CSE251
Basics of Computer Graphics
Module: Rasterization Module

Avinash Sharma

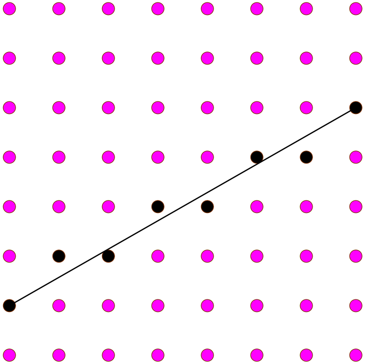
Spring 2019

(Point) Pipeline in Action

- Points are transformed from Object to World to Canonical to Window coordinates.
- Each 3D point maps to a pixel (i, j) in the window space.
- Lines are made out of two points. Triangles and polygons are made out of 3 or more points.

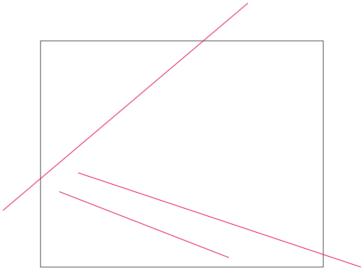
Lines in Action

- Lines are *rasterized* to the pixel grid of the window.
- Find pixels that lie closest to the line. Results in **aliasing**.
- Each pixel needs to be given a color and depth.



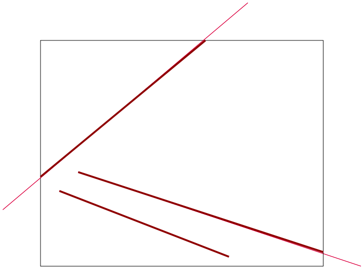
Clipping Lines

- End points map to window coordinates independently.
- World lines needn't map nicely onto points inside the window.



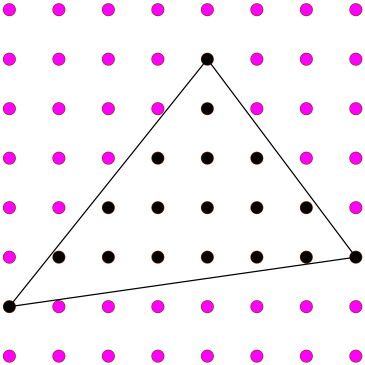
Clipping Lines

- End points map to window coordinates independently.
- World lines needn't map nicely onto points inside the window.
- **Clipping:** Finding part of the line that is *inside* the window.
- Clip first and then rasterize.



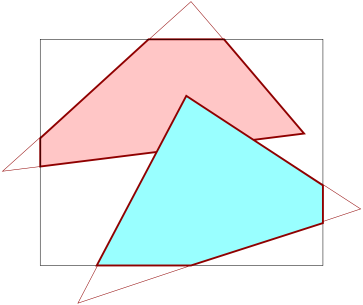
Triangles in Action

- Un-filled triangles are uninteresting. Filled ones represent surfaces.
- Triangles are **scan converted** or **rasterized** to include all pixels inside it.
- Each pixel needs to have a colour and depth.

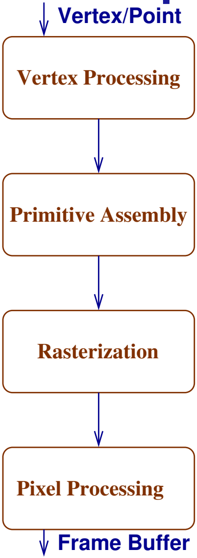


Clipping Triangles

- Only parts of the triangle may lie in the window.
- First clip a triangle to a (planar!) *polygon* that lies inside.
- Scan convert the polygon subsequently.



Primitive Pipeline



- From points, lines, triangles/polygons
- **Vertex** stage: process vertices independently
- Primitive stage: triangle assembly
- Rasterization: Clip & Determine the pixels inside the primitive
- **Pixel** stage: process each pixel independently

Linear Interpolation of Properties

- Each pixel needs: colour, depth, and texture coordinate.
- Assumption: Properties vary linearly across the plane.
- If we know the colour, texture coordinate, and depth at the vertices of the polygon (or line), these can be interpolated to pixels on the inside linearly!
- Colour: 3-vector, texture coord: 2-vector, depth: scalar.
- Rasterization step interpolates these values and gives to each pixel.
- Is the interpolation valid?

Vertex Processing

- Apply ModelView and Projection matrices to the vertex
- Find/send colour: either given or compute from physics!
- Find/send texture coordinates: usually given
- Find/send normals: usually given.
- Can process vertices of a primitive independently
- Modern GPUs: This stage is **programmable!** Can write own *vertex shader* to replace the standard processing.

Rasterization

- Apply viewport transformation.
- Clip primitive to the window or the viewport.
- Evaluate which pixels are part of the primitive.
- Interpolate values for each pixel and queue the pixels or *fragments* for further processing.
- This is computationally quite expensive and is usualy done by a dedicated hardware unit.
- A queue of fragments with associated data are built by this stage.

Pixel or Fragment Processing

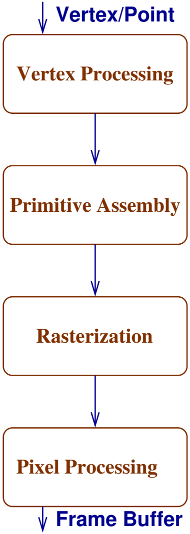
- The pixels generated by the rasterization stage are processed in arbitrary order by this stage.
- Depth value is available already. Can look up Z-buffer to keep or discard the fragment.
- Interpolated colour value can be sent to frame buffer.
- Texture image can be accessed using the texture coordinates. The final colour can be a combination of interpolated and texture colours.
- Modern GPUs: This stage is **programmable!** Can write own *fragment shader* to replace the standard processing.

Programmable GPUs

- Graphics Processing Units are *programmable* today
- Novel shading and lighting can be performed by writing appropriate vetex and pixel **shaders**, beyond OpenGL
- GPUs used parallel processing with 2-4 vertex and 32-64 pixel processing units, all working in parallel. Together, considerable computing power was in a GPU
- Clever idea: Use the power for other processing: matrix multiplication, FFT, sorting, image processing, etc.
- ***GPGPU***: General Processing on GPUs

Primitive Pipeline: Summary

- Basic primitives: Points, Lines, Triangles/Polygons.
- Each constructed fundamentally from points.
- Points map to pixels on screen. Primitives are assembled from points.
- Pipeline of operations on a primitive finds the pixels that are part of it. And performs a few operations on each pixel



Scan Conversion or Rasterization

- Primitives are defined using points, which have been mapped to the screen coordinates.
- In vector graphics, connect the points using a pen directly.
- In Raster Graphics, we create a discretized image of the whole screen onto the **frame buffer** first. The image is scanned automatically onto the display periodically.
- This step is called **Scan Conversion** or **Rasterization**.

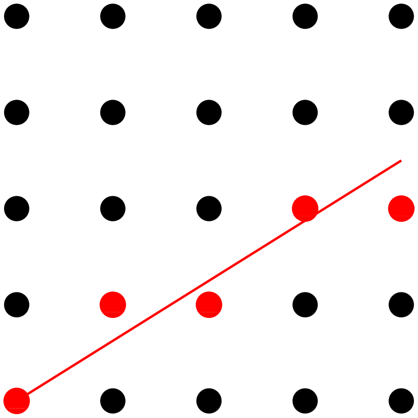
Scan Converting a Point

- The 3D point has been transformed to its screen coordinates (u, v).
- Round the coordinates to frame buffer array indices (i, j).
- Current colour is defined/known. Frame buffer array is initialized to the background colour.
- Perform: frameBuf[i, j] $\leftarrow$ currentColour
- Function WritePixel(i, j, colour) does the above.
- If *PointSize* > 1, assign the colour to a number of points in the neighbourhood!

Scan Converting a Line

- Identify the grid-points that lie on the line and colour them.
- Problem: Given two end-points on the grid, find the pixels on the line connecting them.
- Incremental algorithm or Digital Differential Analyzer (DDA) algorithm.
- Mid-Point Algorithm

Line on an Integer Grid



Incremental Algorithm

Function DrawLine(x_1, y_1, x_2, y_2, **colour**)

  $\Delta x \leftarrow x_2 - x_1, \Delta y \leftarrow y_2 - y_1$, slope $\leftarrow \Delta y / \Delta x$

  $x \leftarrow x_1, y \leftarrow y_1$

  While $(x < x_2)$

    WritePixel $(x$, round(y), **colour**)

    $x \leftarrow x + 1,\ \ y \leftarrow y +$ slope

  EndWhile

  WritePixel $(x_2, y_2$, **colour**)

EndFunction

Incremental Algorithm With Integers

Function DrawLine(x_1, y_1, x_2, y_2, **colour**)

$\Delta x \leftarrow x_2 - x_1, \Delta y \leftarrow y_2 - y_1, sl \leftarrow 0, x \leftarrow x_1, y \leftarrow y_1$

While ($x < x_2$)

WritePixel (x, y, **colour**)

$x \leftarrow x + 1$, sl += Δy.

if (sl $\geq \Delta x$) $\{y \leftarrow y + 1$, sl -= $\Delta x\}$

EndWhile

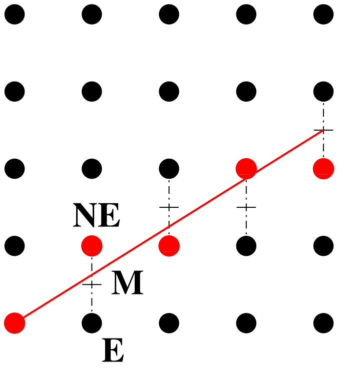
WritePixel (x_2, y_2, **colour**)

EndFunction

Points to Consider

- If abs(slope) > 1, step through y values, adding inverse slopes to x at each step.
- Simple algorithm, easy to implement.
- Floating point calculations were expensive once!
- Can we do with integer arithmetic only?
  Yes: **Bresenham's Algorithm, Mid-Point Line Algorithm**.

Two Options at Each Step!



Mid-Point Line Algorithm

- Line equation: $ax + by + c = 0, \quad a > 0$.
  Let $0 <$ slope $= \Delta y / \Delta x = -a/b < 1.0$
- $F(x, y) = ax + by + c > 0$ for below the line, < 0 for above.
- **NE** if $d = F(\mathbf{M}) > 0$; **E** if $d < 0$; else any!
- $d_{\mathbf{E}} = F(M_{\mathbf{E}}) = d + a, \quad d_{\mathbf{NE}} = d + a + b$
- Therefore, $\Delta_{\mathbf{E}} = a, \quad \Delta_{\mathbf{NE}} = a + b$
- Initial value: $d_0 = F(x_1 + 1, y_1 + \frac{1}{2}) = a + b\ /2$
- Similar analysis for other slopes. Eight cases in total.

Pseudocode

```
Function DrawLine (x1, y1, x2, y2, colour)
        a ← (y2 − y1), b ← (x1 − x2), x ← x1, y ← y1
        d ← 2a + b, ΔE ← 2a, ΔNE ← 2(a + b)
        While (x < x2)
                WritePixel(x, y, colour)
                if (d < 0)            // East
                        d ← d + ΔE, x ← x + 1
                else            // North-East
                        d ← d + ΔNE, x ← x + 1, y ← y + 1
        EndWhile
        WritePixel(i, j, colour)
EndFunction
```

Example: $(10, 10)$ to $(20, 17)$

$$F(x, y) = 7x - 10y + 30, \;\; a = 7, \;\; b = -10$$
$$d_0 = 2 * 7 - 10 = 4, \;\; \Delta_{\mathbf{E}} = 2 * 7 = 14, \;\; \Delta\mathbf{NE} = -6$$

$$\begin{aligned}
d > 0 &: \mathbf{NE}\ (11, 11), \;\; d = 4 + -6 = -2 \\
d < 0 &: \mathbf{E}\ (12, 11), \;\; d = -2 + 14 = 12 \\
d > 0 &: \mathbf{NE}\ (13, 12), \;\; d = 12 + -6 = 6 \\
d > 0 &: \mathbf{NE}\ (14, 13), \;\; d = 6 + -6 = 0 \\
d = 0 &: \mathbf{E}\ (15, 13), \;\; d = 0 + 14 = 14 \\
d > 0 &: \mathbf{NE}\ (16, 14), \;\; d = 14 + -6 = 8
\end{aligned}$$

Later, **NE** (17, 15), **NE** (18, 16), **E** (19, 16), **NE** (20, 17).

Patterned Line

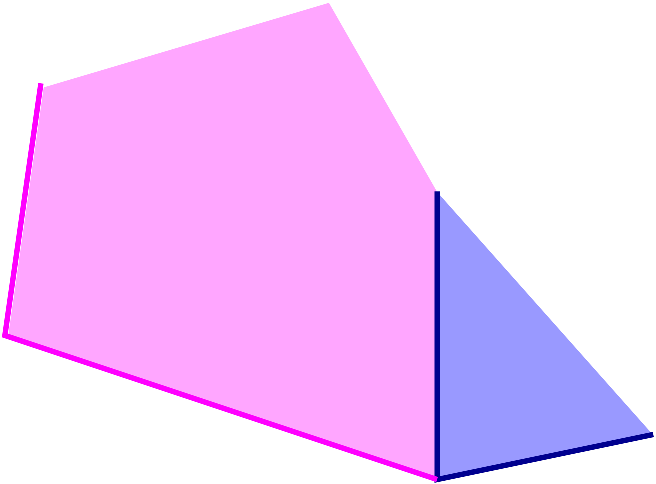
- Represent the pattern as an array of booleans/bits, say, 16 pixels long.
- Fill first half with 1 and rest with 0 for dashed lines.
- Perform WritePixel(x, y) only if pattern bit is a 1.

```
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.



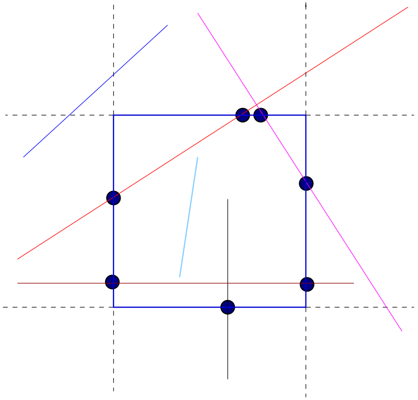
Clipping

- 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

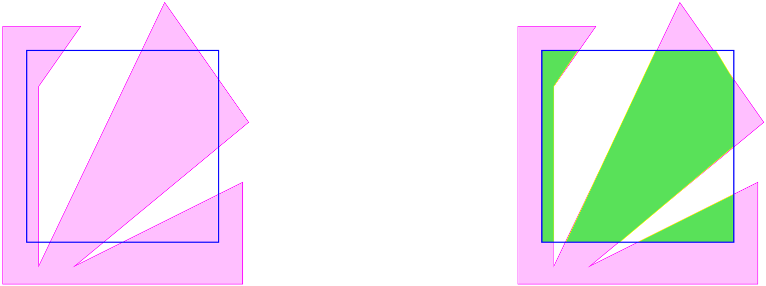


Popular: Cohen-Sutherland Algorithm

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.

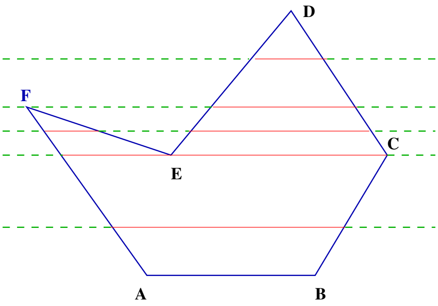
Function FilledRectangle (x_m, x_M, y_m, y_M, **colour**)
    for $x_m \leq x \leq x_M$ do
        for $y_m \leq y \leq y_M$ do
            WritePixel (x, y. **colour**)
EndFunction

- 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.

Relevant Data Structures

Edge Bucket (EB)

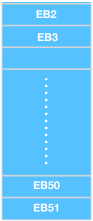
yMax	yMin	x	sign	dX	dY	sum

- **yMax:** Maximum Y position of the edge
- **yMin:** Minimum Y position of the edge
- **x:** The current x position along the scan line, initially starting at the same point as the **yMin** of the edge
- **sign:** The sign of the edge's slope (either -1 or 1)
- **dX:** The absolute delta x (difference) between the edge's vertex points
- **dY:** The absolute delta y (difference) between the edge's vertex points
- **sum:** Initiated to zero. Used as the scan lines are being filled to x to the next position

Relevant Data Structures (cont.)

Edge Table (ET)

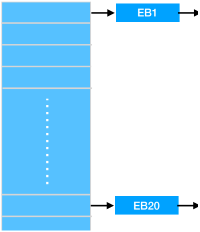
- When creating edges, the vertices of the edge need to be ordered from left to right
- The edges are maintained in increasing yMin order
- The edges are removed from the ET once the Active List is done processing them
- The algorithm is done filling the polygon once all of the edges are removed from the ET



Active edge List (AL)

- Edges are pushed into the AL from the Edge Table once an edge's yMin is equal to the current scan line being processed
- Edges will always be in the AL in pairs
- Edges in the AL are maintained in increasing x order.
- The AL will be re-sorted after every pass

AL



Relevant Data Structures (cont.)

Polygon Scan Conversion Steps:

```
1. Create ET
    1. Process the vertices list in pairs, start with
[numOfVertices-1] and [0].
    2. For each vertex pair, create an edge bucket
2. Sort ET by yMin
3. Process the ET
    1. Start on the scan line equal to theyMin of the first
edge in the ET
    2. While the ET contains edges
        1. Check if any edges in the AL need to be removes
(when yMax == current scan line)
            1. If an edge is removed from the AL, remove
the associated the Edge Bucket from the Edge Table.
        2. If any edges have a yMin == current scan line,
add them to the AL
        3. Sort the edges in AL by X
        4. Fill in the scan line between pairs of edges in
AL
        5. Increment current scan line
        6. Increment all the X's in the AL edges based on
their slope
            1. If the edge's slope is vertical, the
bucket's x member is NOT incremented.
```

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.

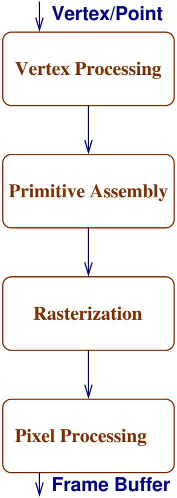
Scan Conversion: Summary

- 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