Course News

Assignment 2
- Due Monday, Feb 28

Homework 3
- Discussed in labs this week

Homework 4
Reading
- Chapters 8, 9
- Hidden surface removal, shading

The Rendering Pipeline

Shading

Input to Scan Conversion:
- Vertices of triangles (lines, quadrilaterals...)
- Color (per vertex)
  - Specified with glColor
  - Or: computed with lighting
- World-space normal (per vertex)
  - Left over from lighting stage

Shading Task:
- Determine color of every pixel in the triangle

Shading

How can we assign pixel colors using this information?

Easiest: flat shading
- Whole triangle gets one color (color of 1st vertex)
Better: Gouraud shading
- Linearly interpolate color across triangle
Even better:
  - Linearly interpolate the normal vector
  - Compute lighting for every pixel
  - Note: not supported by rendering pipeline as discussed so far

Flat Shading

Simplest approach calculates illumination at a single point for each polygon

- Obviously inaccurate for smooth surfaces
Flat Shading Approximations

If an object really is faceted, is this accurate?

- For point sources, the direction to light varies across the facet
- For specular reflectance, direction to eye varies across the facet

Improving Flat Shading

What if we evaluate Phong lighting model at each pixel of the polygon?
Better, but result still clearly faceted

For smoother-looking surfaces, we introduce vertex normals at each vertex
Usually different from facet normal
Used only for shading
Think of as a better approximation of the real surface that the polygons approximate

Vertex Normals

Vertex normals may be
- Provided with the model
- Computed from first principles
- Approximated by averaging the normals of the facets that share the vertex

Gouraud Shading Artifacts

often appears dull, chalky, lacks accurate specular component
- If included, will be averaged over entire polygon

\[ C_1 \]
\[ \rightarrow \]
\[ C_2 \]
\[ \rightarrow \]
\[ C_3 \]

This interior shading missed!
This vertex shading spread over too much area

Mach bands

- Eye enhances discontinuity in first derivative
- Very disturbing, especially for highlights
**Phong Shading**

Linearly interpolating surface normal across the facet, applying Phong lighting model at every pixel
- Same input as Gouraud shading
- Pro: much smoother results
- Cons: considerably more expensive

*Not the same as Phong lighting*

- Common confusion
- Phong lighting: empirical model to calculate illumination at a point on a surface

---

**Phong Shading Difficulties**

- Computationally expensive
  - Per-pixel vector normalization and lighting computation!
  - Floating point operations required
- Lighting after perspective projection
  - Messes up the angles between vectors
  - Have to keep eye-space vectors around
- No direct support in standard rendering pipeline
  - But can be simulated with texture mapping, procedural shading hardware (see later)

---

**How to Interpolate?**

- Need to propagate vertex attributes to pixels
  - Interpolate between vertices:
    - \( z \) (depth)
    - \( r,g,b \) color components
    - \( N_x, N_y, N_z \) surface normals
    - \( u,v \) texture coordinates (talk about these later)
  - Three equivalent ways of viewing this (for triangles)
    1. Linear interpolation
    2. Barycentric coordinates
    3. Plane Equation

---

**Shading Artifacts: Silhouettes**

Polygonal silhouettes remain
- Gouraud
- Phong

---

**1. Linear Interpolation**

*Interpolate quantity along L and R edges* (as a function of \( y \))
- Then interpolate quantity as a function of \( x \)
Linear Interpolation

Most common approach, and what OpenGL does

- Perform Phong lighting at the vertices
- Linearly interpolate the resulting colors over faces
  - Along edges
  - Along scanlines

Same as Barycentric Coordinates!

interior: mix of \( c_1, c_2, c_3 \)

edge: mix of \( c_1, c_2 \)

2. Barycentric Coordinates

Have seen this before

- Barycentric Coordinates: weighted combination of vertices, with weights summing to 1

\[
P = \alpha \cdot P_1 + \beta \cdot P_2 + \gamma \cdot P_3
\]

\[\alpha + \beta + \gamma = 1\]

\[0 \leq \alpha, \beta, \gamma \leq 1\]

Barycentric Coordinates

Convex combination of 3 points

\[
x = \alpha \cdot x_1 + \beta \cdot x_2 + \gamma \cdot x_3
\]

with \( \alpha + \beta + \gamma = 1 \), \( 0 \leq \alpha, \beta, \gamma \leq 1 \)

\( \alpha, \beta, \text{ and } \gamma \) are called barycentric coordinates

3. Plane Equation

Observation: Quantities vary linearly across image plane

- E.g.: \( r = Ax + By + C \)
  - \( r \) = red channel of the color
  - \( A, B, C \) = same for \( g, b \), \( Nx, Ny, Nz \), \( x, y, z \)...

From info at vertices we know:

\[
\begin{align*}
    r_1 &= Ax_1 + By_1 + C \\
    r_2 &= Ax_2 + By_2 + C \\
    r_3 &= Ax_3 + By_3 + C
\end{align*}
\]

- Solve for \( A, B, C \)
- One-time set-up cost per triangle and interpolated quantity
Discussion

Which algorithm to use when?
- Scanline interpolation
  - Together with trapezoid scan conversion
- Plane equations
  - Together with edge equation scan conversion
- Barycentric coordinates
  - Not useful in the current context
  - But: method of choice for ray-tracing
    - Whenever you only need to compute the value for a single pixel

Clipping

Wolfgang Heidrich

Line Clipping

Purpose
- Originally: 2D
  - Determine portion of line inside an axis-aligned rectangle (screen or window)
- 3D
  - Determine portion of line inside axis-aligned parallelepiped (viewing frustum in NDC)
  - Simple extension to the 2D algorithms

Outcodes (Cohen, Sutherland '74)
- 4 flags encoding position of a point relative to top, bottom, left, and right boundary

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0010</td>
<td>0000</td>
<td>0001</td>
<td>0011</td>
</tr>
<tr>
<td>0110</td>
<td>0100</td>
<td>0101</td>
<td></td>
</tr>
</tbody>
</table>

\[ x_{\text{min}} \leq x \leq x_{\text{max}} \]
\[ y_{\text{min}} \leq y \leq y_{\text{max}} \]

Line segment:
- \((p1, p2)\)

Trivial cases:
- \(\text{OC}(p1) == 0 \&\& \text{OC}(p2) == 0\)
  - Both points inside window, thus line segment completely visible (trivial accept)
- \((\text{OC}(p1) \& \text{OC}(p2)) == 0\) (i.e. bitwise “and”)
  - There is (at least) one boundary for which both points are outside (same flag set in both outcodes)
  - Thus line segment completely outside window (trivial reject)
\[ y = y_{\text{max}} \\
\]

\[ x = x_{\text{min}} \]

**α-Clipping**
- Handling of all the non-trivial cases
- Improvement of earlier algorithms (Cohen/Sutherland, Cyrus/Beck, Liang/Barsky)

Define window-edge-coordinates of a point \( p=(x,y)^T \)

\[ \text{WEC}_x(p) = x - x_{\text{min}} \]
\[ \text{WEC}_y(p) = y - y_{\text{min}} \]
\[ \text{WEC}_x(p) = x_{\text{max}} - x \]
\[ \text{WEC}_y(p) = y_{\text{max}} - y \]

Negative if outside!

**α-Clipping: algorithm**
alphaClip (p1, p2, window) {
    Determine window-edge-coordinates of p1, p2
    Determine outcodes OC(p1), OC(p2)
    Handle trivial accept and reject
    \( \alpha_1 = 0; \) // line parameter for first point
    \( \alpha_2 = 1; \) // line parameter for second point
    ...
}

**α-Clipping: algorithm (cont.)**
...
// now clip point p1 against all edges
if (OC(p1) & LEFT_FLAG) {
    \( \alpha = \text{WEC}_x(p1) / (\text{WEC}_x(p1) - \text{WEC}_x(p2)) \)
    \( \alpha_1 = \max(\alpha_1, \alpha) \);
}
Similarly clip p1 against other edges
...

**α-Clipping: example for clipping p1**

Start configuration After clipping to left After clipping to top
**Line Clipping**

**α-Clipping: algorithm (cont.)**

```cpp
... 
if (OC(p2) & LEFT_FLAG) {
    α = WECₜ(p2) / (WECₜ(p1) - WECₜ(p2));
    α₂ = min(α₂, α);
}
... 
```

Similarly clip p₁ against other edges

```cpp
... 
```

**Line Clipping**

**Example**

- ![Diagram](diagram1.png)
- ![Diagram](diagram2.png)

**Line Clipping**

**Another Example**

- ![Diagram](diagram3.png)
- ![Diagram](diagram4.png)

**Line Clipping in 3D**

**Approach:**
- Clip against parallelepiped in NDC (after perspective transform)
- Means that the clipping volume is always the same!
  - OpenGL: $x_{min} = y_{min} = -1, x_{max} = y_{max} = 1$
  - Boundary lines become boundary planes
  - But outcodes and WECs still work the same way
  - Additional front and back clipping planes
  - $z_{min} = 0, z_{max} = 1$ in OpenGL

**Extensions**
- Algorithm can be extended to clipping lines against
  - Arbitrary convex polygons (2D)
  - Arbitrary convex polytopes (3D)
**Line Clipping**

*Non-convex clipping regions*
- E.g.: windows in a window system

**Polygon Clipping**

*Objective*
- 2D: clip polygon against rectangular window
  - Or general convex polygons
  - Extensions for non-convex or general polygons
- 3D: clip polygon against parallelepiped

*Not just clipping all boundary lines*
- May have to introduce new line segments

**Classes of Polygons**
- Triangles
- Convex
- Concave
- Holes and self-intersection

*Sutherland/Hodgeman Algorithm (74)*
- Arbitrary convex or concave object polygon
  - Restriction to triangles does not simplify things
  - Convex subject polygon (window)
Polygon Clipping

Sutherland-Hodgeman Algorithm (74)

Approach: clip object polygon independently against all edges of subject polygon.

clipPolygonToEdge( p[n], edge ) {
  for( i = 0; i < n; i++ ) {
    if( p[i] inside edge ) {
      if( p[i-1] inside edge ) // p[-1]= p[n-1]
        output p[i];
      else {
        p= intersect( p[i-1], p[i], edge );
        output p, p[i];
      }
    } else...
  } // end of algorithm

clipPolygonToEdge (cont)

Clipping against one edge (cont)

p[i] inside: 2 cases

\[
\begin{align*}
\text{inside} & \quad \text{outside} \\
p[i-1] & \quad p[i]
\end{align*}
\]

Output: p[i]

Clipping against one edge (cont)

\[p[i] outside: 2 cases\]

\[
\begin{align*}
\text{inside} & \quad \text{outside} \\
p[i-1] & \quad p[i]
\end{align*}
\]

Output: p

Example

p0 p1 p2 p3 p4 p5

\[
\begin{align*}
\text{inside} & \quad \text{outside} \\
p0 & \quad p1
\end{align*}
\]
Polygon Clipping

Sutherland/Hodgeman Algorithm
- Inside/outside tests: outcodes
- Intersection of line segment with edge: window-edge coordinates
- Similar to Cohen/Sutherland algorithm for line clipping

Sutherland/Hodgeman Algorithm
- Discussion:
  - Works for concave polygons
  - But generates degenerate cases

Sutherland/Hodgeman Algorithm
- For Rendering Pipeline:
  - Re-triangulate resulting polygon
  (can be done for every individual clipping edge)

Other Polygon Clipping Algorithms
- Weiler/Atherton ’77:
  - Arbitrary concave polygons with holes both as subject and as object polygon
- Vatti ’92:
  - Self intersection allowed as well
- … many more
  - Improved handling of degenerate cases
  - But not often used in practice due to high complexity

Coming Up:
- More clipping, hidden surface removal