Chapter 8: Implementation- Clipping and Rasterization

Clipping

- Fundamentals
- Cohen-Sutherland
- Parametric
- Polygons
- Circles and Curves
- Text

Basic Concepts:

The purpose of clipping is to remove objects or parts of objects that lie outside the view volume. Operationally the acceleration in the conversion to pixels (scan conversion) is sufficient to make the process worthwhile. Find parts of primitives within clip region
- often rectangular
- basic case

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

Clipping usually applied to:
- points
- lines
- polygons
- curves
- text

Clipping versus Scissoring:

Clipping is carried out on the transformed coordinates in the canonical view volume (CVV). Scissoring is carried out on the pixel in memory, and is usually used to mask of parts of an image.

**Clipping:**
- Operates with primitives
- Discards, creates or modifies
- Computationally expensive
- Usually accelerates rasterization

**Scissoring:**
- operates with pixels
- brute force
- is used for arbitrary regions (masks)
Figure 1 shows the pipeline process of Clipping and Scissoring

![Pipeline Process Diagram]

**Cohen-Sutherland Clipping Algorithm**

The Cohen-Sutherland algorithm uses the limits of the view volume to divide the world up into regions and assigning codes called *outcodes* to the regions. Line segments can be classified by the outcodes of their endpoints.

- Assign *outcodes* to line midpoints
- Test outcodes => accept, reject, split
- repeat until all line segments accepted or rejected

The algorithm works by testing the outcodes at each end of a line segment, together with the logical and of the outcodes. From this 4 conditions can be found:

- The line is within the view volume: render the line
- One end of the line is in the view volume: clip the line and render it.
- The line may enter the view volume: clip to the clipping planes and retest
- The line may not be entered the view volume: delete it. Table 1 shows the various conditions for testing.
Table 1: Condition test for total accept and Total Reject cases

<table>
<thead>
<tr>
<th>O₁</th>
<th>O₂</th>
<th>O₁ &amp; O₂</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Don’t care</td>
<td>Both ends in view volume: render</td>
</tr>
<tr>
<td>0</td>
<td>&gt;0</td>
<td>Don’t care</td>
<td>One end in and one out of view volume: clip line to view volume</td>
</tr>
<tr>
<td>&gt;0</td>
<td>0</td>
<td>Don’t care</td>
<td>Part of line may be in view: more tests</td>
</tr>
<tr>
<td>&gt;0</td>
<td>&gt;0</td>
<td>0</td>
<td>Line cannot enter view volume</td>
</tr>
</tbody>
</table>

An Example:
1. oA = 1001; oB = 0010
2. (oA AND oB) = 0 so split using first non-zero bit giving AC & CB, throw away “external” segment AC
3. oC = 0000; oB = 0010
4. oC = 0; oB ≠ 0 so split using the first non-zero bit giving CD & DB, throw away the “external” segment DB
5. oC = 0000; oD = 0000 so trivially accept

Pros:

- The Cohen-Sutherland algorithm is best when:
  - the clipping region is large compared to the world so there are lots of trivial accepts
  - the clipping region is small compared to the world so there are lots of trivial rejects
- Find the intersection using slope-intercept form: where x or y is set to edge coordinate
  \[ Y = y_1 + m(x - x_1), \quad x = x_1 + (y - y_1)/m \]

Cons
The calculations involve floating point divisions both to find x and the value of m
Looking at the example we can see that redundant clipping may occur

Parametric Clipping Algorithms
This algorithm works by deriving the line equation in parametric form. Then the properties of the intersections with the sides of the view volume are qualitatively analyzed. Only if absolutely necessary are the intersections carried out.

For a line defined by 2 points (x1, y1) and (x2, y2) the parametric form is:
\[ x = x_1 + u \Delta x, \quad y = y_1 + u \Delta y \]
Where \( \Delta x = x_2 - x_1 \) , \( \Delta y = y_2 - y_1 \)

Clearly \( u = 0 \) at \((x_1, y_1)\) and \( u = 1 \) at \((x_2, y_2)\).

Also if \( \Delta y = 0 \) or \( \Delta x = 0 \), the line is parallel to an axis and can be handled by comparisons of the \( x \) or \( y \) values.
If both increments are non zero the line will cut all 4 boundaries of the view volume. Consider the intersection of the line with the lower boundary view volume \( x = x_{\text{min}} \). The intersection can be found by solving
\[ x_{\text{min}} = x_1 + u_1 \Delta x \]
for \( u_1 \), but this involves a floating point division, which we wish to avoid.
The trick is to find qualitative information on the value of \( u_1 \) without having to find the actual value.
To do this write the equation as \( u_1 \Delta x = x_{\text{min}} - x_1 \)
If \( \Delta x \) and \( x_{\text{min}} - x_1 \) have opposite signs then \( u_1 < 0 \) : so the intersection is not on the segment, and if \( \Delta x > x_{\text{min}} - x_1 \) then \( u_1 > 1 \) : so the intersection if not on the segment.
The same process can be applied to the intersections with \( y_{\text{min}} \), \( x_{\text{max}} \), \( y_{\text{max}} \), and respectively \( u_2 \), \( u_3 \), \( u_4 \).

We can also rank the \( u \) values by without division
\( u_1 < u_4 \Rightarrow (x_{\text{min}} - x_1)/\Delta x \Rightarrow (y_{\text{max}} - y_1)/\Delta y < ((x_{\text{min}} - x_1)\Delta y < (y_{\text{max}} - y_1)\Delta x) \)
We have the 4 intersections parameter values in order and know if each is less than 0 greater than 1 or in the line segment.
Consider the case where the order is $u_2 < u_1 < u_3 < u_4$ with $u_2 < 0$ and $u_4 > 1$

The only form of line that satisfies these conditions is of the form:

All the possible permutations of $u_1$ to $u_4$ can be similarly classified. As a result clipping can be applied only when need. The main parametric clipping algorithm is the Ling-Barsky algorithm that uses a slightly modified version of this approach. The Ling-Barsky algorithm is extendable to 3D.

Clipping in 3D

In 3D we have to clip lines against plains. The Cohen-Sutherland algorithm can be applied in a modified form. The Ling-Barsky algorithm can be applied but not other parametric clipping algorithm.
Clipping polygon not equal to line clipping polygon edges (new points must be added to close up polygons). The Sutherland-Hodgman algorithm clips all the lines forming the polygon against each edge of the view volume in turn. The fig 3 shows the various cases considered for polygon clipping.
Conceptually:
• Maintain input and out vertex lists
• Traverse input list, storing vertices into output list
• Clip against each edge in turn, using output from previous clip as input
• Fig 4 shows pipeline of clippers

Fig 4: Pipeline of Clipper

• Optimize by passing each vertex to next clipper immediately (i.e. pipelining)

Circles and Curves:

Non-linear equations are used for intersection tests. Bounding information is utilized, for example:
Check bounding box for trivial accept/reject
If necessary, repeat for quadrants.
If necessary, repeat for octants
Either: test intersection analytically
Or: scissor on a pixel-by-pixel basis
Clipping Text

Text is represented:
• As a collection of lines and curves (outline)
  GLUT_STROKE clipped as a polygon
• As a bitmap
  GLUT_BITMAP scissored in color buffer
Since characters usually come in a string, we can accelerate clipping by using individual character bounds to identify where to clip the string. Fig 5 shows character clipping.

![Figure 5: Character Clipping](image)

Scan Conversion (Rasterization)

Contents:
• DDA algorithm
• Bresenham's algorithm
• Additional issues
• Filled polygons
• Edge tables

What we are given
• We have a line defined on the plane of real numbers (x, y).
• We have a discrete model of the (x, y) plane consisting of a regular array of rectangles called pixels, which can be colored.

What we want to do
• Map the line onto the pixel array, while satisfying or optimizing the following constraints.
  - Maintain constant brightness
  - Differing pen styles or thicknesses may be required
  - Shape of endpoints if line thicker than one pixel
  - Minimize ‘jaggies’
The Pixel Space:

The pixel space is a rectangle on the x y plane, bounded by $0 \leq x \leq xmax$ and $0 \leq y \leq ymax$. Each axis is divided into an integer number of pixels, $Nx$ and $Ny$. The pixels there for have width $W = xmax / Nx$ and height $H = ymax / Ny$.

Pixels are referred to using integer coordinates, that either refer to the location of their lower left hand corners or their centres. Knowing the W and H values allows the pixel to be defined. Assume that W and H are equal so pixels are square. Fig 6 shows the pixel representation.

![Fig 6: Coordinate System](image)

Digital Differential Analyzer Algorithm

This algorithm assumes that the gradient satisfies $0 \leq m \leq 1$ other cases handled by symmetry. If a line segment is to be drawn from $(x_1, y_1)$ to $(x_2, y_2)$ the gradient can be found form $m \Delta y / \Delta x$.

We now need a function round(x) that converts floats to integers and we can draw the line as follows:

Remember there is **no scaling** involved in moving from projection to viewport coordinates.
Bresenham’s Algorithm:

The DDA algorithm needed to carry out a floating point addition and a rounding operation for every iteration. Bresenham’s algorithm operates only on integers requiring only that the start and end points of a line segment are rounded.

Assume \(0 \leq m \leq 1\) and pixel \((i, j)\) is the current pixel. The value of \(y\) on the line \(x=i+1\) must lie between \(j\) and \(j+1\). Because the gradient is in the range 0-45° the next pixel must be either E, \((i+1, j)\) or NE, \((i+1, j+1)\). Assume pixels identified by their centers.

We now have a decision problem, and so must find a decision variable that can be tested to determine which pixel to color. First rewrite the equation of a straight line:

\[
y = mx + c \\
mx - y + c = 0 \\
(\Delta y/\Delta x) x - y + c = 0 \\
F(x, y) = x\Delta y - y\Delta x + c\Delta x = x\Delta y - y\Delta x + B = 0
\]
The useful properties of this function are:

- If $F(x, y) < 0$, then the point is above the line
- If $F(x, y) > 0$, then the point is below the line

Function $F(x, y)$ clearly has the properties required of a decision variable: $dn = F(xn, yn)

Consider an arbitrary pixel $(xp, yp)$ that is on a line segment. We need to test a point to determine which of the possible next 2 pixels to colour. A suitable point is $(xp+1, yp+1/2)$ this will be on the vertical boundary between pixels E and NE and at the horizontal midpoint. If $F(xp + 1, yp +1/2) \leq 0$, the next pixel is E, else the next pixel is NE.

Examining the next step we see it is not independent of the decision we have just made. If E was chosen then the next test will be $F(xp + 2, yp +1/2)$ else if NE was chosen the next test will be $F(xp + 2, yp +3/2)$

Now let $dn = F(xp + 1, yp +1/2)$ and substitute the values $x+2, y+1/2$ and $y+3/2$ back in the definition of F and we have:

If E was chosen then the next test will be $dn + 1 = dn + \Delta y$
Else if NE was chosen the next test will be $dn + 1 = dn + (\Delta y - \Delta x)$

All we need now is a formula for $d1$ and we can form the line. To find $d1$:

\[
d1=F(x0 + 1, y0 +1/2)\Delta y(x0 + 1) - \Delta x(y0 +1/2) + B
\]
\[
d1 = (x0\Delta y - y0\Delta x + B) + \Delta y - \Delta x/2
\]
\[
d1 = F(x0, y0) +\Delta y - \Delta x/2
\]
\[
d1 = d0 + \Delta y - \Delta x/2
\]

But $d0$ is the start point and we know that it is on the line so $d0 = 0$ hence $d1 = \Delta y - \Delta x/2$

This still has a factor of $1/2$ in it and we wanted only integers. Simple multiply everything by 2.
void bresenham(int x0, int y0, int x1, int y1){
    int dx = x1 - x0;
    int dy = y1 - y0;
    int d = (2*dy) - dx;
    int incr_E = 2*dy;
    int incr_NE = 2*(dy - dx);
    int x = x0, y = y0;
    color_pixel(x0, y0);
    while(x<x1){
        if(d<=0){
            d += incr_E;
            x++;
        }else{
            d += incr_NE;
            x++;
            y++;
        }
        color_pixel(x, y);
    }
}

Additional Issues:

Line intensity:
- Intensity is function of slope
- Lines with different slopes have different number of pixels per unit length
- To draw two such lines with the same intensity must make pixel intensity a function of the gradient (or simply use antialiasing)

Clip rectangles:
- If line segment has been clipped...
- At the clipping boundary opposite:
  - Integer x value
  - Real y value
- The pixel at the edge produced by the clipping algorithm P' is the same one that would be drawn without clipping
- Subsequent pixels may not be as the clipped line has different slope
- Solution:
  - Draw edge pixel
  - Initialize F(M) for next column over
  - Use original (not clipped) gradient
Filled Polygons

for each scan line
  calculate list of edge intersections
  sort list in increasing x
  for each pair (p1, p2) in list
    fill span from p1 to p2

Fig. 7: An Example
Span Filling

The different cases to be considered during Polygon filling are:

1. **Given an intersection at some (fractional) x value, how do we determine which side is interior?**
   Round down when inside, round up when outside.

2. **What about intersections at integer values?**
   Leftmost pixels of span inside, rightmost outside.

3. **What about shared vertices?**
   Only count the $y_{\text{min}}$ vertex of an edge.

4. **What about horizontal edges?**
   Don’t count these vertices at all (top edges drawn only)

![Fig 8: scan lines at value of y](image)

**Edge Table:**

This is the same scan line algorithm used in for visible surface determination. There is a global list of edges maintained each with whatever interpolation information is required. At any point on a scan within a polygon we interpolate the color intensity from the 2 current edges. The figure 9 below shows the active edge table for scan line $y=3$.
Filled-Polygon Algorithm

Basic Algorithm

sort edges in ET on minimum $y_{\text{min}}$

$y = \text{first } y \text{ value with non-empty bucket}$

AET = empty

**REPEAT**

add edges from ET($y$) to AET

sort AET on minimum $x_{\text{init}}$

fill segment defined by consecutive (x,y) pairs

remove edge $e$ when $e.y_{\text{max}} = y$

$y++$

for each $e$ in AET, $x_{\text{init}} = x_{\text{init}} + 1/m$

**UNTIL** AET and ET are empty

<table>
<thead>
<tr>
<th>$y_{\text{max}}$</th>
<th>$x_{\text{init}}$</th>
<th>$1/m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>5/3</td>
</tr>
</tbody>
</table>

As with the scanline algorithm floating point operations can be avoided

- use a decision variable
- modify edge table entries to include the decision variable and its increments

<table>
<thead>
<tr>
<th>$y_{\text{max}}$</th>
<th>$x_{\text{init}}$</th>
<th>cnt.</th>
<th>$dx$</th>
<th>$dy$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2</td>
<td>0</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>