Lecture 9A:
The Rasterization Pipeline
(and its implementation on GPUs)

Computer Graphics
CMU 15-462/15-662, Spring 2017
ACTIVISION ONLINE TECH TALK
THURSDAY FEB 16 5PM ROOM 4405

TOUR OF (CALL OF) DUTY
A Look At The Online Tech Behind A Large-Scale FPS

PRESENTED BY
BEN STRAGNELL
SENIOR DIRECTOR, ONLINE TECHNOLOGY
DAVID LANDAU
SENIOR ONLINE PROGRAMMER

TOPICS COVERED INCLUDE
MATCHMAKING
DEDICATED SERVERS
BACKEND INFRASTRUCTURE
IN-GAME NETWORKING

OFFICE HOURS
CAN’T MAKE THE TALK?
COME CHAT WITH OUR PROGRAMMING, RESEARCH ART, ANIMATION & MOTION CAPTURE DIRECTORS
12.30PM - 3.00PM ROOM 6121
A1 Due Tomorrow, A2 Out Now

- Start building up “Scotty3D”; first part is 3D modeling

(Start from the cube you described in Lecture 1!)

https://15462-s17.github.io/Scotty3D/Assignment2
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.
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.

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

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
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:
\[ \text{depth} = 0.75 \]

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

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)

After processing red 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

```cpp
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.

Green triangle in front of yellow triangle

Yellow triangle in front of green triangle
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 $\alpha_A$

A over B $\neq$ B over A

“Over” is not commutative

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:

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

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

Composited color:

\[
C = \alpha_B B + (1 - \alpha_B) \alpha_A A
\]
Over operator: premultiplied alpha

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

Non-premultiplied alpha:

$$
A = [A_r \ A_g \ A_b]^T \\
B = [B_r \ B_g \ B_b]^T \\
C' = \alpha_B B + (1 - \alpha_B)\alpha_A A
$$

Premultiplied alpha:

$$
A' = [\alpha_A A_r \ \alpha_A A_g \ \alpha_A A_b \ \alpha_A]^T \\
B' = [\alpha_B B_r \ \alpha_B B_g \ \alpha_B B_b \ \alpha_B]^T \\
C'' = B + (1 - \alpha_B)A
$$

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.
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:

$$A = \begin{bmatrix} A_r & A_g & A_b \end{bmatrix}^T$$
$$B = \begin{bmatrix} B_r & B_g & B_b \end{bmatrix}^T$$
$$C = \alpha_B B + (1 - \alpha_B)\alpha_A A$$
$$\alpha_C = \alpha_B + (1 - \alpha_B)\alpha_A$$

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

$$C = \begin{bmatrix} 0.75 & 0 & 0 \end{bmatrix}^T$$
$$\alpha_C = 0.75$$

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: $\overline{\text{over}(C, \overline{\text{over}(B, A)})}$

Closed form of non-premultiplied alpha:

$$C = \frac{1}{\alpha_C} (\alpha_B B + (1 - \alpha_B)\alpha_A A)$$
Another problem with non-premultiplied alpha

Consider pre-filtering a texture with an alpha matte

Desired filtered result

input color

filtered color

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

$0.25 \times ((0, 1, 0, 1) + (0, 1, 0, 1) + (0, 0, 0, 0) + (0, 0, 0, 0)) = (0, 0.5, 0, 0.5)$

Result of filtering premultiplied image

filtered result composited over white
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 textures with alpha channel
Color buffer update: semi-transparent surfaces

Color buffer values and tri_color are represented with premultiplied alpha

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

update_color_buffer(tri_d, tri_color, x, y) {
    if (pass_depth_test(tri_d, zbuffer[x][y]) {
        // update color buffer
        // Note: no depth buffer update
        color[x][y] = over(tri_color, color[x][y]);
    }
}

What is the assumption made by this implementation?
**Triangles must be rendered in back to front order!**

What if triangles are rendered in front to back order?
**Modify code: over(color[x][y], tri_color)**
Putting it all together

Consider rendering a mixture of opaque and transparent triangles

Step 1: render opaque surfaces 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")
OpenGL/Direct3D graphics pipeline *

Structures rendering computation as a series of operations on vertices, primitives, fragments, and screen samples

### Operations on vertices
- **Vertex Processing**
  - Vertex stream

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

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

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

- **Screen sample operations (depth and color)**

---

* Several stages of the modern OpenGL pipeline are omitted

---

**Input:** Vertices in 3D space

**Vertices in positioned in normalized coordinate space**

**Triangles positioned on screen**

**Fragments (one fragment per covered sample)**

**Shaded fragments**

**Output:** image (pixels)
OpenGL/Direct3D graphics pipeline

Operations on vertices
- Vertex stream
  - Vertex Processing
    - Transform matrices

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

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

Operations on screen samples
- Screen sample operations (depth and color)
  - Screen sample operations (depth and color)

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

- SIMD Exec
- Cache

- SIMD Exec
- Cache

- SIMD Exec
- Cache

- SIMD Exec
- Cache

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

- Texture
- Texture

- Texture
- Texture

- Tessellate
- Tessellate

- Tessellate
- Tessellate

- Clip/Cull Rasterize
- Clip/Cull Rasterize

- Clip/Cull Rasterize
- Clip/Cull Rasterize

- Zbuffer / Blend
- Zbuffer / Blend

- Zbuffer / Blend
- Zbuffer / Blend

- Scheduler / Work Distributor

GPU Memory

Take Kayvon’s parallel computing course (15-418) for more details!
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)
Things you should know:

- What is the depth buffer (Z-buffer) and how is it used for hidden surface removal?
- Where does the depth for each sample / fragment come from? Where is it computed in the graphics pipeline?
- Is the depth represented in the depth buffer the actual distance from the camera? If not, what is it?
- What is the meaning of the alpha parameter in the \([R \ G \ B \ a]\) color representation?
- Be able to use alpha to do compositing with the “Over” operator. $C = \alpha_B B + (1 - \alpha_B)\alpha_A A$
- Is “Over” commutative? If not, create a counterexample.
- What is premultiplied alpha, and how does it work?
- Be able to use premultiplied alpha for “Over” composition.
- Why is premultiplied alpha better?
- How do we properly render a scene with mixed opaque and semi-transparent triangles? What is the rendering order we should use? When is the depth buffer updated?
- Draw a rough sketch of the graphics pipeline. Think about transforming triangles into camera space, doing perspective projection, clipping, transforming to screen coordinates, computing colors for samples, computing colors for pixels, the depth test, updating color and depth buffers.