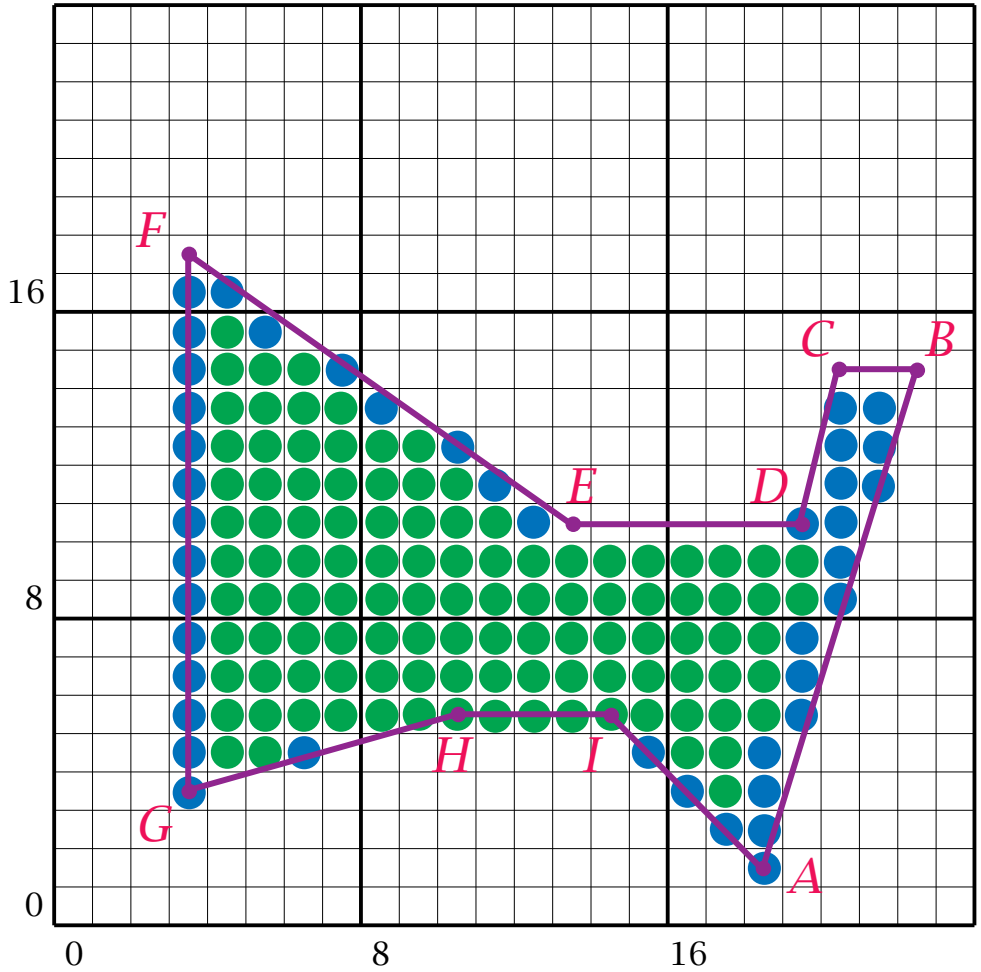
Scanline Fill Algorithm

- Terminology
- Generalities
- Scan-Line Polygon Fill Algorithm
- Boundary-Fill Algorithm
- Flood-Fill Algorithm

Interior Pixel Convention

- Pixels that lie in the interior of a polygon belong to that polygon, and can be filled.
- Pixels that lie on a left boundary or a lower boundary of a polygon belong to that polygon, and can be filled.
- Pixels that have centers that fall outside the polygon, are said to be exterior and should not be drawn.
- Pixels that lie on a right or an upper boundary do not belong to that polygon, and should not drawn.

Example



The scan extrema (blue) and interior (green) pixels that are obtained using our interior pixel convention for the given polygon (purple).

Basic Scan-Fill Algorithm (Foley et al., pp. 92–99)

1. For each non-horizontal edge of the polygon boundary identify the upper and lower endpoints, (x_l, y_l) and (x_u, y_u), such that $y_u > y_l$, and contstruct a record for each that contains
   - y_u, the y-coordinate at the upper endpoint
   - $x = x_l$, the current x-intersection
   - $w = 1/m = (x_u - x_l)/(y_u - y_l)$, the reciprocal of the slope of the edge
2. Set the AET (the active edge table) to be empty.
3. Apply a bucket sort algorithm to sort the edges using the y_l as the primary key, and x_l as the secondary, and w as the tertiary. N.B., Each bucket contains a list. The set of buckets is called the ET (edge table):

Example – Buckets

23 ϕ

⋮ ⋮

11 ϕ

10 $\{y_F, x_E, w_{EF}\} \rightarrow \{y_C, x_D, w_{DC}\}$

9 ϕ

⋮ ⋮

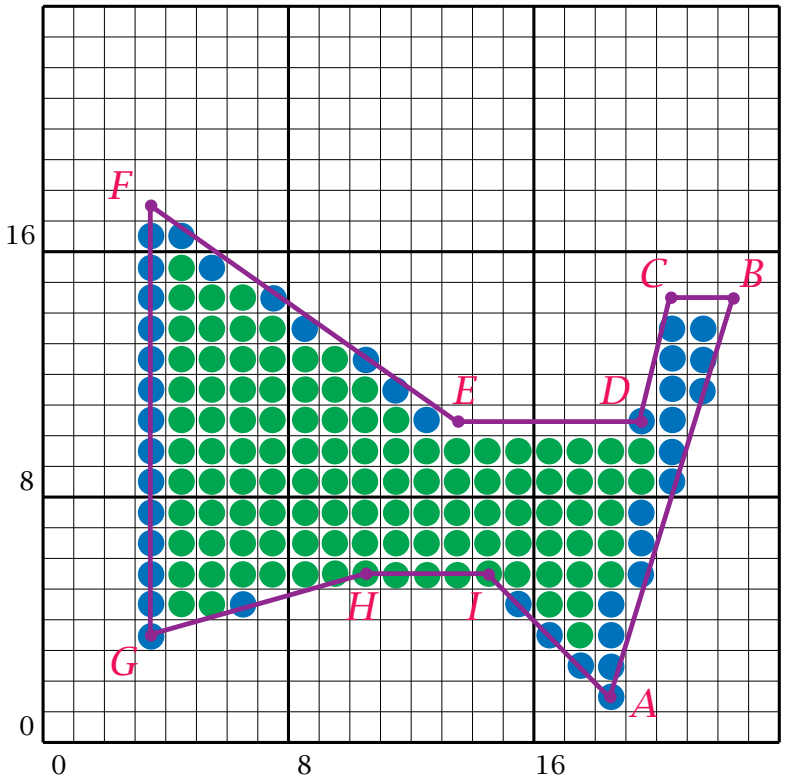
4 ϕ

3 $\{y_F, x_G, w_{GF}\} \rightarrow \{y_H, x_G, w_{GH}\}$

2 ϕ

1 $\{y_I, x_A, w_{AI}\} \rightarrow \{y_B, x_A, w_{AB}\}$

0 ϕ

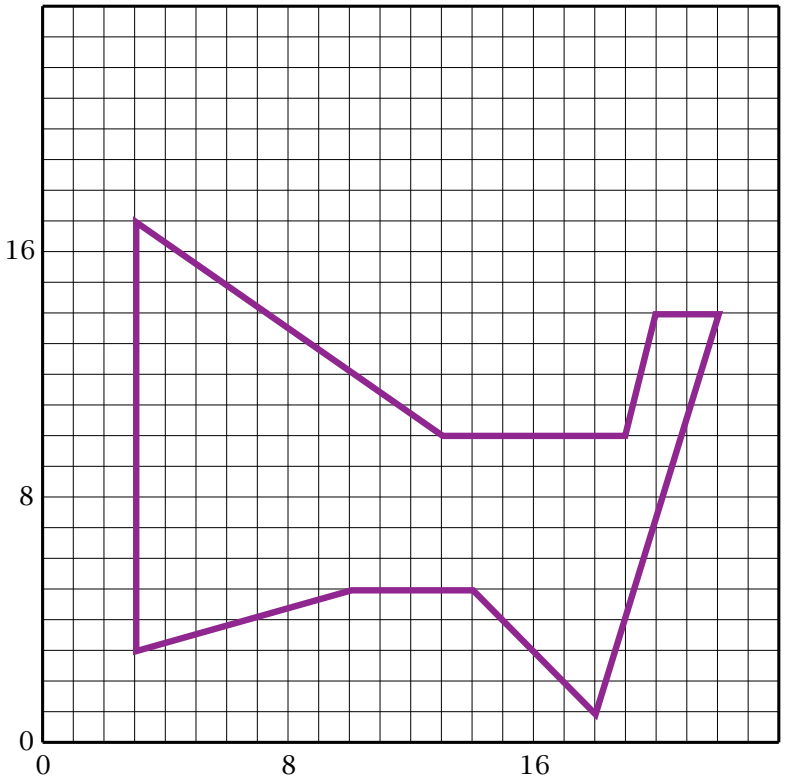


Basic Scan-Fill Algorithm (cont.)

4. Set y equal to the smallest index in the ET that has a non empty bucket.
5. Repeat until the ET and the AET are empty:
   (a) Move any edges from bucket y in the ET to the AET.
   (b) Remove any edges from the AET that have a y_u equal to y.
   (c) Sort the AET according to x.
   (d) Fill in the requisite pixels between the even and odd adjacent pairs of intersections in the AET: round up, $\lceil x \rceil$ the x-coordinate of "left" intersections, round down, $\lceil x - 1 \rceil$ that of the "right" intersections.
   (e) Increment y by one.
   (f) Update $x \leftarrow x + w$ for every nonvertical edge in the AET.

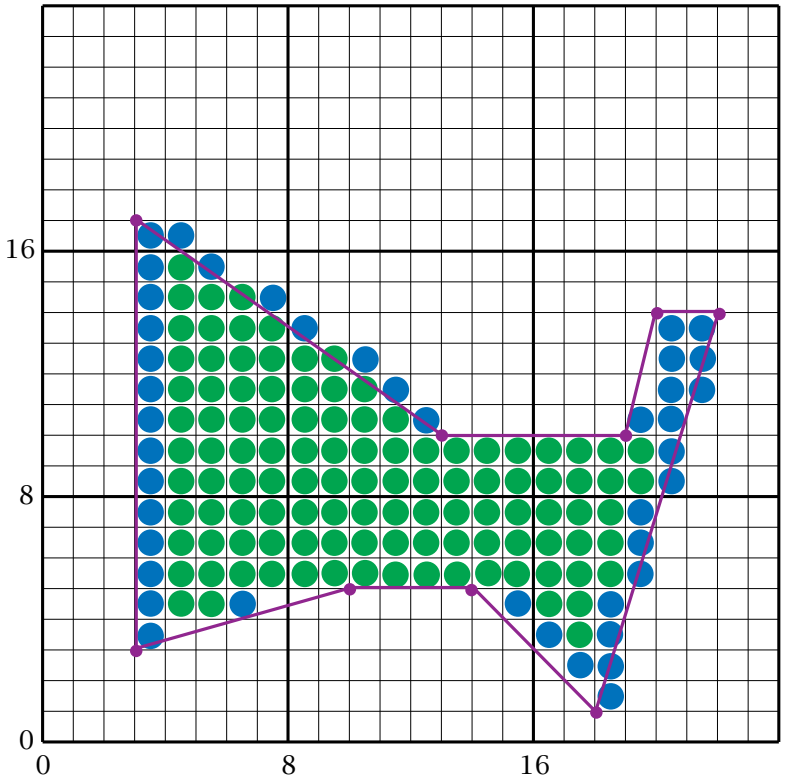
Screen Grid Coordinates

See page 114 of Hearn and Baker.



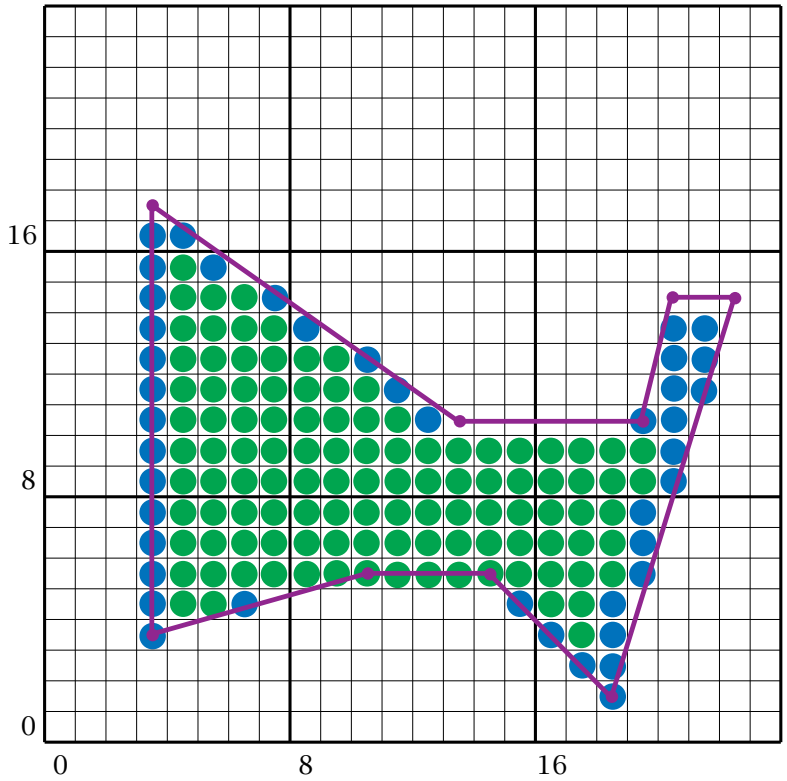
Integer coordinates correspond to the grid intersections: rounding can now be implemented via truncation.

Screen Grid Coordinates



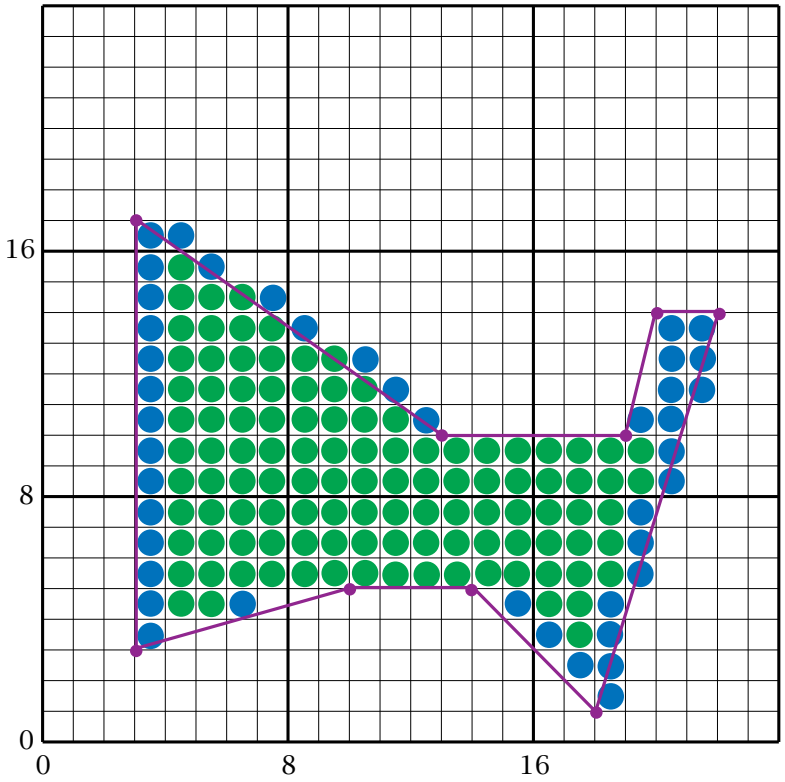
Integer coordinates correspond to the grid intersections: rounding can now be implemented via truncation. (See page 114 of Hearn and Baker.)

Pixel Center Coordinates



Integer coordinates correspond to the pixel centers.

Screen Grid Coordinates



Integer coordinates correspond to the grid intersections: rounding can now be implemented via truncation. (See page 114 of Hearn and Baker.)

Scan-Fill Algorithm — The Code

The edge data structure

```
typedef struct tEdge {
  int yUpper;
  float xIntersect, dxPerScan;
  struct tEdge * next;
} Edge;

typedef struct tdcPt {
  int x;
  int y;
} dcPt;
```

Scan-Fill Algorithm — The Code (cont.)

```
void scanFill (int cnt, dcPt * pts) {
  Edge * edges[WINDOW_HEIGHT], * active;
  int i, scan;

  for (i=0; i<WINDOW_HEIGHT; i++) {
    edges[i] = (Edge *) malloc (sizeof (Edge));
    edges[i]->next = NULL;
  }
  buildEdgeList (cnt, pts, edges);
  active = (Edge *) malloc (sizeof (Edge));
  active->next = NULL;
```

```
  for (scan=0; scan<WINDOW_HEIGHT; scan++) {
    buildActiveList (scan, active, edges);
    if (active->next) {
      fillScan (scan, active);
      updateActiveList (scan, active);
      resortActiveList (active);
    }
  }
  /* Free edge records that have been malloc'ed ... */
}
```

Scan-Fill Algorithm — The Code (cont.)

```
void scanFill (int cnt, dcPt * pts) {
  Edge * edges[WINDOW_HEIGHT], * active;
  int i, scan;

  for (i=0; i<WINDOW_HEIGHT; i++) {
    edges[i] = (Edge *) malloc (sizeof (Edge));
    edges[i]->next = NULL;
  }
  buildEdgeList (cnt, pts, edges);
  active = (Edge *) malloc (sizeof (Edge));
  active->next = NULL;
```

```
  for (scan=0; scan<WINDOW_HEIGHT; scan++) {
    buildActiveList (scan, active, edges);
    if (active->next) {
      fillScan (scan, active);
      updateActiveList (scan, active);
      resortActiveList (active);
    }
  }
  /* Free edge records that have been malloc'ed ... */
}
```

Scan-Fill Algorithm — The Code (cont.)

```
void buildEdgeList (int cnt, dcPt * pts, Edge * edges[]) {
  Edge * edge;
  dcPt v1, v2;
  int i, yPrev = pts[cnt - 2].y;

  v1.x = pts[cnt-1].x; v1.y = pts[cnt-1].y;
  for (i=0; i<cnt; i++) {
    v2 = pts[i];
    if (v1.y != v2.y) {                    /* nonhorizontal line */
      edge = (Edge *) malloc (sizeof (Edge));
      if (v1.y < v2.y)                     /* up-going edge      */
         makeEdgeRec (v1, v2, yNext (i, cnt, pts), edge, edges);
      else                                 /* down-going edge    */
         makeEdgeRec (v2, v1, yPrev, edge, edges);
    }
```

```
    yPrev = v1.y;
    v1 = v2;
  }
}
/* For an index, return y-coordinate of next nonhorizontal line */
int yNext (int k, int cnt, dcPt * pts) {
  int j;
  if ((k+1) > (cnt-1))
    j = 0;
  else
    j = k + 1;
  while (pts[k].y == pts[j].y)
    if ((j+1) > (cnt-1))
      j = 0;
    else
      j++;
  return (pts[j].y);
}
```

Scan-Fill Algorithm — The Code (cont.)

```
void buildEdgeList (int cnt, dcPt * pts, Edge * edges[]) {
  Edge * edge;
  dcPt v1, v2;
  int i, yPrev = pts[cnt - 2].y;

  v1.x = pts[cnt-1].x; v1.y = pts[cnt-1].y;
  for (i=0; i<cnt; i++) {
    v2 = pts[i];
    if (v1.y != v2.y) {                    /* nonhorizontal line */
      edge = (Edge *) malloc (sizeof (Edge));
      if (v1.y < v2.y)                     /* up-going edge      */
        makeEdgeRec (v1, v2, yNext (i, cnt, pts), edge, edges);
      else                                 /* down-going edge    */
        makeEdgeRec (v2, v1, yPrev, edge, edges);
    }
```

Scan-Fill Algorithm — The Code (cont.)

```
/* Store lower-y coordinate and inverse slope for each edge.  Adjust
   and store upper-y coordinate for edges that are the lower member
   of a monotically increasing or decreasing pair of edges */
void makeEdgeRec
  (dcPt lower, dcPt upper, int yComp, Edge * edge, Edge * edges[])
{
  edge->dxPerScan =
    (float) (upper.x - lower.x) / (upper.y - lower.y);
  edge->xIntersect = lower.x;
  if (upper.y < yComp)
    edge->yUpper = upper.y - 1;
  else
    edge->yUpper = upper.y;
  insertEdge (edges[lower.y], edge);
}
```

Scan-Fill Algorithm — The Code (cont.)

```
/* Inserts edge into list in order of increasing xIntersect field. */
void insertEdge (Edge * list, Edge * edge) {
  Edge * p, * q = list;

  p = q->next;
  while (p != NULL) {
    if (edge->xIntersect < p->xIntersect)
      p = NULL;
    else {
      q = p;
      p = p->next;
    }
  }
  edge->next = q->next;
  q->next = edge;
}
```

Scan-Fill Algorithm — The Code (cont.)

```
void scanFill (int cnt, dcPt * pts) {
  Edge * edges[WINDOW_HEIGHT], * active;
  int i, scan;

  for (i=0; i<WINDOW_HEIGHT; i++) {
    edges[i] = (Edge *) malloc (sizeof (Edge));
    edges[i]->next = NULL;
  }
  buildEdgeList (cnt, pts, edges);
  active = (Edge *) malloc (sizeof (Edge));
  active->next = NULL;
```

```
  for (scan=0; scan<WINDOW_HEIGHT; scan++) {
    buildActiveList (scan, active, edges);
    if (active->next) {
      fillScan (scan, active);
      updateActiveList (scan, active);
      resortActiveList (active);
    }
  }
  /* Free edge records that have been malloc'ed ... */
}
```

Scan-Fill Algorithm — The Code (cont.)

```
void buildActiveList (int scan, Edge * active, Edge * edges[])
{
  Edge * p, * q;

  p = edges[scan]->next;
  while (p) {
    q = p->next;
    insertEdge (active, p);
    p = q;
  }
}
```

Scan-Fill Algorithm — The Code (cont.)

```
void scanFill (int cnt, dcPt * pts) {
  Edge * edges[WINDOW_HEIGHT], * active;
  int i, scan;

  for (i=0; i<WINDOW_HEIGHT; i++) {
    edges[i] = (Edge *) malloc (sizeof (Edge));
    edges[i]->next = NULL;
  }
  buildEdgeList (cnt, pts, edges);
  active = (Edge *) malloc (sizeof (Edge));
  active->next = NULL;
```

```
  for (scan=0; scan<WINDOW_HEIGHT; scan++) {
    buildActiveList (scan, active, edges);
    if (active->next) {
      fillScan (scan, active);
      updateActiveList (scan, active);
      resortActiveList (active);
    }
  }
  /* Free edge records that have been malloc'ed ... */
}
```

Scan-Fill Algorithm — The Code (cont.)

```
void fillScan (int scan, Edge * active) {
  Edge * p1, * p2;
  int i;

  p1 = active->next;
  while (p1) {
    p2 = p1->next;
    for (i=p1->xIntersect; i<p2->xIntersect; i++)
      setPixel ((int) i, scan);
    p1 = p2->next;
  }
}
```

Scan-Fill Algorithm — The Code (cont.)

```
void scanFill (int cnt, dcPt * pts) {
  Edge * edges[WINDOW_HEIGHT], * active;
  int i, scan;

  for (i=0; i<WINDOW_HEIGHT; i++) {
    edges[i] = (Edge *) malloc (sizeof (Edge));
    edges[i]->next = NULL;
  }
  buildEdgeList (cnt, pts, edges);
  active = (Edge *) malloc (sizeof (Edge));
  active->next = NULL;
```

```
  for (scan=0; scan<WINDOW_HEIGHT; scan++) {
    buildActiveList (scan, active, edges);
    if (active->next) {
      fillScan (scan, active);
      updateActiveList (scan, active);
      resortActiveList (active);
    }
  }
  /* Free edge records that have been malloc'ed ... */
}
```

Scan-Fill Algorithm — The Code (cont.)

```
/* Delete completed edges. Update 'xIntersect' field for others */
void updateActiveList (int scan, Edge * active) {
  Edge * q = active, * p = active->next;

  while (p)
    if (scan >= p->yUpper) {
      p = p->next;
      deleteAfter (q);
    } else {
      p->xIntersect = p->xIntersect + p->dxPerScan;
      q = p;
      p = p->next;
    }
}
```

```
void deleteAfter (Edge * q) {
  Edge * p = q->next;

  q->next = p->next;
  free (p);
}
```

Scan-Fill Algorithm — The Code (cont.)

```
void scanFill (int cnt, dcPt * pts) {
  Edge * edges[WINDOW_HEIGHT], * active;
  int i, scan;

  for (i=0; i<WINDOW_HEIGHT; i++) {
    edges[i] = (Edge *) malloc (sizeof (Edge));
    edges[i]->next = NULL;
  }
  buildEdgeList (cnt, pts, edges);
  active = (Edge *) malloc (sizeof (Edge));
  active->next = NULL;
```

```
  for (scan=0; scan<WINDOW_HEIGHT; scan++) {
    buildActiveList (scan, active, edges);
    if (active->next) {
      fillScan (scan, active);
      updateActiveList (scan, active);
      resortActiveList (active);
    }
  }
  /* Free edge records that have been malloc'ed ... */
}
```

Scan-Fill Algorithm — The Code (cont.)

```
void resortActiveList (Edge * active) {
  Edge * q, * p = active->next;

  active->next = NULL;
  while (p) {
    q = p->next;
    insertEdge (active, p);
    p = q;
  }
}
```

Remarks

- The intersection update can be implemented more efficiently using integer arithmetic. If $m > 1$, for example:

```
int x = xMin;
int numerator   = xMax - yMin;
int denominator = yMax - yMin;
int counter = denominator;
...
/* When updating the edge intersections */
counter +=  numerator;
if (counter > denominator) {
  x++;
  counter -= denominator;
}
...
```

Remarks (cont.)

- The fill can be a periodic pattern using either

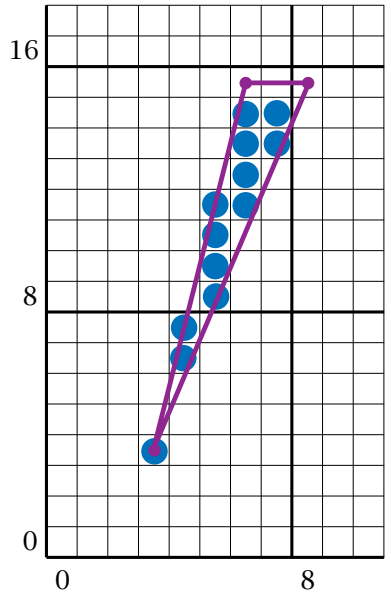
```
if (pattern[x % M][y % N])
   setPixel(x, y);
```

or if setPixel takes a value for a third argument:

```
   setPixel(x, y, pattern[x % M][y % N]);
```

Problems

- What happens if a vertex is shared by more than one polygon, e.g. three triangles?
- What happens if the polygon intersects itself?
- What happens for a “sliver”?

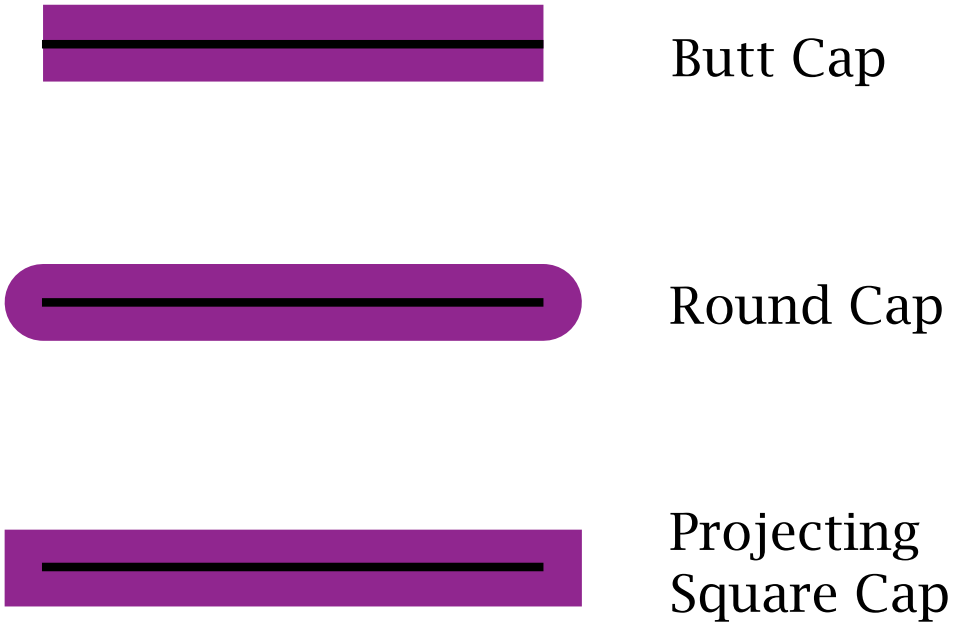


A sliver

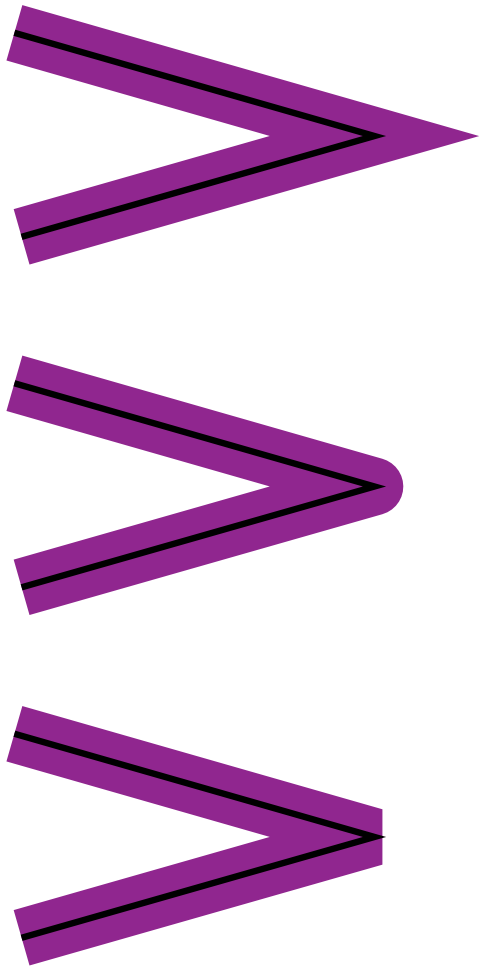
Attributes

1. dashed lines
2. line thickness
   (a) parallel lines
   (b) vertical or horizontal spans
   (c) rectangular pens
   (d) scan-line fill
3. antialiasing

Line Endcaps



Line Joins



Miter Join

Round Join

Bevel Join