Graphics & Visualization

Chapter 2

Rasterization Algorithms
Rasterization

- 2D display devices consist of discrete grid of pixels
- **Rasterization**: converting 2D primitives into a discrete pixel representation
- The complexity of rasterization is $O(Pp)$, where $P$ is the number of primitives and $p$ is the number of pixels
- There are 2 main ways of viewing the grid of pixels:
  - Half – Integer Centers
  - Integer Centers (shall be used)
- **Connectedness**: which are the neighbors of a pixel?
  - 4 – connectedness
  - 8 – connectedness
- Challenges in designing a rasterization algorithm:
  - Determine the pixels that accurately describe the primitive
  - Efficiency
Rasterization (2)

- Half – Integer Centers

- 4 – Connectedness

- Integer Centers

- 8 – Connectedness
Mathematical Curves

- Two mathematical forms:
  - **Implicit Form:**
    
    e.g.: $f(x, y) < 0$, implies point $(x, y)$ is 'inside' the curve
    $f(x, y) = 0$, implies point $(x, y)$ is on the curve
    $f(x, y) > 0$, implies point $(x, y)$ is 'outside' the curve

  - **Parametric Form:**
    
    - Function of a parameter $t \in [0, 1]$
    - $t$ corresponds to arc length along the curve
    - The curve is traced as $t$ goes from 0 to 1
    
    e.g.: $l(t) = (x(t), y(t))$
Mathematical Curves (2)

• Examples:
  - **Implicit Form:**
    - **line:** \( l(x, y) \equiv ax + by + c = 0 \)
      where \( a, b, c \) : line coefficients
      - if \( l(x, y) = 0 \) then point \( (x, y) \) is on the curve
      - else if \( l(x, y) < 0 \) then point \( (x, y) \) is on one half-plane
      - else if \( l(x, y) > 0 \) then point \( (x, y) \) is on the other half-plane
    - **circle:** \( c(x, y) \equiv (x - x_c)^2 + (y - y_c)^2 - r^2 = 0 \)
      where \( (x_c, y_c) \) : the center of the circle & \( r \) : circle’s radius
      - if \( c(x, y) = 0 \) then point \( (x, y) \) is on the circle
      - else if \( c(x, y) < 0 \) then point \( (x, y) \) is inside the circle
      - else if \( c(x, y) > 0 \) then point \( (x, y) \) is outside the circle
Examples:

- **Parametric Form:**
  - **line:** \( l(t) = (x(t), y(t)) \)
    
    where \( x(t) = x_1 + t (x_2 - x_1) \),
    
    \( y(t) = y_1 + t (y_2 - y_1) \),
    
    \( t \in [0,1] \)
  
  - **circle:** \( c(t) = (x(t), y(t)) \)
    
    where \( x(t) = x_c + r \cos(2\pi t) \),
    
    \( y(t) = y_c + r \sin(2\pi t), \)
    
    \( t \in [0,1] \)
Finite Differences

- Functions that define primitives need to be evaluated on the pixel grid for each pixel ⇒ wasteful
- Cut this cost by taking advantage of finite differences
- Forward differences (fd):
  - First (fd): \( \delta f_i = f_{i+1} - f_i \)
  - Second (fd): \( \delta^2 f_i = \delta f_{i+1} - \delta f_i \)
  - \( k \)th (fd): \( \delta^k f_i = \delta^{k-1} f_{i+1} - \delta^{k-1} f_i \)
- Implicit functions can be used to decide if the pixel belongs to the primitive
  e.g.: pixel\((x, y)\) is included if \( |f(x, y)| < e \),
  where \( e \): related to the line width
Finite Differences (2)

- Examples:
  - Evaluation of the **line** function incrementally:
    - from pixel \((x, y)\) to pixel \((x+1, y)\)
      - Calculation of the forward differences of the implicit line equation in the \(x\) direction from pixel \(x\) to pixel \(x+1\):
        \[
        \delta_x l(x, y) = l(x + 1, y) - l(x, y) = a
        \]
        Compute \(l(x, y) + \delta_x l(x, y) = l(x, y) + a\)
    - from pixel \((x, y)\) to pixel \((x+1, y)\)
      - Calculation of the forward differences of the implicit line equation in the \(y\) direction from pixel \(y\) to pixel \(y+1\):
        \[
        \delta_y l(x, y) = l(x, y + 1) - l(x, y) = b
        \]
        Compute \(l(x, y) + \delta_y l(x, y) = l(x, y) + b\)
Finite Differences (3)

- Examples:
  - Evaluation of the circle function incrementally:
    \[ \text{from pixel } (x, y) \text{ to pixel } (x+1, y) \]
    Calculation of the forward differences of the implicit circle equation.
    Since it has degree 2 there are two forward differences in the x direction from pixel x to pixel x+1:
    \[
    \delta_x c(x, y) = c(x+1, y) - c(x, y) = 2(x - x_c) + 1 \\
    \delta_x^2 c(x, y) = \delta_x c(x+1, y) - \delta_x c(x, y) = 2
    \]
    Compute \[ \delta_x c(x, y) = \delta_x c(x-1, y) + \delta_x^2 c(x, y) \]
    \[ c(x+1, y) = c(x, y) + \delta_x c(x, y) \]
    \[ \text{from pixel } (x, y) \text{ to pixel } (x, y+1): \text{ similar by adding } \delta_y c(x, y) \text{ and } \delta_y^2 c(x, y) \]
Line Rasterization

- Desired qualities of a line rasterization algorithm:
  - Selection of the nearest pixels to the mathematical path of the line
  - Constant line width, independent of the slope of the line
  - No gaps
  - High efficiency

The 8 octants with an example line in the first octant
Line Rasterization Algorithm 1

- Draw a line from pixel \( p_s = (x_s, y_s) \) to pixel \( p_e = (x_e, y_e) \) in the first octant

- Slope of the line: \( s = \frac{y_e - y_s}{x_e - x_s} \), \( y = y_s + \text{round}(s \cdot (x - x_s)) \), \( x = x_s, \ldots, x_e \)

Algorithm:

```c
line1 ( int xs, int ys, int xe, int ye, colour c ) {
    float s; int x, y;
    s = (ye - ys) / (xe - xs); (x, y) = (xs, ys);
    while (x <= xe) {
        setpixel (x, y, c);
        x = x + 1;
        y = ys + round(s * (x - xs));
    }
}
```
• Using \texttt{line1} algorithm in the first and second octants:
Line Rasterization Algorithm 2

- Avoid rounding operation by splitting y value into an integer and a float part e
- Compute its value incrementally

Algorithm:

```c
line2 ( int xs, int ys, int xe, int ye, colour c ) {
    float s, e; int x, y;
    e = 0;    s = (ye - ys) / (xe - xs);    (x, y) = (xs, ys);
    while (x <= xe) {
        /* assert -1/2 <= e < 1/2 */
        setpixel(x, y, c);
        x = x + 1;
        e = e + s;
        if (e >= 1/2) {
            y = y + 1;
            e = e - 1;
        }
    }
}
```
Line Rasterization Algorithm 2 (2)

- Algorithm line2 resembles the leap year calculation

- The slope is added to the \( e \) variable at each iteration until it makes up more than half a unit & then the line leaps up by 1.
- The integer \( y \) variable is incremented and \( e \) is correspondingly reduced, so that the sum of the 2 variables is unchanged.

- Similarly, the year has approximately 365.25 days but calendars are designed with an integer number of days.
- We add a day every 4 years to make up for the error being accumulated.
Bresenham Line Algorithm

- Replace the floating point variables in line2 by integers
- Multiplying the leap decision variables by $dx = x_e - x_s$ makes $s$ and $e$ integers
- The leap decision becomes $e \geq \left\lfloor \frac{dx}{2} \right\rfloor$ because $e$ is integer
- $\left\lfloor \frac{dx}{2} \right\rfloor$ can be computed by a numerical shift
- For more efficiency replace the test $e \geq \left\lfloor \frac{dx}{2} \right\rfloor$ by $e \geq 0$ using

  an initial subtraction of $\left\lfloor \frac{dx}{2} \right\rfloor$ from $e$
Bresenham Line Algorithm (2)

- Floating point variables are replaced by integers

**Algorithm**

```c
line3 ( int xs, int ys, int xe, int ye, colour c ) {
    int x, y, e, dx, dy;
    e = - (dx >> 1); dx = (xe - xs); dy=(ye - ys); (x, y)=(xs, ys);
    while (x <= xe) {
        /* assert -dx <= e < 0 */
        setpixel(x, y, c);
        x = x + 1;
        e = e + dy;
        if (e >= 0) {
            y = y + 1;
            e = e - dx;
        }
    }
}
```
Bresenham Line Algorithm (3)

- Suitable for lines in the first octant
- Changes for other octants according to the following table

<table>
<thead>
<tr>
<th>Octant</th>
<th>Major Axis</th>
<th>Minor Axis Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>increasing</td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td>increasing</td>
</tr>
<tr>
<td>3</td>
<td>y</td>
<td>decreasing</td>
</tr>
<tr>
<td>4</td>
<td>x</td>
<td>increasing</td>
</tr>
<tr>
<td>5</td>
<td>x</td>
<td>decreasing</td>
</tr>
<tr>
<td>6</td>
<td>y</td>
<td>decreasing</td>
</tr>
<tr>
<td>7</td>
<td>y</td>
<td>increasing</td>
</tr>
<tr>
<td>8</td>
<td>x</td>
<td>decreasing</td>
</tr>
</tbody>
</table>

- Meets the requirements of a good line rasterization algorithm
Circle Rasterization

- Circles possess 8–way symmetry
- Compute the pixels of one octant
- Pixels of other octants are derived using the symmetry
Circle Rasterization Algorithm

• The following algorithm exploits 8-way symmetry

*Algorithm:*

```c
set8pixels ( int x, y, colour c ) {
    setpixel(x, y, c);
    setpixel(y, x, c);
    setpixel(y, -x, c);
    setpixel(x, -y, c);
    setpixel(-x, -y, c);
    setpixel(-y, -x, c);
    setpixel(-y, x, c);
    setpixel(-x, y, c);
}
```
Bresenham Circle Algorithm

- The radius of the circle is $r$
- The center of the circle is pixel $(0, 1)$
- The algorithm starts with pixel $(0, r)$
- It draws a circular arc in the second octant
- Coordinate $x$ is incremented at every step
- If the value of the circle function becomes non-negative (pixel not inside the circle), $y$ is decremented
To center the selected pixels on the circle use a circle function which is displaced by half a pixel upwards; the circle center becomes \((0, \frac{1}{2})\)

\[
c(x, y) = x^2 + (y - \frac{1}{2})^2 - r^2 = 0
\]

Initialize the error variable to:

\[
c(0, r) = (r - \frac{1}{2})^2 - r^2 = \frac{1}{4} - r
\]

Since error is an integer variable the \(\frac{1}{4}\) can be dropped

\(e\) keeps the value of the implicit circle function

For the incremental evaluation of \(e\) use the finite differences of that function for the 2 possible steps of the algorithm

\[
c(x+1, y) - c(x, y) = (x+1)^2 - x^2 = 2x + 1
\]

\[
c(x, y-1) - c(x, y) = (y-\frac{3}{2})^2 - (y-\frac{1}{2})^2 = -2y + 2
\]
Algorithm:

circle ( int r, colour c )  {
    int x, y, e;
    x = 0;    y = r;    e = - r;
    while (x <= y) {
        /* assert e == x^2 + (y - 1/2)^2 - r^2 */
        set8pixels(x, y, c);
        e = e + 2 * x + 1;
        x = x + 1;
        if (e >= 0) {
            e = e - 2 * y + 2;
            y = y - 1;
        }
    }
}
Point in Polygon Tests

• Polygon: \(\text{n vertices (}v_0, \ldots, v_{n-1}\text{)}\) \(\text{form a closed curve}
  \begin{align*}
  \text{n edges} \\
  v_0, v_1, \ldots, v_{n-1}, v_0
  \end{align*}\)

• **Jordan Curve Theorem:** A continuous simple closed curve in the plane separates the plane into 2 regions. The ‘inside’ and the ‘outside’

• For efficient rasterization we need to know if a pixel \(p\) is inside a polygon \(P\). There are two types of inclusion tests:
  - Parity test
  - Winding number
Point in Polygon Tests (2)

• Parity Test:
  - Draw a half line from pixel p in any direction
  - Count the number of intersections of the line with the polygon P
  - If \( \# \text{intersections} == \text{odd number} \) then p is inside P
  - Otherwise p is outside P
Point in Polygon Tests (3)

- **Winding Number Test:**
  - $\omega(P, p)$ counts the # of revolutions completed by a ray from $p$ that traces $P$
  
  \[ \omega(P, p) = \frac{1}{2\pi} \int d\varphi \]
  
  - For every counterclockwise revolution $\omega(P, p) ++$
  - For every clockwise revolution $\omega(P, p)--$
  - If $\omega(P, p)$ is odd then $p$ is inside $P$
  - Otherwise $p$ is outside $P$
Point in Polygon Tests (4)

- The winding number test for point in polygon:

- Simple computation of the winding number:

- The sign test for point in convex polygon:

\[
\text{sign}(l_0(p)) = \text{sign}(l_1(p)) = \ldots = \text{sign}(l_{n-1}(p))
\]
Polygon Rasterization

- **Basic Polygon Rasterization Algorithm:**
  - Based on the parity test
  - Steps:
    1. Compute intersections $I(x, y)$ of every edge with all the scanlines it intersects & store them in a list
    2. Sort the intersections by $(y, x)$
    3. Extract spans from the list & set the pixels between them
Singularities

• Basic Polygon Rasterization Algorithm:
  ■ inefficient due to the cost of intersection computations

• Problem:
  ■ if a polygon vertex falls exactly on a scanline:
    count 2, 1 or 0 intersections?

• Solutions:
  ■ regard edge as closed on the vertex with min y and open on
    the vertex with max y
  ■ ignore horizontal edges
Singularities (2)

- Rule for Treating Intersection Singularities

- Effect of Singularities Rule on Singularities

![Diagram showing the rule for treating intersection singularities and the effect of the singularities rule on singularities.](image-url)
Scanline Polygon Rasterization Algorithm

- Takes advantage of scanline coherence & edge coherence
- Uses an Edge Table (ET) and an Active Edge Table (AET)

**Algorithm:**

1. Construct the polygon ET, containing the maximum y, the min x and the inverse slope of each edge \((y_{\text{max}}, x_{\text{min}}, 1/s)\). The record of an edge is inserted in the bucket of its minimum y coordinate.

2. For every scanline y that intersects the polygon in an upward sweep
   (a) Update the AET edge intersections for the current scanline: \(x = x + 1/s\).
   (b) Insert edges from y bucket of ET into AET.
   (c) Remove edges from AET whose \(y_{\text{max}} \leq y\).
   (d) Re-sort AET on x.
   (e) Extract spans from the AET and set their pixels.
Scanline Polygon Rasterization Algorithm (2)

- A polygon and its Edge Table (ET)

Example states of the AET

\[
\begin{align*}
y &= 4 & 8 & \frac{13}{5} & 3/5 & \quad 8 & [4] & 1/4 & \\
y &= 3 & 8 & 2 & 3/5 & \quad 4 & [17/4] & -1/4 & \\
y &= 2 & 3 & \frac{3}{2} & -1 & \quad 4 & [9/2] & -1/4 & \\
\end{align*}
\]
The edges that populate the AET change at polygon vertices according to the following figure:

- remove
- insert
- replace

Updating the AET
Critical points Polygon Rasterization Algorithm

- Uses the local minima (critical points) explicitly in order to make ET redundant and to avoid its expensive creation

- An example polygon (above) and the contents of the AET for 3 scanlines (below)

\[
\begin{align*}
\text{y} &= H_3 \quad 6 \quad -1 \quad x_9 \quad 6 \quad +1 \quad x_{10} \quad 8 \quad -1 \quad x_{11} \quad 3 \quad +1 \quad x_{12} \quad 3 \quad -1 \quad x_{13} \quad 1 \quad +1 \quad x_{14} \\
\text{y} &= H_2 \quad 11 \quad -1 \quad x_5 \quad 9 \quad +1 \quad x_6 \quad 9 \quad -1 \quad x_7 \quad 1 \quad +1 \quad x_8 \\
\text{y} &= H_1 \quad 12 \quad -1 \quad x_1 \quad 13 \quad +1 \quad x_2 \quad 15 \quad -1 \quad x_3 \quad 15 \quad +1 \quad x_4
\end{align*}
\]
Critical points Polygon Rasterization Algorithm

**Algorithm:**

1. Find and store the critical points of the polygon.
2. For every scanline \( y \) that intersects the polygon in an upward sweep
   (a) For every critical point \( c (x_c, y_c) \) \( \mid y - 1 < y_c \leq y \) track the perimeter of the polygon in both directions starting at \( c \). Tracking stops if scanline \( y \) is intersected or a local maximum is found. For every intersection with scanline \( y \) create an \( AET \) record \( (v, \pm 1, x) \) containing the start vertex number \( v \) of the intersecting edge, the tracking direction along the perimeter of the polygon \((-1\text{ or }+1\text{ depending on whether it is clockwise or counterclockwise})\) and the \( x \) coordinate of the point of intersection.
   (b) For every \( AET \) record that pre-existed step (a), track the polygon perimeter in the direction stored within it. If an intersection with scanline \( y \) is found, the record’s start vertex number and intersection \( x \) coordinate are updated. If a local maximum is found the record is deleted from the \( AET \).
   (c) Sort the \( AET \) on \( x \) if necessary.
   (d) Extract spans from the \( AET \) and set their pixels.
Triangle Rasterization Algorithm

- **Triangle**: simplest, planar, convex polygon
- Determine the pixels covered by a triangle → perform an inside test on all the pixels of the triangle’s bounding box
- The inside test can be the evaluation of the 3 line functions defined by the triangle edges
- For each pixel \( p \) of the bounding box, if the 3 line functions give the same sign, then \( p \) is inside the triangle, otherwise outside
- For efficiency, the line functions are incrementally evaluated using their forward differences
Algorithm:
triangle1 ( vertex v0, v1, v2, colour c ) {
    line l0, l1, l2;
    float e0, e1, e2, e0t, elt, e2t;
    /* Compute the line coefficients (a,b,c) from the vertices */
    mkline(v0, v1, &l0); mkline(v1, v2, &l1); mkline(v2, v0, &l2);
    /* Compute bounding box of triangle */
    bb_xmin = min(v0.x, v1.x, v2.x);
    bb_xmax = max(v0.x, v1.x, v2.x);
    bb_ymin = min(v0.y, v1.y, v2.y);
    bb_ymax = max(v0.y, v1.y, v2.y);
    /* Evaluate linear functions at (bb_xmin, bb_ymin) */
    e0 = l0.a * bb_xmin + l0.b * bb_ymin + l0.c;
    e1 = l1.a * bb_xmin + l1.b * bb_ymin + l1.c;
    e2 = l2.a * bb_xmin + l2.b * bb_ymin + l2.c;
Algorithm (continued):

```c
for (y=bb_ymin; y<=bb_ymax; y++) {
    e0t = e0; elt = e1; e2t = e2;
    for (x=bb_xmin; x<=bb_xmax; x++) {
        if (sign(e0)==sign(e1)==sign(e2))
            setpixel(x, y, c);
        e0 = e0 + l0.a;
        e1 = e1 + l1.a;
        e2 = e2 + l2.a;
    }
    e0 = e0t + l0.b;
    e1 = elt + l1.b;
    e2 = e2t + l2.b;
}
```
Triangle Rasterization Algorithm (4)

• If the bounding box is large, triangle1 is wasteful
• Another approach: **Edge Walking**
  - 3 Bresenham line rasterization algorithms are used to walk the edges of the triangle
  - Trace is done per scanline by synchronizing the line rasterizers
  - The endpoints of a span of inside pixels are computed for every scanline that intersects the triangle and the pixels of the span are set
  - Special attention to special cases
• Simplicity of the above algorithms makes them ideal for hardware implementation
Area Filling Algorithms

- A simple approach is **flood fill**

**Algorithm:**

```c
flood_fill ( polygon P, colour c ) {
    point s;
    draw_perimeter ( P, c );
    s = get_seed_point ( P );
    flood_fill_recur ( s, c );
}
```

```c
flood_fill_recur ( point (x,y), colour fill_colour ); {
    colour c;
    c = getpixel(x,y); /* read current pixel colour */
    if ( c != fill_colour ) {
        setpixel(x,y,fill_colour);
        flood_fill_recur((x+1,y), fill_colour );
        flood_fill_recur((x-1,y), fill_colour );
        flood_fill_recur((x,y+1), fill_colour );
        flood_fill_recur((x,y-1), fill_colour );
    }
}
```
Area Filling Algorithms (2)

- For 4 – connected areas the above 4 recursive calls are sufficient
- For 8 – connected areas 4 extra recursive calls must be added
  - `flood_fill_recur((x+1,y+1), fill_colour);`
  - `flood_fill_recur((x+1,y-1), fill_colour);`
  - `flood_fill_recur((x-1,y+1), fill_colour);`
  - `flood_fill_recur((x-1,y-1), fill_colour);`
- Basic problem its inefficiency
Perspective Correction

• The rasterization of primitives is performed in 2D screen space while the properties of primitives are associated with 3D object vertices

• The general projection transformation does not preserve ratios of distances → it is incorrect to linearly interpolate the values of properties in screen space

• **Perspective Correction** used to obtain the correct value at a projected point

• Based on the fact that projective transformations preserve cross ratios
Perspective Correction (2)

• Example:
  - Let \( \mathbf{ad} \) be a line segment and \( \mathbf{b} \) its midpoint in 3D space
  - Let \( \mathbf{a}', \mathbf{d}', \mathbf{b}' \) be the perspective projections of the points \( \mathbf{a}, \mathbf{d}, \mathbf{b} \)

\[
\begin{align*}
\frac{ac}{cd} &= \frac{a'c'}{c'd'} \\
\frac{ab}{bd} &= \frac{a'b'}{b'd'} \\
\end{align*}
\]

\[
\begin{align*}
\frac{ab}{bd} &= 1 \\
\frac{a'b'}{b'd'} &= q
\end{align*}
\]

• Heckbert provides an efficient solution to perspective correction:
  - Perspective division of a property:
    - Let \([x, y, z, w, c]^T\) be the pre-perspective coordinates of a vertex, where \(c\) is the value of a property \(\rightarrow [x/w, y/w, z/w, c/w, 1/w]^T\) are the coordinates of the projected vertex
Spatial Anti-aliasing

- The primitive rasterization algorithms represent the pixel as a point.
- Pixels are **not** mathematical points but have a small area → aliasing effects.
- Aliasing effects:
  - jagged appearance of object silhouettes
  - improperly rasterized small objects
  - incorrectly rasterized detail
Anti-aliasing Techniques

• Anti-aliasing trades intensity resolution to gain spatial resolution

2 categories of anti-aliasing techniques:

• **Pre-filtering:**
  - extract high frequencies before sampling
  - treat the pixel as a finite area
  - compute the % contribution of each primitive in the pixel area

• **Post-filtering:**
  - extract high frequencies after sampling
  - increase sampling frequency
  - results are averaged down
Pre-filtering Anti-aliasing Methods

Anti-aliased Polygon Rasterization: Catmull’s Algorithm

- Consider each pixel as a square window
- Clip all overlapping polygons
- Estimate the visible area of each polygon as a % of the pixel

A general polygon clipping algorithm is needed, such as Greiner-Horman (section 1.8.3)
Catmull’s Algorithm

**Algorithm:**

1. Clip all polygons against the pixel window → $P_0...P_{n-1}$ : the surviving polygon pieces

2. Eliminate hidden surfaces:
   (a) order by depth polygons $P_0...P_{n-1}$
   (b) clip against the area formed by subtracting the polygons from the (remaining) pixel window in depth order → $P_0...P_{m-1}$ ($m \leq n$) the visible parts of polygons & $A_0...A_{m-1}$ their respective areas

3. Compute final pixel color: $A_0C_0 + A_1C_1 + ... + A_{m-1}C_{m-1} + A_BC_B$
   where $C_i$: the color of polygon I & $A_B, C_B$: background area & its color

- Not practically viable:
  - Extraordinary computations
  - A polygon may not have constant color in a pixel (texture)
Pre-filtering Anti-aliasing Methods (2)

Anti-aliased Line Rasterization

• Bresenham algorithm
  - uses binary decision to select the closest pixel to the mathematical path of the lines → jagged lines & polygon edges
• Lines must have certain width → modeled as thin parallelograms
  - binary decision is wrong
  - color value depends on the % of the pixel that is covered by the line
Anti-aliased Line Rasterization

- An example:
  - Line in the 1st octant with slope $s = -\frac{a}{b}$
  - 2 pixels partially covered by the line
  - Determine the portions of the triangles $A_1$ & $A_2$
  - Color of the top pixel = color of line at a portion $A_2$
  - Color of the bottom pixel = color of line at a portion $(1-A_1)$
  - The areas of the triangles: $A_1 = \frac{d^2}{2s}$, $A_2 = \frac{(s-d)^2}{2s}$
Post-filtering Anti-aliasing Methods

• More than 1 sample per pixel → image at a higher resolution
• The results are averaged down to the resolution of the pixel grid
• Most common technique due to its simplicity
• An example:
  ■ to create an $1024 \times 1024$ image, take $3072 \times 3072$ samples
    ◆ 9 samples per pixel (3 horizontally $\times$ 3 vertically)
  ■ $3 \times 3$ virtual image pixels correspond to 1 final image pixel
  ■ the final pixel’s color is the average of the 9 samples
Post-filtering Algorithm

**Algorithm:**

1. The (continuous) image is sampled at \( s \) times the final pixel resolution (\( s \) horizontally \( \times \) \( s \) vertically) creating a virtual image \( I_u \).

2. The virtual image is low-pass filtered to eliminate the high frequencies that cause aliasing.

3. The filtered virtual image is re-sampled at the pixel resolution to produce the final image \( I_f \).

- **Use** \( s \times s \) convolution filter \( h \) instead of averaging the \( s \times s \) samples

- **Steps:**
  - Place the filter over the virtual image pixel
  - Compute the final image value: 
    \[
    I_f(i, j) = \sum_{p=0}^{s-1} \sum_{q=0}^{s-1} I_v(i*s + p, j*s + q) \cdot h(p, q)
    \]
  - Move the filter
Post-filtering Algorithm (2)

- **Examples of convolution filters:**

  - To avoid color shifts, normalize:
    \[
    \sum_{p=0}^{s-1} \sum_{q=0}^{s-1} h(p, q) = 1
    \]

- **To avoid color shifts, normalize:**

  - The larger the \( s \) is \( \rightarrow \) better results

- **Drawbacks:**
  - \( \uparrow s \rightarrow \uparrow \) image generation time & \( \uparrow \) memory required
  - no matter how big \( s \) becomes, the aliasing problem will remain
  - not sensitive to image complexity \( \rightarrow \) a lot of wasted computations
More Post-filtering Algorithms

- **Adaptive post-filtering:**
  - Increases the sampling rate where high frequencies exist
  - More complex algorithm

- **Stochastic post-filtering:**
  - Samples the continuous image at non-uniformly spaced positions
  - Aliasing effects are converted to noise (human eye ignores them)

![Regular vs Stochastic Sampling](image-url)
2D Clipping Algorithms

- Avoid giving out-of-range values to a display device
- **Clipping object (window):** display device usually modeled as rectangular parallelogram which defines the within-range values
- **Subject:** primitive of a modeled scene
- Generalization from 2D to 3D is relatively straightforward
- Subject relation to the clipping object
  - Subject entirely inside: rasterize it
  - Subject outside: do not rasterize
  - Subject intersects the clipping object: compute the intersection with a 2D clipping algorithm & rasterize the result
Point Clipping

• Point clipping is a trivial case:
  ■ is point \((x, y)\) inside the clipping object?

• If the clipping object is a rectangular parallelogram:
  ■ Exploit its opposite vertices \((x_{\text{min}}, y_{\text{min}}), (x_{\text{max}}, y_{\text{max}})\)

• Inclusion Test:

  If \(x_{\text{min}} \leq x \leq x_{\text{max}}\) \& \(y_{\text{min}} \leq y \leq y_{\text{max}}\)

  Then the point is entirely inside and must be rasterized

  Else the point is entirely outside and must NOT be rasterized
Cohen – Sutherland (CS) Algorithm

• Perform a low-cost test which decides if a line segment is entirely inside or entirely outside the clipping window

• For each non-trivial line segment compute its intersection with one of the lines defined by the window boundary

• Recursively apply the algorithm to both resultant line segments
Line Clipping - CS Algorithm (2)

- The plane of the clipping window is divided into 9 regions.
- Each region is assigned a 4-bit binary code.
- The code bits are set according to the following rules:
  - **First Bit**: Set 1 for \( y > y_{\text{max}} \), else set 0.
  - **Second Bit**: Set 1 for \( y < y_{\text{min}} \), else set 0.
  - **Third Bit**: Set 1 for \( x > x_{\text{max}} \), else set 0.
  - **Fourth Bit**: Set 1 for \( x < x_{\text{min}} \), else set 0.

![Diagram of clipping window and code mapping](image)
Line Clipping - CS Algorithm (3)

- Let the 4-bit codes of the endpoints of a line segment be $c_1, c_2$
- Each endpoint is assigned a 4-bit code according to the above rules
- Then the low-cost inclusion tests are:
  - If $c_1 \lor c_2 = 0000$
    - Then the line segment is entirely inside
  - If $c_1 \land c_2 \neq 0000$
    - Then the line segment is entirely outside
Line Clipping - CS Algorithm (4)

• Example:

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Code</th>
<th>Endpoint</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0001</td>
<td>e</td>
<td>0100</td>
</tr>
<tr>
<td>b</td>
<td>0101</td>
<td>f</td>
<td>0010</td>
</tr>
<tr>
<td>c</td>
<td>0000</td>
<td>g</td>
<td>0001</td>
</tr>
<tr>
<td>d</td>
<td>0000</td>
<td>h</td>
<td>1010</td>
</tr>
</tbody>
</table>

• \textbf{ab} is entirely outside since 0001 \land 0101 \neq 0000
• \textbf{cd} is entirely inside since 0000 \lor 0000 = 0000
• For \textbf{ef} & \textbf{gh} the extent tests are not conclusive \Rightarrow compute the intersection points
• Intersect \textbf{ef} with line \(y = y_{\min}\) since the 2\textsuperscript{nd} bit of the code is different at \textbf{e} & \textbf{f}
• Continue with the \textbf{if} line segment as the 2\textsuperscript{nd} bit of the code of the \textbf{f} vertex has value 0 (inside)
• For \textbf{gh} compute one of the intersection points \textbf{k} & continue with \textbf{gk} which then computes the intersection \textbf{j} & recurses with a trivial inside decision for \textbf{jk}
Algorithm:

CS_Clip ( vertex p1, p2, float xmin, xmax, ymin, ymax ) {
    int c1, c2;  vertex i;  edge e;
    c1 = mkcode (p1);  c2 = mkcode (p2);
    if ((c1 | c2) == 0)
        /* p1p2 is inside */
    else if ((c1 & c2) != 0)
        /* p1p2 is outside */
    else {
        e = /* window line with (c1 bit != c2 bit) */
        i = intersect_lines (e, (p1,p2));
        if outside (e, p1)
            CS_Clip(i, p2, xmin, xmax, ymin, ymax);
        else
            CS_Clip(p1, i, xmin, xmax, ymin, ymax);
    }
}
Line Clipping - Skala Algorithm

Skala Algorithm:

- Gain in efficiency over CS algorithm by classifying the vertices of the clipping window relative to the line segment being clipped.
- A binary code $c_i$ is assigned to each clipping window vertex $v_i = (x_i, y_i)$ as follows:

$$c_i = \begin{cases} 
1, & l(x_i, y_i) \geq 0 \\
0, & \text{otherwise} 
\end{cases}$$

where $l(x, y)$ is the function defined by the line segment to be clipped.

- $c_i$ indicates the side of the line segment that vertex $v_i$ lies in.
Line Clipping - Skala Algorithm (2)

- The codes are computed by taking the vertices in a consistent order around the clipping window (e.g. counterclockwise)

- A clipping window edge is intersected by the line segment for every change in the coding of the vertices (from 0 to 1 or from 1 to 0)

- A pre-computed table directly gives the clipping window edges intersected by the line segment from the code vector \([c_0, c_1, c_2, c_3]\) and this replaces the recursive case of the CS algorithm
Line Clipping – LB Algorithm

Liang – Barsky (LB) Algorithm

• Solves the line clipping problem without using recursive calls
• Compared to CS algorithm, LB is more than 30% more efficient
• Can be easily extended to a 3D clipping object
• LB is based on the parametric equation of the line segment to be clipped from \( p_1(x_1, y_1) \) to \( p_2(x_2, y_2) \):

\[
P = p_1 + t (p_2 - p_1), \quad t \in [0, 1]
\]

or

\[
x = x_1 + t \Delta x, \quad y = y_1 + t \Delta y
\]

where

\[
\Delta x = x_2 - x_1, \quad \Delta y = y_2 - y_1
\]
Line Clipping – LB Algorithm (2)

• For the part of the line segment that is inside the clipping window:

\[
\begin{align*}
    x_{\text{min}} & \leq x_1 + t \Delta x \leq x_{\text{max}} , \\
    y_{\text{min}} & \leq y_1 + t \Delta y \leq y_{\text{max}}
\end{align*}
\]

or

\[
\begin{align*}
    -t \Delta x & \leq x_1 - x_{\text{min}} , \\
    t \Delta x & \leq x_{\text{max}} - x_1 , \\
    -t \Delta y & \leq y_1 - y_{\text{min}} , \\
    t \Delta y & \leq y_{\text{max}} - y_1
\end{align*}
\]
The above inequalities have the common form:
\[ t p_i \leq q_i , \]
where
\[ p_1 = -\Delta x , \quad q_1 = x_1 - x_{\text{min}} \]
\[ p_2 = \Delta x , \quad q_2 = x_{\text{max}} - x_1 \]
\[ p_3 = -\Delta y , \quad q_3 = y_1 - y_{\text{min}} \]
\[ p_4 = \Delta y , \quad q_4 = y_{\text{max}} - y_1 \]
• Notice the following:
  - If $p_i = 0$ the line segment is parallel to the window edge $i$ and the clipping problem is trivial
  - If $p_i \neq 0$ the parametric value of the point of intersection of the line segment with the line defined by window edge $i$ is $t_i = q_i / p_i$
  - If $p_i < 0$ the directed line segment is incoming with respect to window edge $i$
  - If $p_i > 0$ the directed line segment is outgoing with respect to window edge $i$
Line Clipping – LB Algorithm (5)

- Therefore \( t_{in} \) and \( t_{out} \) can be computed as:

\[
t_{in} = \max\left(\left\{ \frac{q_i}{p_i} \mid p_i < 0, \ i:1..4\right\} \cup \{0\}\right), \quad t_{out} = \min\left(\left\{ \frac{q_i}{p_i} \mid p_i > 0, \ i:1..4\right\} \cup \{1\}\right)
\]

- Sets \{0\}, \{1\} clamp the starting and ending parametric values at the end points of the line segment.

- If \( t_{in} \leq t_{out} \), the values \( t_{in} \) and \( t_{out} \) are plugged into parametric line equation to get the actual starting – ending points of the clipped segment.

- Otherwise there is no intersection with the clipping window.
LB example:

- Compute: $\Delta x = 2.5$ and $\Delta y = 2.5$
- Compute: $p_1 = -2.5$, $q_1 = -0.5$
  $p_2 = 2.5$, $q_2 = 3.5$
  $p_3 = -2.5$, $q_3 = -0.5$
  $p_4 = 2.5$, $q_4 = 3.5$.
- Compute: $t_{in} = \max(\{\frac{q_1}{p_1}, \frac{q_3}{p_3}\} \cup \{0\}) = 0.2$, $t_{out} = \min(\{\frac{q_2}{p_2}, \frac{q_4}{p_4}\} \cup \{1\}) = 1$
- Since $t_{in} < t_{out}$ compute endpoints $p_1'(x_1', y_1')$, $p_2'(x_2', y_2')$ of the clipped line segment using the parametric equation:

\[
\begin{align*}
x_1' &= x_1 + t_{in} \Delta x = 0.5 + 0.2 \cdot 2.5 = 1 \\
y_1' &= y_1 + t_{in} \Delta y = 0.5 + 0.2 \cdot 2.5 = 1 \\
x_2' &= x_1 + t_{out} \Delta x = 0.5 + 1 \cdot 2.5 = 3 \\
y_2' &= y_1 + t_{out} \Delta y = 0.5 + 1 \cdot 2.5 = 3
\end{align*}
\]
Polygon Clipping

- In 2D polygon clipping the subject and clipping object are both polygons (subject polygon, clipping polygon)
- Why is polygon clipping important?

- Polygon clipping cannot be regarded as multiple line clipping
Sutherland – Hodgman (SH) Algorithm:

- Clips an arbitrary subject polygon against a convex clipping polygon
- Has $m$ pipeline stages which correspond to the $m$ edges of the clipping polygon
- Stage $i \mid i: 0…m-1$ clips the subject polygon against the line defined by edge $i$ of the clipping polygon
- The input to stage $i \mid i: 1…m-1$ is the output of stage $i-1$
- Polygon is restricted to be convex
For each stage of the SH algorithm there are the following 4 relationships between a clipping line and an object polygon edge $v_kv_{k+1}$.

- **Case 1:** 1 output
  - Inside
  - Output vertex

- **Case 2:** 1 output
  - Outside

- **Case 3:** 0 outputs
  - Inside
  - Clipping line

- **Case 4:** 2 outputs
  - Outside
  - Clipping line

- *output vertex*
Example of the 1st stage of the SH algorithm:

<table>
<thead>
<tr>
<th>$v_k$</th>
<th>$v_{k+1}$</th>
<th>Case</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_0$</td>
<td>$v_1$</td>
<td>2</td>
<td>$i_1$</td>
</tr>
<tr>
<td>$v_1$</td>
<td>$v_2$</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>$v_2$</td>
<td>$v_3$</td>
<td>4</td>
<td>$i_2, v_3$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>$v_4$</td>
<td>2</td>
<td>$i_3$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>$v_5$</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>$v_5$</td>
<td>$v_6$</td>
<td>4</td>
<td>$i_4, v_6$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>$v_0$</td>
<td>1</td>
<td>$v_0$</td>
</tr>
</tbody>
</table>
• **Algorithm:**

```c
poly SH_Clip (polygon C, S) { /*C must be convex*/
    int i, m;
    edge e;
    polygon InPoly, OutPoly;
    m = getedgenumber(C);
    InPoly = S;
    for (i=0; i<m; i++) {
        e = getedge(C, i);
        SH_Clip_Edge(e, InPoly, OutPoly);
        InPoly = OutPoly
    }
    return OutPoly
}
```
Polygon Clipping – SH Algorithm (5)

- **Algorithm:**

```c
SH_Clip_Edge ( edge e, polygon InPoly, OutPoly ) {
    int k, n; vertex vk, vkplus1, i;
    n = getedgenumber(InPoly);
    for (k=0; k<n; k++) {
        vk = getvertex(InPoly,k); vkplus1=getvertex(InPoly,(k+1) mod n);
        if (inside(e, vk) and inside(e, vkplus1))
            /* Case 1 */
            putvertex(OutPoly,vkplus1)
        else if (inside(e, vk) and !inside(e, vkplus1)) {
            /* Case 2 */
            i = intersect_lines(e, (vk,vkplus1)); putvertex(OutPoly,i)
        } else if (!inside(e, vk) and !inside(e, vkplus1))
            /* Case 3 */
        else {
            /* Case 4 */
            i = intersect_lines(e, (vk,vkplus1)); putvertex(OutPoly,i);
            putvertex(OutPoly,vkplus1)
        }
    }
}
```
Polygon Clipping – SH Algorithm (6)

- The complexity of SH algorithm is $O(mn)$ where $m$ and $n$ are the numbers of vertices of the clipping and subject polygons respectively.

- No complex data structures or operations are required so the SH algorithm is quite efficient.

- The SH algorithm is appropriate for hardware implementation since the clipping polygon, in general, is constant.
Greiner – Hormann Algorithm

- Suitable for general clipping polygons (C) and subject polygons (S)
- The polygons can be arbitrary closed polygons, even self intersecting
- The complexity of step 1 and 2 is $O(mn)$ where $m$ and $n$ are the numbers of vertices of the C and S polygon respectively
- The overall complexity of the GF algorithm is $O(mn)$
- In practice, the complex data structures used in GF algorithm makes it less efficient than the SH algorithm
Polygon Clipping – GH Algorithm (2)

- GH algorithm is based on the winding number test for point $p$ in polygon $P$, symbolically $\mapsto \omega(P, p)$

- $\omega(P, p)$ does not change so long as the topological relation of the point $p$ and the polygon $P$ remains constant

- If $p$ crosses $P$ the $\omega(P, p)$ is incremented or decremented

- If $\omega(P, p)$ is odd then $p$ is inside $P$, otherwise it is outside
The 3 steps of the GH algorithm:

1. Trace the perimeter $S$ starting from a vertex $v_{s0}$. An imaginary stencil toggles between on and off state every time the perimeter of $C$ is crossed. Its initial state is on if $v_{s0}$ is inside $C$ and off otherwise. It thus computes the part of the perimeter of $S$ that is inside $C$.

2. As step 1 but reverse the roles of $S$ and $C$. The part of the perimeter of $C$ that is inside $S$ is thus computed.

3. The union of the results of steps 1 and 2 is the result of clipping $S$ against $C$ (or equivalently $C$ against $S$).
Polygon Clipping – GH Algorithm (4)

- GH algorithm example:
  (a) The initial $S, C$ polygons
  (b) After step 1 of GH
  (c) After step 2 of GH
  (d) The final result
Polygon Clipping – GH Algorithm (5)

- GH algorithm computes the intersection of the areas of 2 polygons, $C \cap S$
- It easily generalizes to compute $C \cup S$, $C - S$ and $S - C$ by changing the initial states of the stencils for $S$ and $C$
- Obviously there are 4 possible combinations of the initial state
- These generalizations are not useful for the clipping problem