Texture Mapping in OpenGL

Computer Graphics
Instructor: Sungkil Lee

Texture Mapping Revisited

Textures?

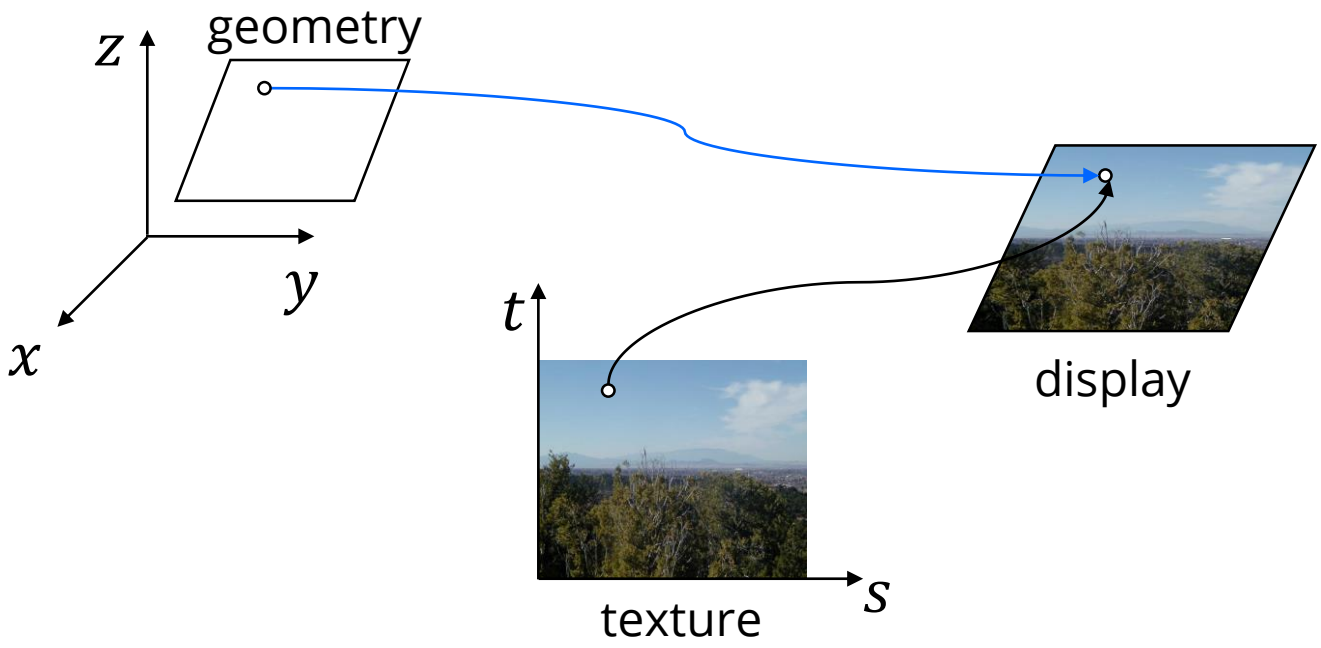
- **A “texture” is a repeated pattern in some spaces.**
  - In physical space, the texture indicates the repeated ***image*** patterns.
  - In temporal space, it refers to the repeated sound patterns.

- **In CG, textures usually refer to images in GPU memory.**
  - Initially, they meant repeated spatial patterns in the geometric surfaces.
  - Nowadays, they simply mean images in GPU memory.
    - e.g., texture memory dedicated to GPU

Texture Mapping

- **Definition of texture mapping:**
  - a technique of defining surface materials (especially shading parameters) in such a way that vary as a function of position on the surface.

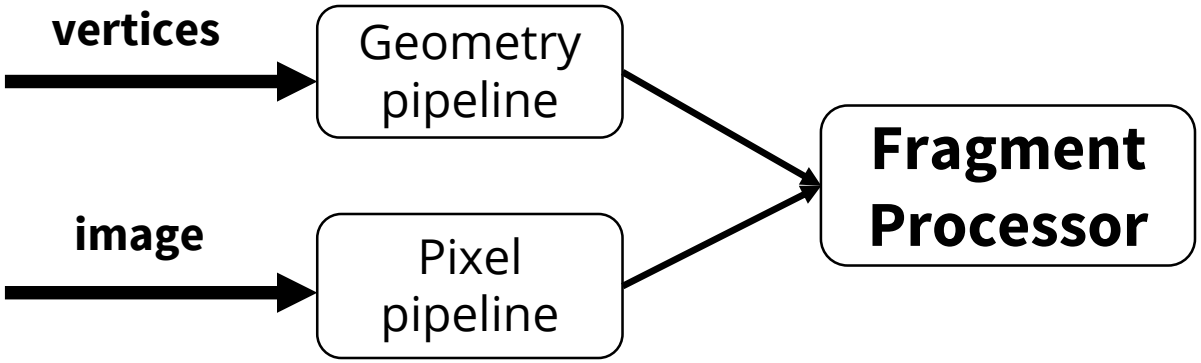
- **Very simple in comparison to geometric modeling.**
  - However, it produces complex-looking effects at reduced cost.



Texture Mapping in OpenGL

Pixel Pipeline in OpenGL

- **Images and geometry flow through separate pipelines that join during fragment processing**
  - Texture pipeline is opaque, which is not visible in programming.
  - Complex textures do not affect geometric complexity.



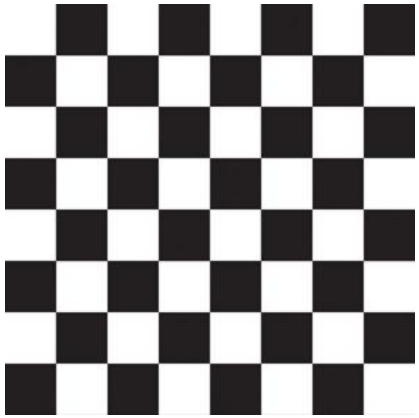
Texture Mapping in Three Steps

- **Texture specification**
  - Read or generate an image
  - Enable texturing
  - Generate a texture object
- **Assign texture coordinates to vertices**
  - Proper mapping function is left to application
  - Texture coordinates provided for many models
- **Texture parameter specification**
  - Wrapping modes, filtering, mipmapping, ...

Generating Texture Image

- **Define a texture image from an array of texels in memory:**
  - You may generate an image by application code.
  - Example: checkerboard generation

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

// create a 64x64 checker-board pattern
for( int y=0; y < 64; y++ )
for( int x=0; x < 64; x++ )
{
     GLubyte c = (((y&0x8)==0)^((x&0x8)==0))*255;
     image[y][x][0]  = c;
     image[y][x][1]  = c;
     image[y][x][2]  = c;
}
```



Reading Texture Image

- **Loading an external image (e.g., jpeg, png)**
  - http://nothings.org/
  - Include <stb_image.h>, and use "*#define STB_IMAGE_IMPLEMENTATION*" to enable implementation.

```
extern "C"
{
   uchar* stbi_load(const char* file, int* x,int* y,int* comp,int req_comp);
   void stbi_image_free( void* retval_from_stbi_load );
}

int width;      // output width of the loaded image
int height;     // output height of the loaded image
int comp;       // output number of channels of the loaded image
uchar* image = stbi_load( "lena.png", &width, &height, &comp, 0 );
```

Reading Texture Image

- **Vertical flipping may be required**

```
...

// flip image vertically
image = (uchar*) malloc( sizeof(uchar)*width*height*comp );
for( int y=0, stride=width*comp; y < height; y++ )
   memcpy( image+(height-1-y)*stride, image0+y*stride, stride );

...
```

- **cg_load_image() in cgut.h**
  - image* cg_load_image(const char* image_path);
  - abstracts all the necessary handling, including 4-byte boundary alignment and vertical flipping

Enable Texturing

- **OpenGL supports 1-3 dimensional texture targets**
  - `GL_TEXTURE_1D, GL_TEXTURE_1D_ARRAY, GL_TEXTURE_2D, GL_TEXTURE_2D_ARRAY, GL_TEXTURE_3D, ...`
  - A 2D texture is the most common choice.

- **Enable texture mapping**

```
glEnable( GL_TEXTURE_2D );    // enable texturing for 2D
```

Texture Object Generation

- **Generate and bind a texture object as usual.**

```
GLuint texture_object;
glGenTextures( 1, &texture_object );
glBindTexture( GL_TEXTURE_2D, texture_object );
```

Setting up Texture Object Data

```
glTexImage2D( target, level, internalFormat, width, height, border, format, type, texels );
```

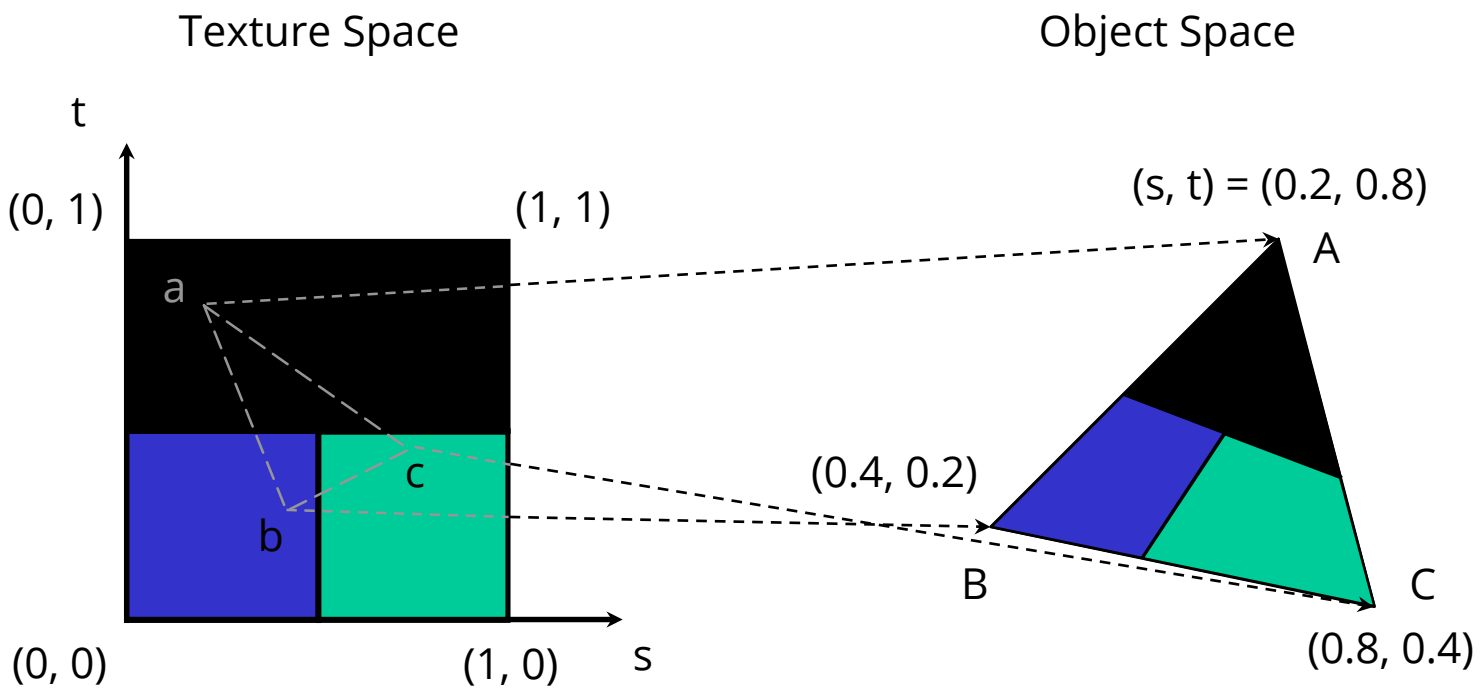
- target: type of texture, e.g. GL_TEXTURE_2D
- level: mipmap level (default: 0, discussed later)
- internalFormat: internal data format (usually, GL_RGB8 or GL_RGB)
- width, height: width and height of texels in the image
- border: used for smoothing (use 0)
- format: `GL_RGB` for 3-component images
- type: `GL_UNSIGNED_BYTE` for 8-bit images
- texels: pointer to the texel array

- **Example**
  - 512x512 size, 3 components (RGB), memory of each component is 1 byte

```
glTexImage2D( GL_TEXTURE_2D,0,GL_RGB8,512,512,0,GL_RGB,GL_UNSIGNED_BYTE,my_texels);
```

Mapping a Texture

- **Based on parametric texture coordinates**
  - We are using backward mapping: given (x,y,z), derive (s,t).



Typical Example

- **One more thing to do before draw() function:**
  - Connecting the texture object to the uniform variable

```
void render()
{
     glUniform1i( glGetUniformLocation( program, "TEX"), 0 );

     ...
}
```

Vertex Shader: texture.vert

- **Do similarly, but pass the texcoord to the fragment shader.**

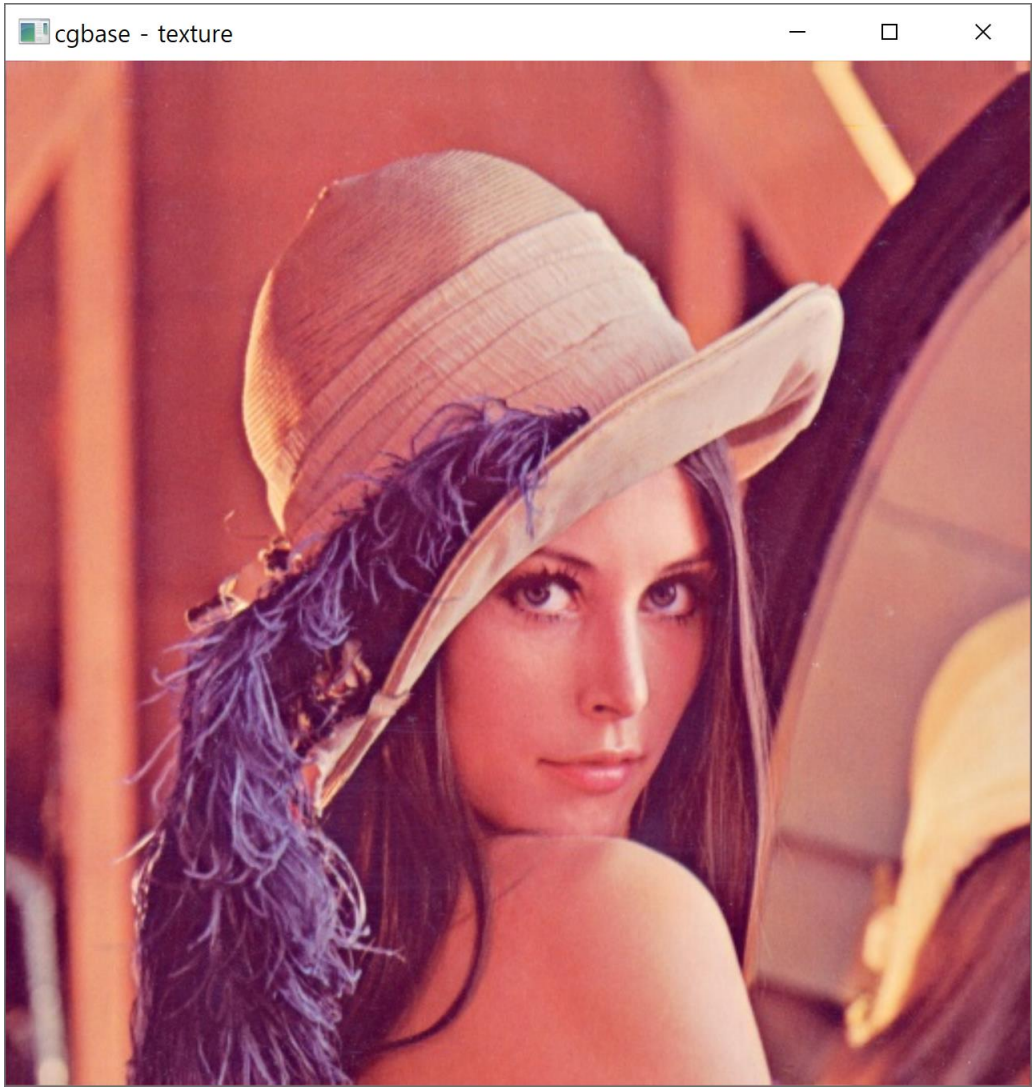
```
...

out vec2 tc;

void main()
{
    ...

    // pass texture coordinate to fragment shader
    tc = texcoord;
}
```

Fragment Shader: texture.frag

- **Fetch the color from the texture sampler.**

```
in vec2 tc;
out vec4 fragColor;

uniform sampler2D TEX; // texture sampler object

void main()
{
    fragColor = texture( TEX, tc );
}
```

Example



Multiple Textures

- **Texture slots are defined more than one.**
  - Many systems support at least 16 textures at the same time.
  - In render(), specify on which slot you are working.
  - Set the value of the uniforms as the slot.

```
void render()
{
     glActiveTexture( GL_TEXTURE0 );
     glBindTexture( GL_TEXTURE_2D, tex0 );
     glSetUniform1i( glGetUniformLocation("TEX0"), 0 );

     glActiveTexture( GL_TEXTURE1 );
     glBindTexture( GL_TEXTURE_2D, tex1 );
     glSetUniform1i( glGetUniformLocation("TEX1"), 1 );

     …
}
```

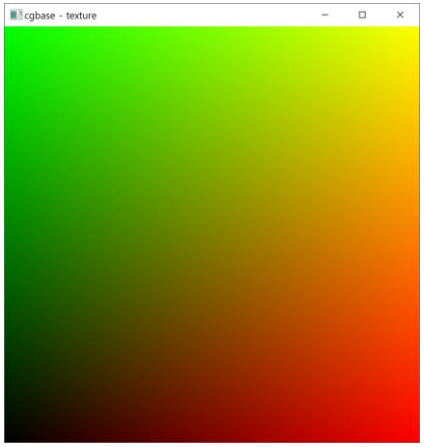
Fragment Shader: texture.frag

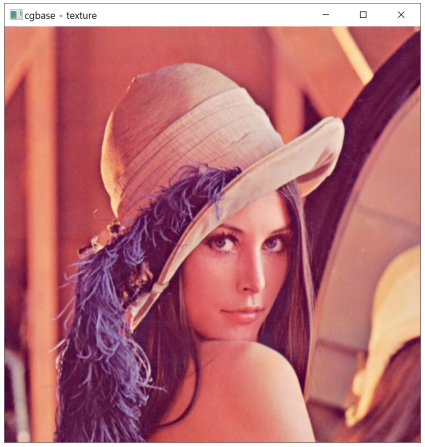
- **Fetch the color from the multiple texture samplers.**

```
in vec2 tc;
out vec4 fragColor;

uniform sampler2D TEX0; // first texture sampler object
uniform sampler2D TEX1; // second texture sampler object
uniform sampler2D TEX2; // third texture sampler object
uniform int       mode;

void main()
{
    if(mode==1)       fragColor = texture( TEX0, tc );
    else if(mode==2)  fragColor = texture( TEX1, tc ).rrrr;
    else if(mode==3)  fragColor = texture( TEX2, tc ).aaaa;
    else              fragColor = vec4(tc,0,0);
}
```

Example

- **Cyclic toggling of multiple textures and texcoords**





Texture Sampling Revisited

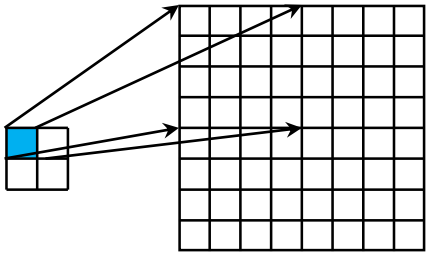
Resampling

- **Both *textures* and *fragments* of surfaces are raster images.**
  - Recall raster images are sampled representation of continuous function.

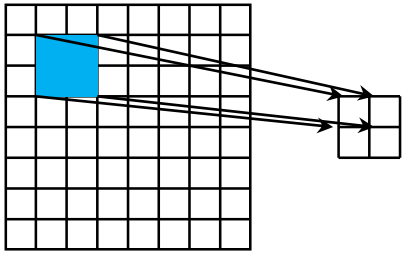
- **Aliasing:**
  - Insufficient sampling rates may cause the incorrect reconstruction of continuous signals.
  - Hence, we need resampling, when the resolutions of textures and fragments do not match.

Magnification vs. Minification

- **Texture magnification:**
  - Texture resolution < object surface resolution
  - In this case, aliasing is not that severe, but we may get better reconstruction by interpolation.
- **Texture minification:**
  - Texture resolution > object surface resolution
  - Aliasing is a serious problem; we need to pre-integrate the signals for better reconstruction.



Magnification



Minification

Magnification

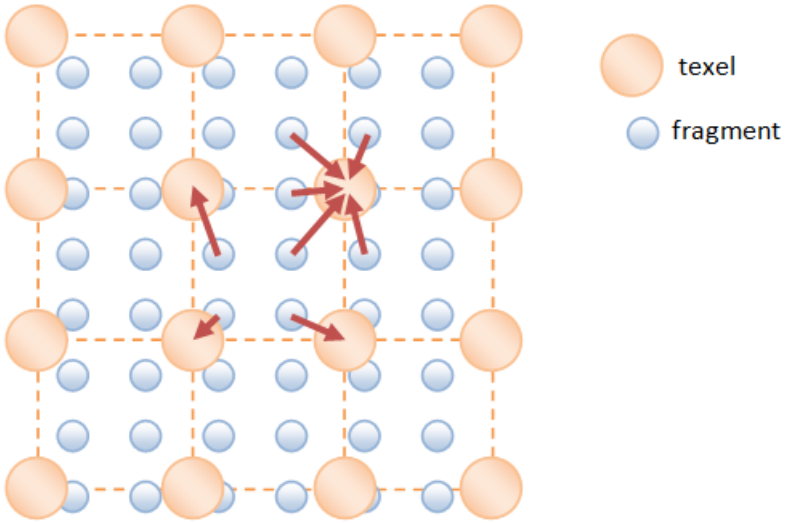
- **Adjacent pixels in the window space map to the same texel.**
  - We can use the texel as it is: nearest neighbor sampling
  - We can apply interpolation according to the window space position.
    - Bilinear, bicubic interpolation, …

[Akenine-Moeller et al.: Real-time Rendering]

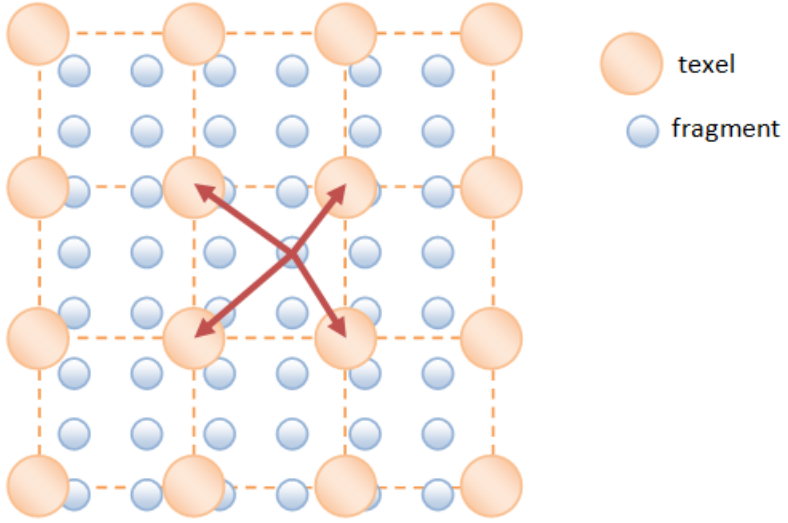
nearest neighbor | bilinear interpolation | bicubic interpolation

Magnification

- **Point (nearest-neighbor) sampling vs. Bilinear interpolation**



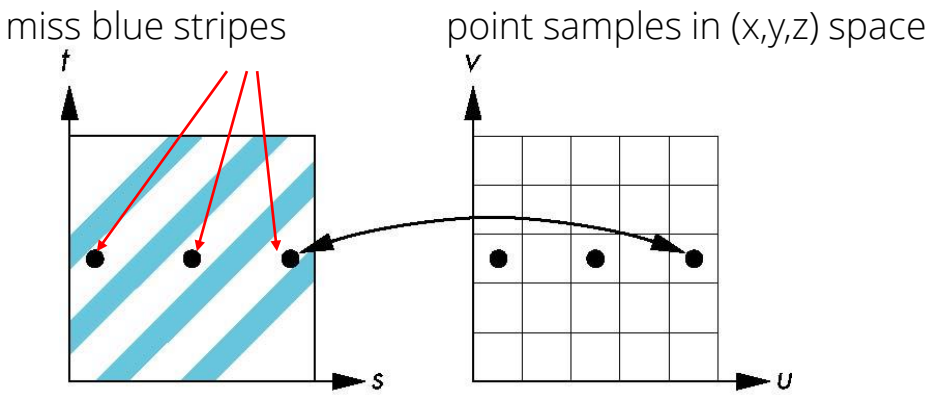
Magnification – Nearest Point Sampling

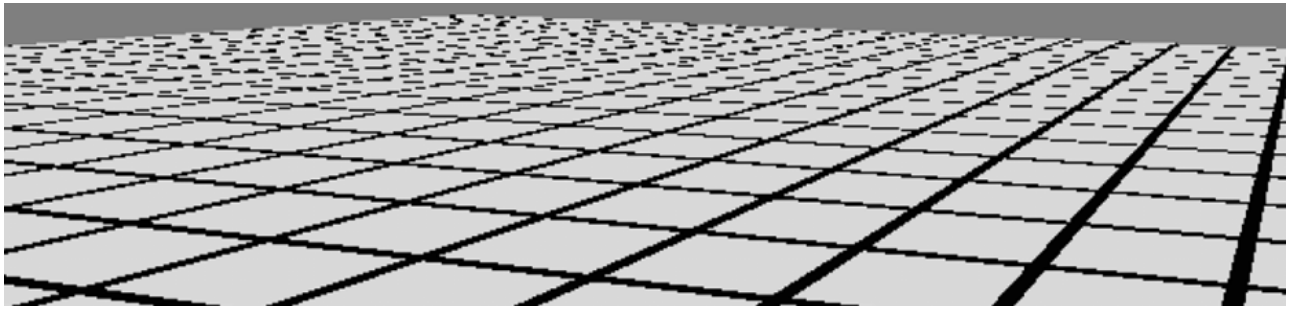


Magnification – Bilinear Interpolation

Minification

- **The aliasing problem from undersampling**
  - Many samples correspond to the single fragment, but point sampling only fetches a single sample among them.

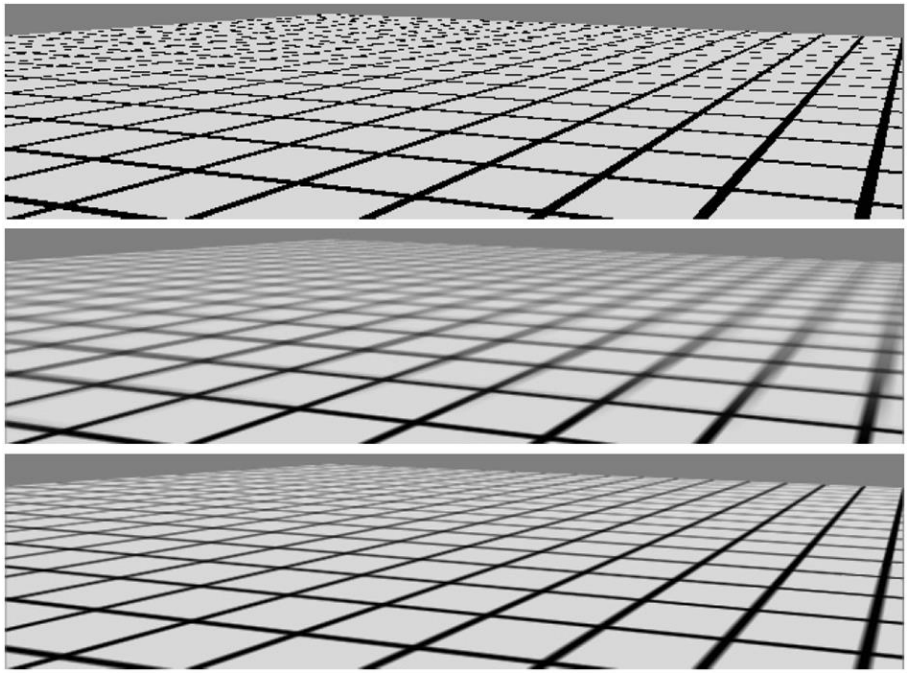




Undersampling in minification is the most pronounced at the farther surfaces, which suffers from aliasing.

Pre-integration for Minification

- **Pre-integration (area averaging)**
  - Prior to the texture look-up, we can average the multiple texels.
  - **Mipmapping** is a a standard pre-integration technique in OpenGL



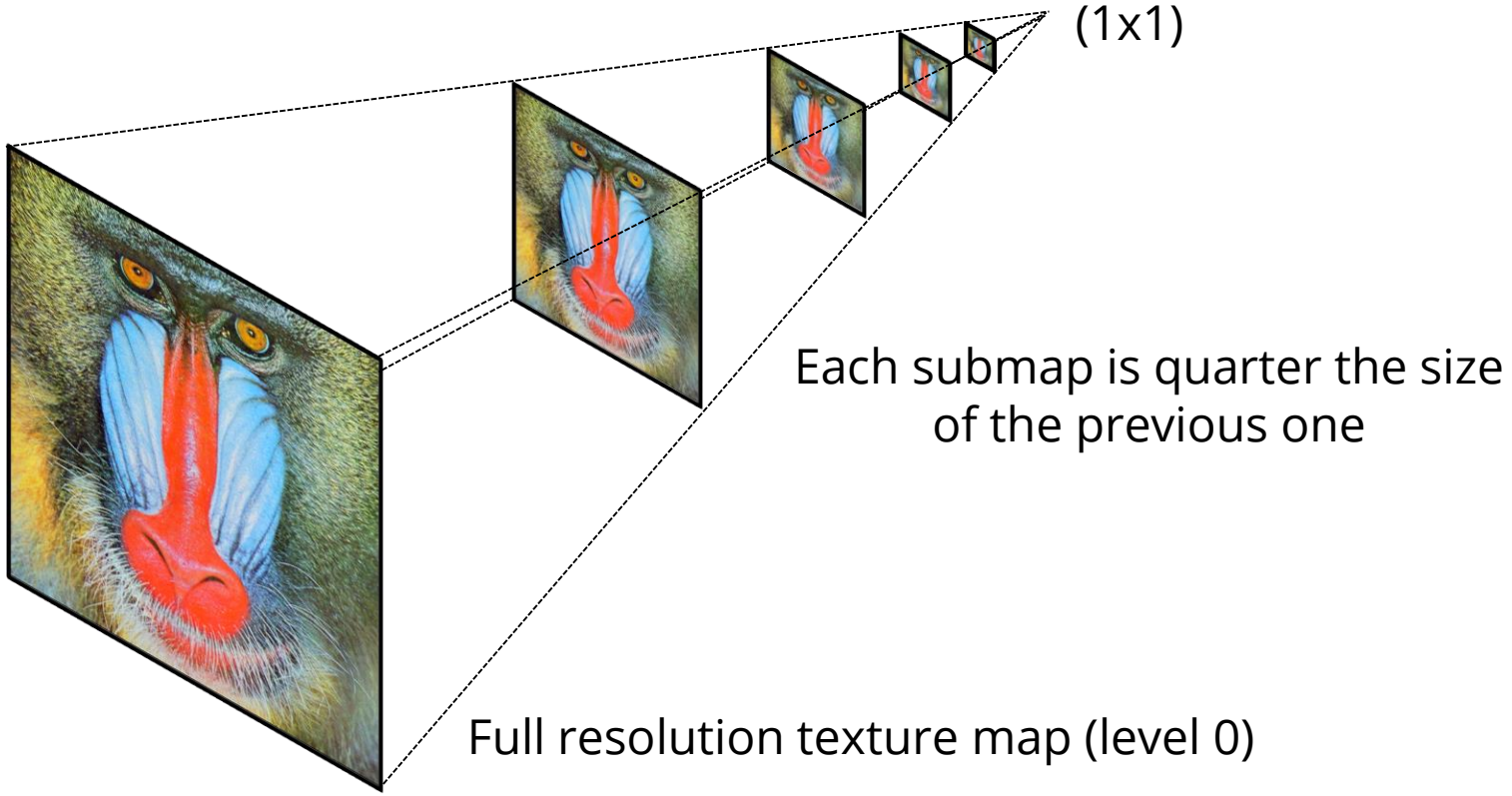
nearest neighbor (point sampling)

mipmapping

summed-area tables

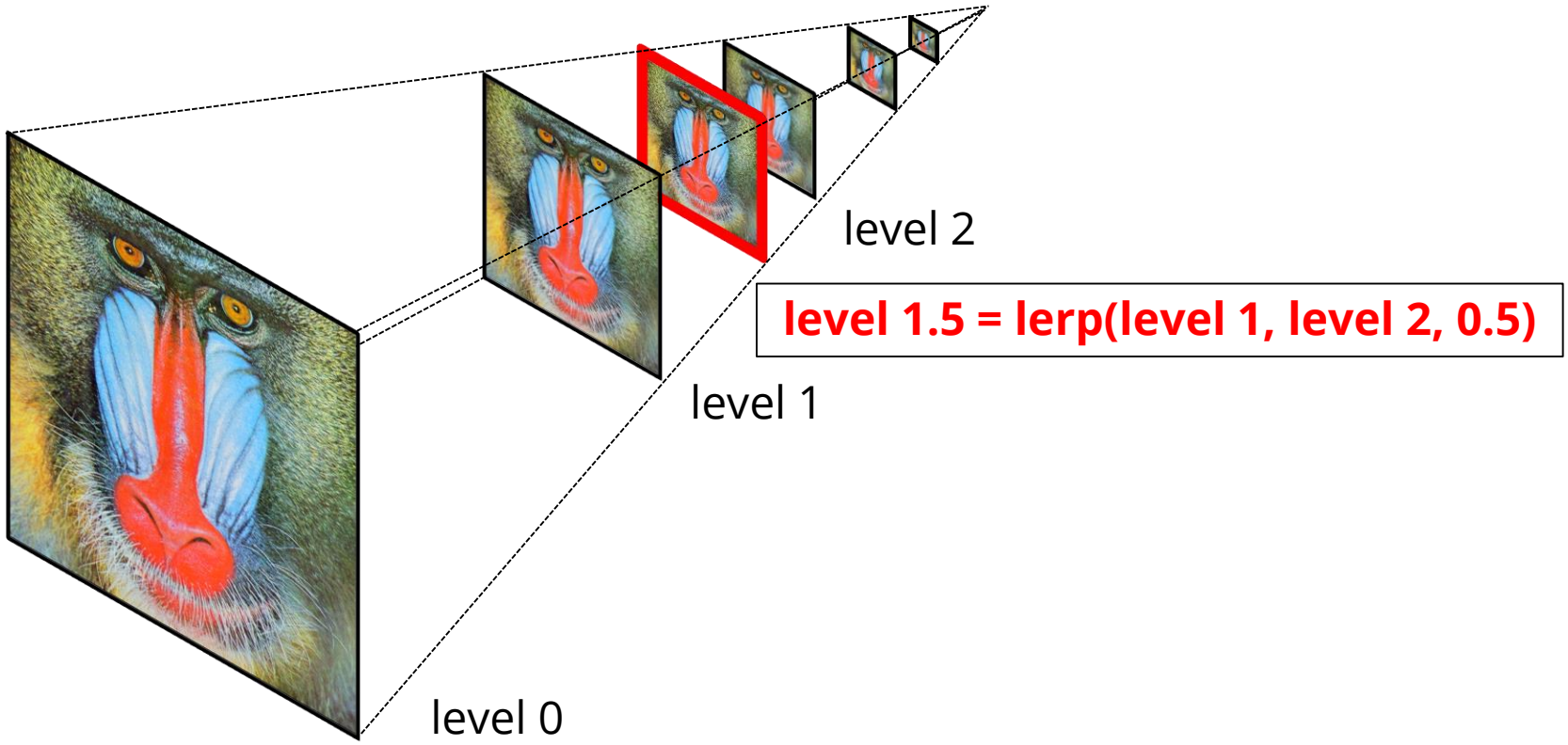
Pre-integration via Mipmapping

- **A texel in a mipmap level n represents spatial averages of 2^n.**
  - Thereby, we can fetch a pre-integrated single texel in a area.
  - To find a mipmap level, we need a bit of computation, but this is automatically provided in OpenGL.



Tri-linear Interpolation in Mipmapping

- **Trilinear interpolation: looking up non-integer mipmap level**
  - A mipmap level can be a real number, but mipmaps are pre-built for integer levels.
  - OpenGL (with **GL_LINEAR_MIPMAP_LINEAR**) linearly interpolates between two (bilinearly filtered) integer levels. For example, level 1.5 interpolates levels 1 and 2, by the factor of 0.5.



Mipmapped Textures

- ***Mipmapping*** **allows prefiltered texture maps of decreasing resolutions**
  - Lessens interpolation errors for smaller textured objects
  - Declare mipmap level during texture definition
  - Or use glTexStorage2D function for all level of texture generation

```
glTexImage2D( GL_TEXTURE_2D, 0, ... )
for( int level=1; level < mip_levels; level++ )
   glTexImage2D( GL_TEXTURE_2D, level, ... );
glGenerateMipmap( GL_TEXTURE_2D );
```

```
// the code above equivalent to
glTexStorage2D( GL_TEXTURE_2D, mip_levels, ... );
glGenerateMipmap(GL_TEXTURE_2D );
```

Texture Parameters

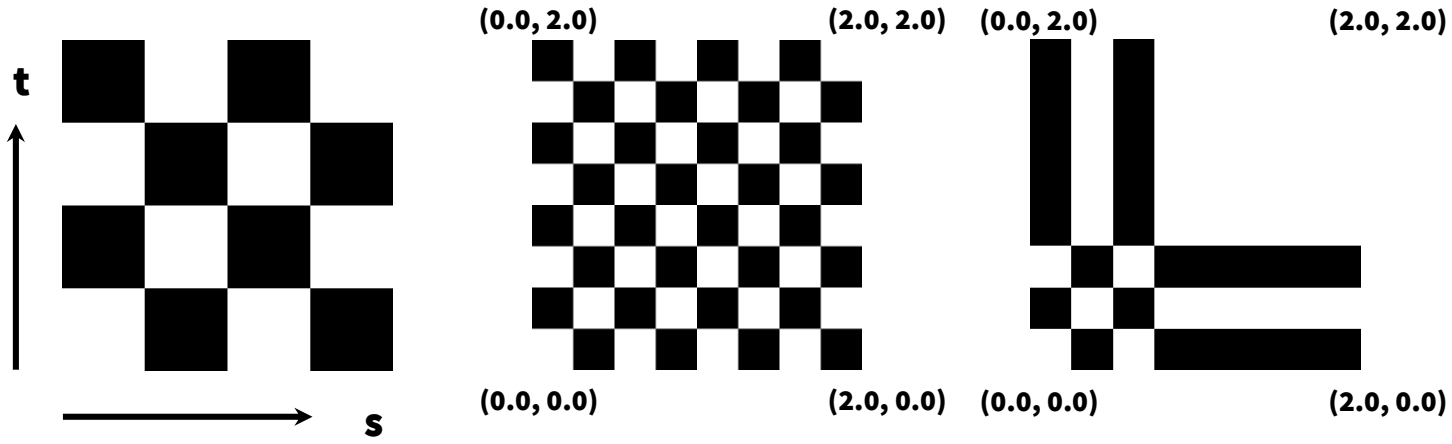
Texture Parameters

- **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.
  - Filter modes allow us to use area averaging instead of point samples.
  - Mipmapping allows us to use textures at multiple resolutions.

Wrapping Modes

- **Clamping:**
  - if $s, t > 1$, use 1, if $s, t < 0$, use 0.
- **Repeating:**
  - use s, t modulo 1

```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
```



texture GL_REPEAT GL_CLAMP_TO_EDGE

Magnification and Minification

- **Modes determined by**

```
glTexParameteri( GL_TEXTURE_2D, GL_TEXURE_MAG_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_2D, GL_TEXURE_MIN_FILTER, GL_LINEAR );
```

GL_NEAREST

GL_LINEAR

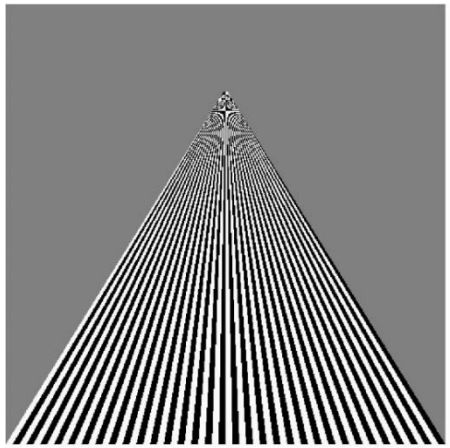
- **When you have mipmaps, use this instead:**

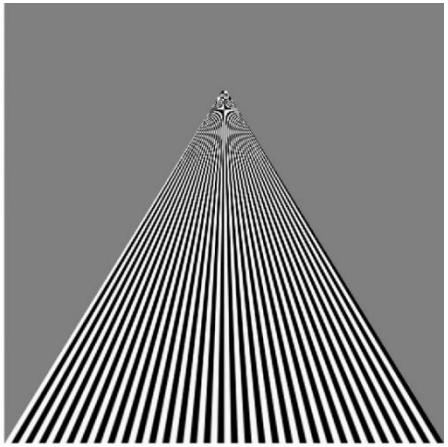
```
glTexParameteri( GL_TEXTURE_2D, GL_TEXURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR );
```

Example

- **Texture filtering examples**

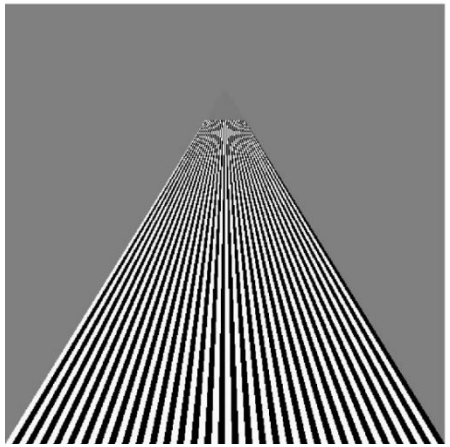
point sampling (GL_NEAREST)

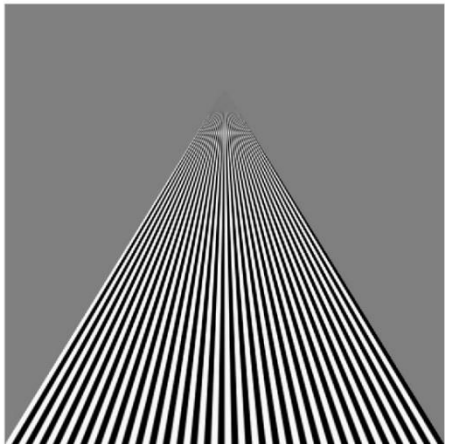




linear filtering (GL_LINEAR)

mipmapped point sampling (GL_NEAREST_MIPMAP_NEAREST)





mipmapped linear sampling (GL_LINEAR_MIPMAP_LINEAR)

Putting All Together

```
// this function will be avaialble as cg_create_texture() in other samples
GLuint create_texture( const char* image_path, bool mipmap, GLenum wrap, GLenum filter )
{
     // load image and set internal format and format from image
     ...

     GLuint texture;
     glGenTextures( 1, &texture );
     glBindTexture( GL_TEXTURE_2D, texture );
     glTexImage2D( GL_TEXTURE_2D,0,internal_format,w,h,0,format,GL_UNSIGNED_BYTE,img->ptr );

     // build mipmap
     if( mipmap ) ...

     // set up texture parameters
     glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, wrap );
     ...

     return texture;
}
```

Textured Shading

Diffuse Texture Mapping

- **Diffuse mapping in Blinn-Phong Illumination Model**
  - Uses images to fill inside of polygons ($\mathbf{k}_d$)
  - Since this is common, texture mapping usually refers to diffuse mapping.

- **Fetch the color from the texture sampler.**
  - Use the color for Kd (diffuse reflectance).
  - That's it; very simple.

```
...

// uniform vec4  Kd; // do not use diffuse per-object material color

void main()
{
   ...
   vec4 Kd = texture( TEX, tc ); // replace uniform Kd with texture sampling

   ...
}
```

Example