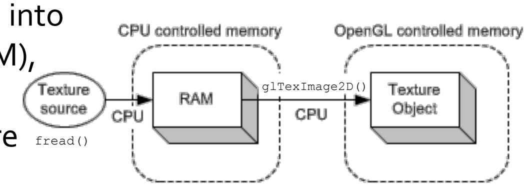
EECS 487: Interactive Computer Graphics

Lecture 25:

- Texture Loading: Pixel-Buffer Object
- Texture Filtering

Texture Loading and PBO

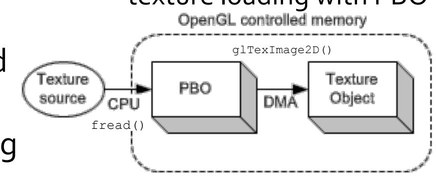
Reading texture file into client memory (RAM), and then writing it from RAM to texture object can be slow



texture loading without PBO

Pixel Buffer Object (PBO) allows fast data transfer between graphics card and file through DMA (Direct Memory Access), bypassing RAM

texture loading with PBO



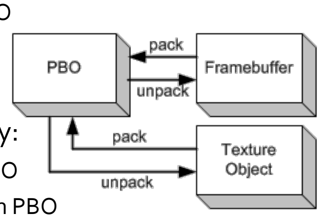
[Ahn]

Pixel Buffer Object (PBO)

Stores pixel data into buffer objects

Same mechanisms as VBO, with two additional "targets" (or types of buffer):

- `GL_PIXEL_PACK_BUFFER`, used by:
  - `glReadPixel`: read from framebuffer to PBO
  - `glGetTexImage`: read from texture to PBO
  - "packed to be shipped off"
- `GL_PIXEL_UNPACK_BUFFER`, used by:
  - `glDrawPixel`: write to framebuffer from PBO
  - `glTex(Sub)Image2D`: write to texture from PBO
  - "unpacked to be used"



[Ahn]

Pixel Buffer Object Setup

As with other OpenGL objects, first generate buffer object handle(s):

```
glGenBuffers(GLsizei n, GLuint *pbods);
```

Next bind PBO descriptor to a type of buffer

```
glBindBuffer(target, pbod);
// target is GL_PIXEL_PACK_BUFFER or
// GL_PIXEL_UNPACK_BUFFER
```

and allocate space for it:

```
glBufferData(target, size, data, usage);
// data: set to NULL to simply allocate space (no data copy)
// usage: GL_STREAM_{DRAW,READ}
```

Populating Pixel Buffer Object

As with VBO, we could populate the PBO by copying over texture image stored in client-side memory using

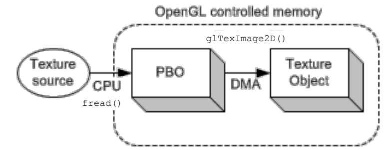
```
glBufferData(target, size, data, usage);
glBufferSubData(target, offset, size, data);
// data: pointer to data in client-side memory (RAM)
```

Or we could bypass client-side memory by mapping graphics-system memory to client address space

```
void *glMapBuffer(GLenum target, GLenum access);
// target: same as glBindBuffer()
// access: GL_WRITE_ONLY, GL_READ_ONLY, GL_READ_WRITE
```

returns a pointer to the mapped memory

Write to PBO Bypassing RAM



Setup PBO and map it to client address:

- bind and allocate PBO

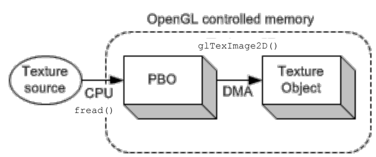
```
int pbod; glGenBuffers(1, &pbod);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbod);
glBufferData(GL_PIXEL_UNPACK_BUFFER, DATASIZE,
             NULL, GL_STREAM_DRAW);
```

- to map PBO to client address space use:

```
char *texture=glMapBuffer(GL_PIXEL_UNPACK_BUFFER,
             GL_WRITE_ONLY); // app writes into PBO
```

[Ahn]

Write to PBO Bypassing RAM



To bypass RAM, read from file directly into PBO

- read into PBO from texture file with handle `fin`

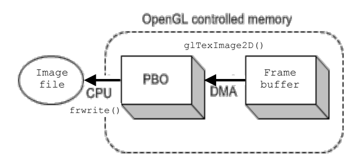
```
fin >> texture;
```

- now we can unmap buffer from client address space and write/unpack the PBO to texture object:

```
glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER);
glTexImage2D(..., offset /* instead of pointer */)
```

[Ahn]

Read from PBO Bypassing RAM



Setup PBO and pack it with framebuffer content:

- bind and allocate PBO

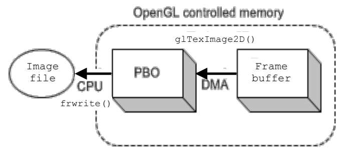
```
int pbod; glGenBuffers(1, &pbod);
glBindBuffer(GL_PIXEL_PACK_BUFFER, pbod);
glBufferData(GL_PIXEL_PACK_BUFFER, DATASIZE,
             NULL, GL_STREAM_READ);
```

- next, specify the framebuffer to read from and pack it into the bound PBO

```
glReadBuffer(GL_FRONT);
glReadPixels(..., offset /* instead of pointer */)
```

[Ahn]

Read from PBO Bypassing RAM



To read from PBO directly to file:

- map PBO to client address space

```
char *image = glMapBuffer(GL_PIXEL_PACK_BUFFER,
     GL_READ_ONLY); // app reads from PBO
```

- finally, dump the PBO directly to image file with handle `fout` and unmap buffer from client address space:

```
fwrite(image, sizeof(char), DATASIZE, fout);
glUnmapBuffer(GL_PIXEL_PACK_BUFFER);
```

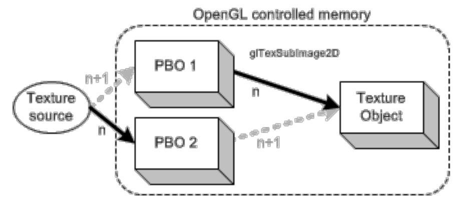
[Ahn]

Double Buffering

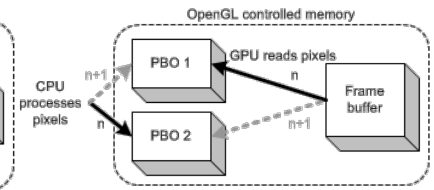
Since file⇔PBO transfer is done by the CPU and PBO⇔texture/framebuffer is done by the GPU, the two can happen asynchronously

- `glMapBuffer()` waits if GPU is busy with buffer
- `glBufferData()` with NULL pointer detaches existing buffer object, which will be freed when GPU is done with it

We can use double buffering to speed things up:



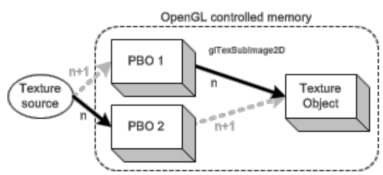
double-buffered unpack



double-buffered pack

[Ahn]

Double Buffered Unpack

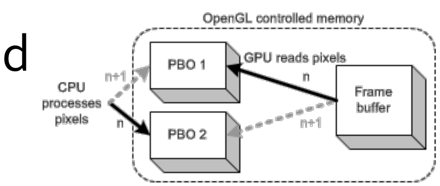



```
int i=0;
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbods[i]);
glBufferData(GL_PIXEL_UNPACK_BUFFER, size, 0, GL_STREAM_DRAW);
texture = glMapBuffer(GL_PIXEL_UNPACK_BUFFER, GL_WRITE_ONLY);
fin >> texture; // blocking
glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER);
while (not done) {
  glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbods[i]);
  glTexSubImage2D(); // non-blocking

  i = (i+1)%2;
  glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbods[i]);
  glBufferData(GL_PIXEL_UNPACK_BUFFER, size, 0,
      GL_STREAM_DRAW); // to prevent MapBuffer() from blocking
      // if TexSubImage2D() from previous iteration is not done
  texture = glMapBuffer(GL_PIXEL_UNPACK_BUFFER, GL_WRITE_ONLY);
  fin >> texture; // blocking
  glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER);
}
```

[Ahn]

Double Buffered Pack




```
int i=0;
glReadBuffer(GL_FRONT);
glBindBuffer(GL_PIXEL_PACK_BUFFER, pbods[i+1]);
glBufferData(GL_PIXEL_PACK_BUFFER, size, 0, GL_STREAM_READ);
glBindBuffer(GL_PIXEL_PACK_BUFFER, pbods[i]);
glBufferData(GL_PIXEL_PACK_BUFFER, size, 0, GL_STREAM_READ);
glReadPixels(..., 0 /* offset */); // non-blocking
while (not done) {
  glBindBuffer(GL_PIXEL_PACK_BUFFER, pbods[i]);
  // MapBuffer blocks until ReadPixel is done
  image = glMapBuffer(GL_PIXEL_PACK_BUFFER, GL_READ_ONLY);
  fwrite(image, sizeof(char), size, fout);
  glUnmapBuffer(GL_PIXEL_PACK_BUFFER);

  i = (i+1)%2;
  glBindBuffer(GL_PIXEL_PACK_BUFFER, pbods[i]);
  // fwrite() is blocking
  glReadPixels(..., 0 /* offset */); // non-blocking
}
```

[Ahn]

Texture Filtering

Mipmapping

- mip == "multum in parvo" (many things in a small place)

Summed-area table

Anisotropic filtering

Texture Filtering

Mipmapping

- mip == "*multum in parvo*" (many things in a small place)

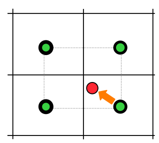
Summed-area table

Anisotropic filtering

Texture Value Interpolation

Interpolated texture coordinates (s, t) are continuous values, texture image is discretely indexed
How to compute the color of a pixel?

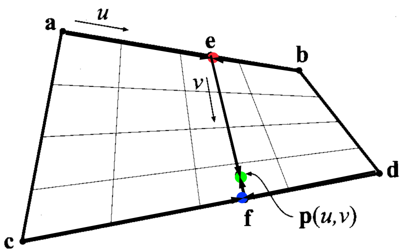
Nearest neighbor (point sample), use color of closest texel:



Simple and fast, but low quality

OpenGL:
```
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MIN_FILTER,
                GL_NEAREST);
```

Bilinear Interpolation



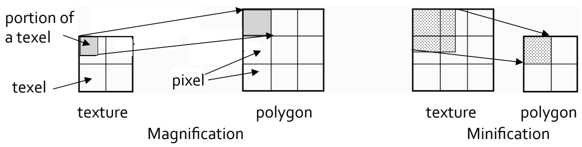
Linear interpolation in 2D:

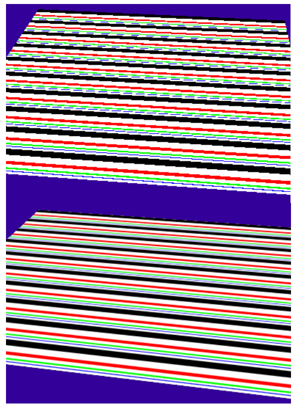
$$\mathbf{e} = (1-u)\mathbf{a} + u\mathbf{b}$$
$$\mathbf{f} = (1-u)\mathbf{c} + u\mathbf{d}$$
$$\mathbf{p}(u,v) = (1-v)\mathbf{e} + v\mathbf{f} = (1-u)(1-v)\mathbf{a} + u(1-v)\mathbf{b} + (1-u)v\mathbf{c} + uv\mathbf{d}$$

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

Heckbert89

Fitting Texture to Primitive



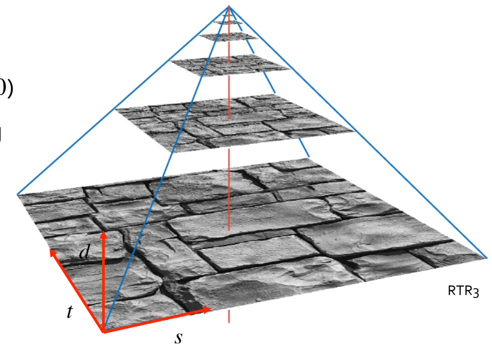


Magnification: texture is too small for polygon/triangle (not whole surface)
- nearest neighbor point sample: texel repeated, causing aliasing
- (bi)linear interpolation: blurring

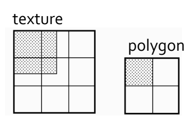
Minification: many texels per pixel
- nearest neighbor point sample: aliasing causing moire pattern
- mipmapping with trilinear interpolation
  - `GL_LINEAR_MIPMAP_LINEAR`

Minification: Mipmapping

Many texels map (shrunk) into a single pixel
- need to average effects of many texels: expensive
- precompute/prefilter texture maps of decreasing resolutions: lessens interpolation errors for smaller textured objects
- image pyramid
  - halve width (s) and height (t) when going upwards (d, $d \geq 0$)
  - filtering while down sampling
    - simple box filter (average over 4 "parent texels" to form a "child texel")
    - or some other, better filter

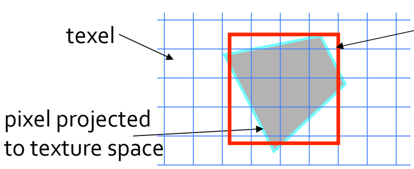


Finding the Mip Level



texel/pixel coverage: 2.5:1

One simple way to compute d (level of detail):



A = approx. area of quadrilateral (in #texels/pixel)

$$d = \log_2 \sqrt{A}$$

- compute number of texels per pixel
- approximate coverage with square
- e.g., given a texture of 128×128 texels
  - for a 128×128 polygon, $d = \log_2(1) = 0$
  - for a 64×64 polygon, 4 texels per pixel, $d = \log_2(\sqrt{4}) = 1$

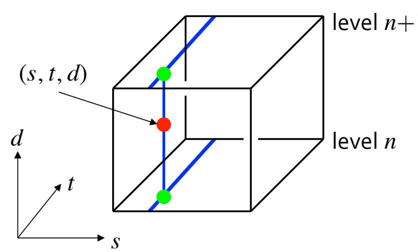
d too large:
⇒ gives overblur

d too small:
⇒ fails to anti-alias

Akenine-Möller02

Trilinear Interpolation

Given texels in 2 levels, do trilinear interpolation:
- bilinear interpolation in each level
- linear interpolation across levels

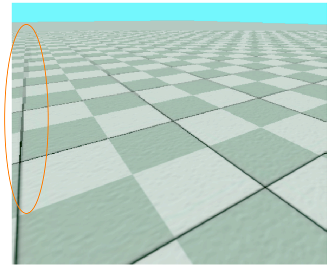


(can also use nearest neighbor instead)

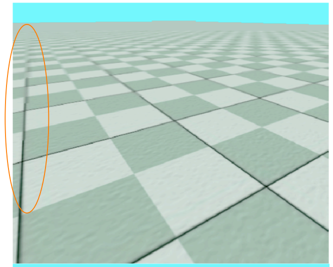
Akenine-Möller02

GL_TEXTURE_MIN_FILTER

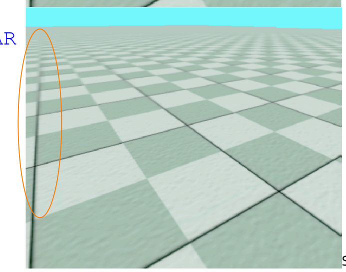
GL_NEAREST_MIPMAP_NEAREST



GL_LINEAR_MIPMAP_NEAREST



GL_LINEAR_MIPMAP_LINEAR (trilinear)



Schulze08

Use textures from different mipmap levels as one moves towards the horizon

Specifying the Mipmap Image

Manually specify a different texture image for each level:

```
glTexImage2D(target, level, internalFormat, width,
             height, border, format, type, teximage)
// target: GL_TEXTURE_2D
// level: mipmap level, 0 if not mipmapping
// teximage: pointer to image in memory
```

Or generate mipmap pyramid automatically by using one of:

- `glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE); //` must be called BEFORE `glTexImage2D()`
- `glGenerateMipmap(GL_TEXTURE_2D);`
  `//` must be called AFTER `glTexImage2D()`, used with FOB
- `gluBuild2DMipmaps(); // deprecated`

Setting Mipmap Parameters

```
glTexParameteri(target, pname, param);
```

where

- `target` is `GL_TEXTURE_2D`
- `pname` is a parameter name that you want to change:
  - `GL_TEXTURE_WRAP_T`
  - `GL_TEXTURE_WRAP_S`
  - `GL_TEXTURE_MIN_FILTER`
  - `GL_TEXTURE_MAG_FILTER`
- `param` is the parameter value to change to

For example:

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
  GL_LINEAR_MIPMAP_NEAREST) // or GL_NEAREST_MIPMAP_NEAREST
  or GL_LINEAR_MIPMAP_LINEAR (trilinear)
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
  GL_LINEAR) // or GL_NEAREST
```

Limitations of Mipmapping

1. Area over which to compute pixel value (i.e., texel coverage) is always as a square (isotropic filtering)
2. Fixed filters: only a pre-determined, fixed number of area sizes are available, i.e., mip levels are fixed in number and pre-determined

Result: overblurred

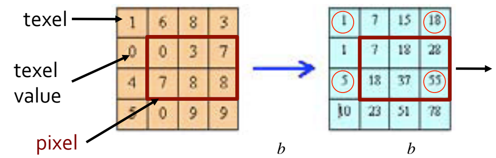
Summed-Area Tables

Pre-compute area-sum, but filtering (size of area to average from) to fit pixel is done on-the-fly, only when texel coverage is known

Advantages:
- no pre-determined mip levels, texture can be shrunk to custom size
- no need to keep multiple tables
- texel coverage can be rectangular in shape (but still isotropic)

Summed-Area Tables

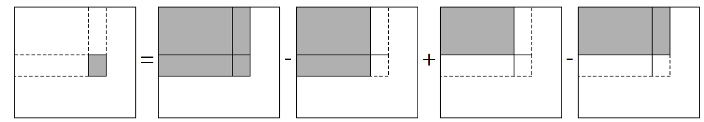
Table contains two-dimensional cumulative distribution function: keep sum of everything above and to the left



Average texel value in area covered by pixel: $(55-5+1-18 = 33)/6$

Recall from calculus: $\int_a^b f(x)dx = \int_{-\infty}^b f(x)dx - \int_{-\infty}^a f(x)dx$

or in discrete form: $\sum_{i=k}^m f[i] = \sum_{i=0}^m f[i] - \sum_{i=0}^k f[i]$



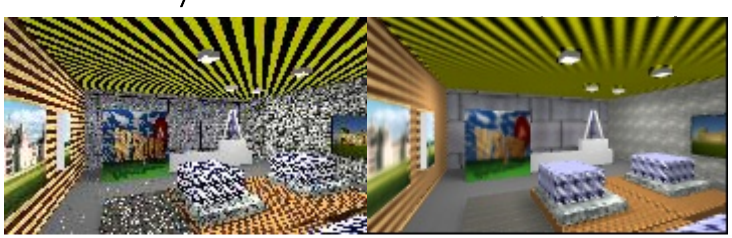
Popovic09

Summed-Area Tables

Disadvantages:
- requires four table lookups
- and more memory to keep the larger summed values (2-4 times the original image)

aschilda

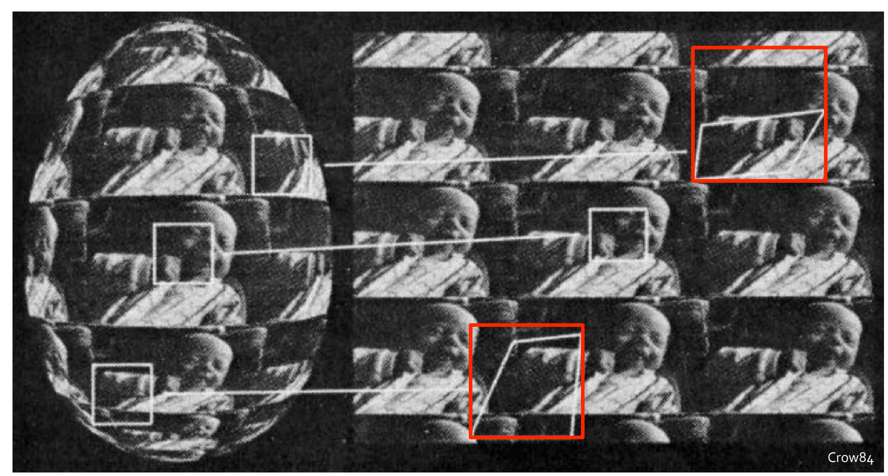
Gives less blurry textures



nearest neighbor bilinear

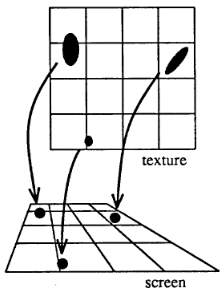
Problem with Isotropic Filtering

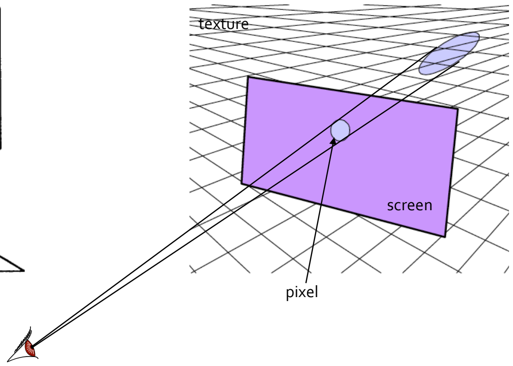
Uniform averaging (isotropic filtering) in screen space becomes non-uniform (anisotropic) in texture space



Problem with Isotropic Filtering

Texture distortion happens not only due to surface curving, but also due to perspective projection



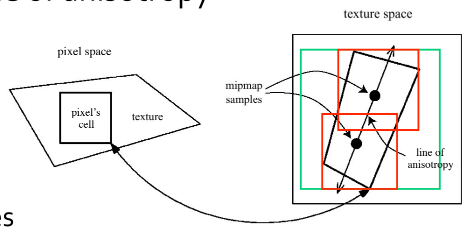


Anisotropic Filtering

Summed-area table is constrained to axis-aligned rectangle

Alternative: approximate quad with several smaller mipmap samples along line of anisotropy

- line of anisotropy along the longer of the quad edges
- use the shorter of the quad edges to determine level
- number of samples = ratio of long/short quad edges



Pixel color is weighted average of the samples

RTR