Clipping and Scan Conversion

March 19, 2002
Frank Pfenning
Carnegie Mellon University
http://www.cs.cmu.edu/~fp/courses/graphics/

The Graphics Pipeline, Revisited

- Must eliminate objects outside viewing frustum
- Tied in with projections
  - Clipping: object space (eye coordinates)
  - Scissoring: image space (pixels in frame buffer)
- Introduce clipping in stages
  - 2D (for simplicity)
  - 3D (as in OpenGL)
- In a later lecture: scissoring
Transformations and Projections

- Sequence applied in many implementations
  1. Object coordinates to
  2. Eye coordinates to
  3. Clip coordinates to
  4. Normalized device coordinates to
  5. Screen coordinates

Clipping Against a Frustum

- General case of frustum (truncated pyramid)
  - Clipping is tricky because of frustum shape
Perspective Normalization

- Solution:
  - Implement perspective projection by perspective normalization and orthographic projection
  - Perspective normalization is a homogeneous tfm.

See [Angel Ch. 5.8]

The Normalized Frustum

- OpenGL uses \(-1 \leq x,y,z \leq 1\) (others possible)
- Clip against resulting cube
- Clipping against programmer-specified planes is different and more expensive
- Often a useful programming device
The Viewport Transformation

- Transformation sequence again:
  1. Camera: From object coordinates to eye coords
  2. Perspective normalization: to clip coordinates
  3. Clipping
  4. Perspective division: to normalized device coords.
  5. Orthographic projection (setting $z_p = 0$)
  6. Viewport transformation: to screen coordinates
- Viewport transformation can distort
- Often in OpenGL: resize callback

Line-Segment Clipping

- General: 3D object against cube
- Simpler case:
  - In 2D: line against square or rectangle
  - Before scan conversion (rasterization)
  - Later: polygon clipping
- Several practical algorithms
  - Avoid expensive line-rectangle intersections
  - Cohen-Sutherland Clipping
  - Liang-Barsky Clipping
  - Many more [see Foley et al.]
Clipping Against Rectangle

- Line-segment clipping: modify endpoints of lines to lie within clipping rectangle
- Could calculate intersections of line (segments) with clipping rectangle (expensive)

Cohen-Sutherland Clipping

- Clipping rectangle as intersection of 4 half-planes
- Encode results of four half-plane tests
- Generalizes to 3 dimensions (6 half-planes)
Outcodes

- Divide space into 9 regions
- 4-bit outcode determined by comparisons

\[ \begin{align*}
0101 & \quad 0100 & \quad 0110 \\
0011 & \quad 0010 & \\
0101 & \quad 0110 & \\
0001 & \quad 0000 & \quad 0010 \\
1001 & \quad 1000 & \quad 1010 \\
\end{align*} \]

- \( b_0 : y > y_{\text{max}} \)
- \( b_1 : y < y_{\text{min}} \)
- \( b_2 : x > x_{\text{max}} \)
- \( b_3 : x < x_{\text{min}} \)

- \( o_1 = \text{outcode}(x_1, y_1) \) and \( o_2 = \text{outcode}(x_2, y_2) \)

Cases for Outcodes

- Outcomes: accept, reject, subdivide

\[ \begin{align*}
0101 & \quad 0100 & \quad 0110 \\
0011 & \quad 0010 & \\
0101 & \quad 0110 & \\
0001 & \quad 0000 & \quad 0010 \\
1001 & \quad 1000 & \quad 1010 \\
\end{align*} \]

- \( o_1 = o_2 = 0000: \text{accept} \)
- \( o_1 \neq o_2 \neq 0000: \text{reject} \)
- \( o_1 = 0000, o_2 \neq 0000: \text{subdiv} \)
- \( o_1 \neq 0000, o_2 = 0000: \text{subdiv} \)
- \( o_1 \neq o_2 = 0000: \text{subdiv} \)
Cohen-Sutherland Subdivision

• Pick outside endpoint ($o \neq 0000$)
• Pick a crossed edge ($o = b_0b_1b_2b_3$ and $b_k \neq 0$)
• Compute intersection of this line and this edge
• Replace endpoint with intersection point
• Restart with new line segment
  – Outcodes of second point are unchanged
• Must converge (roundoff errors?)

Liang-Barsky Clipping

• Starting point is parametric form

$$p(\alpha) = (1 - \alpha)p_1 + \alpha p_2, \quad 0 \leq \alpha \leq 1$$

$$x(\alpha) = (1 - \alpha)x_1 + \alpha x_2$$
$$y(\alpha) = (1 - \alpha)y_1 + \alpha y_2$$

• Compute four intersections with extended clipping rectangle
• Will see that this can be avoided
Ordering of intersection points

• Order the intersection points
• Figure (a): $1 > \alpha_4 > \alpha_3 > \alpha_2 > \alpha_1 > 0$
• Figure (b): $1 > \alpha_4 > \alpha_2 > \alpha_3 > \alpha_1 > 0$

Liang-Barsky Efficiency Improvements

• Efficiency improvement 1:
  – Compute intersections one by one
  – Often can reject before all four are computed
• Efficiency improvement 2:
  – Equations for $\alpha_3$, $\alpha_2$
    $y_{max} = (1 - \alpha_3)y_1 + \alpha_3y_2$
    $x_{min} = (1 - \alpha_2)x_1 + \alpha_2x_2$
    $\alpha_3 = \frac{y_{max} - y_1}{y_2 - y_1}$, $\alpha_2 = \frac{x_{min} - x_1}{x_2 - x_1}$
  – Compare $\alpha_3$, $\alpha_2$ without floating-point division
Line-Segment Clipping Assessment

- **Cohen-Sutherland**
  - Works well if many lines can be rejected early
  - Recursive structure (multiple subdiv) a drawback
- **Liang-Barsky**
  - Avoids recursive calls (multiple subdiv)
  - Many cases to consider (tedious, but not expensive)
  - Used more often in practice (?)

Outline

- **Line-Segment Clipping**
  - Cohen-Sutherland
  - Liang-Barsky
- **Polygon Clipping**
  - Sutherland-Hodgeman
- **Clipping in Three Dimensions**
- **Scan Conversion**
  - DDA algorithm
  - Bresenham’s algorithm
Polygon Clipping

- Convert a polygon into one or more polygons
- Their union is intersection with clip window
- Alternatively, we can first tesselate concave polygons (OpenGL supported)

Concave Polygons

- Approach 1: clip and join to a single polygon

- Approach 2: tesselate and clip triangles
Sutherland-Hodgeman I

- Subproblem:
  - Input: polygon (vertex list) and single clip plane
  - Output: new (clipped) polygon (vertex list)
- Apply once for each clip plane
  - 4 in two dimensions
  - 6 in three dimension
  - Can arrange in pipeline

Sutherland-Hodgeman II

- To clip vertex list (polygon) against half-plane:
  - Test first vertex. Output if inside, otherwise skip.
  - Then loop through list, testing transitions
    - In-to-in: output vertex
    - In-to-out: output intersection
    - out-to-in: output intersection and vertex
    - out-to-out: no output
  - Will output clipped polygon as vertex list
- May need some cleanup in concave case
- Can combine with Liang-Barsky idea
Other Cases and Optimizations

• Curves and surfaces
  – Analytically if possible
  – Through approximating lines and polygons otherwise

• Bounding boxes
  – Easy to calculate and maintain
  – Sometimes big savings

Outline

• Line-Segment Clipping
  – Cohen-Sutherland
  – Liang-Barsky

• Polygon Clipping
  – Sutherland-Hodgeman

• Clipping in Three Dimensions

• Scan Conversion
  – DDA algorithm
  – Bresenham’s algorithm
Clipping Against Cube

- Derived from earlier algorithms
- Can allow right parallelepiped

Cohen-Sutherland in 3D

- Use 6 bits in outcode
  - \( b_4 \): \( z > z_{\text{max}} \)
  - \( b_5 \): \( z < z_{\text{min}} \)
- Other calculations as before
Liang-Barsky in 3D

- Add equation \( z(\alpha) = (1 - \alpha) z_1 + \alpha z_2 \)
- Solve, for \( p_0 \) in plane and normal \( n \):
  \[
  y_{\text{max}} = (1 - \alpha_3) y_1 + \alpha_3 y_2 \\
  x_{\text{min}} = (1 - \alpha_2) x_1 + \alpha_2 x_2 \\
  \alpha_3 = \frac{y_{\text{max}} - y_1}{y_2 - y_1} \\
  \alpha_2 = \frac{x_{\text{min}} - x_1}{x_2 - x_1}
  \]
- Yields
  \[
  \alpha = \frac{n \cdot (p_0 - p_1)}{n \cdot (p_2 - p_1)}
  \]
- Optimizations as for Liang-Barsky in 2D

Perspective Normalization

- Intersection simplifies for orthographic viewing
  - One division only (no multiplication)
  - Other Liang-Barsky optimizations also apply
- Otherwise, use perspective normalization
  - Reduces to orthographic case
  - Applies to oblique and perspective viewing

Normalization of oblique projections
Summary: Clipping

- Clipping line segments to rectangle or cube
  - Avoid expensive multiplications and divisions
  - Cohen-Sutherland or Liang-Barsky
- Clipping to viewing frustum
  - Perspective normalization to orthographic projection
  - Apply clipping to cube from above
- Client-specific clipping
  - Use more general, more expensive form
- Polygon clipping
  - Sutherland-Hodgeman pipeline

Outline

- Line-Segment Clipping
  - Cohen-Sutherland
  - Liang-Barsky
- Polygon Clipping
  - Sutherland-Hodgeman
- Clipping in Three Dimensions
- Scan Conversion
  - DDA algorithm
  - Bresenham’s algorithm
Rasterization

- Final step in pipeline: rasterization (scan conv.)
- From screen coordinates (float) to pixels (int)
- Writing pixels into frame buffer
- Separate z-buffer, display, shading, blending
- Concentrate on primitives:
  - Lines
  - Polygons (Thursday)

DDA Algorithm

- DDA (“Digital Differential Analyzer”)
- Represent
  \[ y = mx + h \quad \text{where} \quad m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x} \]
- Assume \( 0 \leq m \leq 1 \)
- Exploit symmetry
- Distinguish special cases
DDA Loop

- Assume `write_pixel(int x, int y, int value)`

```
For (ix = x1; ix <= x2; ix++)
{
    y += m;
    write_pixel(ix, round(y), color);
}
```

- Slope restriction needed
- Easy to interpolate colors

Bresenham’s Algorithm I

- Eliminate floating point addition from DDA
- Assume again $0 \leq m \leq 1$
- Assume pixel centers halfway between ints
Bresenham’s Algorithm II

- Decision variable $a - b$
  - If $a - b > 0$ choose lower pixel
  - If $a - b \leq 0$ choose higher pixel
- Goal: avoid explicit computation of $a - b$
- Step 1: re-scale $d = (x_2 - x_1)(a - b) = \Delta x (a - b)$
- $d$ is always integer

Bresenham’s Algorithm III

- Compute $d$ at step $k + 1$ from $d$ at step $k!$
- Case: $j$ did not change ($d_k > 0$)
  - $a$ decreases by $m$, $b$ increases by $m$
  - $(a - b)$ decreases by $2m = 2(\Delta y / \Delta x)$
  - $\Delta x (a - b)$ decreases by $2\Delta y$
Bresenham’s Algorithm IV

• Case: j did change \((d_k \leq 0)\)
  – \(a\) decreases by \(m-1\), \(b\) increases by \(m-1\)
  – \((a - b)\) decreases by \(2m - 2 = 2(\Delta y/\Delta x - 1)\)
  – \(\Delta x(a-b)\) decreases by \(2(\Delta y - \Delta x)\)

Bresenham’s Algorithm V

• So \(d_{k+1} = d_k - 2\Delta y\) if \(d_k > 0\)
• And \(d_{k+1} = d_k - 2(\Delta y - \Delta x)\) if \(d_k \leq 0\)
• Final (efficient) implementation:

```c
void draw_line(int x1, int y1, int x2, int y2) {
    int x, y = y0;
    int dx = 2*(x2-x1), dy = 2*(y2-y1);
    int dydx = dy-dx, D = (dy-dx)/2;
    for (x = x1 ; x <= x2 ; x++) {
        write_pixel(x, y, color);
        if (D > 0) D -= dy;
        else {y++; D -= dydx;}
    }
}
```
Bresenham’s Algorithm VI

- Need different cases to handle other m
- Highly efficient
- Easy to implement in hardware and software
- Widely used

Summary

- Line-Segment Clipping
  - Cohen-Sutherland
  - Liang-Barsky
- Polygon Clipping
  - Sutherland-Hodgeman
- Clipping in Three Dimensions
- Scan Conversion
  - DDA algorithm
  - Bresenham’s algorithm
Preview

- Scan conversion of polygons
- Anti-aliasing
- Other pixel-level operations
- Assignment 5 due Thursday
- Assignment 6 (written) out Thursday