Buffers, Textures, Compositing, and Blending

David Carr
Virtual Environments, Fundamentals
Spring 2005
Based on Slides by E. Angel

Overview

- Buffers
  - Additional OpenGL buffers
  - Reading and writing buffers
  - Blending
- Texture mapping
  - Mapping Methods
  - Texture mapping
  - Environmental Mapping
  - Bump Mapping
- Basic strategies
  - Forward versus backward mapping
  - Point sampling versus area averaging
- Compositing and Blending
  - Blending for translucent surfaces
  - Compositing images
  - Antialiasing

Buffers
**Buffer**

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

**OpenGL Buffers**

- **Color buffers**
  - Front
  - Back
  - Auxiliary
    - Undisplayed
    - System dependent
    - Overlay
    - Chroma key animation
- **Depth**
- **Accumulation**
  - High-resolution buffer
- **Stencil**
  - Holds masks
  - Chroma key animation

**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
Writing Model

- Read destination pixel before writing source

![Diagram of writing model](image)

Writing Modes

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

<table>
<thead>
<tr>
<th>a</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>d</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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>XOR</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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

XOR Mode

- Recall from Chapter 3 that we can use XOR by enabling logic operations and selecting the XOR write mode
- 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

\[
\begin{align*}
S &\leftarrow S \oplus M \\
M &\leftarrow S \oplus M \\
S &\leftarrow S \oplus M
\end{align*}
\]

swaps the S and M

![Diagram of XOR mode](image)
The Pixel Pipeline

- OpenGL has a separate pipeline for pixels
  - Writing pixels involves:
    - Moving pixels from processor memory to the frame buffer
    - Format conversions
    - Mapping, Lookups, Tests
  - Reading pixels
    - Format conversion

Raster Position

- Raster position is the association between the bitmap and object vertices
- OSG maintains raster position as part of the Geometry
  - Each vertex gets a corresponding 2D position in the bitmap
  - Geometry.setTexCoordArray(int unit, Vec2Array)
    - Each element in the Vec2Array is a bitmap coordinate (usually 0.-1.)
- The raster position is a geometric entity
  - Passes through geometric pipeline
  - Eventually yields a 2D position in screen coordinates
  - This position in the frame buffer is where the raster primitive is drawn

Buffer Selection

- OpenGL can draw into or read from any of the color buffers (front, back, auxiliary)
- Default to the back buffer
- 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 fonts
- Bitmaps can have a color associated with them

Drawing Bitmaps

```
gl.glBitmap(width, height, x0, y0, xi, yi, bitmap)
```

- First raster position
- Second raster position
- Offset from raster position
- Increments in raster position after bitmap drawn

Pixel Maps

- OpenGL works with rectangular arrays of pixels called pixel maps or images
- JavaOSG uses an Image class
- Pixels are in one byte (8 bit) chunks
  - Luminance (gray scale) images 1 byte/pixel
  - RGB 3 bytes/pixel
- Image read/write via supported plug-ins
  - Formats: JPEG, PNG, GIF, …
Texture Mapping

Application

• Want to:
  - Map realistic variation onto an object
  - Limit model complexity
• Solution directly map a digital pattern to an object
• Example, mapping reflections to Geri’s glasses

The Limits of Geometric Modeling

• Graphics cards can render over 10 million polygons per second
• But, that is insufficient for many phenomena
  - Clouds
  - Grass
  - Terrain
  - Skin
Modeling an Orange

- Consider the problem of modeling an orange (the fruit)
- Start with an orange-colored sphere
  - Too simple
- Replace sphere with a more complex shape
  - Does not capture surface characteristics (small dimples)
  - Takes too many polygons to model all the dimples

Modeling an Orange (Alternate)

- Take a picture of a real orange, scan it, and "paste" onto simple geometric model
  - This process is texture mapping
- Still might not be sufficient because resulting surface will be smooth
  - Need to change local shape
  - Bump mapping

Three Types of Mapping

- Texture Mapping
  - Uses images to fill inside of polygons
- Environmental (reflection mapping)
  - Uses a picture of the environment for texture maps
  - Allows simulation of highly specular surfaces
- Bump mapping (Note! doesn’t work in JavaOSG)
  - Emulates altering normal vectors during the rendering process
Where Does Mapping Take Place?

- Mapping techniques are implemented at the end of the rendering pipeline
  - Very efficient because few polygons pass down the geometric pipeline

Is It Simple?

- Although the idea is simple
  - Map an image to a surface
  - There are 3 or 4 coordinate systems involved

Coordinate Systems

- Parametric coordinates
  - May be used to model curved surfaces
- Texture coordinates
  - Used to identify points in the image to be mapped
- World Coordinates
  - Conceptually, where the mapping takes place
- Screen Coordinates
  - Where the final image is really produced
Texture Mapping

- parametric coordinates
- texture coordinates
- world coordinates
- screen coordinates

Mapping Functions

- Basic problem is how to find the maps
- Consider mapping from texture coordinates to a point on a surface
- Appear to need three functions:
  \[ x = x(s, t) \]
  \[ y = y(s, t) \]
  \[ z = z(s, t) \]
- But we really want to go the other way:
  \( (x, y, z) \)

Backward Mapping

- We really want to go backwards
  - Given a pixel, we want to know to which point on an object it corresponds
  - Given a point on an object, we want to know to which point in the texture it corresponds
  - Need a map of the form:
    \[ s = s(x, y, z) \]
    \[ t = t(x, y, z) \]
- Such functions are difficult to find in general
Two-Part Mapping

- One solution to the mapping problem is to first map the texture to a simple intermediate surface
- Example: map to cylinder

Cylindrical Mapping

- Parametric cylinder
  \[ x = r \cos 2\pi u \]
  \[ y = r \sin 2\pi u \]
  \[ z = v h \]
- Maps rectangle in u,v space to cylinder of:
  - Radius \( r \) and height \( h \) in world coordinates
    \[ s = u \]
    \[ t = v \]
- Maps from texture space

Spherical Map

- We can similarly use a parametric sphere
  \[ x = r \cos 2\pi u \cos 2\pi v \]
  \[ y = r \sin 2\pi u \cos 2\pi v \]
  \[ z = r \sin 2\pi u \sin 2\pi v \]
- But, must decide where to put the distortion
- Spheres are used in environmental maps
Box Mapping

- Easy to use with simple orthographic projection
- Also used in environmental maps

Second Mapping

- Map from intermediate object to actual object
  - Normal vectors from intermediate to actual
  - Normal vectors from actual to intermediate
  - Vectors from center of intermediate

Aliasing

- Point sampling of the texture can lead to aliasing errors
Area Averaging

- A better but slower option is to use area averaging
- Note that pre image of pixel is curved

osgExample

- Remember TextureDemo.java
- The tower bitmap is mapped to the pyramid
- Five vertices to image locations:
  - 0 \rightarrow (0.00, 0.00)
  - 1 \rightarrow (0.25, 0.00)
  - 2 \rightarrow (0.50, 0.00)
  - 3 \rightarrow (0.75, 0.00)
  - 4 (tip) \rightarrow (0.50, 1.00)

Another osgExample

- LightDemo2.java
Compositing and Blending

Overview

• The $\alpha$ component in RGB$\alpha$ 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 (RGBα) 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
  \[ s = [s_r, s_g, s_b, s_\alpha] \]
  \[ d = [d_r, d_g, d_b, d_\alpha] \]
- Source and destination colors
  \[ b = [b_r, b_g, b_b, b_\alpha] \]
  \[ c = [c_r, c_g, c_b, c_\alpha] \]
- Blend as
  \[ c' = [b_r s_r + c_r d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_\alpha s_\alpha + c_\alpha d_\alpha] \]
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: $\alpha$ as the source blending factor
    $(1-\alpha)$ as the destination blending factor
  • Note this formula is correct even 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 `gl.glDepthMask(GL.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

![Fog Functions Diagram]
Fog Effects

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

- 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

no overlap

overlap
Area Averaging

• Use average area $\alpha_1 + \alpha_2 - \alpha_1 \cdot \alpha_2$ as blending factor

Accumulation Buffer

• Compositing and blending are limited by resolution of the frame buffer
  - Typically 6 bits per color component
• The accumulation buffer is a high resolution buffer (16 or more bits per component) that avoids this problem
• Write into it or read from it with a scale factor
• Slower than direct compositing into the frame buffer

Applications

• Composites
• Image Filtering (convolution)
• Whole scene anti-aliasing
• Motion effects
Summary

- Buffers
  - Additional OpenGL buffers
  - Reading and writing buffers
  - Blending
- Texture mapping
  - Mapping Methods
  - Texture mapping
  - Environmental Mapping
  - Bump Mapping
- Basic strategies
  - Forward versus backward mapping
  - Point sampling versus area averaging
- Compositing and Blending
  - Blending for translucent surfaces
  - Compositing images
  - Antialiasing

Questions?