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
Buffer

Define a buffer by its spatial resolution \((n \times m)\) and its depth \(k\), the number of bits/pixel.
OpenGL Frame Buffer
OpenGL Buffers

• 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

![Diagram showing writing into frame buffer](image-url)
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 Pipeline Diagram](image-url)
Pixel Storage Modes

• Deal with byte-ordering conventions
• Memory alignment

```
glPixelStore[if](GLenum pname, TYPE param);
pname:
  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,
  GL_UNPACK_ALIGNMENT
```
Pixel Transfer Operations

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

```c
GLfloat s, b;

glPixelTransferf(GL_RED_SCALE, s);
glPixelTransferf(GL_BLUE_BIAS, b);
```
Pixel Mapping Operations

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

```c
glPixelMap[ui us f]v(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

\[ \text{glLogicOp(GLenum \ op)} \]
Logical 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>s</td>
</tr>
<tr>
<td>GL_COPY_INVERTED</td>
<td>~s</td>
</tr>
<tr>
<td>GL_NOOP</td>
<td>d</td>
</tr>
<tr>
<td>GL_INVERT</td>
<td>~d</td>
</tr>
<tr>
<td>GL_AND</td>
<td>s &amp; d</td>
</tr>
<tr>
<td>GL_NAND</td>
<td>~(s &amp; d)</td>
</tr>
<tr>
<td>GL_OR</td>
<td>s</td>
</tr>
<tr>
<td>GL_NOR</td>
<td>~(s</td>
</tr>
<tr>
<td>GL_XOR</td>
<td>s ^ d</td>
</tr>
<tr>
<td>GL_EQUIV</td>
<td>~(s ^ d)</td>
</tr>
<tr>
<td>GL_AND_REVERSE</td>
<td>s &amp; ~d</td>
</tr>
<tr>
<td>GL_AND_INVERTED</td>
<td>~(s &amp; d)</td>
</tr>
<tr>
<td>GL_OR_REVERSE</td>
<td>s</td>
</tr>
<tr>
<td>GL_OR_INVERTED</td>
<td>~(s</td>
</tr>
</tbody>
</table>
Writing Modes

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

<table>
<thead>
<tr>
<th>s</th>
<th>d</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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</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>1</td>
<td>0</td>
<td>1</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

\[ S \leftarrow S_0 \]
\[ M \leftarrow S \oplus M \]
\[ S \leftarrow S \oplus M \]
\[ S \leftarrow S \oplus M \]

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_BIT_MAP_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)

offset from raster position

increments in raster position after bitmap drawn

first raster position

second raster position

Angel: Interactive Computer Graphics 3E
© Addison-Wesley 2002
Example: Checker Board

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, 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

\[ \text{glReadPixels}(x, y, width, height, format, type, myimage) \]

- start pixel in frame buffer
- size
- type of image
- type of pixels
- pointer to processor memory

```
GLubyte myimage[512][512][3];
glReadPixels(0, 0, 512, 512, GL_RGB,
             GL_UNSIGNED_BYTE, myimage);
```

```
glDrawPixels(width, height, format, type, myimage)
```

starts at current raster position
OpenGL Pixel Functions

\texttt{glReadPixels(x, y, width, height, format, type, myimage)}

- \texttt{x, y}: start pixel in frame buffer
- \texttt{width, height}: size
- \texttt{format, type}: type of image, type of pixels
- \texttt{myimage}: pointer to processor memory

\texttt{GLubyte myimage[512][512][3];}
\texttt{glReadPixels(0, 0, 512, 512, GL_RGB,}
\texttt{ GL_UNSIGNED_BYTE, myimage);};

\texttt{glDrawPixels(width, height, format, type, myimage)}

starts at current raster position
OpenGL Pixel Functions

\[ \text{glReadPixels}(x, y, width, height, format, type, myimage) \]

- Start pixel in frame buffer
- Size
- Type of image
- Type of pixels
- Pointer to processor memory

\[
\text{GLubyte myimage[786432];}
\]

\[
\text{glReadPixels(0,0, 512, 512, GL_RGB, GL_UNSIGNED_BYTE, myimage);}\]

\[
\text{glDrawPixels(width, height, format, type, myimage)}\]

starts 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_BYTE
- GL_BITMAP
- GL_UNSIGNED_SHORT
- GL_SHORT
- GL_UNSIGNED_INT
- GL_INT
- GL_FLOAT
- GL_UNSIGNED_BYTE_3_3_2
- GL_UNSIGNED_INT_8_8_8_8
- 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
• Header

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

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);
Reading the Header (cont)

```c
fscanf(fd, "%c", &c);
while (c == '#') {
    fscanf(fd, "^[^\n] ", b);
    printf("\%s\n", b);
    fscanf(fd, "%c", &c);
}
ungetc(c, fd);

skip over comments by
looking for # in first column
```
Reading the Data

```c
fscanf(fd, "%d %d %d", &n, &m, &k);
printf("%d rows  %d columns  max value= %d
",n,m,k);

nm = n*m;
image=malloc(3*sizeof(GLuint)*nm);
s=255./k;  /* scale factor */

for(i=0;i<nm;i++)
{
    fscanf(fd,"%d %d %d",&red, &green, &blue );
    image[3*nm-3*i-3]=red;
    image[3*nm-3*i-2]=green;
    image[3*nm-3*i-1]=blue;
}
```
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 need can use

```c
glPixelStorei(GL_UNPACK_SWAPBYTES, 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
  translucency = 1 – opacity ($\alpha$)

opaque surface $\alpha = 1$
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

\[ b = [b_r, b_g, b_b, b_\alpha] \]
\[ c = [c_r, c_g, c_b, c_\alpha] \]

source and destination colors

\[ s = [s_r, s_g, s_b, s_\alpha] \]
\[ d = [d_r, d_g, d_b, d_\alpha] \]

Blend as

\[ \text{pixel} = [b_r s_r + c_r d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_\alpha s_\alpha + c_\alpha d_\alpha] \]
OpenGL Blending and Compositing

- Must enable blending and pick source and destination factors
  
  ```
  glEnable(GL_BLEND)
  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
  \[ R'_1 = \alpha_1 R_1 + (1 - \alpha_1) R_0, \ldots \]
• 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 lose 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 `glDepthMask(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

\[ e^{-z^2} \]

\[ 1 - 0.5z \]

Attenuation vs Distance
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

http://www.engin.swarthmore.edu/~jshin1
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

![Antialiasing Diagram](image-url)
Area Averaging

• Use average area $\alpha_1 + \alpha_2 - \alpha_1 \alpha_2$ 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`
Accumulation Buffer

• `glAccum(Glenum operation, GLfloat value);`

• Operation
  - `GL_ACCUM`: Multiplies by `value` then adds the RGBA values from the buffer selected for reading by `glReadBuffer` to the accumulation buffer
  - `GL_LOAD`: Multiplies by `value` then copies the RGBA values from the buffer selected for reading by `glReadBuffer` to the accumulation buffer
  - `GL_ADD`: Adds `value` to each RGBA in the accumulation buffer
  - `GL_MULT`: Multiplies each RGBA in the accumulation buffer by `value`
  - `GL_RETURN`: Transfers accumulation buffer values to the color buffer(s) currently selected for writing by `glDrawBuffer`
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
Interactive Depth-of-Field

Depth of Field Test

[+/-] focus: 176.00
[?] jitter: 2.00
press F to show the focal plane.
OpenGL Perspective

\texttt{glFrustum}(\texttt{xmin}, \texttt{xmax}, \texttt{ymin}, \texttt{ymax}, \texttt{near}, \texttt{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

![Diagram](image)

- `front plane`
- `aspect = w/h`
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
  - Disadvantage: accuracy of the geometry
  - 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

Figure 54. Mirror Reflection of the Viewpoint

Figure 55. Mirror Reflection of the Scene
Render the Reflected Geometry

• 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
Clipping using the stencil

- 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

Figure 57: Stencil Reflection Steps
Reflection Effect