Shading
Clipping
Wolfgang Heidrich

Course News
Assignment 2
Due March 2
Homework 4
Out today
Reading
Chapter 8

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

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 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
  - 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
- Pros: 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)

Shading Artifacts: Silhouettes

- Polygonal silhouettes remain
- Gouraud vs Phong

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

1. Linear Interpolation

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

I_{\text{total}} = k_a I_{\text{ambient}} + \sum_{k=1}^{n} I_k \left( k_d \left( \mathbf{n} \cdot \mathbf{l}_k \right) + k_s \left( \mathbf{v} \cdot \mathbf{r}_k \right)^n \right)

Remember: normals used in diffuse and specular terms
discontinuity in normal's rate of change harder to detect
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!

edge: mix of $c_1, c_2$

interior: mix of $c_1, c_2, c_3$

2. Barycentric Coordinates
Have seen this before
- Barycentric Coordinates: weighted combination of vertices, with weights summing to 1
  \[ P = \alpha \cdot P_1 + \beta \cdot P_2 + \gamma \cdot P_3 \]
  \[ \alpha + \beta + \gamma = 1 \]
  \[ 0 \leq \alpha, \beta, \gamma \leq 1 \]

\[ P_1 (0,0,0) \]
\[ P_2 (0,1,0) \]
\[ P_3 (1,0,0) \]

Barycentric Coordinates
- Convex combination of 3 points
  \[ x = \alpha \cdot x_1 + \beta \cdot x_2 + \gamma \cdot x_3 \]
  with $\alpha + \beta + \gamma = 1$, $0 \leq \alpha, \beta, \gamma \leq 1$
- $\alpha$, $\beta$, and $\gamma$ are called barycentric coordinates

How to compute areas?
- Cross products!
  \[ A_i = \frac{1}{2} (x_2 - x_1) \times (x - x_i) \]

3. Plane Equation
Observation: Quantities vary linearly across image plane
- E g: $r = Ax + By + C$
  - $r$ = red channel of the color
  - Same for $g$, $b$, $Nx$, $Ny$, $Nz$, $z$
- From info at vertices we know:
  \[ r_1 = Ax_1 + By_1 + C \]
  \[ r_2 = Ax_2 + By_2 + C \]
  \[ r_3 = Ax_3 + By_3 + C \]
- 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

Line Clipping

*Outcodes (Cohen, Sutherland '74)*
- 4 flags encoding position of a point relative to top, bottom, left, and right boundary
- E.g.:  
  - OC(p1)=0010  
  - OC(p2)=0000  
  - OC(p3)=1001
  - OC(p4)=1010  
  - 0010  
  - 0000  
  - 0001  
  - 0110  
  - 0100  
  - 0101

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)
Line Clipping

\[ \alpha \text{-Clipping} \]
- Line segment defined as: \( p_1 + \alpha(p_2 - p_1) \)
- Intersection point with one of the borders (say, left):
  \[ x_i + \alpha (x_i - x_i) = x_{\text{ext}} \quad \Leftrightarrow \quad \alpha = \frac{x_{\text{ext}} - x_i}{x_i - x_i} \]
  \[ = \frac{x_{\text{ext}} - x_i}{(x_i - x_{\text{ext}}) - (x_i - x_{\text{ext}})} \]
  \[ = \frac{\text{WEC}_l(x_i)}{\text{WEC}_l(x_i) - \text{WEC}_l(x_i)} \]
  \[ = x_{\text{ext}} - x_i \]

\[ \alpha \text{-Clipping: algorithm} \]
alphaClip( \( p_1, p_2, \text{window} \) )
- Determine window-edge-coordinates of \( p_1, p_2 \)
- 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
- ...

\[ \alpha \text{-Clipping: example for clipping } p_1 \]

Similarly clip \( p_1 \) against other edges

...
Line Clipping

\( \alpha \)-Clipping: algorithm (cont.)

```c
// now clip point p2 against all edges
if( OC(p2) & LEFT_FLAG ) {
    \( \alpha = \text{WEC}_L(p2)/(\text{WEC}_L(p1) - \text{WEC}_L(p2)) \)
    \( \alpha^2 = \min(\alpha^2, \alpha) \)
}
```

Similarly clip p1 against other edges

```c
...
```

Line Clipping

Example

```
\( t*p1 + (1-t)*p2 \)
\( t = \text{clip}(p1, p2) \)
```

\( p1 \) \( p2 \)
\( \text{Start configuration} \) \( \text{After clipping } p1 \) \( \text{After clipping } p2 \)

Line Clipping

Another Example

```
\( t*p1 + (1-t)*p2 \)
\( t = \text{clip}(p1, p2) \)
```

\( p1 \) \( p2 \)
\( \text{Start configuration} \) \( \text{After clipping } p1 \) \( \text{After clipping } p2 \)

Line Clipping in 3D

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

Line Clipping

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

Polygon Clipping

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

Polygon Clipping

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

Polygon Clipping

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

**Sutherland-Hodgemann Algorithm (’74)**
- Approach: clip object polygon independently against all edges of subject polygon

```java
Clipping against one edge:
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
```

**Clipping against one edge (cont)**

- p[i] inside: 2 cases
  - p[i-1]
  - p[i]

```
Output: p[i]  Output: p, p[i]
```

- p[i] outside: 2 cases
  - p[i-1]
  - p[i]

```
Output: p  Output: nothing
```

**Example**

- inside
- outside
Polygon Clipping

**Sutherland/Hodgeman Algorithm**
- Discussion:
  - Works for concave polygons
  - But generates degenerate cases
- Clipping against individual edges independent
  - Great for hardware (pipelining)
  - All vertices required in memory at the same time
  - Not so good, but unavoidable
  - Another reason for using triangles only in hardware rendering

**Other Polygon Clipping Algorithms**
- Weller/Aehlert 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:
**Friday**
- More clipping, hidden surface removal