Algorithm 9.6 **Scan line polygon fill algorithm**

Input: Set of vertices of the polygon
Output: Interior pixels with specified color
From the vertices, determine the maximum and minimum scan lines (i.e., maximum and minimum y values) for the polygon.
Set scanline = minimum
repeat
  for Each edge (pair of vertices (x_1, y_1) and (x_2, y_2)) of the polygon **do**
    if ($y_1 \leq$ scanline $\leq y_2$) OR ($y_2 \leq$ scanline $\leq y_1$) **then**
      Determine edge–scanline intersection point
    end if.
  end for
  Sort the intersection points in increasing order of x coordinate
  Apply specified color to the pixels that are within the intersection points
  Set scanline = scanline + 1
until scanline = maximum

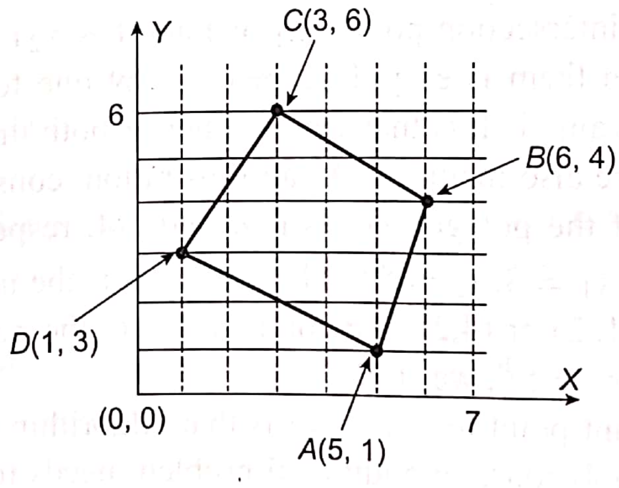


Fig. 9.5 Illustrative example of the scan line polygon fill algorithm

Algorithm 9.4 **Seed fill algorithm**

```
Input: Boundary pixel color, specified color, and the seed (interior pixel) p
Output: Interior pixels with specified color
Push(p) to Stack
repeat
    Set current pixel = Pop(Stack)
    Apply specified color to the current pixel
    for Each of the four connected pixels (four-connected) or eight connected pixels (eight-connected)
    of current pixel do
        if (connected pixel color ≠ boundary color) OR (connected pixel color ≠ specified color) then
            Push(connected pixel)
        end if
    end for
until Stack is empty
```

Algorithm 9.5 **Flood fill algorithm**

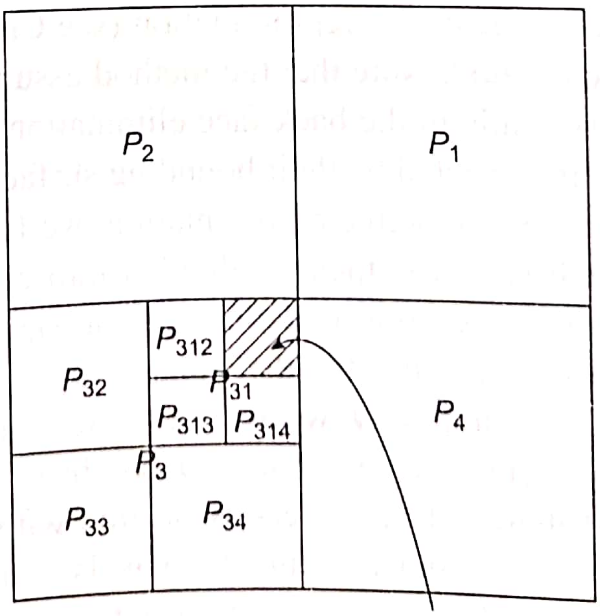
```
Input: Interior pixel color, specified color, and the seed (interior pixel) p
Output: Interior pixels with specified color
Push(p) to Stack
repeat
    Set current pixel = Pop(Stack)
    Apply specified color to the current pixel
    for Each of the four connected pixels (four-connected) or eight connected pixels (eight-connected)
    of current pixel do
        if (Color(connected pixel) = interior color then
            Push(connected pixel)
        end if
    end for
until Stack is empty
```

Algorithm 9.3 **Midpoint circle drawing algorithm**

```
Input: The radius of the circle r
Output: Set of pixels P to render the line segment
Compute p = 5/4 − r
Set x = 0, y = RoundOff(r)
Add the four axis points (0, y), (y, 0), (0, −y) and (−y, 0) to P
repeat
  if p < 0 then
    Set p = p + 2x + 3
    Set x = x + 1
  else
    Set p = p + 2(x − y) + 5
    Set x = x + 1, y = y − 1
  end if
  Add (x, y) and the seven symmetric points {(y, x), (y, −x), (x, −y), (−x, −y), (−y, −x), (−y, x), (−x, y)} to P
until x ≥ y
```

Algorithm 8.4 **Warnock's Algorithm**

```
Input: The screen region
function Warnock (Projected region P)
Divide the input region P into four equal sized subregions P1, P2, P3 and P4
for each subregion Pi do
    if there is no surface in Pi or Pi equals the pixel size then
        Assign background color to Pi
    else if the nearest surface completely overlaps Pi then
        Assign the surface color to Pi
    else
        Warnock (Pi)
    end if
end for
```



Subregion P_{311} overlapped by the surface

Algorithm 8.3 **Painter's Algorithm**

```
Input: List of surfaces S = {s_1, s_2, ··· s_n}, in sorted order (of increasing maximum depth value).
Output: Final frame buffer values.
Set a flag Reorder=OFF
repeat
  Set s = s_n (i.e., the last element of S)
  for for each surface s_i in S where 1 ≤ i < n do
    if z_min(s) < z_max(s_i) (that means, there is depth overlap) then
      if (bounding rectangles of the two surfaces on the view plane do not overlap then
        Set i = i + 1 and continue loop.
      else if s is completely behind s_i then
        Set i = i + 1 and continue loop
      else if s_i is completely in front of s then
        Set i = i + 1 and continue loop
      else if projections of s and s_i do not overlap then
        Set i = i + 1 and continue loop
      else
        Swap the positions of s and s_i in S
        Set Reorder = ON
        Exit inner loop
      end if
    end if
  end for
  if Reorder = OFF then
    Invoke rendering routine for s
    Set S = S − s
  else
    Set Reorder = OFF
  end if
until S = NULL
```

Algorithm 8.1 **Depth-buffer algorithm**

```
Input: Depth-buffer DB[][] initialized to 1.0, frame buffer FB[][] initialized to background color
  value, list of surfaces S, list of projected points for each surface.
Output: DB[][] and FB[][] with appropriate values.
for each surface in S do
    for each projected pixel position of the surface i, j, starting from the top-leftmost projected pixel
    position do
        Calculate depth d of the projected point on the surface.
        if d <DB[i][j] then
            Set DB[i][j]=d
            Set FB[i][j]=surface color
        end if
    end for
end for
```

Algorithm 7.5 shows a quick and easy way of creating a triangle mesh from a convex polygon.

Algorithm 7.5 **Algorithm to create triangle mesh from a convex polygon**

Input: Set of vertices $V = \{v_1, v_2, \cdots, v_n\}$, the triangle set $V_T = NULL$
Output: Set of triangles V_T
repeat
    Take first three vertices from V to form the vertex set v_t representing a triangle
    Add v_t to V_T
    Reset V by removing from it the middle vertex of v_t
until V contains only three vertices
Add V to V_T and Return V_T

Let us consider an example to understand the idea of Algorithm 7.5. Suppose we want to create a triangle mesh from the polygon shown in Fig. 7.10.

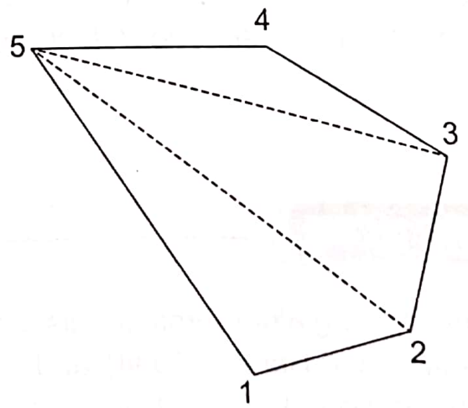


Fig. 7.10 Polygon for creating triangular mesh

Algorithm 7.4 **Weiler–Atherton fill-area clipping algorithm**

1: Start from a vertex inside the window.
2: Process the edges of the polygon fill-area in any particular order (clockwise or anti-clockwise). Continue the processing till an edge of the fill-area is found that crosses a window boundary from inside to outside. The intersection point of the edge with the window boundary is the *exit-intersection* point. Record the intersection point.
3: From the exit-intersection point, process the window boundaries in the same direction (clockwise or anti-clockwise). Continue processing till another intersection point (of a fill-area edge with a window boundary) is found.
4: **if** the intersection point is a *new* point not yet processed **then**
5:     Record the intersection point
6:     Continue processing the fill-area edges till a previously processed vertex is encountered
7: **end if**
8: Form the output vertex list V_{out} for this section of the clipped fill-area
9: **if** all the polygon fill-area edges have been processed **then**
10:     Output V_{out}
11: **else**
12:     Return to the exit-intersection point
13:     Continue processing the fill-are a edges in the same order (clockwise or anti-clockwise) till another intersection point (of a fill-area edge with a window boundary) is found
14:     Go to the line 4
15: **end if**

Algorithm 7.3 **Sutherland–Hodgeman fill-area clipping algorithm**

Input: Four clippers: $c_l = x_{\min}, c_r = x_{\max}, c_t = y_{\max}, c_b = y_{\min}$ corresponding to the left, right, top, and bottom window boundaries, respectively. The polygon is specified in terms of its vertex list $V_{in} = \{v_1, v_2, \cdots, v_n\}$, where the vertices are named anti-clockwise.
for each clipper in the order c_l, c_r, c_t, c_b **do**
  Set output vertex list $V_{out} = NULL, i = 1, j = 2$
  repeat
    Consider the vertex pair v_i and v_j in V_{in}
    if v_i is *inside* and v_j *outside* of the clipper **then**
      ADD the intersection point of the clipper with the edge (v_i, v_j) to V_{out}
    else if both the vertices are *inside* the clipper **then**
      ADD v_j to V_{out}
    else if v_i is *outside* and v_j *inside* of the clipper **then**
      ADD the intersection point of the clipper with the edge (v_i, v_j) and v_j to V_{out}
    else
      ADD NULL to V_{out}
    end if
  until all edges (i.e., consecutive vertex pairs) in V_{in} are checked
  Set $V_{in} = V_{out}$
end for
Return V_{out}

Algorithm 7.2 Liang–Barsky line clipping algorithm

Input: A line segment with end points $P(x_1, y_1)$ and $Q(x_2, y_2)$, the window parameters $(x_{min}, x_{max}, y_{min}, y_{max})$. A window boundary is denoted by k where k can take the values 1, 2, 3, or 4 corresponding to the left, right, below, and above boundary, respectively.

Output: Clipped line segment

Calculate $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$

Calculate $p_1 = -\Delta x$, $q_1 = x_1 - x_{min}$

Calculate $p_2 = \Delta x$, $q_2 = x_{max} - x_1$

Calculate $p_3 = -\Delta y$, $q_3 = y_1 - y_{min}$

Calculate $p_4 = \Delta y$ $q_4 = y_{max} - y_1$

if $p_k = 0$ and $q_k < 0$ for any $k = 1, 2, 3, 4$ **then**

  Discard the line as it is completely outside the window

else

  Compute $r_k = \frac{q_k}{p_k}$ for all those boundaries k for which $p_k < 0$. Determine parameter $u_1 = \max\{0, r_k\}$.

  Compute $r_k = \frac{q_k}{p_k}$ for all those boundaries k for which $p_k > 0$. Determine parameter $u_2 = min\{1, r_k\}$.

  if $u_1 > u_2$ **then**

    Eliminate the line as it is completely outside the window

  else if $u_1 = 0$ **then**

    There is one intersection point, calculated as $x_2 = x_1 + u_2\Delta x$, $y_2 = y_1 + u_2\Delta y$

    Return the two end points (x_1, y_1) and (x_2, y_2)

  else

    There are two intersection points, calculated as: $x_1' = x_1 + u_1\Delta x$, $y_1' = y_1 + u_1\Delta y$ and $x_2 = x_1 + u_2\Delta x$, $y_2 = y_1 + u_2\Delta y$

    Return the two end points (x_1', y_1') and (x_2, y_2)

  end if

end if

Algorithm 7.1 **Cohen–Sutherland line clipping algorithm**

Input: A line segment with end points PQ and the window parameters $(x_{\min}, x_{\max}, y_{\min}, y_{\max})$
Output: Clipped line segment (NULL if the line is completely outside)
for each end point with coordinate (x,y), where $\text{sign}(a) = 1$ if a is positive, 0 otherwise **do**
    Bit 3 = sign $(y - y_{\max})$
    Bit 2 = sign $(y_{\min} - y)$
    Bit 1 = sign $(x - x_{\max})$
    Bit 0 = sign $(x_{\min} - x)$
end for
if both the end point region codes are 0000 **then**
    RETURN PQ.
else if logical AND (i.e., bitwise AND) of the end point region codes $\neq$ 0000 **then**
    RETURN NULL
else
    for each boundary b_i where b_i = above, below, right, left, **do**
        Check corresponding bit values of the two end point region codes
        if the bit values are same, **then**
            Check next boundary
        else
            Determine b_i-line intersection point using line equation
            Assign region code to the intersection point
            Discard the line from the end point outside b_i to the intersection point (as it is outside the window)
            if the region codes of both the intersection point and the remaining end point are 0000 **then**
                Reset PQ with the *new* end points.
            end if
        end if
    end for
    RETURN modified PQ
end if