# Johns Hopkins
# Engineering for Professionals

# 605.767 Applied Computer Graphics

Brian Russin

# Module 9B
# Texture Review Part 2 - OpenGL

# OpenGL Texture Mapping

- OpenGL (and its predecessor GL) were the first widely used graphics API to support texture mapping
  - OpenGL has extensive support for texture mapping
- OpenGL Programming Guide states: texture mapping is a "fairly involved process"
- Texture mapping is both memory and computationally intensive
  - Likely why it has only recently become a widespread graphics technique
  - Successful graphics systems today provide hardware support for texture mapping
- Many good Web references
  - **Advanced OpenGL Texture Mapping** by Nate Miller
  - https://www.flipcode.com/archives/Advanced_OpenGL_Texture_Mapping.shtml

# Steps in OpenGL Texture Mapping

- Specify the texture
  - Can consist of 1-4 elements per texel up to an (R,G,B, alpha) quadruple
  - Supports MipMapping - specifying a texture at many resolutions
- Indicate how the texture is to be applied to each pixel
  - Decal - texture is 'painted' on top of the pixel
  - Modulate - scales the pixel color
    - Useful for combining effects of lighting with texturing
    - Often use white reflection coefficients for underlying material
      - Creates a light intensity which then is multiplied with the texture color
  - Blend - constant color is blended with the pixel value
- Enable texture mapping
- Draw the scene: supply texture and geometric coordinates
  - Provides the mapping of texture coordinates into object coordinates

# Texture Specification

```
void glTexImage2D(GLenum target, GLint level, GLint internalFormat,
                  GLsizei width, GLsizei height, GLint border,
                  GLenum format, GLenum type, const GLvoid* pixels)
```

Defines a two-dimensional texture. Target is set to `GL_TEXTURE_2D`. Level is used to specify multiple resolutions of the texture map (MipMapping). Internal format describes how the texture data is stored internally. Width and height specify the texture dimensions. Format and type specify the format of the pixel data supplied. Pixels supplies the texture image.

- Width and height
  - Form must be 2m+2b (where b is the border width)
  - Maximum size is implementation specific (256x256 is common)
  - Minimum size is 64x64
- Border width
  - Can be used for tiling together texture maps to get a map larger than the maximum
  - Format and type specify the format of the pixel data supplied

# Format of the Image Data

- Format of the incoming image data can be:
  - GL_COLOR_INDEX
  - GL_RGB
  - GL_RGBA
  - GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA
  - GL_LUMINANCE, GL_LUMINANCE_ALPHA
- Type is the data type used
  - GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT, GL_FLOAT, or GL_BITMAP

# Internal Format for Texture Storage

- Internal format indicates which components are selected for use in describing the texels
  - R, G, B,  or intensity
- OpenGL 1.0 uses the following:
  - GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB, GL_RGBA
- OpenGL 1.1, 38 additional internal pixel formats can be specified
  - e.g., GL_RGB, GL_RGBA, GL_LUMINANCE, GL_RGB4, GL_R3_G3_B2
  - Some require less storage, but more computation
    - RGBA is often the most efficient (but uses the most storage)
    - GL_RGBA8 – 32 bytes per texel
    - GL_R3_G3_B2 – 8 bytes per texel
  - Implementations may not support a particular one – will map to nearest format it supports

# Reading Image Files

- Recommend using an open source image library
  - https://www.khronos.org/opengl/wiki/Image_Libraries
  - STB Image (https://github.com/nothings/stb)
    - Cross-platform, header-based
    - Does not load directly into OpenGL
  - SOIL – Simple OpenGL Image Library
  - DevIL (http://openil.sourceforge.net/)
    - Mimics OpenGL calls
  - GLI – OpenGL Image (http://www.g-truc.net/project-0024.html)
  - FreeImage – does not load to OpenGL

# Steps to Use STB to Get Image Data

- Include base image library
  - #include “stb/stb_image.h”
- Load an image, get metadata and image data: stbi_load
  - unsigned char *stbi_load(char const *filename, int *x, int *y, int *channels_in_file, int desired_channels);
  - Desired channel options
    - STBI_grey - 1 channel
    - STBI_grey_alpha - 2 channels
    - STBI_rgb - 3 channels
    - STBI_rgb_alpha - 4 channels
  - The returned value is the image data loaded into memory
  - *x, *y, *channels_in_file return the width (x), height (y), and number of channels
  - Use stbi_info to get information from the image without loading
  - Use stbi_failure_reason() to get failure information

# Example of loading image data with STB

```
struct ImageData
{
    int             w = 0;
    int             h = 0;
    int             channels = 0;
    unsigned char *data = nullptr;
};

ImageData im_data;
im_data.data = stbi_load(file_path.c_str(),
                         &im_data.w,
                         &im_data.h,
                         &im_data.channels,
                         STBI_rgb_alpha);
```

# Other Methods for Use with Texture Images

- Texture proxy
  - Place holder for a texture
    - Useful for determining if OpenGL can accommodate a texture of specific size and internal format
  - Use GL_PROXY_TEXTURE_2D in glTexImage2D
  - Use **glGetTexLevelParameteriv** to test
- Replacing all or part of a texture image
  - **glTexSubImage2D**
  - Often used with video sources
    - Create a texture, replace texture data with new video images
- One dimensional textures
  - Example application: contouring
    - OpenGL Red Book Example: texgen.c
  - **glTexImage1D, glTexSubImage1D, glCopyTexImage1D,** and **glCopyTexSubImage1D**
- Controlling perspective correction
  - **glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST)**

# Texture Objects

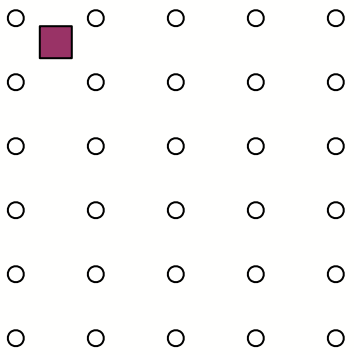
- Texture object stores texture data and makes it readily available
  - Performance improvements achieved as applications can bind or reuse an existing texture rather than reloading it
- **glGenTextures(GLsizei n, GLuint *textureNames)**
  - Retrieves a set of n texture names (integer IDs)
- **glIsTexture(GLuint textureName)**
  - Returns true if the texture has been bound and not subsequently deleted
- **glBindTexture(GLenum target, GLuint textureName)**
  - First time for a valid name: creates the texture object
  - Previously created name: the texture object becomes active
  - 0: stops using texture objects and returns to the default texture
- **glDeleteTextures(GLsizei n, GLuint *textureNames)**
  - Deletes the specified texture objects

# OpenGL Mip Mapping

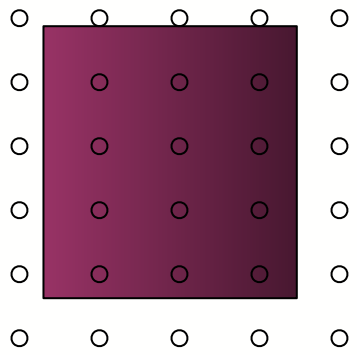
- OpenGL supports mip-mapping
  - Mip-mapping requires some extra computation
  - If not used textures mapped onto smaller objects might shimmer as object or view is moved
  - **glGenerateMipmap(GL_TEXTURE_2D)**
  - Some systems, especially older systems, require image dimensions to be a power of 2
- To take effect the application must select an appropriate filtering method using **glTexParameter**

# Magnification vs. Minification

- A single pixel on the screen can map to:
  - A portion of a texel (magnification)
  - Numerous texels (minification)
    - Depending on transformations and texture mapping applied



**Magnification**
Pixel maps to a portion of a texel
Many pixels can map to the same texel



**Minification**
Pixel maps to several texels

# Controlling Filtering

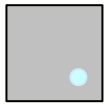
- OpenGL provides several options for filtering, averaging, or interpolating texels
  - Trade-offs between image quality and speed
- Use **glTexParameter**
  - GL_TEXTURE_MAG_FILTER
    - Magnification: many pixel fragment values map to one texel
      - GL_NEAREST or GL_LINEAR
    - Always uses the largest texture map even if mip-mapping is provided
  - GL_TEXTURE_MIN_FILTER
    - Minification: many texel values map to one pixel fragment
      - GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR

Example:

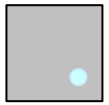
```
glTexParameter(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER, GL_LINEAR)
```

# Filtering (cont.)

- GL_NEAREST
  - Texel with coordinates nearest the center of the pixel is used
    - Produces severe aliasing if a pixel actually maps to many texels
  - Round-off to nearest texel
- GL_LINEAR
  - Weighted linear average of the 2x2 texel array nearest the pixel center is used
    - Bilinear filtering
    - Provides smoother results with a bit more computation
    - If near the edge of the texture map some complications arise depending on whether a border is present and whether GL_REPEAT or GL_CLAMP is in effect



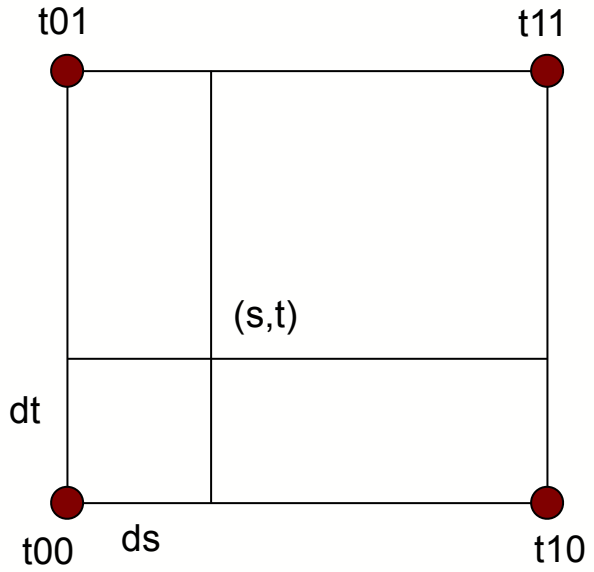
GL_NEAREST: Maps to the texel at the lower right



GL_LINEAR: uses a weighted linear average of the 4 nearest texels

# Bilinear Filtering

- Uses a weighted average of the 4 nearest texels
  - w00 = (1.0f - ds) * (1.0f - dt);
  - w01 = (1.0f - ds) * dt;
  - w10 = ds * (1.0f - dt);
  - w11 = ds * dt;
  - ds,dt: percentage of s,t between texels



# MipMap Filtering Options

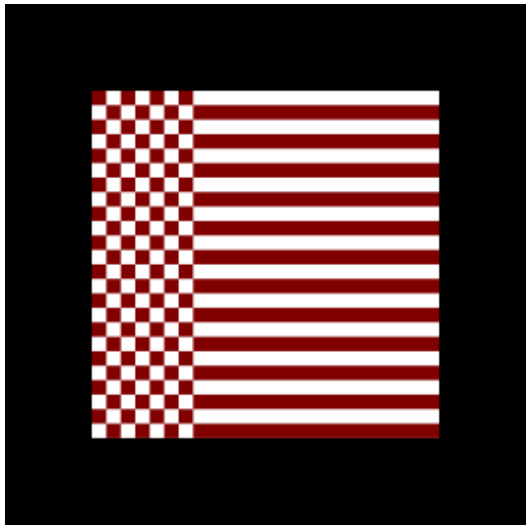
- Mip map filtering options
  - GL_NEAREST_MIPMAP_NEAREST
    - Chooses nearest texel in an individual mip map
      - Best choice of mip map
  - GL_LINEAR_MIPMAP_NEAREST
    - Interpolates a 2x2 array of texels in the nearest individual mip map
  - GL_NEAREST_MIPMAP_LINEAR or GL_LINEAR_MIPMAP_LINEAR
    - Interpolate texel values from the two nearest and best choices of mip maps
    - GL_LINEAR_MIPMAP_LINEAR will provide the best results of any filtering but is the most computation
      - Known as trilinear mipmapping or trilinear texture filtering
      - Bilinear filtering for two nearest MipMaps
      - Linear filtering between two results of bilinear filtering
        - w1 = lambda - (int)lambda;
        - w0 = 1.0f - w1;
- Texture **level of detail (LOD) bias** may be set to bias the computed lambda parameter used in texturing for mipmap level of detail selection
  - Provides a means to blur or sharpen textures.
  - Available in OpenGL 1.4 and above

# Repeating or Clamping Textures

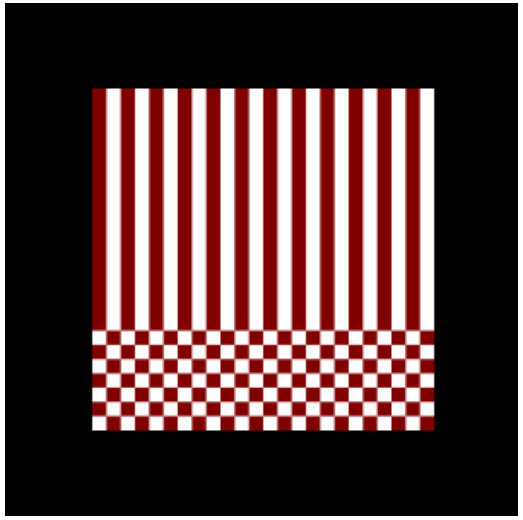
- Application can assign texture coordinates outside [0,1] and specify whether they are to be clamped or repeated
  - Repeating textures - the integer part of the coordinate is ignored - thus repeating the texture image
    - Want the image texels at the top edge to match the bottom, also left match right edge
    - Useful for repeating a pattern like a brick wall
    - Watch using negative coordinates (wrap at 0 problem)
  - Clamping textures: any values less than 0.0 are set to 0.0 and any values greater than 1.0 are set to 1.0
    - Useful when you want a single copy of the texture to appear on a large surface
    - Picture on a wall
- Use **glTexParameter**()
  - GL_TEXTURE_WRAP_S or GL_TEXTURE_WRAP_T
  - With GL_CLAMP or GL_REPEAT

# Clamping vs. Repeating

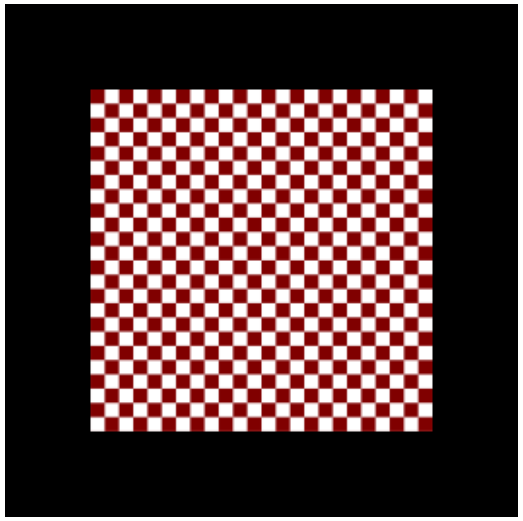
Wrap S : GL_CLAMP
Wrap T : GL_CLAMP



Wrap S : GL_CLAMP
Wrap T : GL_REPEAT



Wrap S : GL_REPEAT
Wrap T : GL_CLAMP



Wrap S : GL_REPEAT
Wrap T : GL_REPEAT

# Texture Mapping with GLSL

- To allow use of a texture by a shader your application should:
  - Select a specific texture unit and make it active
    - **glActiveTexture(GLenum texture)**
      - GL_TEXTUREi (GL_TEXTURE0 is initial value)
  - Create a texture object and bind it to the active texture unit
    - **glBindTexture**
  - Set texture parameters e.g., wrapping, filtering
    - **glTexParameter**
  - Define the texture
    - **glTexImage**
- glEnable and glTexEnv (for modulate vs. decal) are not required
  - Texture function is expressed within shader code

# Texture Within Vertex Shaders

- Incoming texture coordinate is one of the per vertex attributes
- Vertex shader passes the texture coordinate to the fragment shader
  - Use perspective correct interpolation (smooth qualifier)

```
#version 150
in vec2 st;
in vec3 vertexPosition;
smooth out vec2 texCoord;
uniform mat4 pvm;
void main()  {
    texCoord = st;
    gl_Position = pvm * vec4(vertexPosition,
1.0);
}
```

# Texture Within Fragment Shaders

- Uses interpolated texture coord within a **texture function** that uses the texture supplied as a uniform sampler
  - **Samplers** are uniform variable types used within a shader to access a texture
  - Built-in texture functions allow texture access within a shader

```
#version 150
smooth in vec2 texCoord;
uniform sampler2D MyTexture;
out vec4 fragColor;
void main()  {
   vec4 color = texture(MyTexture, texCoord);
   fragColor = color;
}
```

# GLSL Samplers

- Samplers are uniform variable types used within a shader to access a texture
  - Application must provide a value for the sampler prior to its execution
  - Pass the texture unit where texture is bound to OpenGL
    - Get the sampler uniform location
      - tex = glGetUniformLocation(programObj, “MyTexture”);
    - Bind the texture to texture unit i
    - Pass i as an integer by glUniform.
      - glUniform1i(tex, i);
        - i is the texture unit – not the texture object!
      - Examples:
        - glUniform1i(tex, 0); // correct
        - glUniform1i(tex, my_tex); // incorrect! (i.e. glGenTextures(1, &my_tex))
        - glUniform1i(tex, GL_TEXTURE0); // incorrect!

# Built-in Texture Functions

- Built-in functions perform texture access within a shader
  - **texture(sampler, tex_coord)**
    - overloaded function for various sampler types
  - 1st argument is a uniform variable sampler
    - sampler1D, sampler2D, sampler3D, etc.
  - 2nd argument is a texture coordinate
    - Generally the interpolated variable passed from the vertex shader
    - must match the corresponding sampler type

```
vec4 color = texture(MyTexture, texCoord);
```

# Modulate vs. Decal

- Fragment shader determines how texture color affects the fragment color
  - Many possibilities!
- Decal - texture color becomes the fragment color
- Modulate – texture color is modulated by lighting
  - Note: material properties should have equal RGB intensities
    - Texture applies the “color” and lighting modulates the intensity

```
cf = lightingColor;              // Compute lighting / intensity
ct = texture(tex, texCoord);     // Color from texture
gl_FragColor = vec4(ct * cf);
```