Depth and Transparency
Today: Wrap up the rasterization pipeline!

Remember our goal:
- Start with INPUTS (triangles)
  - possibly w/ other data (e.g., colors or texture coordinates)
- Apply a series of transformations: STAGES of pipeline
- Produce OUTPUT (final image)

**INPUT (TRIANGLES)**

<table>
<thead>
<tr>
<th>VERTICES</th>
<th>RASTERIZATION PIPELINE</th>
<th>OUTPUT (BITMAP IMAGE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: (1, 1, 1)</td>
<td>E: (1, 1, -1)</td>
<td></td>
</tr>
<tr>
<td>B: (-1, 1, 1)</td>
<td>F: (-1, 1, -1)</td>
<td></td>
</tr>
<tr>
<td>C: (1, -1, 1)</td>
<td>G: (1, -1, -1)</td>
<td></td>
</tr>
<tr>
<td>D: (-1, -1, 1)</td>
<td>H: (-1, -1, -1)</td>
<td></td>
</tr>
</tbody>
</table>

**TRIANGLES**

- EHF, GFH, FGB, CBG
- GHC, DCH, ABD, CDB
- HED, ADE, EFA, BAF
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)
- Sample texture maps (filtering, mipmapping)
- Interpolate vertex attributes (barycentric coordinates)
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

near ➜ ➜ ➜ ➜ ➜ ➜ ➜ far
Example: rendering three opaque triangles
Occlusion using the depth-buffer (Z-buffer)

Processing yellow triangle:
depth = 0.5

---

Color buffer contents

Depth buffer contents

---

near — sample passed depth test

far
Occlusion using the depth-buffer (Z-buffer)

After processing yellow triangle:

Color buffer contents

Depth buffer contents

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

Processing blue triangle:
depth = 0.75

Color buffer contents

Depth buffer contents

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

After processing blue triangle:

Color buffer contents

Depth buffer contents

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

Processing red triangle:
depth = 0.25

Color buffer contents

Depth buffer contents

— sample passed depth test
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

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

```cpp
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
}
```
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.
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.
Depth + Supersampling

Q: Does depth buffer work with super sampling?

A: Yes! If done per (super) sample.

(Here: green triangle occludes yellow triangle)
Depth + Supersampling

Color of super samples after rasterizing with depth buffer
Color buffer contents (4 samples per pixel)
Final resampled result

Note anti-aliasing of edge due to filtering of green and yellow samples
Summary: occlusion using a depth buffer

- Store one depth value per (super) sample—not 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**

**α** channel

![Leaf](image1.png)

![Alpha channel of leaf](image2.png)

![Koala](image3.png)

![Alpha channel of koala](image4.png)

**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 vs. 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:

$\left( C_r, C_g, C_b, \alpha_C \right) \implies (C_r/\alpha_C, C_g/\alpha_C, C_b/\alpha_C)$

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.
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>.5(1,0,0) + (1-.5).5(1,0,0)</td>
<td>(.5,0,0,.5)+(1-.5)(.5,0,0,.5)</td>
</tr>
<tr>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>(0.75,0,0)</td>
<td>(.75,0,0.75)</td>
</tr>
<tr>
<td></td>
<td>divide by $\alpha$</td>
</tr>
</tbody>
</table>

### premultiplied

<table>
<thead>
<tr>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>(.5,0,0,.5)+(.5,0,0,.5)</td>
</tr>
<tr>
<td>↓</td>
</tr>
<tr>
<td>(.75,0,0.75)</td>
</tr>
<tr>
<td>bright red</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>(1,0,0)</td>
</tr>
</tbody>
</table>

alpha

| .5 + (1-.5).5 = .75 |
| \(\alpha = 0.75\) |
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:

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

```c
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, v1z,
v2x, v2y, v2z,
v3x, v3y, v3z,
v4x, v4y, v4z,
v5x, v5y, v5z
};
texcoords ={
v0u, v0v,
v1u, v1v,
v2u, v2v,
v3u, v3v,
v4u, v4v,
v5u, v5v
};

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 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.

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

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

e.g., sample texture map *

*Not the only way to get a color! Later we’ll talk about more general models of materials...
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
# OpenGL/Direct3D graphics pipeline

Our rasterization pipeline doesn’t look much different from “real” pipelines used in modern APIs / graphics hardware.

<table>
<thead>
<tr>
<th>Operations on vertices</th>
<th><strong>Vertex Processing</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vertex stream</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations on primitives (triangles, lines, etc.)</th>
<th><strong>Primitive Processing</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primitive stream</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations on fragments</th>
<th><strong>Fragment Generation (Rasterization)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fragment stream</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations on screen samples</th>
<th><strong>Fragment Processing</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Shaded fragment stream</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations on screen samples (depth and color)</th>
<th><strong>Screen sample operations (depth and color)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Shaded fragments</strong></td>
</tr>
</tbody>
</table>

Input: vertices in 3D space

Vertices in positioned in 3D normalized coordinate space

Triangles projected to 2D screen

Fragments (one fragment per covered sample)

Output: image (pixels)

* Several stages of the modern OpenGL pipeline are omitted
Goal: render very high complexity 3D scenes

- 100’s of thousands to millions of triangles in a scene
- Complex vertex and fragment shader computations
- High resolution screen outputs (~10Mpixel + supersampling)
- 30-120 fps
Graphics pipeline implementation: GPUs

Specialized processors for executing graphics pipeline computations

- Discrete GPU card
- Smartphone GPU (integrated)
- Integrated GPU: part of modern CPU die
Modern GPUs offer ~35 TFLOPs of performance for generic vertex/fragment programs ("compute")

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")
What else do we need to know to generate images like these?

GEOMETRY
How do we describe complex shapes (so far just triangles...)

RENDERING
How does light interact w/ materials to produce color?

ANIMATION
How do we describe the way things move?

(“Moana”, Disney 2016)
Course roadmap

- Introduction
- Drawing a triangle (by sampling)
- Transforms and coordinate spaces
- Perspective projection and texture sampling
- Today: putting it all together: end-to-end rasterization pipeline

Rasterization

Geometry

Next time!

Materials and Lighting