The Rasterization Pipeline

- Application
- Command Stream
- Vertex Processing
- Transformed Geometry
- Rasterization
- Fragments
- Fragment Processing
- Framebuffer Image
- Display

3D transformations; shading
Conversion of primitives to pixels
Blending, compositing, shading
User sees this
Transformation Pipeline
Hidden Surface Elimination

- Occlusion is an important visual cue

- Must be modeled for realistic images
  - No matter what order primitives are specified in
Hidden Surface Elimination

- Ray tracing models this by looking for the closest hit for each ray
Hidden Surface Elimination

- Two kinds of hidden surfaces removed:
  - Portions of objects facing away from camera
    - The “back” side of the spheres
  - Portions of objects hidden by other objects
    - The portion of the plane hidden by the spheres

- How to model this during rasterization?
Back–Face Culling

- If a triangle faces away from the camera, don’t draw it
- Back–facing if $\mathbf{n} \cdot \mathbf{v} < 0$
Painter’s Algorithm

- Simplest way to handle occlusion
- Draw primitives in back-to-front order
  - Overwrite pixels as you go
Painter’s Algorithm

- Need a way to sort primitives by depth
- Such an ordering may not always exist!

[Foley et al.]
Painter’s Algorithm

- Still useful when sorting is possible
The Depth Buffer

- For each pixel, keep track of closest depth
  - Compare with ray tracing

- Draw primitives in any order

- While rasterizing a primitive, mark a pixel if:
  - The pixel lies inside the object
  - Depth of the object is less than depth at pixel

- Discard pixel otherwise
The Depth Buffer
The Depth Buffer

- Stores values between “near” and “far” plane depths
  - Larger the gap between near and far, lesser the precision

- “Depth” measured as z-coordinate in canonical view volume
  - i.e., perspective-correct depth
The Depth Buffer

linear interp. in screen space ≠ linear interp. in world (eye) space
Minimal Pipeline

**Input**
- Position (per-vertex)
- Color (per-triangle)

**Vertex Processing**
- Project position
- Pass-through color

**Rasterization**
- Pass-through color

**Fragment Processing**
- Write color

**Output**
- Color (per-pixel)
Minimal Pipeline
Pipeline with Depth Buffer

**Input**
- Position (per-vertex)
- Color (per-triangle)

**Vertex Processing**
- Project position
- Pass-through color

**Rasterization**
- Interpolate depth
- Pass-through color

**Fragment Processing**
- Write color if depth test passes

**Output**
- Color (per-pixel)
Pipeline with Depth Buffer
Shading Model

- Blinn–Phong shading model:

\[ L = k_a I_a + k_d I \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s I \max(0, \mathbf{n} \cdot \mathbf{h})^p \]

- Need to know \( \mathbf{n}, \mathbf{l}, \text{and} \ \mathbf{h} \) for each fragment

- \( k_a, k_d, k_s, p \) are material properties

- \( I_a \) is a constant

- What is the value of \( I? \)
Light Types

- Many different kinds of light sources
  - Omnidirectional (lightbulbs)
  - Spotlights
  - Distant light (e.g., from the sun)
  - Linear/area light sources (fluorescent lights)
  - Ambient light

- Each light has its own **lighting model**
  - Equation to determine intensity at any point
Ambient Light

- Simplest lighting model
- A constant intensity $I_a$ everywhere
Directional Light

- Represents distant light sources
- Assume a fixed intensity $I$ and direction $\mathbf{l}$ everywhere
Point Light

- Omnidirectional light at a point $l_0$

- Light direction at any point $p$:
  \[ l = \frac{p - l_0}{\|p - l_0\|} \]

- Intensity given by **falloff** model ($r = \|p - l_0\|$):
  \[ I(r) = \frac{1}{k_0 + k_1 r + k_2 r^2} \]
Evaluating Shading

- How to determine \( \mathbf{n} \), \( \mathbf{l} \), and \( \mathbf{h} \) at each fragment?

- Or, when in the pipeline do we evaluate shading?

- Multiple choices:
  - Once per triangle (before vertex processing)
  - Once per vertex (during vertex processing)
  - Once per fragment (during fragment processing)
  - After all triangles are rasterized
Flat Shading

- a.k.a. **per-triangle** shading
- Shading calculated once for each triangle
- Use the actual normal of the triangle
- Evaluate shading at (say) centroid of triangle
Pipeline with Flat Shading

- **Input**
  - Position (per-vertex), normal (per-triangle)
  - Shade using per-triangle normal

- **Vertex Processing**
  - Project position
  - Pass-through color

- **Rasterization**
  - Interpolate depth
  - Pass-through color

- **Fragment Processing**
  - Write color if depth test passes

- **Output**
  - Color (per-pixel)
Flat Shading
Gouraud Shading

- a.k.a per-vertex shading
- Evaluate shading at each vertex
  - Use vertex normals
- Interpolate color at each fragment
Vertex Normals

- How to determine normal at each vertex?

- **Option 1:** Use underlying geometry
  - Easy for a sphere

- **Option 2:** Infer from neighboring triangles
  - e.g., by averaging:

\[
\mathbf{n}_v = \frac{\sum_i \mathbf{n}_i}{\left\| \sum_i \mathbf{n}_i \right\|}
\]
Gouraud Shading

**Input**
- Position, normal (per–vertex)

**Vertex Processing**
- Transform position/normal, project position
- Shade using per–vertex normal

**Rasterization**
- Interpolate depth, color

**Fragment Processing**
- Write color if depth test passes

**Output**
- Color (per–pixel)
Gouraud Shading
Gouraud Shading

- Specular highlights look a little off

- Attribute interpolation is linear:
  \[ A = (1 - \beta - \gamma)A_0 + \beta A_1 + \gamma A_2 \]

- Specular shading is not linear:
  \[ L_s = k_s I \max (\mathbf{n} \cdot \mathbf{h})^p \]

- What if the highlight is smaller than a triangle?
Phong Shading

- a.k.a. *per-pixel* shading
- Interpolate normals at each fragment
- Evaluate shading at each fragment
Phong Shading

Input
• Position, normal (per-vertex)

Vertex Processing
• Transform position and normal
• Project position

Rasterization
• Interpolate depth, normal

Fragment Processing
• Shade using per-fragment normal
• Write color if depth test passes

Output
• Color (per-pixel)
Phong Shading
Phong Shading

- Multiple primitives may rasterize to the same pixel
- Only the closest one needs to be drawn
  - Taken care of by depth buffer
- Shading may be evaluated multiple times per pixel
  - Depending on draw order
- Can be computationally expensive
  - Especially if the shading model is complicated
Deferred Shading

- Don’t output colors during rasterization
- Output positions, normals in a geometry buffer
- In a second pass, evaluate shading
  - Once per pixel
- Looks like per-pixel shading, only faster
Deferred Shading

**Input**
- Position, normal (per-vertex)

**Vertex Processing**
- Transform position and normal
- Project position

**Rasterization**
- Interpolate depth, normal

**Fragment Processing**
- Write position, normal if depth test passes

**Output**
- Shade using position, normal at each pixel
- Color (per-pixel)