Depth and Transparency

Computer Graphics
CMU 15-462/15-662
What we know how to do so far...

- Position objects in the world (3D transformations)
- Project objects onto the screen (perspective projection)
- Sample triangle coverage (rasterization)
- Put samples into frame buffer (depth & alpha)
- Interpolate vertex attributes (barycentric coordinates)

Today
Coverage \( (x,y) \)

Previously discussed how to sample coverage given the 2D position of the triangle's vertices.
What if our triangle is not all the same color (or any other property)?
Consider sampling color(x,y)

What is the triangle’s color at the point \( x \) ?

**Standard strategy:** interpolate color values at vertices.
Linear interpolation in 1D

Suppose we’ve sampled values of a function $f(x)$ at points $x_i$, i.e., $f_i := f(x_i)$

Q: How do we construct a function that “connects the dots” between $x_i$ and $x_{i+1}$?

$t := (x - x_i) / (x_{i+1} - x_i) \in [0, 1]$

$f(t) = f_i + t(f_{i+1} - f_i) = (1 - t)f_i + tf_{i+1}$
Linear interpolation in 2D

Suppose we’ve likewise sampled values of a function \( f(p) \) at points \( p_i, p_j, p_k \) in 2D

Q: How do we “connect the dots” this time? E.g., how do we fit a plane?

\[ p_i = (x_i, y_i) \]
Linear interpolation in 2D

- Want to fit a linear (really, affine) function to three values
- Any such function has three unknown coefficients $a, b, \text{ and } c$:
  $$\hat{f}(x, y) = ax + by + c$$
- To interpolate, we need to find coefficients such that the function matches the sample values at the sample points:
  $$\hat{f}(x_n, y_n) = f_n, \; n \in \{i, j, k\}$$
- Yields three linear equations in three unknowns. Solution?

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \frac{1}{(x_iy_i - x_iy_j) + (x_ky_j - x_jy_k) + (x_iy_k - x_ky_i)} \begin{bmatrix} f_i(y_k - y_j) + f_j(y_i - y_k) + f_k(y_j - y_i) \\ f_i(x_j - x_k) + f_j(x_k - x_i) + f_k(x_i - x_j) \\ f_i(x_ky_j - x_jy_k) + f_j(x_iy_k - x_ky_i) + f_k(x_iy_i - x_iy_j) \end{bmatrix}$$

This is ugly. There has to be a better way to think about this…
1D Linear Interpolation, revisited

- Let's think about how we did linear interpolation in 1D:

\[
\hat{f}(t) = (1 - t)f_i + tf_j
\]

- Can think of this as a linear combination of two functions:

- As we move closer to \( t=0 \), we approach the value of \( f \) at \( x_i \)
- As we move closer to \( t=1 \), we approach the value of \( f \) at \( x_j \)
2D Linear Interpolation, revisited

- We can construct analogous functions for a triangle
- For a given point $x$, measure the distance to each edge; then divide by the height of the triangle:

$$\phi_i(x) = \frac{d_i(x)}{h_i}$$

Interpolate by taking linear combination:

$$\hat{f}(x) = f_i \phi_i + f_j \phi_j + f_k \phi_k$$

Q: Is this the same as the (ugly) function we found before?
2D Interpolation, another way

I claim we can also get the same three basis functions as a ratio of triangle areas:

\[ \phi_i(x) = \frac{\text{area}(x, x_j, x_k)}{\text{area}(x_i, x_j, x_k)} \]

Q: Do you buy it? (Why or why not?)
Barycentric Coordinates

- No matter how you compute them, the values of the three functions $\phi_i(x)$, $\phi_j(x)$, $\phi_k(x)$ for a given point are called barycentric coordinates.

- Can be used to interpolate any attribute associated with vertices. (color*, texture coordinates, etc.)

- Importantly, these same three values fall out of the half-plane tests used for triangle rasterization! (Why?)

- Hence, get them for “free” during rasterization.

$$\text{color}(x) = \text{color}(x_i)\phi_i + \text{color}(x_j)\phi_j + \text{color}(x_k)\phi_k$$

*Note: we haven’t explained yet how to encode colors as numbers! We’ll talk about that in a later lecture…
Occlusion
Occlusion: which triangle is visible at each covered sample point?
Sampling Depth

Assume we have a triangle given by:
- the projected 2D coordinates \((x_i, y_i)\) of each vertex
- the “depth” \(d_i\) of each vertex (i.e., distance from the viewer)

**Q:** How do we compute the depth \(d\) at a given sample point \((x, y)\)?

**A:** Interpolate it using barycentric coordinates—just like any other attribute that varies linearly over the triangle
The depth-buffer (Z-buffer)

For each sample, depth-buffer stores the depth of the closest triangle seen so far

Initialize all depth buffer values to “infinity” (max value)
Depth buffer example
Example: rendering three opaque triangles
Occlusion using the depth-buffer (Z-buffer)

Processing yellow triangle:
depth = 0.5

Color buffer contents

Depth buffer contents

— sample passed depth test
Occlusion using the depth-buffer (Z-buffer)

After processing yellow triangle:

Color buffer contents

Depth buffer contents

near — sample passed depth test
Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:
depth = 0.75

Color buffer contents

Depth buffer contents

---

near  
--- sample passed depth test
Occlusion using the depth-buffer (Z-buffer)

After processing blue triangle:

Color buffer contents

Depth buffer contents

— sample passed depth test
Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:
depth = 0.25

Color buffer contents

Depth buffer contents
Occlusion using the depth-buffer (Z-buffer)

After processing red triangle:

Color buffer contents

Depth buffer contents

near — sample passed depth test
Occlusion using the depth buffer

```c
bool pass_depth_test(d1, d2)
{
    return d1 < d2;
}
```

```c
draw_sample(x, y, d, c) //new depth d & color c at (x,y)
{
    if( pass_depth_test( d, zbuffer[x][y] ) )
    {
        // triangle is closest object seen so far at this
        // sample point. Update depth and color buffers.
        zbuffer[x][y] = d; // update zbuffer
        color[x][y] = c; // update color buffer
    }
    // otherwise, we’ve seen something closer already;
    // don’t update color or depth
}
```
**Depth + Intersection**

**Q:** Does depth-buffer algorithm handle interpenetrating surfaces?

**A:** Of course!

Occlusion test is based on depth of triangles at a given sample point. Relative depth of triangles may be different at different sample points.
Q: Does depth-buffer algorithm handle interpenetrating surfaces?
A: Of course!
Occlusion test is based on depth of triangles at a given sample point. Relative depth of triangles may be different at different sample points.
Summary: occlusion using a depth buffer

- Store one depth value per sample—this is not always going to be one per pixel!

- Constant additional space per sample
  - Hence, constant space for depth buffer
  - Doesn’t depend on number of overlapping primitives!

- Constant time occlusion test per covered sample
  - Read-modify write of depth buffer if “pass” depth test
  - Just a read if “fail”

- Not specific to triangles: only requires that surface depth can be evaluated at a screen sample point

But what about semi-transparent surfaces?
Compositing
Representing opacity as alpha

An “alpha” value $0 \leq \alpha \leq 1$ describes the opacity of an object.

- $\alpha = 1$ (fully opaque)
- $\alpha = 3/4$
- $\alpha = 1/2$
- $\alpha = 1/4$
- $\alpha = 0$ (fully transparent)
Alpha channel of an image

color channels

\[
\alpha \text{ channel}
\]

Key idea: can use \( \alpha \) channel to composite one image on top of another.
Fringing

Poor treatment of color/alpha can yield dark “fringing”:

foreground color  foreground alpha  background color

fringing  no fringing
No fringing
Fringing (…why does this happen?)
Over operator:

Composites image $B$ with opacity $\alpha_B$ over image $A$ with opacity $\alpha_A$

Informally, captures behavior of “tinted glass”

Notice: “over” is not commutative

$$A \text{ over } B \neq B \text{ over } A$$

Koala over NYC

Over operator: non-premultiplied alpha

Composite image $B$ with opacity $\alpha_B$ over image $A$ with opacity $\alpha_A$

A first attempt:

$$A = (A_r, A_g, A_b)$$
$$B = (B_r, B_g, B_b)$$

Composite color:

$$C = \alpha_B B + (1 - \alpha_B)\alpha_A A$$

Composite alpha:

$$\alpha_C = \alpha_B + (1 - \alpha_B)\alpha_A$$
Over operator: premultiplied alpha

Composite image $B$ with opacity $\alpha_B$ over image $A$ with opacity $\alpha_A$

Premultiplied alpha—multiply color by $\alpha$, then composite:

$$A' = (\alpha_A A_r, \alpha_A A_g, \alpha_A A_b, \alpha_A)$$
$$B' = (\alpha_B B_r, \alpha_B B_g, \alpha_B B_b, \alpha_B)$$
$$C' = B' + (1 - \alpha_B)A'$$

Notice premultiplied alpha composites alpha just like how it composites rgb.
(Non-premultiplied alpha composites alpha differently than rgb.)

“Un-premultiply” to get final color:

$$(C_r, C_g, C_b, \alpha_C) \Longrightarrow (C_r/\alpha_C, C_g/\alpha_C, C_b/\alpha_C)$$

Q: Does this division remind you of anything?
Compositing with & without premultiplied $\alpha$

Suppose we upsample an image w/ an $\alpha$ channel, then composite it onto a background:

Q: Why do we get the “green fringe” when we don’t premultiply?
Similar problem with non-premultiplied $\alpha$

Consider pre-filtering (downsampling) a texture with an alpha matte

- Input color
- Input $\alpha$
- Filtered color
- Filtered $\alpha$
- Composited over white

- Premultiplied color
- Premultiplied $\alpha$
- Filtered color
- Filtered $\alpha$
- Composited over white
More problems: applying “over” repeatedly

Composite image $C$ with opacity $\alpha_C$ over $B$ with opacity $\alpha_B$ over image $A$ with opacity $\alpha_A$

Premultiplied alpha is closed under composition; non-premultiplied alpha is not!

Example: composite 50% bright red over 50% bright red (where “bright red” = $(1,0,0)$, and $\alpha = 0.5$)

### non-premultiplied

<table>
<thead>
<tr>
<th>color</th>
<th>premultiplied</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0.5(1,0,0) + (1-0.5)\cdot 0.5(1,0,0)$</td>
<td>$(0.75,0,0)$ too dark!</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>$\alpha = 0.75$</td>
</tr>
<tr>
<td>$0.5 + (1-0.5)\cdot 0.5 = 0.75$</td>
<td></td>
</tr>
</tbody>
</table>
Summary: advantages of premultiplied alpha

- Compositing operation treats all channels the same (color and $\alpha$)
- Fewer arithmetic operations for “over” operation than with non-premultiplied representation
- Closed under composition (repeated “over” operations)
- Better representation for filtering (upsampling/downsampling) images with alpha channel
- Fits naturally into rasterization pipeline (homogeneous coordinates)
Strategy for drawing semi-transparent primitives

Assuming all primitives are semi-transparent, and color values are encoded with premultiplied alpha, here’s a strategy for rasterizing an image:

```plaintext
over(c1, c2)
{
    return c1.rgb + (1-c1.a) * c2.rgb;
}
```

```plaintext
update_color_buffer( x, y, sample_color, sample_depth )
{
    if (pass_depth_test(sample_depth, zbuffer[x][y]))
    {
        // (how) should we update depth buffer here??
        color[x][y] = over(sample_color, color[x][y]);
    }
}
```

Q: What is the assumption made by this implementation?

Triangles must be rendered in back to front order!
Putting it all together

What if we have a mixture of opaque and transparent triangles?

Step 1: render opaque primitives (in any order) using depth-buffered occlusion
   If pass depth test, triangle overwrites value in color buffer at sample

Step 2: disable depth buffer update, render semi-transparent surfaces in back-to-front order.
   If pass depth test, triangle is composited OVER contents of color buffer at sample
End-to-end rasterization pipeline
Goal: turn inputs into an image!

Inputs:

positions = {
    v0x, v0y, v0z,
    v1x, v1y, v1x,
    v2x, v2y, v2z,
    v3x, v3y, v3x,
    v4x, v4y, v4z,
    v5x, v5y, v5x
};

Object-to-camera-space transform $T \in \mathbb{R}^{4 \times 4}$

Perspective projection transform $P \in \mathbb{R}^{4 \times 4}$

Size of output image $(W, H)$

At this point we have almost all the tools we need to make an image…

Let’s review!
Step 1:
Transform triangle vertices into camera space
Step 2:
Apply perspective projection transform to transform triangle vertices into normalized coordinate space.
Step 3: clipping

- Discard triangles that lie complete outside the unit cube (culling)
  - They are off screen, don’t bother processing them further
- Clip triangles that extend beyond the unit cube to the cube
  - (possibly generating new triangles)
Step 4: transform to screen coordinates

Perform homogeneous divide, transform vertex xy positions from normalized coordinates into screen coordinates (based on screen w,h)
Step 5: setup triangle (triangle preprocessing)

Before rasterizing triangle, can compute a bunch of data that will be used by all fragments, e.g.,

- triangle edge equations
- triangle attribute equations
- etc.

\[
\begin{align*}
E_{01}(x, y) &= U(x, y) \\
E_{12}(x, y) &= V(x, y) \\
E_{20}(x, y) &= \frac{1}{w}(x, y) \\
Z(x, y) &=
\end{align*}
\]
Step 6: sample coverage

Evaluate attributes z, u, v at all covered samples
Step 6: compute triangle color at sample point

e.g., interpolate from vertices using barycentric coordinates
Step 7: perform depth test (if enabled)

Also update depth value at covered samples (if necessary)
Step 8: update color buffer* (if depth test passed)

* Possibly using OVER operation for transparency
Our rasterization pipeline doesn’t look much different from “real” pipelines used in modern APIs / graphics hardware.

Operations on vertices
- Vertex stream
- Operations on vertices
- Vertex Processing

Operations on primitives (triangles, lines, etc.)
- Primitive stream
- Operations on primitives
- Primitive Processing

Operations on fragments
- Fragment stream
- Operations on fragments
- Fragment Generation (Rasterization)

Operations on screen samples
- Shaded fragment stream
- Operations on screen samples
- Fragment Processing

Input: vertices in 3D space

Vertices in positioned in 3D normalized coordinate space

Triangles projected to 2D screen

Fragments (one fragment per covered sample)

Shaded fragments

Output: image (pixels)

* Several stages of the modern OpenGL pipeline are omitted
GPU: heterogeneous, multi-core processor

Modern GPUs offer ~35 TFLOPs of performance for generic vertex/fragment programs ("compute")

still enormous amount of fixed-function compute over here

This part (mostly) not used by CUDA/OpenCL; raw graphics horsepower still greater than compute!
Modern Rasterization Pipeline

- Trend toward more generic (but still highly parallel!) computation:
  - make stages programmable
  - replace fixed function vertex, fragment processing
  - add geometry, tessellation shaders
  - generic “compute” shaders (whole other story…)
  - more flexible scheduling of stages

(DirectX 12 Pipeline)
Ray Tracing in Graphics Pipeline

- More recently: specialized pipeline for ray tracing (NVIDIA RTX)

GPU Ray Tracing Demo ("Marbles at Night")
Next time: Texture Mapping and Supersampling