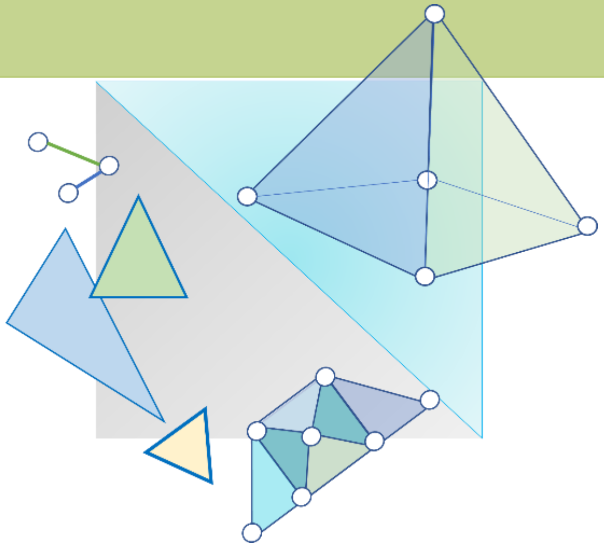
CITS3003 Graphics & Animation

Lecture 19:

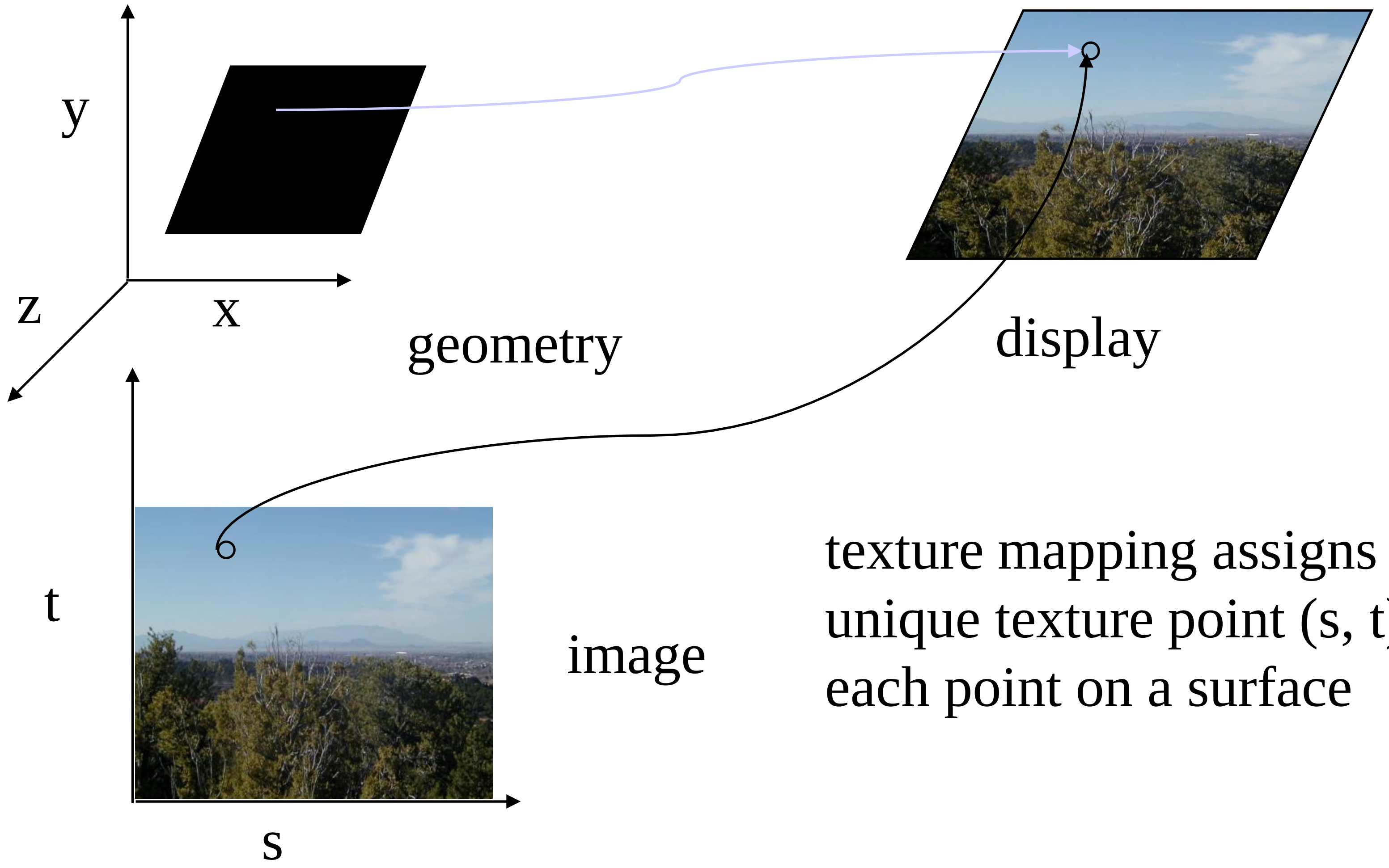
Texture Mapping in OpenGL



Objectives

- Basic strategy & OpenGL pipeline
- Introduce the OpenGL texture functions and options
- Texture parameters
- Texture mapping in real images

Texture Mapping



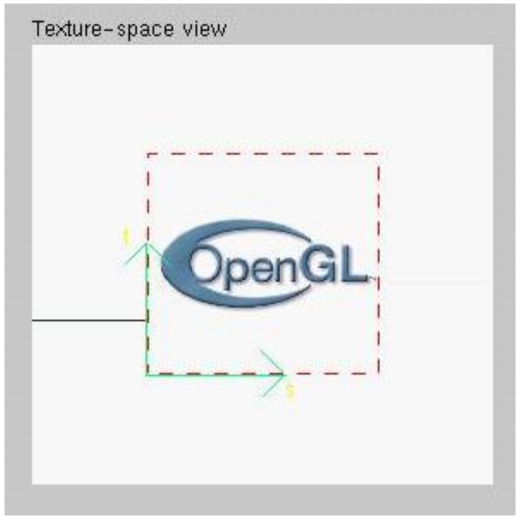
texture mapping assigns a unique texture point (s, t) to each point on a surface

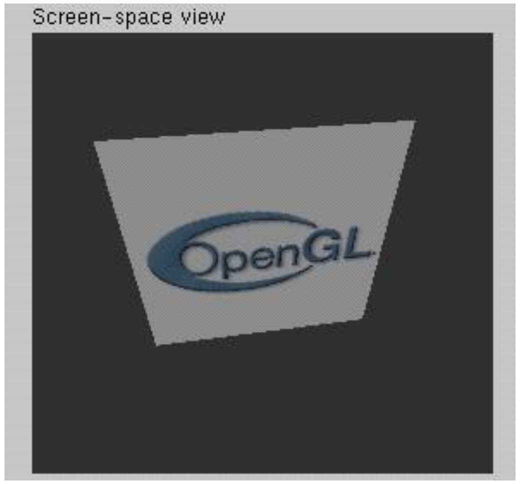
Texture Example

- This is a 256 x 256 image.

- It can be used as a texture map in OpenGL

- The texture has been mapped to a rectangular polygon which is viewed in perspective

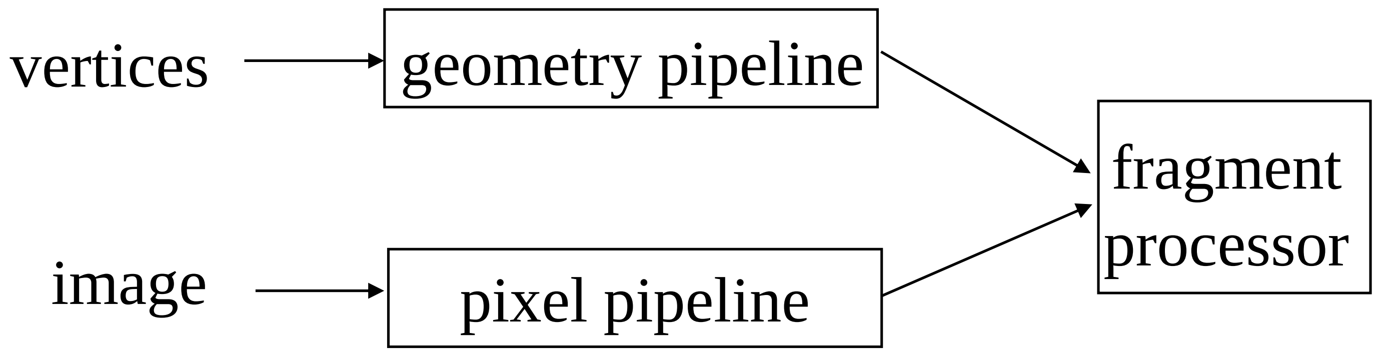




Texture Mapping and the OpenGL Pipeline

- Images and geometry flow through separate pipelines that join during fragment processing

- Hence, "complex" textures do not affect geometric complexity



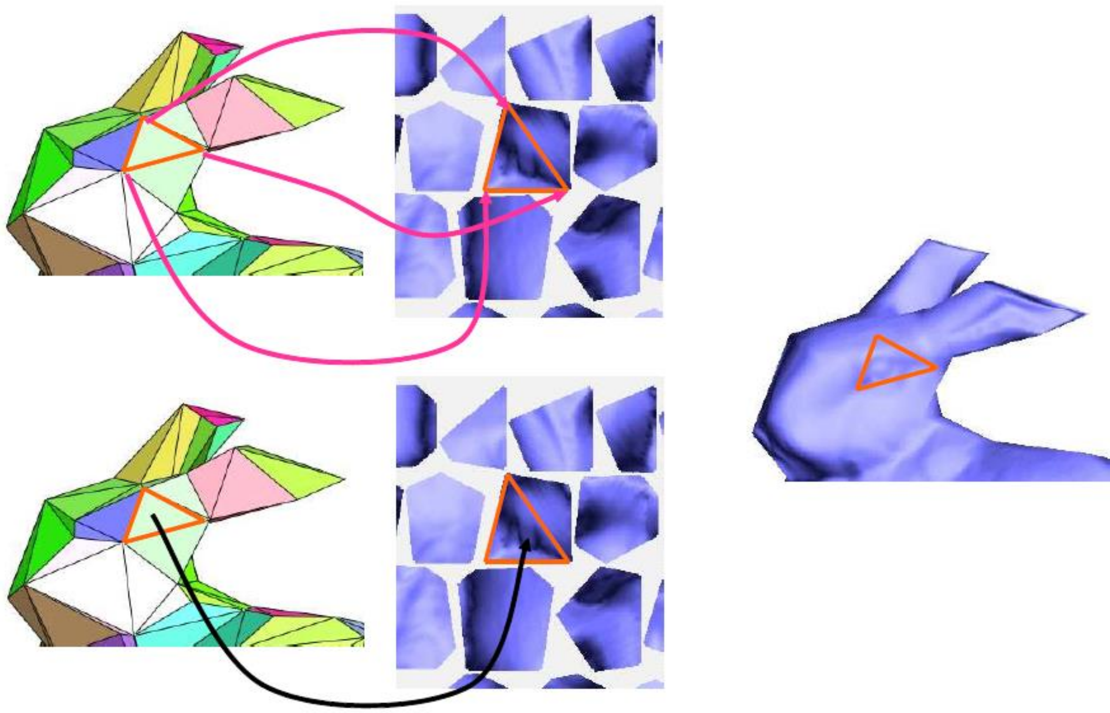
Texture Resolution vs Geometry Resolution

- We can have different resolutions for texture and geometry
- Texture processing is not as complex as geometry processing
- High resolution textures give more realistic appearance
- High resolution texture mapped on low resolution geometry still looks good while being light on the graphics pipeline
- Ground textures are perfect examples of this

A Simple Example of Texture Mapping

Each vertex of a triangle is given texture coordinates s,t , which "map" each vertex to some location in the texture.

The texture coordinates are then interpolated over the triangle giving us texture coordinates for each pixel.



Foundations of 3D Computer Graphics
S.J. Gortler

In the fragment shader, we can grab the color pointed to by the texture coordinates, and use it in our rendering (right).

Basic Steps to Apply Texture

1. Create texture Object
2. Specify the texture
   - read or generate image
   - enable texturing
3. Assign vertices/ object corners to texture coordinates
   - Proper mapping function is left to application
4. Specify texture parameters
   - wrapping, filtering
5. Pass textures to shaders
6. Apply textures in shaders

Step#1: Create Texture Object

- OpenGL has **texture objects** (multiple objects possible)
  - 1 object stores 1 texture image + texture parameters
  - First set up texture object

```
- Gluint mytex;
- glGenTextures(1, mytex); // Get texture identifier
- glBindTexture(GL_TEXTURE_2D, mytex); // Form new
  texture object
```

- Subsequent texture functions use this object

Step#2: Specifying a Texture Image

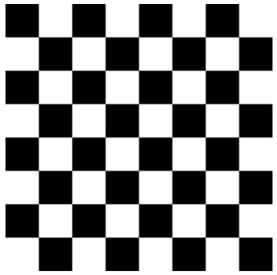
- Define a texture image as an array of *texels* (texture elements) in CPU memory

```
GLubyte my_texels[512][512][3];
```

- Read in Scanned image or camera image

or

- Generate pattern application program



- Enable texture mapping
  - `glEnable(GL_TEXTURE_2D)`
  - OpenGL supports 1-4 dimensional texture maps

Specify Image as a Texture

Let OpenGL know that the image is 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 , **type**:	the format and type of the texels
texels:	pointer to texel array

- Example:

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

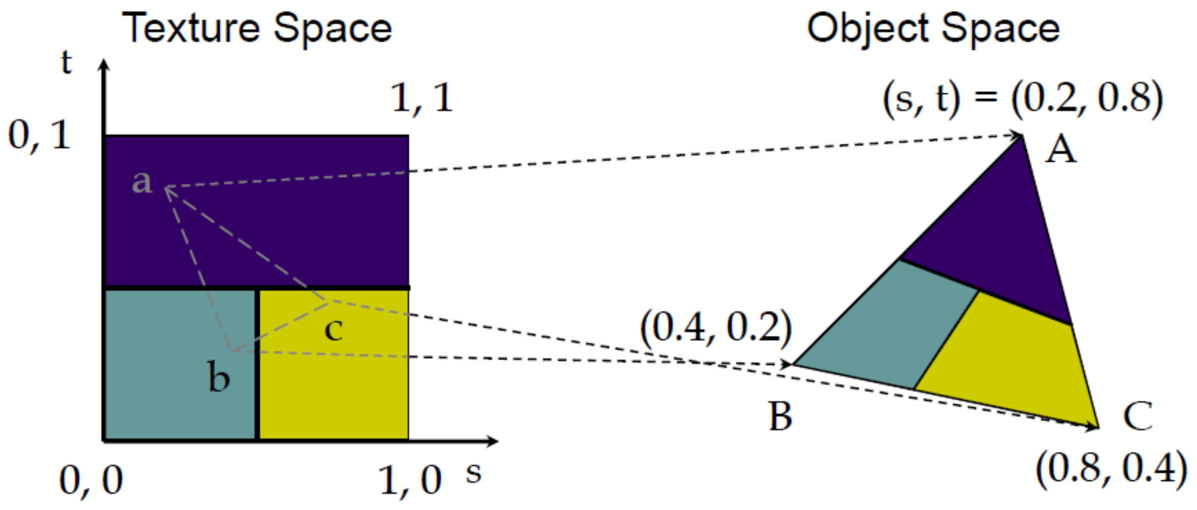
Basic Steps to Apply Texture

1. Create texture Object
2. Specify the texture
   - read or generate image
   - enable texturing
3. Assign vertices/ object corners to texture coordinates
   - Proper mapping function is left to application
4. Specify texture parameters
   - wrapping, filtering
5. Pass textures to shaders
6. Apply textures in shaders

Step#3: Assign Object Corners to Texture Corners

Each object corner (x,y,z) => texture coordinate (s, t)

Programmer establishes this mapping



https://www.opengl.org/archives/resources/code/samples/sig99/advanced99/notes/node52.html

Step#5: Passing Texture to Shaders

```
// pass the vertex coordinates to vertex shader
offset = 0;
GLuint vPosition = glGetAttribLocation(program,"vPosition");

glEnableVertexAttribArray( vPosition );

glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE,0,
                           BUFFER_OFFSET(offset) );


// piggy-back the texture coordinates at the
// end of the buffer and pass it to vertex shader
offset += sizeof(points);

GLuint vTexCoord = glGetAttribLocation(program,"vTexCoord");

glEnableVertexAttribArray( vTexCoord );

glVertexAttribPointer( vTexCoord, 2, GL_FLOAT, GL_FALSE, 0,
                                   BUFFER_OFFSET(offset) );
```

Variable names in shader

Step#6: Apply Texture in Shaders (Vertex Shader)

- 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 texture coordinate to be
                                          //interpolated

texCoord = vTexCoord
color = vColor
gl_Position = modelview * projection * vPosition
```

Step#6: Apply Texture in Shaders (Fragment Shader)

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

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

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

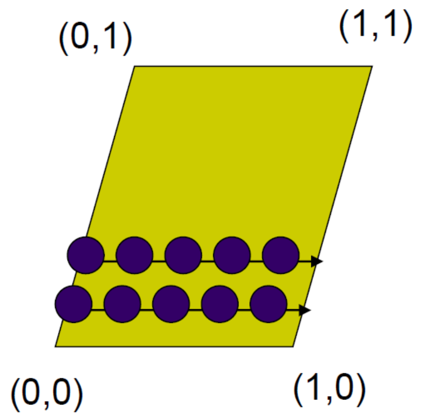
Output color
Of fragment

Original color
of object

Lookup color of
texCoord(s,t) in
texture

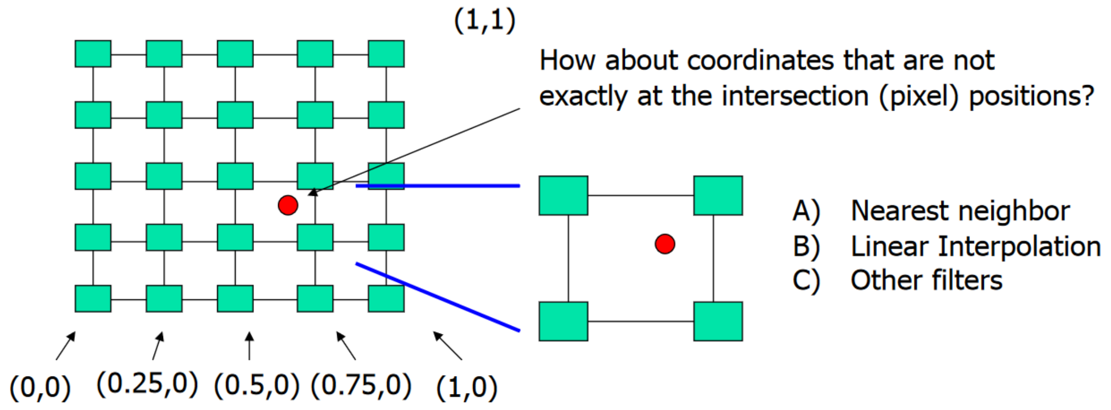
Mapping Textures to Surfaces

- Texture Mapping is performed in rasterization



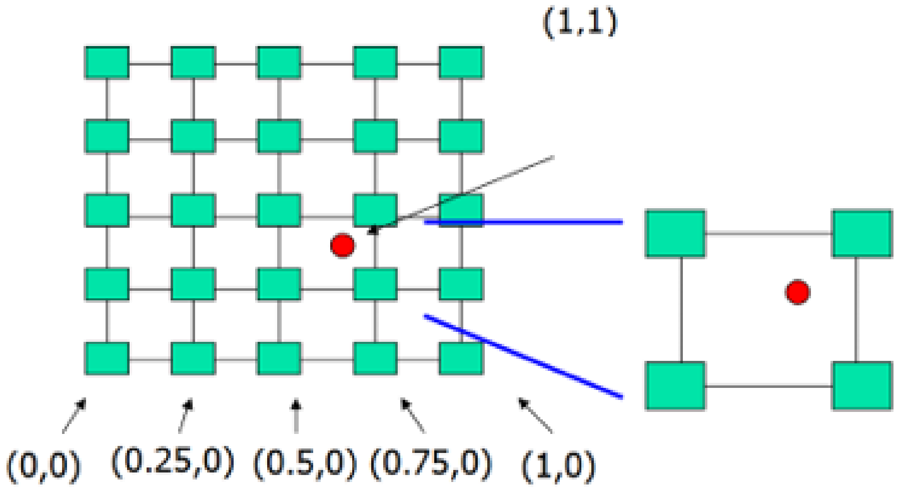
- For each fragment, its texture coordinates (s,t) are interpolated based on corners/vertices' texture coordinates
- The interpolated texture coordinates (s,t) are then used to perform texture lookup

Texture Value Lookup:
For the given texture coordinates (s,t), we can find a unique image value from the texture map.



Texture Sampling

- Aliasing in textures is a major problem. When we map texture coordinates to the texels array, we rarely get a point that is exactly at the center of the texel.



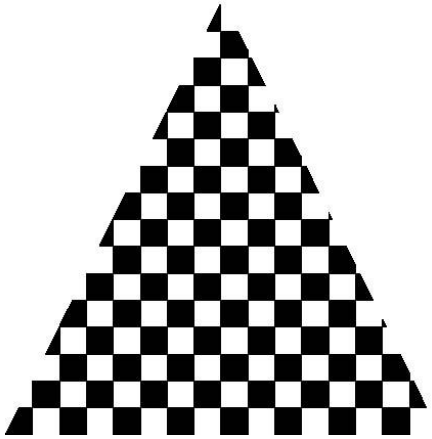
OpenGL supports the following options for sampling textures:

1. **Point sampling** – use the value of the texel that is closest to the texture coordinates output by the rasterizer
2. **Linear filtering** – use the weighted average of a group of texels in neighbourhood of the texture coordinates output by the rasterizer

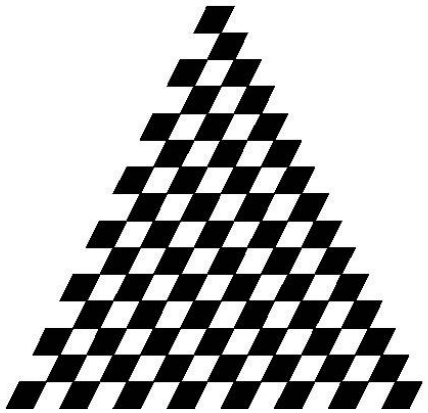
Interpolation

- OpenGL uses interpolation to find proper texels from specified texture coordinates. Distortions may result.

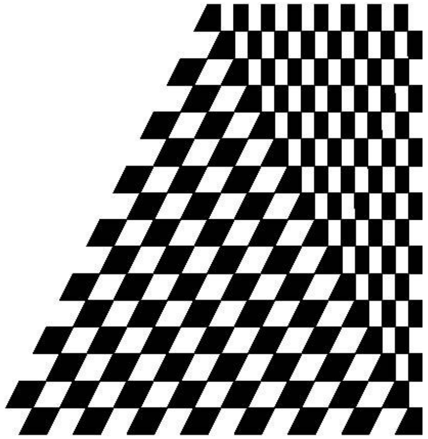
good selection
of tex coordinates



poor selection
of tex coordinates



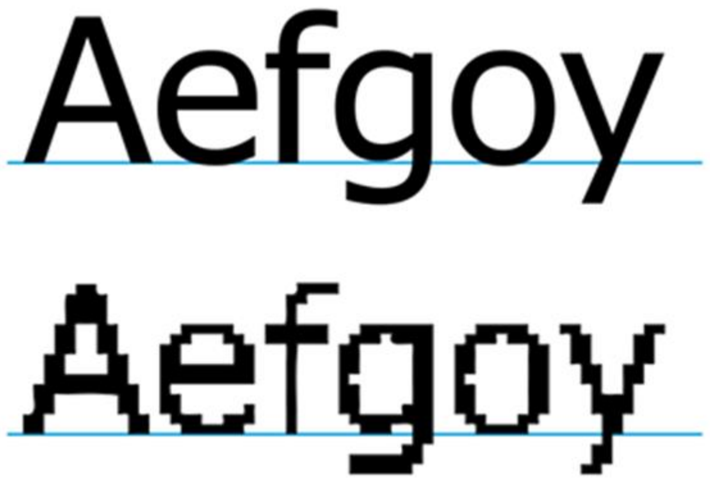
texture stretched
over trapezoid
showing effects of
bilinear interpolation

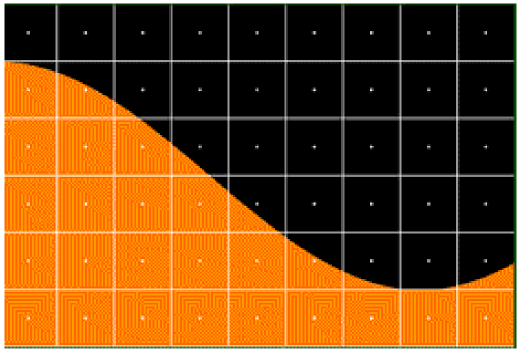


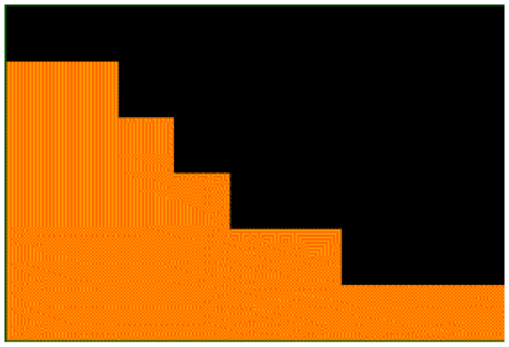
Dealing with Aliasing

Point sampling of texture can lead to aliasing errors

In computer graphics, aliasing is the stair-stepped appearance of smooth curves and lines when there are not enough pixels in the image or on screen to represent them realistically. Also called "stair-stepping" and "jaggies."

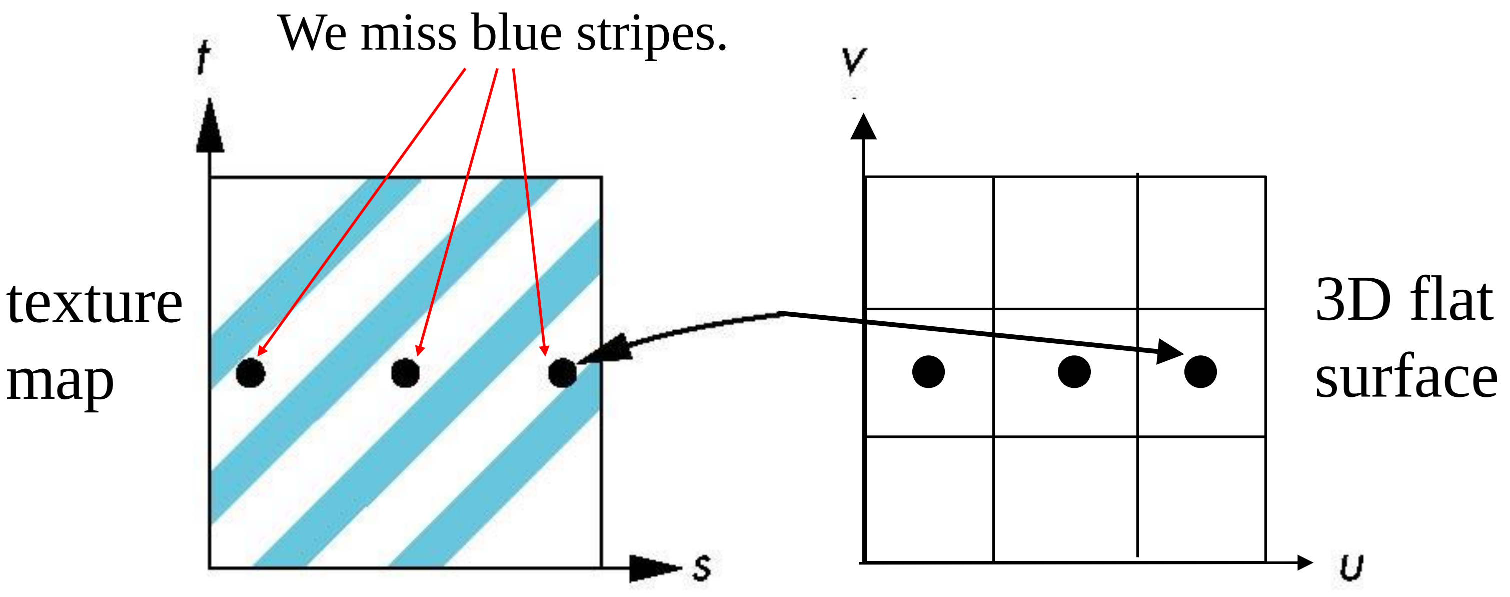






Dealing with Aliasing

Point sampling of texture can lead to aliasing errors



point samples in texture space.

Basic Steps to Apply Texture

1. Create texture Object
2. Specify the texture
   - read or generate image
   - enable texturing
3. Assign vertices/ object corners to texture coordinates
   - Proper mapping function is left to application
4. Specify texture parameters
   - wrapping, filtering
5. Pass textures to shaders
6. Apply textures in shaders

Step#4: Specify 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
  - Texture sampling mode allows us to specify using area averaging instead of point samples
  - Mipmapping allows us to use textures at multiple resolutions
  - Environment parameters determine how texture mapping interacts with shading

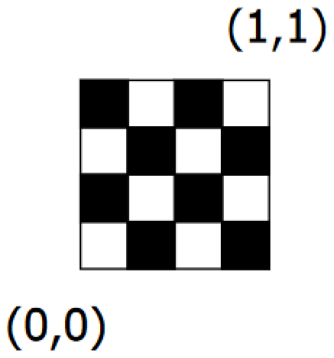
- glTexParameter*(GLenum target, GLenum pname, GLint value);// * can be i or f

- glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);

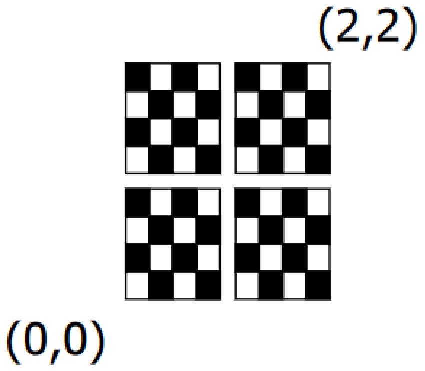
Wrapping Mode

- Want s and t in the range $0 \cdots 1$. Can use clamping or repeat to force them in the $[0,1]$ range.
  - **Clamping:** if $s, t > 1$ use 1, if $s, t < 0$ use 0
  - **Repeat**: use $s, t > 1$ then modulo 1

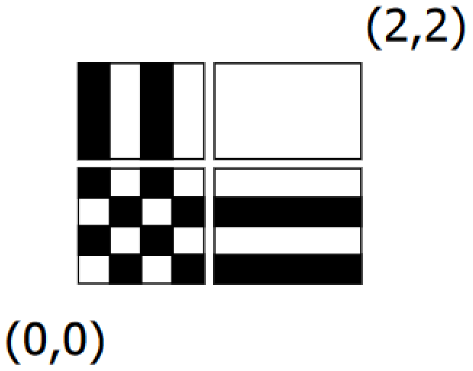
```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
```



texture

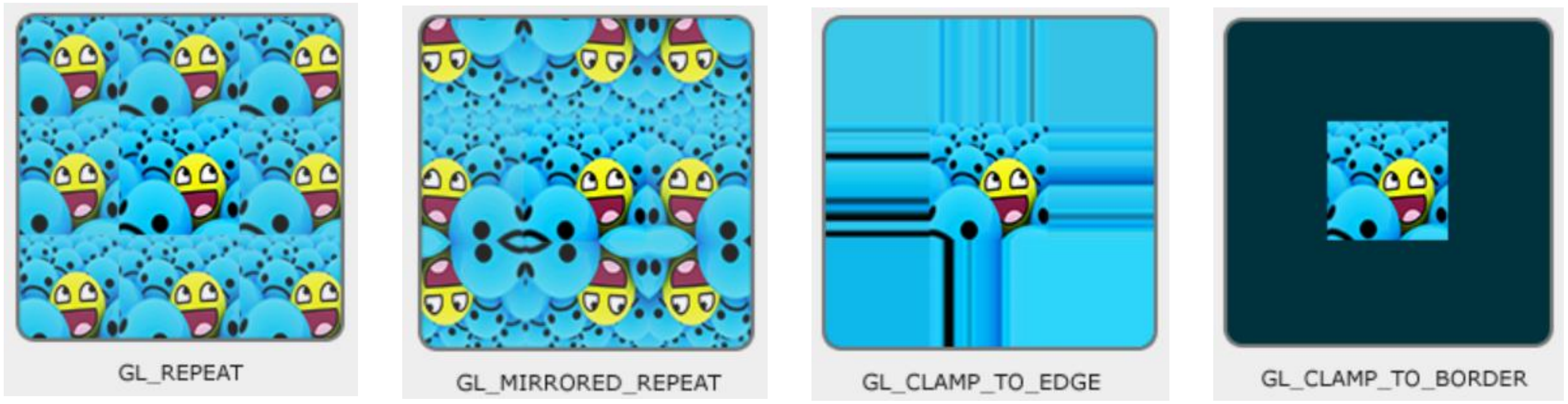


GL_REPEAT



GL_CLAMP

Wrapping Mode



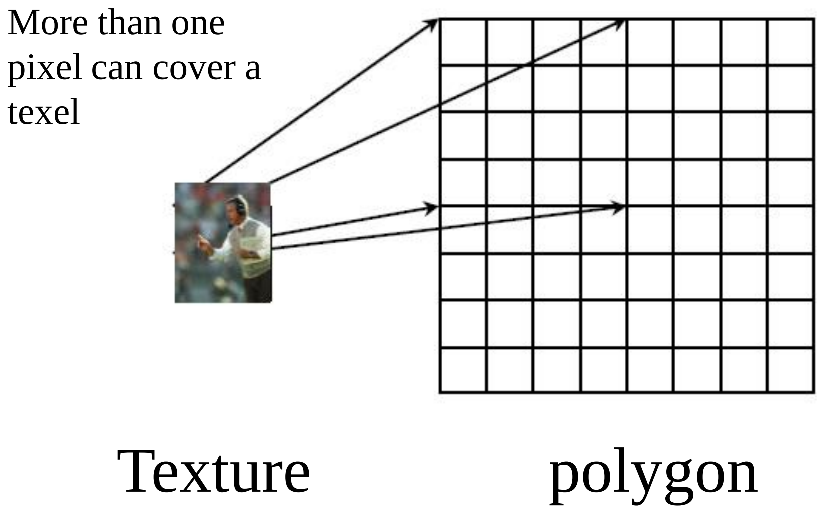
```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT)
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT)
```

https://learnopengl.com/Getting-started/Textures (source)

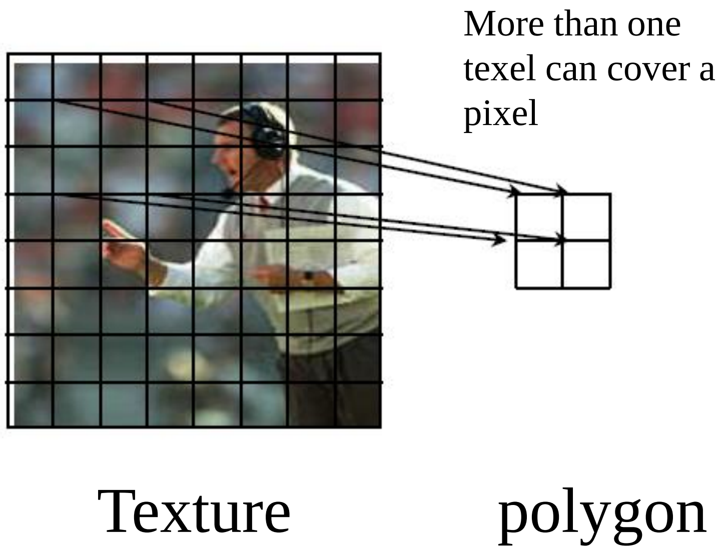
Magnification and Minification

In deciding how to use the texel values to obtain the final texture value, the size of the pixel on the screen may be larger or smaller than the texel:

- **Magnification:** Stretch small texture to fill many pixels
- **Minification:** Shrink large texture to fit few pixels



Magnification



Minification

Magnification and Minification (Cont.)

The filter mode can be specified by calling:

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

Magnification:

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

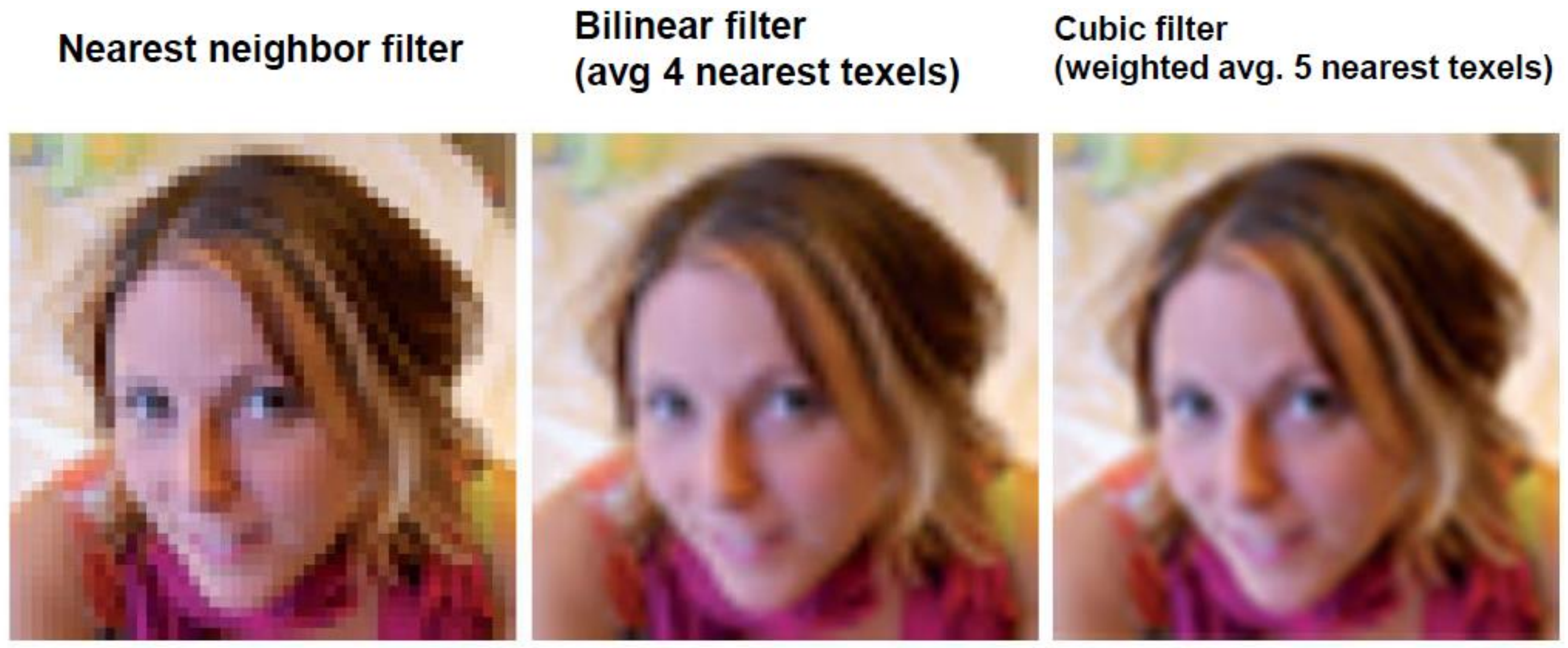
Minification:

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

For point sampling, replace GL_LINEAR by GL_NEAREST

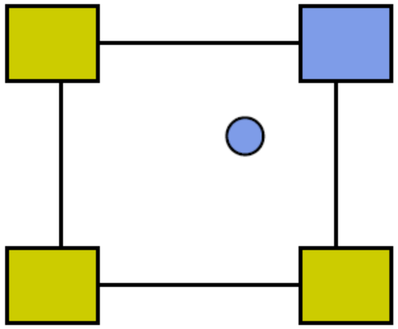
Example: Texture Magnification

48 x 48 image projected (stretched) onto 320 x 320 pixels



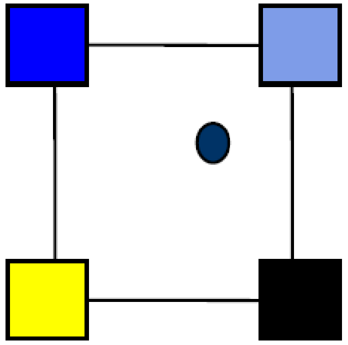
Texture Mapping Parameters

Nearest Neighbor
(lower image quality)



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

Linear Interpolate the
Neighbors(better quality,
slower)



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

Mipmapping

For objects that are project to a smaller area on the screen, we don’t need to keep the original full resolution of the texel array.

OpenGL allows us to create a series of texure arrays at reduced sizes, e.g., for a 64×64 original array, we can set up $64 \times 64, 32 \times 32, 16 \times 16, 8 \times 8, 4 \times 4, 2 \times 2$, and 1×1 arrays by calling:

```
glGenerateMipmap(GL_TEXTURE_2D);
```

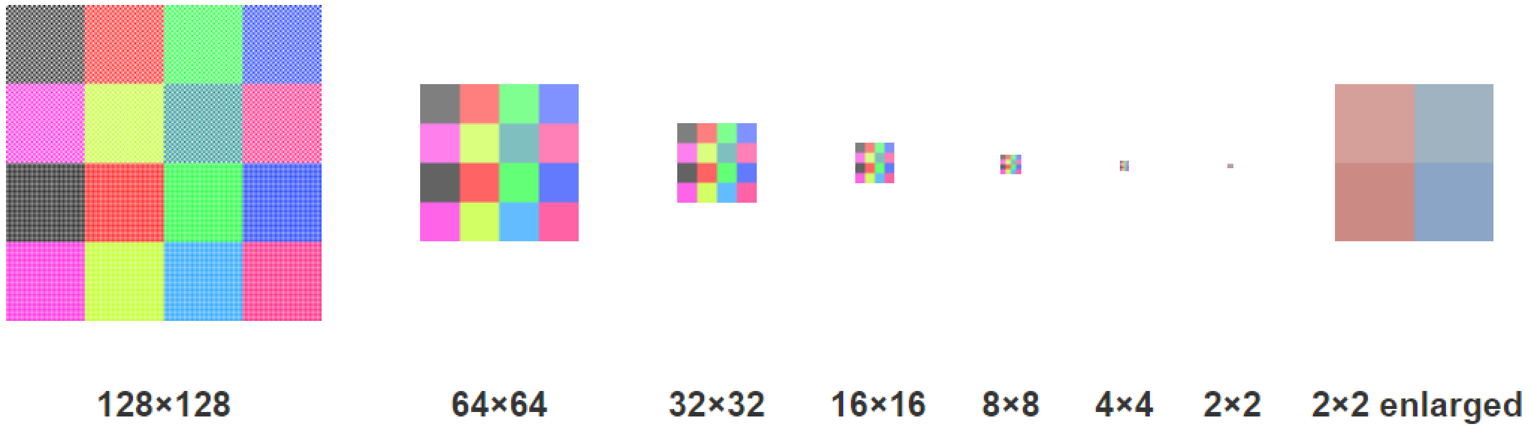


Image source

Mipmapping

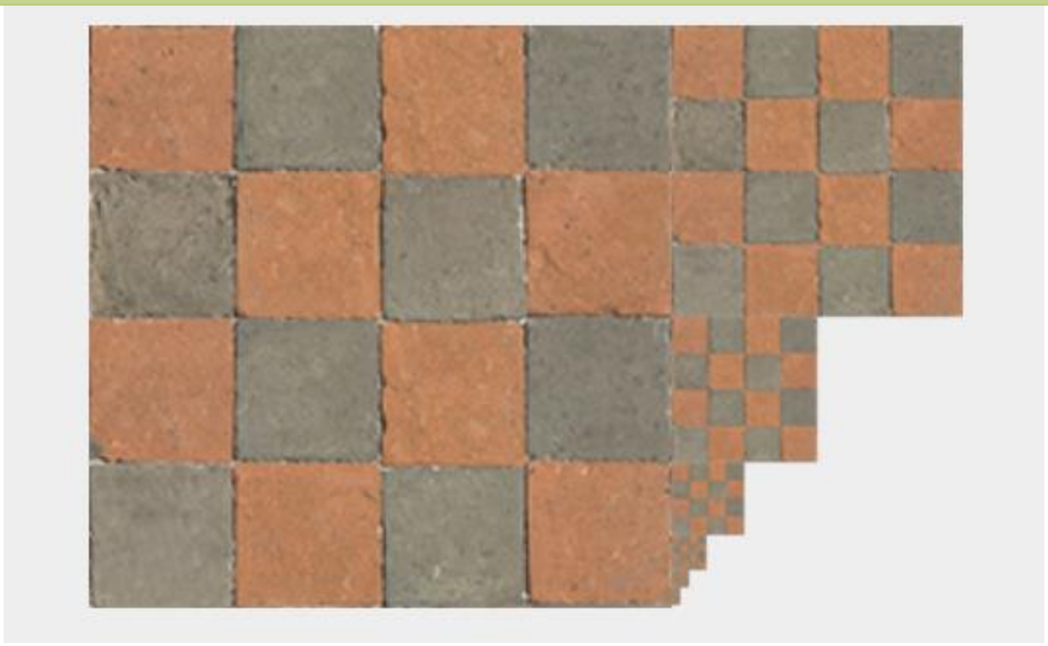
- **Mipmapping** (sometimes called **MIP mapping**) is a technique where an original high-resolution texture map is scaled and filtered into multiple resolutions within the texture file
- *Mipmapping* allows for prefiltered texture maps of decreasing resolutions
- You need to firstly declare the mipmap level during texture definition (by calling, e.g., glTexImage2D). Thus, the order of function calls is (e.g.):

```
glTexImage2D( GL_TEXTURE_2D, 0, … );

glGenerateMipmap(GL_TEXTURE_2D);
```

Indicates that we want to keep all resolutions (all the way to base level 0, i.e. the original texel array).

Mipmapping



•GL_NEAREST_MIPMAP_NEAREST: takes the nearest mipmap to match the pixel size and uses nearest neighbor interpolation for texture sampling.
•GL_LINEAR_MIPMAP_NEAREST: takes the nearest mipmap level and samples that level using linear interpolation.
•GL_NEAREST_MIPMAP_LINEAR: linearly interpolates between the two mipmaps that most closely match the size of a pixel and samples the interpolated level via nearest neighbor interpolation.
•GL_LINEAR_MIPMAP_LINEAR: linearly interpolates between the two closest mipmaps and samples the interpolated level via linear interpolation

https://learnopengl.com/Getting-started/Textures (source)

Put it all Together

```
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 );
```

Partial code from **example1.cpp** in the **CHAPTER07_CODE** folder

Checkerboard Texture

We can create our own texture map in the application.
For example, creating a checkerboard texture:

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

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

Partial code from **example1.cpp** in the **CHAPTER07_CODE** folder

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
}
```

Partial code from **example1.cpp** in the **CHAPTER07_CODE** folder

Texture Mapping Real Images

- If the texture to be mapped is a real image, it already has shading and shadows
  - Due to directional light sources
  - Due to self occlusions and occlusions from other objects

- These shading and shadows will not correspond to the light sources in your graphics world

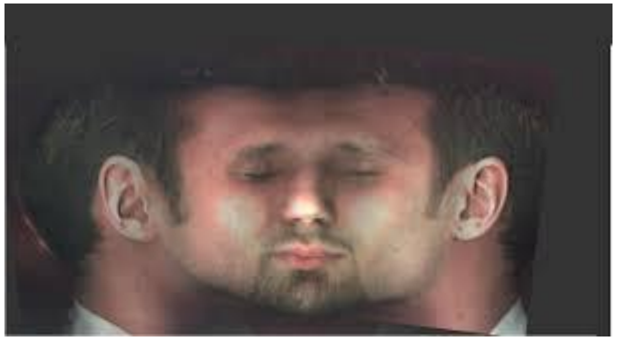
- A good texture map should not include shading or shadows
  - Such an image can only be captured in controlled conditions with perfect diffused lighting i.e. light coming from all directions
  - Multiple large planar light sources can approximate diffused light

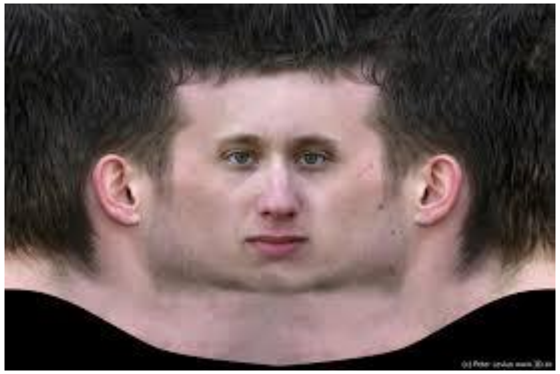
2D Texture on 3D Objects

- Wrapping 2D texture onto a 3D object is problematic, especially if the 3D shape is complex

- Unless the texture was on the 3D object already in the real world and the object was scanned

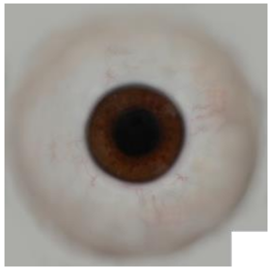
- And/Or some effort was put into unwrapping the texture





Texture Files

If we generate human with clothes option, we get texture files for the skin, hair, eyes, eyelashes, eyebrow, clothes, shoes, etc.



Further Reading

“Interactive Computer Graphics – A Top-Down Approach with Shader-Based OpenGL” by Edward Angel and Dave Shreiner, 6th Ed, 2012

- Sec. 7.6 *Texture Mapping in OpenGL* (including all the subsections)
- For the aliasing problem, see Pages 370-371.

References:

- Foundations of 3D Computer Graphics Steven Gortler
- Prof. Emmanuel Agu, WPI, CS 543 Computer Graphics, Fall Semester 2019