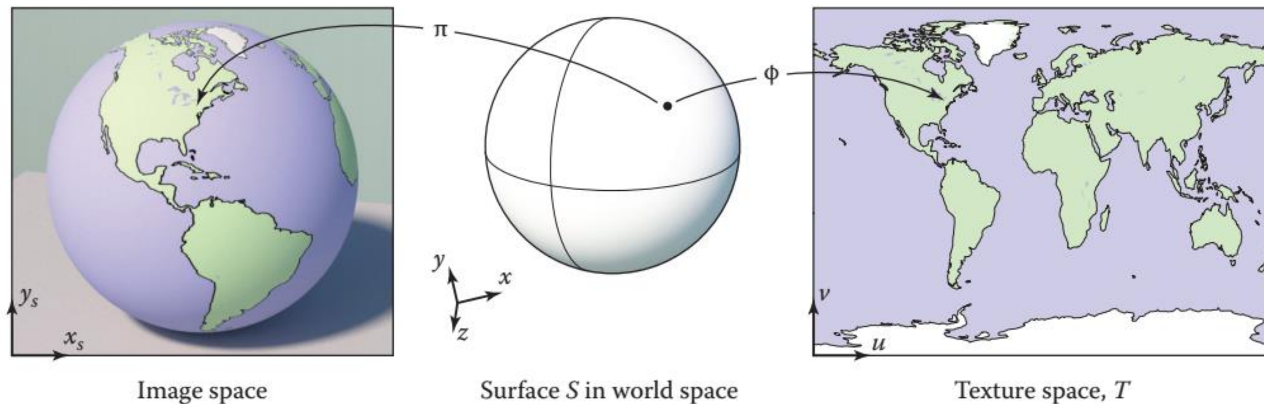
- Only possible coordinates from list are

(b) $0 : (5.00, -1.00)$ $1 : (6.00, -1.00)$ $2 : (6.00, 0.00)$ $3 : (5.00, 0.00)$

- Which give axis-aligned edges
- And parametric lengths of 1

Texture Coordinates

- We've only looked at simple mappings
  - Mapping the texture onto planar surfaces
- More complicated surfaces need more complicated mappings



Example: Mapping onto a sphere

Sphere on the right uses:

For any point P on the sphere, calculate d, that being the unit vector from P to the sphere's origin.

Assuming that the sphere's poles are aligned with the Y axis, UV coordinates in the range $[0, 1]$ can then be calculated as follows

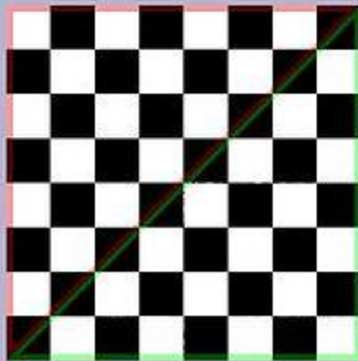
$$u = 0.5 + \frac{\arctan 2(d_z, d_x)}{2\pi}$$

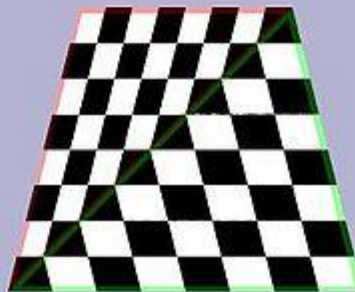
$$v = 0.5 - \frac{\arcsin(d_y)}{\pi}$$

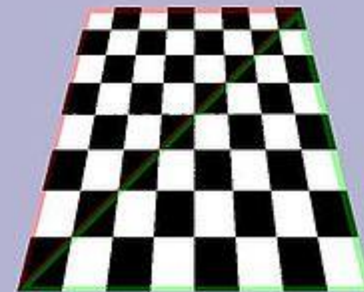
How is the sphere on the left being textured?

Perspective Correct Coordinates

- Linear (affine) interpolation problematic for texture coordinates
  - …and we need per-fragment coordinates
- Won't produce perspective correct texture coordinates


Flat


Affine


Correct

Perspective Correct Coordinates

Affine texture mapping directly interpolates a texture coordinate u_α between two endpoints u_0 and u_1:

$$u_\alpha = (1-\alpha)u_0 + \alpha u_1 \text{ where } 0 \le \alpha \le 1$$

Perspective correct mapping interpolates after dividing by depth z, then uses its interpolated reciprocal to recover the correct coordinate:

$$u_\alpha = \frac{(1-\alpha)\frac{u_0}{z_0} + \alpha\frac{u_1}{z_1}}{(1-\alpha)\frac{1}{z_0} + \alpha\frac{1}{z_1}}$$

-- Wikipedia

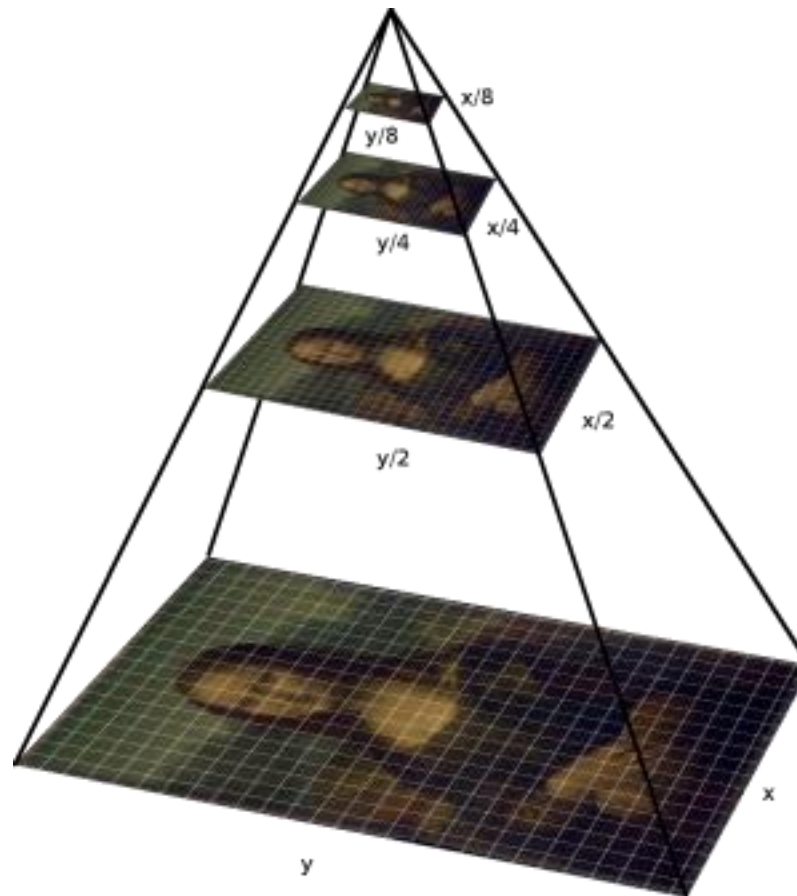
Filtering Textures

- Minification occurs when we have more texels than fragments
- Using NN or Bilinear Filtering can lead to aliasing
- Why?
  - Sparse sampling will can cause us to miss features
  - e.g. a checkerboard pattern could be turned into solid color
- What would a better strategy be?
  - Average all of the texels that map into a fragment
- What is the maximum number of texels fetched per fragment?
  - The entire texture

Mipmapping

- Mipmapping is a method of pre-filtering a texture for minification
  - History: 1983 Lance Williams introduced the word "mipmap" in his paper "Pyramidal Parametrics"
  - mip = "multum in parvo"…. latin: many things in small place(?)
- We generate a pyramid of textures
  - Bottom-level is the original texture
  - Each subsequent level reduces the resolution by ¼ (by ½ along s and t)

Mipmapping



Pre-filtered Image Versions

- Base texture image is say 256x256
  - Then down-sample 128x128, 64x64, 32x32, all the way down to 1x1

Trick: When sampling the texture, pick the mipmap level with the closest mapping of pixel to texel size

Why? Hardware wants to sample just a small (1 to 8) number of samples for every fetch—and want constant time access

Creating a Mipmap

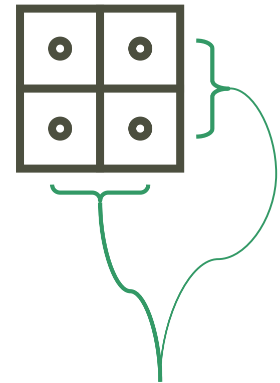
- In WebGL you can manually generate and upload a mipmap
- Or you can have WebGL generate it for you

```
gl.generateMipmap(GL_TEXTURE_2d)
```

- Usually, bilinear filtering is used to minify each level
- …but that's up to the implementation of the library

Mipmap Level-of-detail Selection

- Hardware uses 2x2 pixel entities
  - Typically called quad-pixels or just *quad*
  - Finite difference with neighbors to get change in u and v with respect to window space
    - Approximation to $\partial s/\partial x$, $\partial s/\partial y$, $\partial t/\partial x$, $\partial t/\partial y$
    - Means 4 subtractions per quad (1 per pixel)
- Now compute approximation to gradient length
  - $p = \max(\mathrm{sqrt}((\partial s/\partial x)^2+(\partial s/\partial y)^2), \mathrm{sqrt}((\partial t/\partial x)^2+(\partial t/\partial y)^2))$



Level-of-detail Bias and Clamping

- Convert p length to level-of-detail
  - $\lambda = \log_2(p)$
- Now clamp λ to valid LOD range
  - $\lambda' = \max(minLOD, \min(maxLOD, \lambda))$

Determine Mipmap Levels

- Determine lower and upper mipmap levels
  - b = floor(λ’)) is bottom mipmap level
  - t = floor(λ’+1) is top mipmap level
- Determine filter weight between levels
  - w = frac(λ’) is filter weight

WebGL Computing a Color from a Mipmap

WebGL offers 6 ways to generate a color from a mipmap

NEAREST = choose 1 pixel from the biggest mip

LINEAR = choose 4 pixels from the biggest mip and blend them

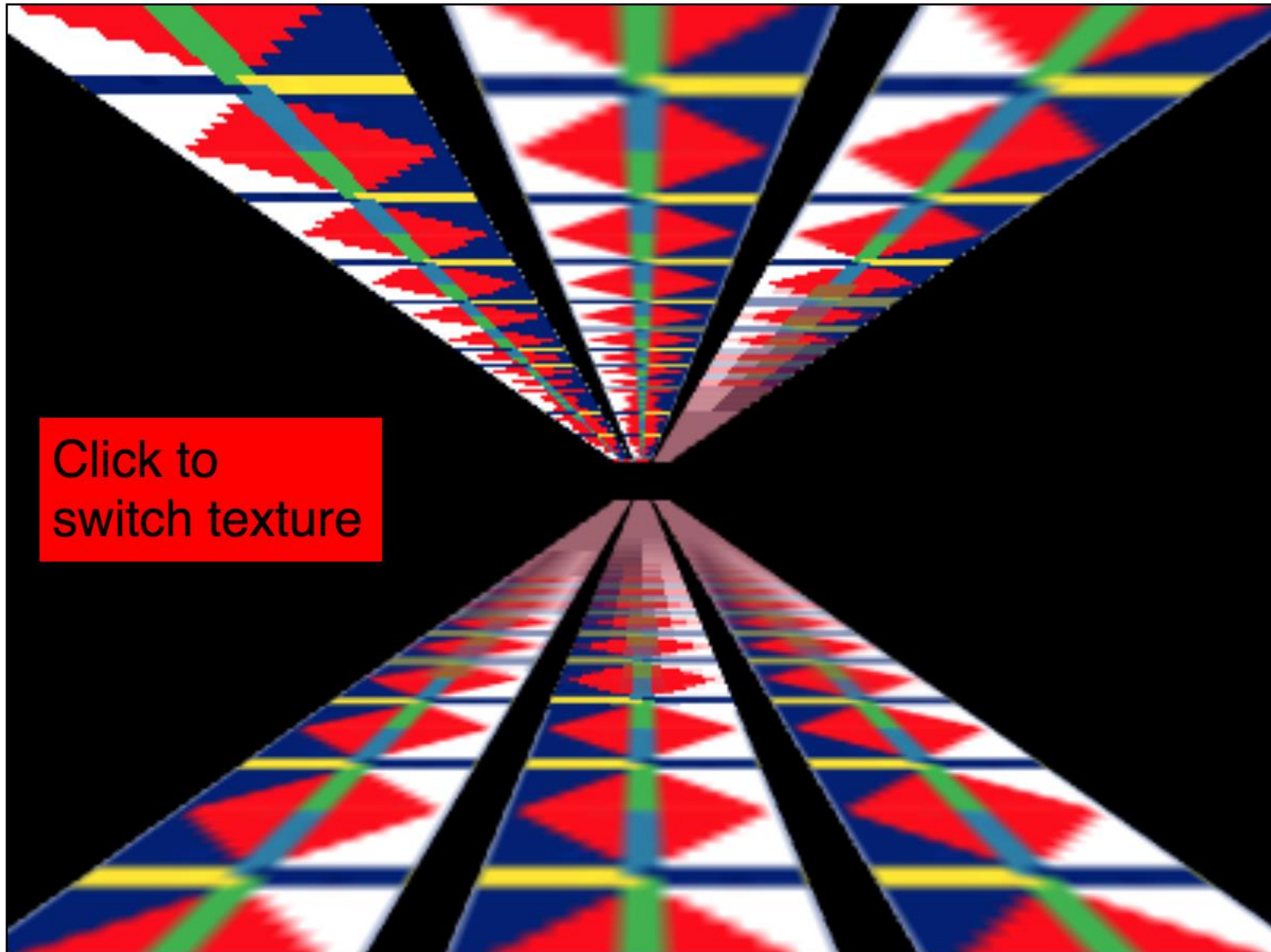
NEAREST_MIPMAP_NEAREST = choose the best mip,
then pick one pixel from that mip

LINEAR_MIPMAP_NEAREST = choose the best mip,
then blend 4 pixels from that mip

NEAREST_MIPMAP_LINEAR = choose the best 2 mips,
choose 1 pixel from each, blend them

LINEAR_MIPMAP_LINEAR = choose the best 2 mips.
choose 4 pixels from each, blend them

Mipmap Texture Filtering



WebGL: Highest Quality Filtering

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR_MIPMAP_LINEAR);

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

Although some WebGL implementations may now support anisotropic texture filtering…which is even better

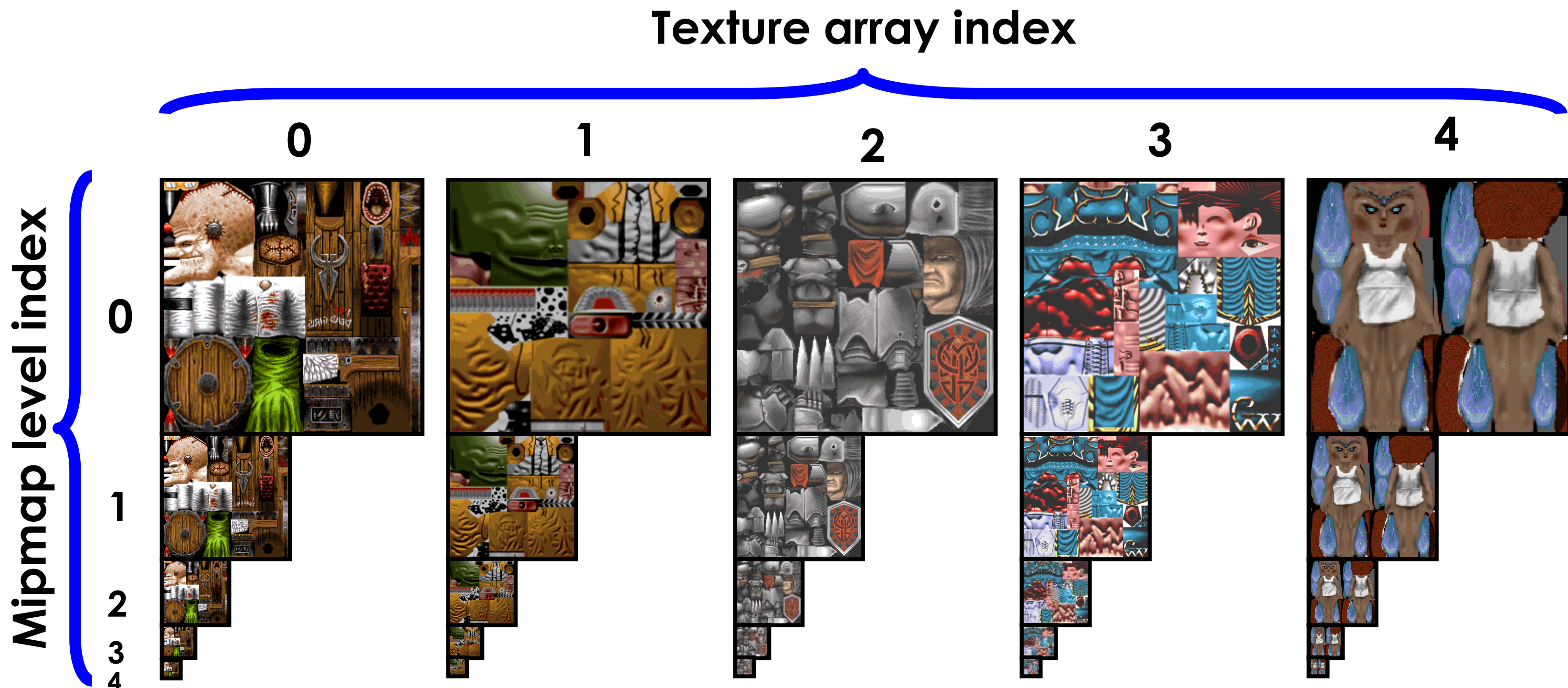
WebGL: Non-power of 2 textures

- You should use textures that are 2^k x 2^k

- You can use textures that are not powers of two
- but must
  - set the wrap mode to CLAMP_TO_EDGE
  - turn off mipmapping by setting filtering to LINEAR or NEAREST…

Texture Arrays

- **Multiple skins packed in texture array**
  - Motivation: binding to one multi-skin texture array avoids texture bind per object



Anisotropic Texture Filtering

- Standard (isotropic) mipmap LOD selection
  - Uses magnitude of texture coordinate gradient (not direction)
  - Tends to spread blurring at shallow viewing angles
- Anisotropic texture filtering considers gradient direction
  - Minimizes blurring

Isotropic

Anisotropic

Texturing in WebGL: Vertex Shader

Need to alter the vertex shader to pass-through texture coordinates

```
attribute vec4 a_position;
attribute vec2 a_texcoord;
uniform mat4 uMVmatrix;
uniform mat4 uPMatrix;
varying vec2 v_texcoord;

void main() {
  gl_Position = uPMatrix* uMVmatrix * a_position;
  // Pass the texcoord to the fragment shader.
  v_texcoord = a_texcoord;
}
```

Texturing in WebGL: Fragment Shader

Need to alter the fragment shader to fetch colors from textures

```
precision mediump float;

// Passed in from the vertex shader.
varying vec2 v_texcoord;

// The texture.
uniform sampler2D u_texture;

void main() {
   gl_FragColor = texture2D(u_texture, v_texcoord);
}
```