Shading and Visibility

Michael Kazhdan

(601.457/657)

HB 13.2 -- 13.8, 14.5
FvDFH 15.4, 15.5, 15.6, 15.7.1, 16.2
Announcements

We will have a midterm next Friday (10/15/21)
- Content: Everything we cover through next Wednesday

Should I update the code-base to support the description of area light sources?

![Hard Shadow](image1.png)
![Soft Shadow](image2.png)

Hard Shadow

Soft Shadow
3D Rendering Pipeline (for direct illumination)

1. 3D Primitives
2. Modeling Transformation
   - 3D Modeling Coordinates
3. Viewing Transformation
   - 3D World Coordinates
4. Lighting
   - 3D Camera Coordinates
5. Projection Transformation
   - 3D Camera Coordinates
6. Clipping
   - 2D Window Coordinates
7. Viewport Transformation
   - 2D Window Coordinates
8. Scan Conversion
   - 2D Viewport Coordinates
9. Image
   - 2D Viewport Coordinates

3D Model

2D Viewport
3D Rendering Pipeline (for direct illumination)

3D Primitives 3D Modeling Coordinates

Modeling Transformation

3D World Coordinates

Viewing Transformation

3D Camera Coordinates

Lighting

3D Camera Coordinates

Projection Transformation

2D Window Coordinates

Clipping

2D Window Coordinates

Viewport Transformation

2D Viewport Coordinates

Scan Conversion

2D Viewport Coordinates

3D Model

Image

2D Viewport
3D Rendering Pipeline (for direct illumination)

3D Primitives
- 3D Modeling Coordinates

Modeling Transformation
- 3D World Coordinates

Viewing Transformation
- 3D Camera Coordinates

Lighting
- 3D Camera Coordinates

Projection Transformation
- 2D Window Coordinates

Clipping
- 2D Window Coordinates

Viewport Transformation
- 2D Viewport Coordinates

Scan Conversion
- 2D Viewport Coordinates

Image
- 2D Viewport

3D Model
- 3D Model
3D Rendering Pipeline (for direct illumination)

3D Primitives

Modeling Transformation

3D Modeling Coordinates

Viewing Transformation

3D World Coordinates

Lighting

3D Camera Coordinates

Projection Transformation

3D Camera Coordinates

Clipping

3D Camera Coordinates

Viewport Transformation

2D Window Coordinates

Scan Conversion

2D Window Coordinates

Image

2D Viewport Coordinates

3D Model

2D Viewport
Overview

• Scan conversion
  ○ Figure out which pixels to fill

• Shading
  ○ Determine a color for each filled pixel

• Depth test
  ○ Determine when the color of a pixel comes from the front-most primitive
Polygon Shading

- Can take advantage of spatial coherence
  - Illumination calculations for pixels covered by same primitive are related to each other

\[
I = I_E + K_A I_{AL} + \sum_i \left( K_D \langle \hat{N}, \hat{L}_i \rangle I_i + K_S \langle \hat{V}, \hat{R}_i \rangle^n I_i \right)
\]
Polygon Shading Algorithms

- **Flat Shading**
- Gouraud Shading
- Phong Shading
Flat Shading

• Take advantage of spatial coherence
  ◦ Make the lighting equation constant over the surface of each primitive

<table>
<thead>
<tr>
<th></th>
<th>Surface Normal</th>
<th>Light Direction</th>
<th>View Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emissive</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Ambient</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Diffuse</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Specular</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

\[
I = I_E + K_A I_{AL} + \sum_i \left( K_D \langle \vec{N}, \vec{L}_i \rangle I_i + K_S \langle \vec{V}, \vec{R}_i \rangle^n I_i \right)
\]

In what follows, assume that the emissive light, \( I_E \), as well as the material properties, \( K_A, K_D, K_S \), and \( n \), are constant.
Flat Shading

• Take advantage of spatial coherence
  ◦ Make the lighting equation constant over the surface of each primitive
    • If the normal is constant over the primitive, and
    • If the light is directional,

\[ I = I_E + K_A I_{AL} + \sum_i \left( K_D \langle \vec{N}, \vec{L}_i \rangle I_i + K_S \langle \vec{V}, \vec{R}_i \rangle^n I_i \right) \]

⇒ The diffuse component is the same for all points on the primitive

In what follows, assume that the emissive light, \( I_E \), as well as the material properties, \( K_A, K_D, K_S \), and \( n \), are constant
Flat Shading

- Take advantage of spatial coherence
  - Make the lighting equation constant over the surface of each primitive
    - If the normal is constant over the primitive, and
    - If the light is directional,
    \[ I = I_E + K_A I_{AL} + \sum_i \left( K_D \langle \vec{N}, \vec{L}_i \rangle I_i + K_S \langle \vec{V}, \vec{R}_i \rangle^n I_i \right) \]

- Emissive
  - If the normal is constant over the primitive, and
  - If the light is directional,

\[ I = I_E + K_A I_{AL} + \sum_i \left( K_D \langle \vec{N}, \vec{L}_i \rangle I_i + K_S \langle \vec{V}, \vec{R}_i \rangle^n I_i \right) \]

In what follows, assume that the emissive light, \( I_E \), as well as the material properties, \( K_A, K_D, K_S \), and \( n \), are constant
Flat Shading

- Take advantage of spatial coherence
  - Make the lighting equation constant over the surface of each primitive
    - If the normal is constant over the primitive, and
    - If the light is directional, and
    - If the direction to the viewer is constant over the primitive
      ⇒ The diffuse component is the same for all points on the primitive
    - If the normal is constant over the primitive,
    - If the light is directional,
      ⇒ The specular component is the same for all points on the primitive

\[ I = I_E + K_A I_{AL} + \sum_i \left( K_D \langle \vec{N}, \vec{L}_i \rangle I_i + K_S \langle \vec{V}, \vec{R}_i \rangle^n I_i \right) \]

In what follows, assume that the emissive light, \( I_E \), as well as the material properties, \( K_A, K_D, K_S \), and \( n \), are constant
Flat Shading

- Take advantage of spatial coherence
  - Make the lighting equation constant over the surface of each primitive
    - If the normal is constant over the primitive, and
    - If the light is directional, and
    - If the direction to the viewer is constant over the primitive
      ⇒ The diffuse component is the same for all points on the primitive
    - If the normal is constant over the primitive,
    - If the light is directional,
      ⇒ The specular component is the same for all points on the primitive

\[
I = I_E + K_A I_{AL} + \sum_i \left( K_D \langle \vec{N}, \vec{L}_i \rangle I_i + K_S \langle \vec{V}, \vec{R}_i \rangle^n I_i \right)
\]

In what follows, assume that the emissive light, \(I_E\), as well as the material properties, \(K_A, K_D, K_S\), and \(n\), are constant
Flat Shading

- Illuminate as though the lights are directional, the polygon is flat, and the camera uses parallel projection
  - $\langle \vec{N}, \vec{L}_i \rangle$ constant over surface
  - $\langle \vec{V}, \vec{R}_i \rangle$ constant over surface
  - $I_i$ constant over surface

\[
I = I_E + K_A I_{AL} + \sum_i \left( K_D \langle \vec{N}, \vec{L}_i \rangle I_i + K_S \langle \vec{V}, \vec{R}_i \rangle^n I_i \right)
\]

In what follows, assume that the emissive light, $I_E$, as well as the material properties, $K_A$, $K_D$, $K_S$, and $n$, are constant
Flat Shading

• One lighting calculation **per polygon**
  ◦ Assign all pixels inside each polygon the same color
Flat Shading

- Objects look like they are composed of polygons
  - OK for faceted objects
  - Not so good for smooth surfaces

Although this is the “simplest” lighting model, it is tricky to implement this on the graphics card.
Polygon Shading Algorithms

- Flat Shading
- **Gouraud Shading**
- Phong Shading
Gouraud Shading

• Represent a polygonal mesh with a normal at each vertex

\[ I = I_E + K_A I_{AL} + \sum_i \left( K_D \langle \vec{N}, \vec{L}_i \rangle I_i + K_S \langle \vec{V}, \vec{R}_i \rangle^n I_i \right) \]
Gouraud Shading

- One lighting calculation \textit{per vertex}
  - Assign pixel colors inside polygon by interpolating colors computed at vertices
Gouraud Shading

- When rasterizing, linearly interpolate colors across and between edges:

\[(I_1, I_2, I_3) \rightarrow (I_{12}, I_{23}) \rightarrow I_{123}\]

- \(I_1, I_2,\) and \(I_3\) are constant per triangle.
- \(I_{12}\) and \(I_{23}\) (and \(I_{13}\)) are constant per scan-line.
- \(I_{123}\) varies across the scan-line
Gouraud Shading

- Produces smoothly shaded polygonal mesh
  - Continuous shading over adjacent polygons

This is the lighting model implemented on the graphics card as part of the fixed pipeline.
Gouraud Shading

- Produces smoothly shaded polygonal mesh
  - Continuous shading over adjacent polygons

What happens with large polygon & spotlight?
Gouraud Shading

• Produces smoothly shaded polygonal mesh
  ◦ Continuous shading over adjacent polygons

What happens with large polygon & spotlight?

Gouraud Shading Demo
Polygon Shading Algorithms

- Flat Shading
- Gouraud Shading
- Phong Shading
Phong Shading

- One lighting calculation **per pixel**
  - Approximate surface normals for points inside polygons by linear interpolation of normals from vertices

\[
I = I_E + K_A I_{AL} + \sum_i \left( K_D \langle \vec{N}, \vec{L}_i \rangle I_i + K_S \langle \vec{V}, \vec{R}_i \rangle^n I_i \right)
\]
Phong Shading

• When rasterizing, interpolate vertex normals down and across scan lines (and make unit length)
Phong Shading

- When rasterizing, interpolate vertex normals down and across scan lines (and make unit length)
- Compute lighting at each pixel
Phong Shading

- When rasterizing, interpolate vertex normals down and across scan lines (and make unit length)
- Compute lighting at each pixel

This was not supported in early generation graphic cards but can now be implemented in the fragment shader of the GPU.
Polygon Shading Algorithms

Wireframe
Flat

Gouraud
Phong
3D Rendering Pipeline (for direct illumination)

3D Primitives

Modeling Transformation

3D Modeling Coordinates

Viewing Transformation

3D World Coordinates

Lighting

3D Camera Coordinates

Projection Transformation

3D Camera Coordinates

Clipping

2D Window Coordinates

Viewport Transformation

2D Window Coordinates

Scan Conversion

2D Viewport Coordinates

Image

3D Model

2D Viewport
Overview

• Scan conversion
  ◦ Figure out which pixels to fill

• Shading
  ◦ Determine a color for each filled pixel

• Depth test
  ◦ Determine when the color of a pixel comes from the front-most primitive
Hidden Surface Removal (HSR)

• Motivation

• Algorithms for HSR
  ◦ Back-face detection
  ◦ Depth sort
  ◦ Ray casting
  ◦ z-buffer
Motivation

In general, we don’t want to draw surfaces that are not visible to the viewer:

• Surfaces may be back-facing
• Surfaces may be covered in 3D
• Surfaces may be covered in the image plane
3D Rendering Pipeline

3D Primitives

Modeling Transformation

3D Modeling Coordinates

Camera Transformation

3D World Coordinates

3D Camera Coordinates

Lighting

3D Camera Coordinates

Projection Transformation

2D Window Coordinates

Clipping

2D Window Coordinates

Viewport Transformation

2D Viewport Coordinates

Scan Conversion

2D Viewport Coordinates

Image

Somewhere in here we have to decide which objects are visible, and which are hidden.
Overview

• Motivation

• Algorithms for HSR
  ◦ Back-face detection
  ◦ BSP-Trees
  ◦ Ray casting
  ◦ z-buffer
Back-face detection

Q: How do we test for back-facing polygons?

A: Dot product of the normal and view directions.

If $\langle \vec{V}, \vec{N} \rangle > 0$, then polygon is back-facing
Back-face detection

This method:
- Does not eliminate shapes overlapping in 3D or 2D
- Does not work for non-solid models and/or models without a well-defined orientation.

In general, back-face expected to remove $\approx$ half of polygon surfaces from removal further visibility tests
A polygon is back-facing if $\langle \vec{V}, \vec{N} \rangle > 0$
3D Rendering Pipeline

A polygon is back-facing if \( \langle \vec{V}, \vec{N} \rangle > 0 \)

Note: When your graphics card does this, it does not use the normals you provide at the vertices for lighting. It uses the geometric normal – the cross-product of the triangle edges – so make sure that the ordering of the vertices is consistent.

By default triangles/polygons are back-facing if the vertices are in clockwise order when viewed from the camera.
View-frustrum culling

If the shape is outside the viewing volume, we don’t have to draw it.
View-frustrum culling

If the shape is outside the viewing volume, we don’t have to draw it.
View-frustrum culling

3D Primitives

Modeling Transformation

Viewing Transformation

Lighting

Projection Transformation

Clipping

Viewport Transformation

Scan Conversion

Image

3D Model Coordinates

3D World Coordinates

3D Camera Coordinates

3D Camera Coordinates

2D Window Coordinates

2D Window Coordinates

2D Viewport Coordinates

2D Viewport Coordinates

Trivial Reject
Ideal Solution

Painter’s Algorithm:

- Sort primitives front to back and draw the back ones first, over-writing pixel values with information from the front primitives as they are processed.

Problem:

- In general you can’t sort the primitives.
- ...Unless you are allowed to split them
BSP-Tree Rendering (Object Precision)

- BSP-Trees recursively partition space by planes
  - Given two primitives on either side of a plane, the one on the opposite side from the camera will always be further away.
  - Draw the further side first, and then draw the closer one.
BSP-Tree Rendering (Object Precision)

• Draw further half first, then the closer one.
  • Draw right side of 1
  • Draw left side of 1
BSP-Tree Rendering (Object Precision)

• Draw further half first, then the closer one.
  • Draw right side of 1
    • Draw left side of 3
    • Draw right side of 3
  • Draw left side of 1
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of 1
    - Draw left side of 3
    - Draw D
    - Draw right side of 3
  - Draw left side of 1
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of 1
    - Draw left side of 3
      - Draw D
    - Draw right side of 3
      - Draw left side of 5
      - Draw right side of 5
  - Draw left side of 1
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of 1
    - Draw left side of 3
      - Draw D
    - Draw right side of 3
      - Draw left side of 5
        - Draw E
        - Draw right side of 5
  - Draw left side of 1
BSP-Tree Rendering (Object Precision)

• Draw further half first, then the closer one.
  • Draw right side of 1
    • Draw left side of 3
      • Draw D
    • Draw right side of 3
      • Draw left side of 5
        • Draw E
      • Draw right side of 5
        • Draw F
  • Draw left side of 1
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of 1
    - Draw left side of 3
    - Draw D
    - Draw right side of 3
    - Draw left side of 5
    - Draw E
    - Draw right side of 5
    - Draw F
  - Draw left side of 1
    - Draw left side of 2
    - Draw right side of 2
BSP-Tree Rendering (Object Precision)

• Draw further half first, then the closer one.
  • Draw right side of 1
    • Draw left side of 3
      • Draw D
    • Draw right side of 3
      • Draw left side of 5
        • Draw E
      • Draw right side of 5
        • Draw F
  • Draw left side of 1
    • Draw left side of 2
      • Draw left side of 4
        • Draw right side of 4
      • Draw right side of 2
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of 1
    - Draw left side of 3
      - Draw D
    - Draw right side of 3
      - Draw left side of 5
        - Draw E
        - Draw right side of 5
          - Draw F
    - Draw left side of 1
      - Draw left side of 2
        - Draw left side of 4
          - Draw A
          - Draw right side of 4
        - Draw right side of 2
BSP-Tree Rendering (Object Precision)

- Draw further half first, then the closer one.
  - Draw right side of 1
    - Draw left side of 3
      - Draw D
    - Draw right side of 3
      - Draw left side of 5
        - Draw E
      - Draw right side of 5
        - Draw F
  - Draw left side of 1
    - Draw left side of 2
      - Draw left side of 4
        - Draw A
      - Draw right side of 4
        - Draw B
    - Draw right side of 2
BSP-Tree Rendering (Object Precision)

• Draw further half first, then the closer one.
  • Draw right side of 1
    • Draw left side of 3
      • Draw D
    • Draw right side of 3
      • Draw left side of 5
        • Draw E
      • Draw right side of 5
        • Draw F
  • Draw left side of 1
    • Draw left side of 2
      • Draw left side of 4
        • Draw A
      • Draw right side of 4
        • Draw B
    • Draw right side of 2
      • Draw C
3D Rendering Pipeline

Binary Space Partition:
- View-independent
- Linear-time depth sort
Ray Casting

- Fire a ray for every pixel
  - If ray intersects multiple objects, take the closest
Ray Casting Pipeline

- 3D Primitives
- Modeling Transformation
- Ray casting
- Lighting
- Image

Ray casting
- $P(p \log n)$ for $p$ pixels and $n$ shapes
- May (or not) use pixel coherence
- Simple, but generally not used
**z-Buffer**

- Store color & depth of closest object at each pixel
  - Initialize depth of each pixel to $\infty$
  - Only update pixels whose depth is closer than the depth stored in the buffer
**z-Buffer**

- Store color & depth of closest object at each pixel
  - Initialize depth of each pixel to $\infty$
  - Only update pixels whose depth is closer than the depth stored in the buffer

---

**Case 1 (Blue before Red):**
Blue $\rightarrow$ $(d = 1) < (d = \infty)$:
Set $RGB = (0,0,1), d = 1$
Red $\rightarrow$ $(d = 2) > (d = 1)$:
Don’t change pixel
**z-Buffer**

- Store color & depth of closest object at each pixel
  - Initialize depth of each pixel to $\infty$
  - Only update pixels whose depth is closer than the depth stored in the buffer

\[ d = 1 \]

**Case 1 (Blue before Red):**
- Blue $\rightarrow (d = 1) < (d = \infty)$:
  - Set RGB = (0,0,1), d = 1
- Red $\rightarrow (d = 2) > (d = 1)$:
  - Don’t change pixel

\[ d = 2 \]

**Case 2 (Red before Blue):**
- Red $\rightarrow (d = 2) < (d = \infty)$:
  - Set RGB = (1,0,0), d = 2
- Blue $\rightarrow (d = 1) < (d = 2)$:
  - Set RGB = (0,0,1), d = 1
z-Buffer

- Store color & depth of closest object at each pixel
  - Initialize depth of each pixel to $\infty$
  - Update only pixels whose depth is closer than in buffer
  - Depths are interpolated from vertices, just like colors
Polygons can be rasterized in any order
- Requires additional memory
  - $z$-buffer size $\approx$ frame buffer
- This is what your graphics card does!
3D Rendering Pipeline (for direct illumination)

3D Primitives → 3D Modeling Coordinates

Modeling Transformation

3D World Coordinates

Viewing Transformation

3D Camera Coordinates

Lighting

3D Camera Coordinates

Projection Transformation

2D Window Coordinates

Clipping

2D Window Coordinates

Viewport Transformation

2D Viewport Coordinates

Scan Conversion

2D Viewport Coordinates

Image

3D Model

2D Viewport
Scan Conversion

How do we average information (e.g. color, normal, depth) from the three vertices of a triangle?

- Interpolate using weights determined by the 2D screen space projection.
- Interpolate using weights determined by the 3D locations.

It’s easier to do the interpolation in 2D.

Is there a difference?
Scan Conversion

• Projective transformations (recall)

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
w'
\end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}
\]

Properties of projective transformations:
- Origin does not necessarily map to origin
- Lines map to lines
- (Weighted) average is not necessarily preserved
- Parallel lines do not necessarily remain parallel
- Closed under composition
Scan Conversion Example

A line segment in 2D projected onto a 1D window.

How should we interpolate the information from vertices $p_1$ and $p_2$ at the pixel corresponding to ray $R$?

$z = 0 \quad z = 1$
Scan Conversion Example

A line segment in 2D projected onto a 1D window.

1. The ray intersects the window directly between the projections of \( p_1 \) and \( p_2 \):
   - We should use equal contributions from \( p_1 \) and \( p_2 \).

\[
z = 0 \quad z = 1
\]
Scan Conversion Example

A line segment in 2D projected onto a 1D window.

1. The ray intersects the window directly between the projections of $p_1$ and $p_2$:
   ⇒ Use equal contributions from $p_1$ and $p_2$.

2. The ray intersects the 2D line segment closer to $p_1$:
   ⇒ Use more information from $p_1$ than from $p_2$. 

$z = 0 \quad z = 1$
Scan Conversion Example

A line segment in 2D projected onto a 1D window.

- How do we interpolate correctly?
Scan Conversion Example

A line segment in 2D projected onto a 1D window.

- How do we interpolate correctly?

**Recall**: The 2D point \((x, z)\) maps to the point \((x/z)\) in 1D.

If \(p_1 = (x_1, z_1)\) and \(p_2 = (x_2, z_2)\), to find the blending value \(\alpha\) for a pixel falling at position \(x\) in the screen we need to solve:

\[
(1 - \alpha)(x_1, z_1) + \alpha(x_2, z_2) \equiv (x, 1)
\]

\[
\left( (1 - \alpha)x_1 + \alpha x_2, (1 - \alpha)z_1 + \alpha z_2 \right) \equiv (x, 1)
\]

\[
\frac{(1 - \alpha)x_1 + \alpha x_2}{(1 - \alpha)z_1 + \alpha z_2} = \frac{x}{1}
\]
Scan Conversion Example

A line segment in 2D projected onto a 1D window.

- How do we interpolate correctly?

Recall: The 2D point \((x, z)\) maps to the point \((x/z)\) in 1D.

If \(p_1 = (x_1, z_1)\) and \(p_2 = (x_2, z_2)\), to find the blending value \(\alpha\) for a pixel falling at position \(x\) in the screen we need to solve:

\[
\frac{(1 - \alpha)x_1 + \alpha x_2}{(1 - \alpha)z_1 + \alpha z_2} = \frac{x}{1}
\]

This is not the same as solving for the blending value in the image plane:

\[
(1 - \alpha) \frac{x_1}{z_1} + \alpha \frac{x_2}{z_2} = \frac{x}{1}
\]

To compute the interpolation weights correctly, we need to perform a perspective divide: