Compositing Techniques

Richard (Hao) Zhang

Introduction to Computer Graphics
CMPT 361 – Lecture 16
Image compositing

- Purely image-space approaches
- Combine weighted pixel values (images) to form new pixel values (images)
- Discrete (image processing style) techniques used to generate certain effects
  - e.g., fog, translucency, motion blur, etc.
- Frequent use of the accumulation buffer
Why composition?

- One pixel may be covered by multiple polygons with different colors or at an edge (fore- and back-ground)
  - Lead to jagged edges
  - Really an under-sampling or aliasing problem
  - Major use of compositing is for anti-aliasing

- Can also generate certain interesting effects without (expensive) explicit rendering, e.g., **fog, motion blur**

- Natural to use when portion of the scene is available as an image, e.g., looking through a transparent glass
α channel

- Add a fourth component – the $\alpha$ value – to the RGB color representation of each pixel
  
  RGB $\rightarrow$ RGBA or RGB$\alpha$

- $\alpha$ values form the $\alpha$ channel

- $\alpha$ often represents coverage at a pixel — a weight:
  
e.g., red has $\alpha = 0.75$ coverage and green has $\alpha = 0.25$
Combine coverages

- Given polygon coverages with respect to a pixel
  - How much does green cover red and vice versa?
  - What is the coverage of the whole pixel?
  - It really depends …
Use rough approximation

- Too expensive to account for all possible situations

\[ \alpha_r + \alpha_g \]
\[ \alpha_g \]
\[ \alpha_r + \alpha_g - \alpha_r \alpha_g \]

- Just use \( \alpha_r + \alpha_g - \alpha_r \alpha_g \); the rest comes from background

- Assume coverage is **randomly distributed**, so given red coverage \( \alpha_r \) and green coverage \( \alpha_g \), then \( \alpha_g \) of red polygon is covered by green polygon
Pixel compositing

In very general terms, at a particular pixel, given

- **source color** \((s_r, s_g, s_b, s_\alpha)\), i.e., new color to be added or blended in

- **destination color** \((d_r, d_g, d_b, d_\alpha)\), i.e., the current color at the pixel, e.g., background

- a source blending factor \((p_r, p_g, p_b, p_\alpha)\)

- a destination blending factor \((q_r, q_g, q_b, q_\alpha)\),

the composite destination color is

\[
(p_r s_r + q_r d_r, p_g s_g + q_g d_g, p_b s_b + q_b d_b, p_\alpha s_\alpha + q_\alpha d_\alpha)
\]
Blending in OpenGL

- Specify colors using RGBA and turn on blending
  
  glEnable(GL_BLEND);

- Set up desired source and destination factors
  
  glBlendFunc(source_factor, destination_factor);
  
  - source pixel: pixel values currently computed
  
  - destination pixel: pixel value currently in frame buffer
Pixel compositing using $\alpha$

If $(r, g, b, \alpha)$ is the source color, then use $\alpha$ as the source blending factor and $(1 - \alpha)$ as the destination blending factor for colors — this keeps $(r, g, b)$ in range.

Example: **anti-aliased line rasterization**

- Default width of a line is 1 pixel
- Some pixels are partially covered
- If all covered pixels get a single color — jaggies
- Use coverage value $\alpha$ to blend line with background, the line would look smoother — anti-aliasing
Pixel compositing using $\alpha$

Other example: anti-aliased polygon rasterization

- Which color should the pixel $P$ get?

- If red – jaggies

- Should $\alpha$-blend the red with the green
Pixel compositing using $\alpha$

So the **coverage value** $\alpha$ in the RGBA value $(r, g, b, \alpha)$ really specifies how much contribution the color $(r, g, b)$ should make to the pixel color in the composite image.

Another use of $\alpha$ channel?
\( \alpha \) as an opacity value

- Alternatively, \( \alpha \) can be seen as opacity value
  - \( \alpha = 1 \) – the color is complete opaque, i.e., it would overwrite another color if this object is in front
  - \( \alpha = 0 \) – the color is completely transparent, i.e., we can completely see through it or overwrite it (e.g., when it is at the background)
  - \( 0 < \alpha < 1 \) – it blends with the existing color at the pixel – can simulate transparent object without ray tracing
Depth cueing and fog

- Depth cueing: color becomes dimmer as the distance from viewer increases – used for vector graphics
- Extend the idea of distance-dependent decay for colored fog effect using color compositing:

\[ C = fC + (1 - f)C_f \]

where \( C \) is the color of an object, \( C_f \) is the color of the fog, and \( f \) is a fog factor, e.g.,

\[ f = e^{-kz^2} \]

where \( z \) is the depth value in eye coordinates, \( k \) controls speed of decay. Other decay function possible.
Order matters in $\alpha$ blending

- The order of rendering matters.

Example:
- background: $C_0$
- Polygon $P_1$: $C_1$, $\alpha_1$
- Polygon $P_2$: $C_2$, $\alpha_2$

$P_1$ after $P_2$:
$$\alpha_1 C_1 + (1 - \alpha_1)(\alpha_2 C_2 + (1 - \alpha_2)C_0)$$

$P_2$ after $P_1$:
$$\alpha_2 C_2 + (1 - \alpha_2)(\alpha_1 C_1 + (1 - \alpha_1)C_0)$$
What does this mean?

- $\alpha$ blending can be used to simulate translucency, but it does not save us from doing visibility computations.

- On the other hand, translucent polygons should be treated differently from opaque ones in visibility since they are not entirely “blocking” other objects.

- Ideally
  - Render opaque polygons first
  - Then render translucent polygons in back-to-front order while keeping the depth buffer read-only: `glDepthMask(GL_FALSE)`
Use of accumulation buffer

- $\alpha$ blending manipulates the color buffer directly and can do this sequentially – computing pixel $P$ does not need to concern nearby pixels.

- However, it may be necessary to apply changes to an image simultaneously, where changes are based on old values, e.g., local averaging using blending.

- An additional buffer – the accumulation buffer – can be used for this, e.g., see glAccum() …
Accumulation buffer (A-buffer)

- Same resolution as the screen
- Part of the frame buffer, along with depth buffer, front and back color buffer, etc.
- Has more depth (can store more bits) than color buffer and typically used as temporary storage
  - Useful when composition in the color buffer overflows
  - Do this in A-buffer to avoid clamping
Image convolution

- Use the same **convolution mask** for local averaging in an image, e.g., smoothing or edge detection

$$H = \frac{1}{16} \begin{bmatrix} 0 & 2 & 0 \\ 2 & 8 & 2 \\ 0 & 2 & 0 \end{bmatrix}$$

A smoothing filter

$$H = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

An edge detection filter

- Accumulation buffer can be used to store partial results
  - e.g., accumulate scaled and shifted version of original image to perform convolution
Multisampling and rendering

- Use A-buffer to store jittered versions of a scene
- Combine them with original rendering of the scene: **multi-sampling or multi-rendering**
- Applications for motion blur and anti-aliasing
Multi-rendering I: anti-aliasing

- Perturb views slightly to get multiple sampling of a scene and combine renderings in the A-buffer
- This is referred to as the **jittering approach**
- The new samples should be very close to the original samples
- Blend the color buffer (original rendering) and result in the A-buffer to reduce aliasing artifacts
Multi-rendering II: motion blur

- Rendering scene multiple times with only one object moving along a path
- Accumulate scenes in A-buffer, weighted to produced dimmer versions of the jittered object
- Then combine with original rendering to get the effect of motion blur