Texture Loading and PBO

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

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

Pixel Buffer Object Setup

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

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

Next bind PBO descriptor to a type of buffer:

```c
glBindBuffer(target, pbod);
```

and allocate space for it:

```c
glBufferData(target, size, data, usage);
```

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
  - `glTexImage2D`: write to texture from PBO
  - “unpacked to be used”

[Image: Ahn]
Populating Pixel Buffer Object

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

```c
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

```c
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

To bypass RAM, read from file directly into PBO

- read into PBO from texture file with handle `fin`
  ```c
  fin >> texture;
  ```
- now we can unmap buffer from client address space and write/unpack the PBO to texture object:
  ```c
  glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER);
glTexImage2D(..., offset /* instead of pointer */)  
  ```

Read from PBO Bypassing RAM

Setup PBO and pack it with framebuffer content:

- bind and allocate PBO
  ```c
  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
  ```c
  glBindBufferRange(GL_FRAMEBUFFER, 0, pbod, 0, DATASIZE);
  ```
- now we can unpack frame buffer contents into the PBO:
  ```c
  glReadBuffer(GL_FRONT);
glReadPixels(..., offset /* instead of pointer */)  
  ```
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);

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
• glBufferSubData() 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

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_WRITEONLY);
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 previous iteration is not done
    // if TexSubImage2D() from previous iteration is not done
    texture = glMapBuffer(GL_PIXEL_UNPACK_BUFFER, GL_WRITEONLY);
    fin >> texture; // blocking
    glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER);
}

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_READONLY);
    fwrite(image, sizeof(char), Size, fout);
    glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER);
    i = (i+1)%2;
    glBindBuffer(GL_PIXEL_PACK_BUFFER, pbods[i]);
    // fwrite() is blocking
    glReadPixels(..., 0 /* offset */); // non-blocking
}
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:
\[
\begin{align*}
    e &= (1 - u)a + ub \\
    f &= (1 - u)c + ud \\
    p(u, v) &= (1 - v)e + vf \\
            &= (1 - u)(1 - v)a + u(1 - v)b + (1 - u)v c + uv d
\end{align*}
\]

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

**Finding the Mip Level**

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

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

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

$\Rightarrow$ gives overblur
$\Rightarrow$ fails to anti-alias

**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

**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)
Specifying the Mipmap Image

Manually specify a different texture image for each level:

```c
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

```c
.glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_NEAREST) // or GL_NEAREST_MIPMAP_NEAREST or GL_LINEAR_MIPMAP_LINEAR (trilinear)
```

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.

Recall from calculus:
\[
\int_a^b f(x) \, dx = \int_a^\infty f(x) \, dx - \int_\infty^b f(x) \, dx
\]
or in discrete form:
\[
\sum_{i=k}^{m} f[i] = \sum_{i=0}^{m} f[i] - \sum_{i=0}^{k} f[i]
\]

Problem with Isotropic Filtering

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

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

Gives less blurry textures

![nearest neighbor](image1.png) ![bilinear](image2.png)
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.