CSE251
Basics of Computer Graphics
Module: Rasterization Module

Avinash Sharma

Spring 2018

Patterned Line

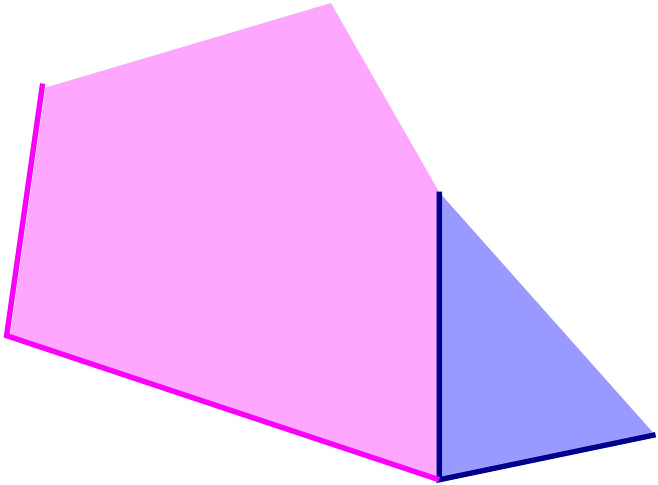
- Represent the pattern as an array of booleans/bits, say, 16 pixels long.
- Fill first half with 1 and rest with 0 for dashed lines.
- Perform WritePixel(x, y) only if pattern bit is a 1.

```
if (pattern[i])   WritePixel(x, y)
```

where **i** is an index variable starting with 0 giving the ordinal number (modulo 16) of the pixel from starting point.

Shared Points/Edges

- It is common to have points common between two lines and edges between two polygons.
- They will be scan converted **twice**. Not efficient. Sometimes harmful.
- Solution: Treat the intervals closed on the left and open on the right. $[x_m, x_M)$ & $[y_m, y_M)$
- Thus, edges of polygons on the top and right boundaries are not drawn.



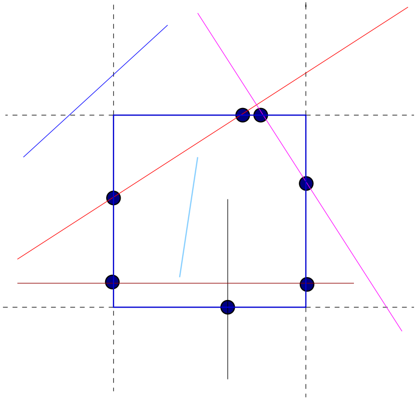
Clipping

- Often, many points map to outside the range in the normalized 2D space.
- Think of the FB as an infinite canvas, of which a small rectangular portion is sent to the screen.
- Let's get greedy: draw only the portion that is visible. That is, **clip** the primitives to a *clip-rectangle*.
- **Scissoring:** Doing scan-conversion and clipping together.

Clipping Points

- Clip rectangle: (x_m, y_m) to (x_M, y_M).
- For (x, y): $\quad x_m \leq x \leq x_M, \quad y_m \leq y \leq y_M$
- Can use this to clip any primitives: Scan convert normally. Check above condition before writing the pixel.
- Simple, but perhaps we do more work than necessary.
- Analytically clip to the rectangle, then scan convert.

Clipping Lines

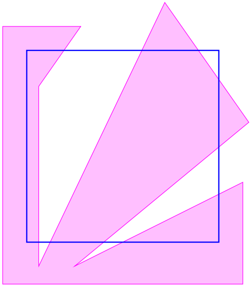


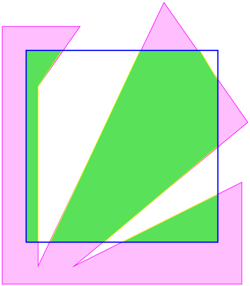
Popular: Cohen-Sutherland Algorithm

Clipping Polygons

- Restrict drawing/filling of a polygon to the inside of the clip rectangle.
- A convex polygon remains convex after clipping.
- A concave polygon can be clipped to multiple polygons.
- Can perform by intersecting to the four clip edges in turn.

An Example





Popular: Sutherland-Hodgman Algorithm

Filled Rectangles

- Write to all pixels within the rectangle.

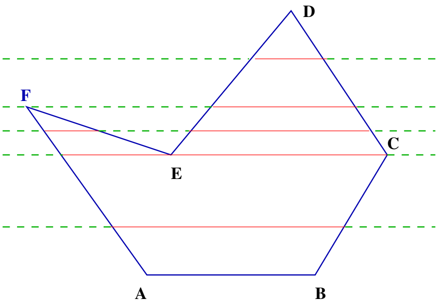
Function FilledRectangle (x_m, x_M, y_m, y_M, colour)
    for $x_m \leq x \leq x_M$ do
        for $y_m \leq y \leq y_M$ do
            WritePixel (x, y. colour)
EndFunction

- How about non-upright rectangles? General polygons?

Filled Polygons

- For each scan line, identify **spans** of the polygon interior. Strictly interior points only.
- For each scan line, the **parity** determines if we are inside or ouside the polygon. Odd is inside, Even is outside.
- Trick: End-points count towards parity enumeration only if it is a y_{min} point.
- Span extrema points and other information can be computed during scan conversion. This information is stored in a suitable data structure for the polygon.

Parity Checking



Edge Coherence

- If scan line y intersects with an edge E, it is likely that $y+1$ also does. (Unless intersection is the y_{max} vertex.)
- When moving from y to $y+1$, the X-coordinate goes from x to $x+1/m$. $1/m = (x_2 - x_1)/(y_2 - y_1) = \Delta x \ / \ \Delta y$
- Store the integer part of x, the numerator (Δx) and the denominator (Δy) of the fraction separately.
- For next scan line, add Δx to numerator. If sum goes $> \Delta y$, increment integer portion, subtract Δy from numerator.

Scan Converting Filled Polygons

- Find intersections of each scan line with polygon edges.
- Sort them in increasing X-coordinates.
- Use parity to find interior spans and fill them.
- Most information can be computed during scan conversion. A list of intersecting polygons stored for each scan line.
- Use edge coherence for the computation otherwise.

Special Concerns

- Fill only strictly interior pixels: Fractions rounded up when even parity, rounded down when odd.
- Intersections at integer pixels: Treat interval closed on left, open on right.
- Intersections at vertices: Count only y_m vertex for parity.
- Horizontal edges: Do not count as y_m!

Filled Polygon Scan Conversion

- Perform all of it together. Each scan line should not be intersected with each polygon edge!
- Edges are known when polygon vertices are mapped to screen coordinates.
- Build up an edge table while that is done.
- Scan conversion is performed in the order of scan lines. Edge coherence can be used; an active edge table can keep track of which edges matter for the current scan line.

Scan Conversion: Summary

- Filling the frame buffer given 2D primitives.
- Convert an analytical description of the basic primitives into pixels on an integer grid in the frame buffer.
- Lines, Polygons, Circles, etc. Filled and unfilled primitives.
- Efficient algorithms required since scan conversion is done repeatedly. Special hardware used these days
- 2D Scan Conversion is all, even for 3D graphics.

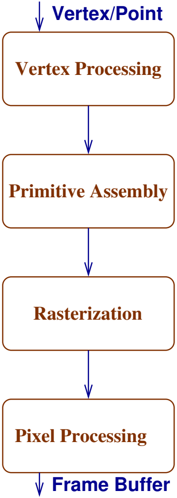
Scan Conversion: Summary

- High level primitives (point, line, polygon) map to window coordinates using transformations.
- Creating the display image on the Frame Buffer is important. Needs to be done efficiently.
- Clipping before filling FB to eliminate futile effort.
- After clipping, line remains line, polygons can become polygons of greater number of sides, etc.
- General polygon algorithm for clipping and scan conversion are necessary.

Now you know ...

- Objects represented/approximated using geometric (1D and 2D) primitives
- Primitives using (2D/3D) points in a natural coord frame
- Points transformed to screen coords in a few steps
- Primitives assembled and converted to pixels on screen
- Colour at each pixle: physics and interpolation
- Visibility evaluation to identify which is closer and farther
- Form image on framebuffer, which appears on the display

Primitive Pipeline



- **Vertex** stage: transform to screen coords, compute lighting in 3D
- Primitive assembly: form polygon/triangle/line
- Rasterization: Clip & Determine pixels inside each primitive
- **Pixel** stage: give **colour**to each pixel, perform Z-buffering