CS 418: Interactive Computer Graphics

Compositing & Blending in WebGL

Eric Shaffer
Lynwood Dunn (1904-1998)

- Visual effects pioneer
- Acme-Dunn optical printer

Run film through a projector and re-photograph it. Can zoom in or out, applies filters etc.
Compositing Example
Academy of Motion Picture Arts & Sciences
Scientific and Engineering Award
To Alvy Ray Smith, Tom Duff, Ed Catmull and Thomas Porter for their Pioneering Inventions in **Digital Image Compositing**.
PRESENTED MARCH 2, 1996
The Over Operator

- Use alpha channel to indicate opacity [Smith]
- Over operator [Porter & Duff S’84]

A over B:

\[
C_{A \over B} = \alpha_A C_A + (1 - \alpha_A) \alpha_B C_B \\
\alpha_{A \over B} = \alpha_A + (1 - \alpha_A) \alpha_B
\]

Alternatively (and better?), you can pre-multiply the colors \(C_A\) and \(C_B\) by the alpha value

\[
C = (\alpha R, \alpha G, \alpha B, \alpha)
\]

A over B w/premultiplied alpha

\[
C_{A \over B} = C_A + (1 - \alpha_A) C_B \\
\alpha_{A \over B} = \alpha_A + (1 - \alpha_A) \alpha_B
\]

Pre-multiplied alpha and Post-multiplied alpha are not equivalent. You will usually get similar results, but not in all situations.
Is Over Associative?

\[ A \text{ over } (B \text{ over } C) = C_A + (1-\alpha_A)(C_B + (1-\alpha_B)C_C) \]
Is Over Associative?

- A over (B over C)
  - \( = C_A + (1-\alpha_A)(C_B + (1-\alpha_B)C_C) \)
  - \( = C_A + (1-\alpha_A)C_B + (1-\alpha_A)(1-\alpha_B)C_C \)
  - \( = C_{AB} + (1 - \alpha_A - (1-\alpha_A)\alpha_B)C_C \)
  - \( = C_{AB} + (1-\alpha_{AB})C_C \)
  - \( = (A \text{ over } B) \text{ over } C \)

- What about \( \alpha \)
  - \( = \alpha_A + (1-\alpha_A) \alpha_{BC} \)
  - \( = \alpha_A + (1-\alpha_A)(\alpha_B + (1-\alpha_B) \alpha_C) \)
  - \( = \alpha_A + (1-\alpha_A)\alpha_B + (1-\alpha_A)(1-\alpha_B)\alpha_C \)
  - \( = \alpha_{AB} + (1-\alpha_{AB})\alpha_C \)
Questions....

- Is post-multiplied alpha associative?
- Is the over operator commutative?
Questions....

- Is post-multiplied alpha associative?
  NO

- Is the over operator commutative?
  NO
Accumulating Opacity

- What if you have multiple layers of surfaces to blend?
- Have to work in sorted order

Back to front: Over operator

\[ C_{out} = C + (1 - \alpha) C_{in} \]
\[ \alpha_{out} = \alpha + (1 - \alpha) \alpha_{in} \]

Could also work front to back…
Hidden Surface Removal

- Hidden Surface Removal
  - ...don’t render surfaces occluded by surfaces in front of them

- Was a significant area of research in early days of CG
  - ...lots of algorithms suggested

- Painter’s Algorithm
  - Render objects in order from back to front
    - i.e. sort your triangles by depth and render deepest first
  - Can anyone imagine any problems with this approach?
Problems with the Painter’s Algorithm

- No correct rendering order for
  - intersecting triangle
  - occlusion cycles

- Sorting is slow...too slow for interactivity in complex scenes
Hidden Surface Removal: Z-Buffer

Key Observation: Each pixel displays color of only one triangle, ignores everything behind it

- Don’t need to sort triangles, just find for each pixel the closest triangle
- Z-buffer: one fixed or floating point value per pixel
- Algorithm:
  For each rasterized fragment \((x,y)\)
  
  If \(z < z\text{buffer}(x,y)\) then
  
  \[
  \text{framebuffer}(x,y) = \text{fragment color}
  \]
  
  \[
  z\text{buffer}(x,y) = z
  \]

Frame Buffer: buffer that stores the colors for the pixels we will render
Z-Buffer

Key Observation: Each pixel displays color of only one triangle, ignores everything behind it
- Don’t need to sort triangles, just find for each pixel the closest triangle
- Z-buffer: one fixed or floating point value per pixel
- Algorithm:
  For each rasterized fragment \((x,y)\)
    If \(z < \text{zbuffer}(x,y)\) then
      \[
      \text{framebuffer}(x,y) = \text{fragment color}
      \]
      \[
      \text{zbuffer}(x,y) = z
      \]
Z-Buffer

Key Observation: Each pixel displays color of only one triangle, ignores everything behind it

- Don’t need to sort triangles, just find for each pixel the closest triangle
- Z-buffer: one fixed or floating point value per pixel
- Algorithm:
For each rasterized fragment \((x,y)\)
  
  If \(z > \text{zbuffer}(x,y)\) then
    
    \[
    \text{framebuffer}(x,y) = \text{fragment color} \\
    \text{zbuffer}(x,y) = z
    \]
Z-Buffer

- Get fragment z-values by interpolating z-values at vertices during rasterization
- True perspective projection destroys z-values, setting them all to \(-d\)
- The perspective distortion we use preserves at least the ordering of z-values
In practice, depths values are typically converted to non-negative integers when stored in the z-buffer.

- Comparison operation needs to be fast...

Imagine having depth values of \{0,1,...,B-1\}

- 0 \rightarrow\text{near clipping plane distance}
- B-1 \rightarrow\text{far clipping plane distance}

Depths occur discretely in “buckets”

- Each bucket covers a range of length \(\Delta z = \frac{f-n}{B}\)

If we use \(b\) bits for the z-buffer values, \(B = 2^b\)

- You usually can’t change the value \(b\)
- To maximize z-buffer effectiveness, **need to minimize** \(f-n\)
Z-Fighting

How can you fix z-fighting?
How can you fix z-fighting?

1. Move co-planar polygons slightly away from each other
2. Move near and far clipping planes as close together as you can
Order Independent Transparency

- Alpha blending works for sorted rendering
  - Front to back
  - Back to front
- Doesn’t work for out-of-order
  - Front, back, middle
- Could need to keep track separately of the front part and the back part
- Could keep a linked list at each pixel
  - A-buffer (Carpenter)
  - Not practical for hardware
Depth Peeling

- Cass Everett, NVIDIA Tech Rep, 2001
- Needs 2 z-buffers (previous, current)
- One rendering pass per layer
- Fragment written to frame buffer if
  - Farther than previous z-buffer
  - Closer than current z-buffer
- After each pass, current z-buffer written to previous z-buffer
- Surviving fragment composited “under” displayed fragment
Depth Peels – Which Layer is Which?
Scissor Test:
cull pixels outside of a rectangular area

Multisample:
anti-aliasing operation

Stencil Test:
uses a stencil buffer to mask pixels
can be used in shadow generation

Depth Buffer Test:
hidden surface removal

Blending:
compositing using alpha channel
Hidden surface removal uses the depth buffer (z-buffer)

Happens after the fragment shader
Blending

//enable blending
gl.Enable(gl.BLEND)

//to set up the parameters of the generic blending equation
//call ONE of the functions below

//OR

WebGL lets you specify the factors and operations in the generic blending equation:

\[ \text{color}_{\text{final}} = \text{factor}_{\text{source}} \times \text{color}_{\text{source}} \text{ op } \text{factor}_{\text{dest}} \times \text{color}_{\text{dest}} \]
**Blending**

\[
\text{color}_{\text{final}} = \text{factor}_{\text{source}} \times \text{color}_{\text{source}} \; \text{op} \; \text{factor}_{\text{dest}} \times \text{color}_{\text{dest}}
\]

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>RGB FACTORS</th>
<th>BLEND FACTOR</th>
<th>ALPHA BLEND FACTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>gl.ZERO</td>
<td>(0, 0, 0)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>gl.ONE</td>
<td>(1, 1, 1)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>gl.SRC_COLOR</td>
<td>(R_s, G_s, B_s)</td>
<td>A_s</td>
<td></td>
</tr>
<tr>
<td>gl.ONE_MINUS_SRC_COLOR</td>
<td>(1, 1, 1) - (R_s, G_s, B_s)</td>
<td>1 - A_s</td>
<td></td>
</tr>
<tr>
<td>gl.DST_COLOR</td>
<td>(R_d, G_d, B_d)</td>
<td>A_d</td>
<td></td>
</tr>
<tr>
<td>gl.ONE_MINUS_DST_COLOR</td>
<td>(1, 1, 1) - (R_d, G_d, B_d)</td>
<td>1 - A_d</td>
<td></td>
</tr>
<tr>
<td>gl.SRC_ALPHA</td>
<td>(A_s, A_s, A_s)</td>
<td>A_s</td>
<td></td>
</tr>
<tr>
<td>gl.ONE_MINUS_SRC_ALPHA</td>
<td>(1, 1, 1) - (A_s, A_s, A_s)</td>
<td>1 - A_s</td>
<td></td>
</tr>
<tr>
<td>gl.DST_ALPHA</td>
<td>(A_d, A_d, A_d)</td>
<td>A_d</td>
<td></td>
</tr>
<tr>
<td>gl.ONE_MINUS_DST_ALPHA</td>
<td>(1, 1, 1) - (A_d, A_d, A_d)</td>
<td>1 - A_d</td>
<td></td>
</tr>
<tr>
<td>gl.CONSTANT_COLOR</td>
<td>(R_c, G_c, B_c)</td>
<td>A_c</td>
<td></td>
</tr>
<tr>
<td>gl.ONE_MINUS_CONSTANT_COLOR</td>
<td>(1, 1, 1) - (R_c, G_c, B_c)</td>
<td>1 - A_c</td>
<td></td>
</tr>
<tr>
<td>gl.CONSTANT_ALPHA</td>
<td>(A_c, A_c, A_c)</td>
<td>A_c</td>
<td></td>
</tr>
<tr>
<td>gl.ONE_MINUS_CONSTANT_ALPHA</td>
<td>(1, 1, 1) - (A_c, A_c, A_c)</td>
<td>1 - A_c</td>
<td></td>
</tr>
<tr>
<td>gl.SRC_ALPHA_SATURATE</td>
<td>(f, f, f)</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Changing the Blending Operator

```glsl
gl.blendEquation(Glenum mode);

Lets you specify the blending operation. Addition is the default.

// colorfinal = factorsource × colorsource + factordest × colordest
gl.blendEquation(GL_FUNC_ADD);

// colorfinal = factorsource × colorsource − factordest × colordest
gl.blendEquation(GL_FUNC_SUBTRACT);

// colorfinal = factordest × colordest − factorsource × colorsource
gl.blendEquation(GL_FUNC_REVERSE_SUBTRACT);
```
Blending

```c
gl.blendFunc(GLenum sfactor, GLenum dfactor);
```

Lets you specify the blending function for both the RGB and Alpha values for both the source and destination

```c
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

Will implement the over operator we saw previously

You can use it to generate semi-transparent imagery

It’s the most commonly used formulation
Pre-multiplied Alpha

- Non-pre-multiplied alpha example: (1.0, 0.0, 0.0, 0.5)
- Pre-multiplied alpha example: (0.5, 0.0, 0.0, 0.5)
- For blending use:

```c
#include <GL/glew.h>

void clamp(float *v)
{
    v[0] = fmax(0.0, fmin(1.0, v[0]));
    v[1] = fmax(0.0, fmin(1.0, v[1]));
    v[2] = fmax(0.0, fmin(1.0, v[2]));
    v[3] = fmax(0.0, fmin(1.0, v[3]));
}

int main()
{
    glEnable(GL_BLEND);
    glBlendFunc(GL.ONE, GL.ONE_MINUS_SRC_ALPHA);
    clamp(
        (GLfloat *)
        (const GLfloat *)glReadPixels(0, 0, 1, 1, GL_RGBA, GL_FLOAT)
    );
    return 0;
}
```

- PNG images use non-pre-multiplied alpha
  - in case you are loading colors from an image
- You can choose to work either way…
Blending and Drawing Order

// 1. Enable depth testing, make sure the depth buffer is writable
// and disable blending before you draw your opaque objects.
gl.enable(gl.DEPTH_TEST);
gl.depthMask(true);
gl.disable(gl.BLEND);

// 2. Draw your opaque objects in any order (preferably sorted on state)

// 3. Keep depth testing enabled, but make depth buffer read-only
// and enable blending

gl.depthMask(false);
gl.enable(gl.BLEND);

// 4. Draw your semi-transparent objects back-to-front

// 5. If you have UI that you want to draw on top of your regular scene, you can finally disable depth testing

gl.disable(gl.DEPTH_TEST);

// 6. Draw any UI you want to be on top of everything else

---

Anyuru, Andreas
Professional WebGL Programming: Developing 3D Graphics for the Web
Blending in WebGL