Buffers, Compositing and Blending

Week 8

David Breen
Department of Computer Science
Drexel University

Based on material from Ed Angel, University of New Mexico
Han-Wei Shen, Ohio State University
David Blythe, Silicon Graphics & Tom McReynolds, Gigapixel

Objectives

- Introduce additional OpenGL buffers
- Learn to read and write buffers
- Learn to use blending
- Learn depth-of-field and reflection effects

Pixel and Geometry Pipelines

- Parallel pipelines (Geometry & Pixel) come together in the Rasterization stage

OpenGL Frame Buffer

- Color buffers can be displayed
  - Front
  - Back
  - Auxiliary
  - Overlay
- Depth
- Accumulation
  - High resolution buffer
- Stencil
  - Holds masks
Writing in Buffers

• Conceptually, we can consider all of memory as a large two-dimensional array of pixels
• We read and write rectangular block of pixels
  – Bit block transfer (bitblt) operations
• The frame buffer is part of this memory

The Pixel Pipeline

• OpenGL has a separate pipeline for pixels
  – Writing pixels involves
    • Moving pixels from processor memory to the frame buffer
    • Unpacking (Format conversions, swapping & alignment)
    • Transfer ops, Map Lookups, Tests
  – Reading pixels
    • Packing (Format conversion, swapping & alignment)

Pixel Storage Modes

• Deal with byte-ordering conventions
• Memory alignment

```c
void glPixelStore(GLenum pname, TYPE param);
```

```c
pnames:
GL_UNPACK_SWAP_BYTES, GL_PACK_SWAP_BYTES,
GL_UNPACK_LSB_FIRST, GL_PACK_LSB_FIRST,
GL_UNPACK_ROW_LENGTH, GL_PACK_ROW_LENGTH,
GL_UNPACK_SKIP_ROWS, GL_PACK_SKIP_ROWS,
GL_UNPACK_ALIGNMENT, GL_PACK_ALIGNMENT
```

Pixel Transfer Operations

• Color Scale & Bias
• Color Index Shift & Offset
• Convolution
• Matrix multiply

```c
void glPixelTransfer(GLenum pname, param);
```

```c
glPixelTransfer(GL_RED_SCALE, s);
glPixelTransfer(GL_BLUE_BIAS, b);
```

Pixel Mapping Operations

• Color components and color & stencil indices can be modified via a table look-up

```c
void glPixelMap(GLenum map, Glint size, TYPE *array);
```

```c
map = GL_PIXEL_MAP_*_TO_*
  I, S, R, G, B, A
```

Pixel Tests

• Scissor `glScissor()`
  – Only draw in a rectangular portion of screen
• Alpha `glAlphaFunc()`
  – Draw based on alpha value
• Stencil `glStencilFunc()`, `glStencilOp()`
  – Draw based on stencil value
• Depth `glDepthFunc()`
  – Draw based on depth value
Writing Model

Read destination pixel before writing source

```
glLogicOp (GLenum op)
```

Writing Modes

- Source and destination bits are combined bitwise
- 16 possible functions (one per column in table)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Logically Pixel Operations

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Resulting Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_CLEAR</td>
<td>0</td>
</tr>
<tr>
<td>GL_SET</td>
<td>1</td>
</tr>
<tr>
<td>GL_COPY</td>
<td>1</td>
</tr>
<tr>
<td>GL_COPY_BITMASK</td>
<td>→</td>
</tr>
<tr>
<td>GL_NOP</td>
<td>d</td>
</tr>
<tr>
<td>GL_PRIV</td>
<td>d</td>
</tr>
<tr>
<td>GL_ADD</td>
<td>s &amp; d</td>
</tr>
<tr>
<td>GL_SUB</td>
<td>s - d</td>
</tr>
<tr>
<td>GL_MIN</td>
<td>s &amp; d</td>
</tr>
<tr>
<td>GL_MAX</td>
<td>s &amp; d</td>
</tr>
<tr>
<td>GL_BAND</td>
<td>s &amp; d</td>
</tr>
<tr>
<td>GL_BAND_INV</td>
<td>s &amp; d</td>
</tr>
<tr>
<td>GL_BOR</td>
<td>s</td>
</tr>
<tr>
<td>GL_XOR</td>
<td>s ^ d</td>
</tr>
<tr>
<td>GL_EQU</td>
<td>s = d</td>
</tr>
<tr>
<td>GL_AND</td>
<td>s &amp; d</td>
</tr>
<tr>
<td>GL_AND_INV</td>
<td>s &amp; ~d</td>
</tr>
<tr>
<td>GL_XOR2</td>
<td>s ^ d</td>
</tr>
</tbody>
</table>

XOR mode

- Applying twice returns bit back to original value
- XOR is especially useful for swapping blocks of memory such as menus that are stored off screen

If S represents screen and M represents a menu
the sequence

```
S ← S₀
M ← S ⊕ M
S ← S ⊕ M
```

swaps the S and M

Raster Position

- OpenGL maintains a raster position as part of the state
- Set by `glRasterPos*()`
  - `glRasterPos3f(x, y, z)`
- The raster position is a geometric entity (like a vertex)
  - Passes through geometric pipeline
  - Eventually yields a 2D position in screen coordinates
  - This position in the frame buffer is where the next raster primitive is drawn

Window Position

- In OpenGL 1.4 `glWindowPos*()` was defined
- Allows you to specify current raster position in window coordinates
Buffer Selection
- OpenGL can draw into or read from any of the color buffers (front, back, auxiliary)
- Default to the back buffer
- Change with `glDrawBuffer` and `glReadBuffer`
- Note that format of the pixels in the frame buffer is different from that of processor memory and these two types of memory reside in different places
  - Need packing and unpacking
  - Drawing and reading can be slow

Bitmaps
- OpenGL treats 1-bit pixels (bitmaps) differently than multi-bit pixels (pixelmaps)
- Bitmaps are masks which determine if the corresponding pixel in the frame buffer is drawn with the present raster color
  - 0 ⇒ color unchanged
  - 1 ⇒ color changed based on writing mode
- Bitmaps are useful for raster text
  - `GLUT_BITMAP_8_BY_13`

Raster Color
- Same as drawing color set by `glColor*()`
- Fixed by last call to `glRasterPos*()`
  
  ```
  glColor3f(1.0, 0.0, 0.0);  
  glRasterPos3f(x, y, z);  
  glColor3f(0.0, 0.0, 1.0);  
  glBitmap( ... );  
  glBegin(GL_LINES);  
  glVertex3f( ... );  
  ```
- Geometry drawn in blue
- Ones in bitmap use a drawing color of red

Drawing Bitmaps
- `glBitmap(width, height, x0, y0, xi, yi, bitmap)`
  
  ```
  glBitmap(64, 64, 0.0, 0.0, 0.0, 0.0, check);  
  ```

Example: Checker Board

```c
GLubyte wb[2] = {0 x 00, 0 x ff};
GLubyte check[4096];
int i, j;
for(i=0; i<64; i++) for (j=0; j<64, j++)
  check[i*8+j] = wb[(i/8+j/8)%2];
glBitmap(64, 64, 0.0, 0.0, 0.0, 0.0, check);
```

Pixel Maps
- OpenGL works with rectangular arrays of pixels called pixel maps or images
- Pixels are in one byte chunks
  - Luminance (gray scale) images 1 byte/pixel
  - RGB 3 bytes/pixel
- Three functions
  - Draw pixels: processor memory to frame buffer
  - Read pixels: frame buffer to processor memory
  - Copy pixels: frame buffer to frame buffer
OpenGL Pixel Functions

`glReadPixels(x, y, width, height, format, type, myimage)`
- start pixel in frame buffer
- size
- type of pixels
- type of image
- pointer to processor
- memory

```c
GLubyte myimage[512][512][3];
glReadPixels(0, 0, 512, 512, GL_RGB,
GL_UNSIGNED_BYTE, myimage);
glDrawPixels(width, height, format, type, myimage)
```

start at current raster position

Formats & Types

- `GL_RGB`
- `GL_RGBA`
- `GL_RED`
- `GL_GREEN`
- `GL_BLUE`
- `GL_ALPHA`
- `GL_DEPTH_COMPONENT`
- `GL_LUMINANCE`
- `GL_LUMINANCE_ALPHA`
- `GL_COLOR_INDEX`
- `GL_STENCIL_INDEX`
- `GL_UNSIGNED_BYTE`
- `GL_UNSIGNED_BYTE_3_3_2`
- `GL_UNSIGNED_SHORT_4_4_4_4`
- `GL_UNSIGNED_SHORT_5_6_5`
- `GL_UNSIGNED_SHORT_5_5_5_1`
- `GL_UNSIGNED_SHORT_4_4_4_4_RGB`
- `GL_UNSIGNED_SHORT_1_5_3_FP`
- `GL_UNSIGNED_INT_8_8_8_8_FLOAT`
- `GL_UNSIGNED_INT_10_10_10_2_FLOAT`
- `GL_UNSIGNED_INT_2_10_10_10_REV`
- `GL_UNSIGNED_INT_8_8_8 UNSIGNED_INT_3_3_2`
- etc.

Image Formats

- We often work with images in a standard format (JPEG, TIFF, GIF)
- How do we read/write such images with OpenGL?
- No support in OpenGL
  - OpenGL knows nothing of image formats
  - Some code available on Web
  - Can write readers/writers for some simple formats in OpenGL

Displaying a PPM Image

- PPM is a very simple format
- Each image file consists of a header followed by all the pixel data

```plaintext
P3
# comment 1
# comment 2
# comment n
rows columns maxvalue
pixels
```
Reading the Header

```c
FILE *fd;
int k, nm;
char c;
int i;
char b[100];
float s;
int red, green, blue;
printf("enter file name\n");
scanf("%s", b);
fd = fopen(b, "r");
fscanf(fd, "%[^\n] \", b);
if(b[0] != 'P' || b[1] != '3'){
    printf("%s is not a PPM file!\n", b);
    exit(0);
}
printf("%s is a PPM file\n", b);
```

check for “P3” in first line

Reading the Data

```c
fscanf(fd, "%d %d %d", &n, &m, &k);
printf("%d rows  %d columns  max value= %d\n", n, m, k);
nm = n*m;
image = malloc(3*sizeof(GLuint)*nm);
s = 255./k;
for(i=0;i<nm;i++){
    fscanf(fd, "%d %d %d", &red, &green, &blue);
    image[3*nm-i-3*0]=red;
    image[3*nm-i-2]=green;
    image[3*nm-i-1]=blue;
}
da吃饭 n=scale factor
```

Scaling the Image Data

We can scale the image in the pipeline

```c
glPixelTransferf(GL_RED_SCALE, s);
glPixelTransferf(GL_GREEN_SCALE, s);
glPixelTransferf(GL_BLUE_SCALE, s);
```

We may have to swap bytes when we go from processor memory to the frame buffer depending on the processor. If so we can use

```c
glPixelStorei(GL_UNPACK_SWAP_BYTES, GL_TRUE);
```

The display callback

```c
void display()
{
glClear(GL_COLOR_BUFFER_BIT);
glRasterPos2i(0,0);
glDrawPixels(n,m,GL_RGB,
GL_UNSIGNED_INT, image);
glFlush();
}
```

Compositing and Blending
Objectives

- Learn to use the A component in RGBA color for:
  - Blending for translucent surfaces
  - Compositing images
  - Antialiasing

Opacity and Transparency

- Opaque surfaces permit no light to pass through
- Transparent surfaces permit all light to pass
- Translucent surfaces pass some light
  \[ \text{translucency} = 1 - \text{opacity} (\alpha) \]

Physical Models

- Dealing with translucency in a physically correct manner is difficult due to:
  - the complexity of the internal interactions of light and matter
  - Using a pipeline renderer
  - Revert to writing model

Writing Model

- Use A component of RGBA (or RGB\(\alpha\)) color to store opacity
- During rendering we can expand our writing model to use RGBA values

Blending Equation

- We can define source and destination blending factors for each component
  \[ \mathbf{b} = [b_r, b_g, b_b, b_\alpha] \]
  \[ \mathbf{c} = [c_r, c_g, c_b, c_\alpha] \]
- source and destination colors
  \[ \mathbf{s} = [s_r, s_g, s_b, s_\alpha] \]
  \[ \mathbf{d} = [d_r, d_g, d_b, d_\alpha] \]
- Blend as
  \[ \text{pixel} = \left( \begin{array}{c} b_r s_r + c_r d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_\alpha s_\alpha + c_\alpha d_\alpha \end{array} \right) \]

OpenGL Blending and Compositing

- Must enable blending and pick source and destination factors
  \[ \text{glEnable(GL_BLEND)} \]
  \[ \text{glBlendFunc(source\_factor, destination\_factor)} \]
- Only certain factors supported:
  - GL_ZERO, GL_ONE
  - GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA
  - GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA
  - See Redbook for complete list
Example

• Suppose that we start with the opaque background color \((R_0, G_0, B_0, 1)\)
  – This color becomes the initial destination color
• We now want to blend in a translucent polygon with color \((R_1, G_1, B_1, \alpha_1)\)
• Select \(GL\_SRC\_ALPHA\) and \(GL\_ONE\_MINUS\_SRC\_ALPHA\) as the source and destination blending factors
  \[ R'_1 = \alpha_1 R_1 + (1-\alpha_1) R_0 \]
• Note this formula is correct if polygon is either opaque or transparent

Clamping and Accuracy

• All the components (RGBA) are clamped and stay in the range (0,1)
• However, in a typical system, RGBA values are only stored to 8 bits
  – Can easily loose accuracy if we add many components together
  – Example: add together n images
    • Divide all color components by \(n\) to avoid clamping
    • Blend with source factor = 1, destination factor = 1
    • But division by \(n\) loses bits

Order Dependency

• Is this image correct?
  – Probably not
  – Polygons are rendered in the order they pass down the pipeline
  – Blending functions are order dependent

Opaque and Translucent Polygons

• Suppose that we have a group of polygons some of which are opaque and some translucent
• How do we use hidden-surface removal?
  • Opaque polygons block all polygons behind them and affect the depth buffer
  • Translucent polygons should not affect depth buffer
    – Render with \(gl\_Depth\_Mask(GL\_FALSE)\) which makes depth buffer read-only
• Sort polygons first to remove order dependency

Fog

• We can composite with a fixed color and have the blending factors depend on depth
  – Simulates a fog effect
• Blend source color \(C_s\) and fog color \(C_f\) by
  \[ C'_s = f C_s + (1-f) C_f \]
• \(f\) is the fog factor
  – Exponential
  – Gaussian
  – Linear (depth cueing)

Fog Functions

\[ \text{Altenuation} \]

\[ 1 - 0.5z \]

\[ z \]
OpenGL Fog Functions

GLfloat fcolor[4] = {...};
glEnable(GL_FOG);
// fog function
glFogf(GL_FOG_MODE, GL_EXP);
// fog function parameter
glFogf(GL_FOG_DENSITY, 0.5);
glFogv(GL_FOG, fcolor);
// near distance to start fog
glFogf(GL_FOG_START, 1.5);
// far distance to stop fog
glFogf(GL_FOG_END, 10.5);

Fog Effect

Line Aliasing

• Ideal raster line is one pixel wide
• All line segments, other than vertical and horizontal segments, partially cover pixels
• Simple algorithms color only whole pixels
• Lead to the "jaggies" or aliasing
• Similar issue for polygons

Antialiasing by Area Averaging

• Pixel shade is proportional to percentage of pixel covered by ideal line

Antialiasing

• Can try to color a pixel by adding a fraction of its color to the frame buffer
  – Fraction depends on percentage of pixel covered by fragment
  – Fraction depends on whether there is overlap

Area Averaging

• Use average area $\alpha_1 + \alpha_2$ as blending factor
OpenGL Antialiasing

- Can enable separately for points, lines, or polygons
  ```
  glEnable(GL_POINT_SMOOTH);
  glEnable(GL_LINE_SMOOTH);
  glEnable(GL_POLYGON_SMOOTH);
  ```
- Pixels along edges are assigned fractional alpha values
  ```
  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  ```

Accumulation Buffer

- Compositing and blending are limited by resolution of the frame buffer
  - Typically 8 bits per color component
- The accumulation buffer is a high resolution buffer (16 or more bits per component) that avoids loss of precision from multiple ops
- Write into it or read from it with a scale factor
- Slower than direct compositing into the frame buffer
  ```
  glAccum(GLenum operation, GLfloat value);
  ```

Specifying Drawing Buffer(s)

- `glDrawBuffer(Glenum mode);`
- Specifies up to four color buffers to be drawn into
- `mode`
  - GL_NONE, GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT, GL_FRONT, GL_BACK, GL_LEFT, GL_RIGHT, GL_FRONT_AND_BACK, GL_AUXi

Applications

- Compositing
- Image Filtering (convolution)
- Whole scene antialiasing
- Motion effects
- Depth-of-field effects

Antialiasing

- `glDrawBuffer(GL_AUX0);`
- `glReadBuffer(GL_AUX0);`
- Loop $n$ time
  - Jitter image plane & draw
    - Less than one pixel
      - `glAccum(GL_ACCUM, 1.0/n);`
      - `glDrawBuffer(GL_BACK);`
      - `glAccum(GL_RETURN, 1.0);`
Interactive Depth-of-Field

- Jitter camera
- Each frustum has common plane “in focus”
- Accumulate images

OpenGL Perspective

`glFrustum(xmin, xmax, ymin, ymax, near, far)`

Using Field of View

- With `glFrustum` it is often difficult to get the desired view
- `gluPerspective(fovy, aspect, near, far)` often provides a better interface

OpenGL Perspective

- `glFrustum` allows for an unsymmetric viewing frustum
- `gluPerspective` does not

Go to `dof.c`
Stencil Buffer

- Specifies which pixels to draw
- Create arbitrary shaped viewports
- `glStencilFunc()` - sets comparison function
- `glStencilOp()` sets stencil operations

Reflections

- One of the most noticeable effects of inter-object lighting
- Direct calculation of the physics (ray tracing) is too expensive
- Our focus is to capture the most significant reflection while minimizing the overhead via rendering the "virtual object"

Image vs. Object Space Methods

- Image space methods: create a texture from a view of the reflected objects and apply it to the reflector
  - Advantage: does not depend on the object geometry
  - Disadvantage: sampling issue and also only an approximation (environment mapping as an example)
- Object space methods: create the actual geometry of the object to be reflected and render it to the reflector
  - Advantage: more accurate reflection for nearby objects
- Both methods need to create the virtual objects

Planar Reflections

- The most common reflection – flat mirror, floor, wall, etc
- Creating virtual objects (or reflected objects) is much easier
- A view independent operation – only consider the relative position of the object and the reflector
- The virtual object is created by transforming the object across the reflector plan

Planar Reflections

- An important task: clip the reflected geometry so it is only visible on the reflector surface
  - Beyond the reflector boundaries and in front of reflector
Clipping using the stencil

- The key is you only want the reflected geometry to appear on the reflector surface.
- Use stencil buffer:
  - Clear the stencil buffer
  - Render the reflector and set the stencil
  - Render the reflected geometry only where the stencil pixels have been set
- The above algorithm is to use the stencil buffer to control where to draw the reflection.

Another method: render the reflected object first, and then render the reflector to set the stencil buffer, then clear the color buffer everywhere except where the stencil is set.

- This method is to use the stencil buffer to control where to erase the incorrect reflection.
- Advantage: when it is faster to use stencil to control clearing the scene than drawing the entire scene with stencil tests.

The stencil erase algorithm

![Stencil Erase Algorithm Image]

Reflection Effect

![Reflection Effect Image]

http://www.canny.org.uk