Lecture 8:
The Rasterization Pipeline
(and its implementation on GPUs)

Computer Graphics
CMU 15-462/15-662, Fall 2017
What you know how to do (at this point in the course)

- Position objects and the camera in the world
- Determine the position of objects relative to the camera
- Project objects onto the screen
- Sample triangle coverage
- Compute triangle attribute values at covered sample points
- Sample texture maps
What else do you need to know to render a picture like this?

Surface representation
How to represent complex surfaces?

Occlusion
Determining which surface is visible to the camera at each sample point

Lighting/materials
Describing lights in scene and how materials reflect light.
**Course roadmap**

**Drawing Things**
- 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

**Geometry**

**Materials and Lighting**

Key concepts:
- Sampling (and anti-aliasing)
- Coordinate Spaces and Transforms
Occlusion
Occlusion: which triangle is visible at each covered sample point?

Opaque Triangles

50% transparent triangles
Assume we have a triangle defined by the screen-space 2D position and distance ("depth") from the camera of each vertex.

\[
\begin{bmatrix}
  p_{0x} & p_{0y} \\
  p_{1x} & p_{1y} \\
  p_{2x} & p_{2y}
\end{bmatrix}^T, \quad d_0, d_1, d_2
\]

How do we compute the depth of the triangle at covered sample point \((x, y)\)?

Interpolate it just like any other attribute that varies linearly over the surface of the triangle.
Occlusion using the depth-buffer (Z-buffer)

For each coverage sample point, depth-buffer stores depth of closest triangle at this sample point that has been processed by the renderer so far.

Closest triangle at sample point \((x,y)\) is triangle with minimum depth at \((x,y)\)

Initial state of depth buffer before rendering any triangles (all samples store farthest distance)

Grayscale value of sample point used to indicate distance

Black = small distance
White = large distance
Depth buffer example
Example: rendering three opaque triangles
Occlusion using the depth-buffer (Z-buffer)

Processing yellow triangle:
depth = 0.5

Grayscale value of sample point used to indicate distance
White = large distance
Black = small distance
Red = sample passed depth test

Color buffer contents

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

After processing yellow triangle:

Color buffer contents

Depth buffer contents

Grayscale value of sample point used to indicate distance:
- White = large distance
- Black = small distance
- Red = sample passed depth test
Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:
depth = 0.75

Grayscale value of sample point used to indicate distance
White = large distance
Black = small distance
Red = sample passed depth test

Color buffer contents

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

After processing blue triangle:

Grayscale value of sample point used to indicate distance
White = large distance
Black = small distance
Red = sample passed depth test

Color buffer contents

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

Processing red triangle:
depth = 0.25

Grayscale value of sample point used to indicate distance
White = large distance
Black = small distance
Red = sample passed depth test

Color buffer contents

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

### After processing red triangle:

<table>
<thead>
<tr>
<th>Color buffer contents</th>
<th>Depth buffer contents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>White = large distance</td>
<td></td>
</tr>
<tr>
<td>Black = small distance</td>
<td></td>
</tr>
<tr>
<td>Red = sample passed depth test</td>
<td>White = large distance</td>
</tr>
</tbody>
</table>
Occlusion using the depth buffer

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

depth_test(tri_d, tri_color, x, y) {

    if (pass_depth_test(tri_d, zbuffer[x][y]) {

        // triangle is closest object seen so far at this
        // sample point. Update depth and color buffers.

        zbuffer[x][y] = tri_d;     // update zbuffer
        color[x][y] = tri_color;   // update color buffer

    }
}

Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

Occlusion test is based on depth of triangles at a given sample point. The relative depth of triangles may be different at different sample points.
Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

Occlusion test is based on depth of triangles at a given sample point. The relative depth of triangles may be different at different sample points.
Does depth buffer work with super sampling?

Of course! Occlusion test is per sample, not per pixel!

This example: green triangle occludes yellow triangle
Color buffer contents
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 coverage sample (not per pixel!)
- Constant space per sample
  - Implication: constant space for depth buffer
- 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

Alpha describes the opacity of an object
- Fully opaque surface: $\alpha = 1$
- 50% transparent surface: $\alpha = 0.5$
- Fully transparent surface: $\alpha = 0$

Red triangle with decreasing opacity

$\alpha = 1$  $\alpha = 0.75$  $\alpha = 0.5$  $\alpha = 0.25$  $\alpha = 0$
Alpha: additional channel of image (rgba)

α of foreground object
Over operator:

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

$B \over A$  

$A \over B$

$A \over B \neq B \over A$

“Over” is not commutative

Koala over NYC
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: non-premultiplied alpha

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

A first attempt:

\[
A = \begin{bmatrix} A_r & A_g & A_b \end{bmatrix}^T
\]
\[
B = \begin{bmatrix} B_r & B_g & B_b \end{bmatrix}^T
\]

Composited color:

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

Appearance of semi-transparent $B$

Appearance of semi-transparent $A$

What $B$ lets through

What $A$ lets through

A over $B$ $\neq$ $B$ over $A$

“Over” is not commutative
Over operator: premultiplied alpha

Composite image B with opacity \( \alpha_B \) over image A with opacity \( \alpha_A \)

Non-premultiplied alpha:

\[
A = \begin{bmatrix} A_r & A_g & A_b \end{bmatrix}^T
\]
\[
B = \begin{bmatrix} B_r & B_g & B_b \end{bmatrix}^T
\]
\[
C' = \alpha_B B + (1 - \alpha_B)\alpha_A A
\]

two multiplies, one add

Premultiplied alpha:

\[
A' = \begin{bmatrix} \alpha_A A_r & \alpha_A A_g & \alpha_A A_b & \alpha_A \end{bmatrix}^T
\]
\[
B' = \begin{bmatrix} \alpha_B B_r & \alpha_B B_g & \alpha_B B_b & \alpha_B \end{bmatrix}^T
\]
\[
C'' = B + (1 - \alpha_B)A
\]

one multiply, one add

Composite alpha:

\[
\alpha_C = \alpha_B + (1 - \alpha_B)\alpha_A
\]

Notice premultiplied alpha composites alpha just like how it composites rgb.
Non-premultiplied alpha composites alpha differently than rgb.
A problem with non-premultiplied alpha

- Suppose we upsample an image with an alpha mask, then composite it onto a background
- How should we compute the interpolated color/alpha values?
- If we interpolate color and alpha separately, then blend using the non-premultiplied “over” operator, here’s what happens:

Notice black “fringe” that occurs because we’re blending, e.g., 50% blue pixels using 50% alpha, rather than, say, 100% blue pixels with 50% alpha.
Eliminating fringe w/ premultiplied “over”

- If we instead use the premultiplied “over” operation, we get the correct alpha:

  \[
  \text{upsampled color} + (1-a) \times \text{background} = \text{composite image w/ no fringe}
  \]
Eliminating fringe w/ premultiplied “over”

- If we instead use the premultiplied “over” operation, we get the correct alpha:
Similar problem with non-premultiplied alpha

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

Desired filtered result

Downsampling non-premultiplied alpha image results in 50% opaque brown.

Result of filtering premultiplied image

\[
0.25 \times ((0, 1, 0, 1) + (0, 1, 0, 1) +
(0, 0, 0, 0) + (0, 0, 0, 0)) = (0, 0.5, 0, 0.5)
\]
More problems: applying “over” repeatedly

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

Non-premultiplied alpha is not closed under composition:

$$A = [A_r\ A_g\ A_b]^T$$
$$B = [B_r\ B_g\ B_b]^T$$
$$C = \alpha_B B + (1 - \alpha_B)\alpha_A A$$
$$\alpha_C = \alpha_B + (1 - \alpha_B)\alpha_A$$

Consider result of compositing 50% red over 50% red:

$$C = [0.75\ 0\ 0]^T$$
$$\alpha_C = 0.75$$

Wait… this result is the premultiplied color!

“Over” for non-premultiplied alpha takes non-premultiplied colors to premultiplied colors (“over” operation is not closed)

Cannot compose “over” operations on non-premultiplied values: $\overline{\overline{C, over(B, A)}}$

Q: What would be the correct UN-premultiplied RGBA for 50% red on top of 50% red?
Summary: advantages of premultiplied alpha

- Simple: compositing operation treats all channels (RGB and A) the same
- More efficient than non-premultiplied representation: “over” requires fewer math ops
- Closed under composition
- Better representation for filtering (upsampling/downsampling) textures with alpha channel
Strategy for drawing semi-transparent primitives

Assuming all primitives are semi-transparent, and RGBA values are encoded with premultiplied alpha, here’s one strategy for creating a correctly rasterized image:

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

```cpp
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(tri_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

Now 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
("real-time graphics pipeline")
Goal: turn these inputs into an image!

Inputs:

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

```c
list_of_texcoords = {
    v0u, v0v,
    v1u, v1v,
    v2u, v2v,
    v3u, v3v,
    v4u, v4v,
    v5u, v5v  
};
```

Object-to-camera-space transform: \( T \)

Perspective projection transform \( P \)

Size of output image \((W, H)\)

At this point we should have all the tools we need, but 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

Camera-space positions: 3D

Normalized space positions
Step 3: clipping

- Discard triangles that lie completely 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 *

* So far, we’ve only described computing triangle’s color at a point by interpolating per-vertex colors, or by sampling a texture map. Later in the course, we’ll discuss more advanced algorithms for computing its color based on material properties and scene lighting conditions.
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)
OpenGL/Direct3D graphics pipeline

* Several stages of the modern OpenGL pipeline are omitted.
OpenGL/Direct3D graphics pipeline *

**Pipeline inputs:**
- Input vertex data
- Parameters needed to compute position on vertices in normalized coordinates (e.g., transform matrices)
- Parameters needed to compute color of fragments (e.g., textures)
- “Shader” programs that define behavior of vertex and fragment stages

* several stages of the modern OpenGL pipeline are omitted
Shader programs

Define behavior of vertex processing and fragment processing stages
Describe operation on a single vertex (or single fragment)

Example GLSL fragment shader program

```glsl
uniform sampler2D myTexture;
uniform vec3 lightDir;
varying vec2 uv;
varying vec3 norm;

void diffuseShader()
{
    vec3 kd;
    kd = texture2d(myTexture, uv);
    kd *= clamp(dot(-lightDir, norm), 0.0, 1.0);
    gl_FragColor = vec4(kd, 1.0);
}
```

Shader function executes once per fragment.

Outputs color of surface at sample point corresponding to fragment.

(This shader performs a texture lookup to obtain the surface’s material color at this point, then performs a simple lighting computation)
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 (2-4 Mpixel + supersampling)
- 30-60 fps
Graphics pipeline implementation: GPUs

Specialized processors for executing graphics pipeline computations

Discrete GPU card
(NVIDIA GeForce Titan X)

Integrated GPU: part of modern Intel CPU die
GPU: heterogeneous, multi-core processor

Modern GPUs offer ~2-4 TFLOPs of performance for executing vertex and fragment shader programs

T-OP's of fixed-function compute capability over here

This part (mostly) not used by CUDA/OpenGL; raw graphics horsepower still greater than compute!
Summary

- Occlusion resolved independently at each screen sample using the depth buffer

- Alpha compositing for semi-transparent surfaces
  - Premultiplied alpha forms simply repeated composition
  - “Over” compositing operations is not commutative: requires triangles to be processed in back-to-front (or front-to-back) order

- Graphics pipeline:
  - Structures rendering computation as a sequence of operations performed on vertices, primitives (e.g., triangles), fragments, and screen samples
  - Behavior of parts of the pipeline is application-defined using shader programs.
  - Pipeline operations implemented by highly, optimized parallel processors and fixed-function hardware (GPUs)