Computer Graphics

P03 Raster Graphics Algorithms

Aleksandra Pizurica
Ghent University

Telecommunications and Information Processing
Image Processing and Interpretation Group
Overview

• Scan conversion
• Polygon filling
• Clipping in 2D
Scan Conversion
Raster Display

- **PIXEL** (picture element)
- **SCAN LINE** (a row of pixels)

**RASTER**
(a rectangular array of points or dots)

- **Asset:** Control of every picture element (rich patterns)
- **Problem:** limited **resolution**
Plotting in a raster display

• Assume a bilevel display: each pixel is black or white
• Different patterns of dots are created on the screen by setting each pixel to black or white (i.e. turning it on or off)
• A problem is that all the edges except perfectly horizontal and vertical ones show ‘jaggies’, i.e., staircasing effect.
Line drawing
Line drawing: problem

- Line drawing on a raster grid involves approximation
- The process is called *rasterization* or *scan-conversion*
Line drawing: objectives

- A line segment is defined by its end points \((x_1,y_1)\) and \((x_2,y_2)\) with integer coordinates.
- What is the best way to draw a line from the pixel \((x_1,y_1)\) to \((x_2,y_2)\)? Such a line should ideally have the following properties:
  - pass through endpoints
  - (appear) straight
  - (appear) smooth
  - independent of endpoint order
  - uniform brightness
  - slope-independent brightness
  - efficient

From Computer Graphics Course of Barry McCaul
Line drawing: brute force approach

**Line Characterisation**
- **Explicit**: \( y = mx + B \)
- **Implicit**: \( F(x,y) = ax + by + c = 0 \)
- **Constant slope**: \( \frac{\Delta y}{\Delta x} = m \)

The simplest strategy is:
1) Compute \( m \);
2) Increment \( x \) by 1 starting with the leftmost point;
3) Calculate \( y_i = mx_i + B \);
4) Intensify the pixel at \( (x_i, \text{Round}(y_i)) \), where \( \text{Round}(y_i) = \text{Floor}(0.5+y_i) \)

What is wrong with this approach?
The previous approach is inefficient because each iteration requires floating point operation *Multiply, Addition, and Floor*. We can eliminate the multiplication by noting that:

\[ y_{i+1} = mx_{i+1} + B = m(x_i + \Delta x) + B = y_i + m\Delta x \]

If \( \Delta x = 1 \), then \( y_{i+1} = y_i + m \). Thus, a unit of change in \( x \) changes \( y \) by \( m \), which is the slope of the line.

A simple incremental algorithm:
In order to implement the presented incremental algorithm, the slope $m$ has to be between 0 and 1 (then we are able to “step” along $x$ axis).

Other slopes can be handled by suitable reflections around the principal axes.

$m < 1$, can step along $x$.  

$m > 1$, cannot step along $x$.  
To handle this, apply a suitable reflection.
Midpoint Line Algorithm: intro

• What is wrong with the incremental algorithm?
  ▪ Required floating-point operations (Round)
  ▪ The time-consuming floating-point operations are unnecessary because both endpoints are integers

• Bresenham (1965) developed a classic algorithm that is attractive because it uses only integer arithmetics

• In Bresenham's algorithm, instead of incrementing y and then rounding it at each step, we just go to the right, or to the right and up using only integer quantities.
The problem becomes to decide on which side of the line the midpoint lies.

- Assume the line slope is \( m \), \( 0 \leq m \leq 1 \),
- Previously selected pixel: \( P(x_p, y_p) \)
- Choose between the pixel one increment to the right (the east pixel, \( E \)) or the pixel one increment to the right and one increment up (the northeast pixel, \( NE \)).
- \( Q \): the intersection point of the line being scan-converted with the grid line \( x = x_p + 1 \).
- \( M \): the midpoint between \( E \) and \( NE \).
- If \( M \) lies above the line, pixel \( E \) is closer to the line; if \( M \) is below the line, pixel \( NE \) is closer to the line.
Midpoint Line Algorithm: point positioning

Explicit line form \( y = \frac{dy}{dx} x + B \) gives:

\[ F(x, y) = x \cdot dy - y \cdot dx + Bdx = 0 \]

Comparing with the implicit form \( F(x, y) = ax + by + c = 0 \) yields

\[ a = dy, \; b = -dx, \; \text{and} \; c = Bdx \]

It can be shown that

\[ F(x, y) = \begin{cases} + & \text{if} \; (x,y) \; \text{is below the line} \\ 0 & \text{if} \; (x,y) \; \text{is on the line} \\ - & \text{if} \; (x,y) \; \text{is above the line} \end{cases} \]
Midpoint Line Algorithm: decision variable (1)

Since we are trying to decide whether $M$ lies below or above the line, we need only to compute

$$F(M) = F(x_p + 1, y_p + \frac{1}{2})$$

and to test its sign.

Define a decision variable $d$:

$$d = F(x_p + 1, y_p + \frac{1}{2}) = a(x_p + 1) + b(y_p + \frac{1}{2}) + c$$

$$= \begin{cases} 
+ & \text{Choose pixel NE} \\
0 & \text{Choose pixel E} \\
- & \text{Choose pixel E}
\end{cases}$$
Midpoint Line Algorithm: decision variable (2)

If $E$ is chosen, $M$ is incremented by one step in the $x$ direction. Then

$$d_{new} = F(x_p + 2, y_p + \frac{1}{2}) = a(x_p + 2) + b(y_p + \frac{1}{2}) + c$$

$$= a(x_p + 1) + a + b(y_p + \frac{1}{2}) + c = F(x_p + 1, y_p + \frac{1}{2}) + a$$

$$= d_{old} + a = d_{old} + dy$$

$$(d_{new} - d_{old})|_E = \Delta_E = a = dy$$
Midpoint Line Algorithm: decision variable (3)

On the other hand, if \( NE \) is chosen

\[
d_{new} = F(x_p + 2, y_p + \frac{3}{2}) = a(x_p + 2) + b(y_p + \frac{3}{2}) + c
\]

\[
= a(x_p + 1) + a + b(y_p + \frac{1}{2}) + b + c = F(x_p + 1, y_p + \frac{1}{2}) + a + b
\]

\[
= d_{old} + a + b = d_{old} + dy - dx
\]

\[
(d_{new} - d_{old})|_{NE} = \Delta_{NE} = a + b = dy - dx
\]

Initial condition:

\[
F(x_0 + 1, y_0 + \frac{1}{2}) = a(x_0 + 1) + b(y_0 + \frac{1}{2}) + c = F(x_0, y_0) + a + b / 2
\]

\[
d_{start} = a + b / 2 = dy - dx / 2
\]
Midpoint Line Algorithm: summary (1)

• Start from the first endpoint, and the first decision variable is given by $a+b/2$. Using $d_{\text{start}}$, choose the second pixel, and so on.

• At each step, choose between two pixels based on the sign of the decision variable $d$ calculated in the previous iteration.

• Update the decision variable $d$ by adding either $\Delta_E$ or $\Delta_{NE}$ to the old value, depending on the choice of the pixel.

***

• **Implementation note:** To eliminate the fraction in $d_{\text{start}}$, the original $F$ is multiplied by 2; $F(x,y) = 2(ax+by+c)$.
  
  ▪ This multiplies each constant in the decision variable (and the increments $\Delta_E$ and $\Delta_{NE}$) by 2.
  
  ▪ Does not affect the sign of the decision variable, which is all that matters for the midpoint test.
Midpoint Line Algorithm: summary (2)

Initialisation: \[ d_{\text{start}} = 2 \times a + b = 2 \times dy - dx \]
where \( dy = y_1 - y_0 \) and \( dx = x_1 - x_0 \).

Incremental update: 1) if \( E \) was chosen, \( \Delta_E = 2 \times dy \)
\[ d_{\text{new}} = d_{\text{old}} + \Delta_E \]
2) if \( NE \) was chosen, \( \Delta_{NE} = 2 \times (dy - dx) \)
\[ d_{\text{new}} = d_{\text{old}} + \Delta_{NE} \]

• Advantage: The arithmetic needed to evaluate \( d_{\text{new}} \) for any step is a simple integer addition.
  ▪ No time-consuming multiplication involved
  ▪ The incremental update is quite simple, therefore
  ▪ An efficient algorithm

• Note: works for those line with slope \((0, 1)\). What about bigger slopes?
Line drawing: Slope dependent intensity

• Problem: weaker intensity of diagonal lines

• Consider two scan-converted lines in the figure. The diagonal line, B, has a slope of 1 and hence is $\sqrt{2}$ times longer than the horizontal line A. Yet the same number of pixels is drawn to represent each line.

• If the intensity of each pixel is $I$, then the intensity per unit length of line A is $I$, whereas for line B it is only $I/\sqrt{2}$.

![Diagram showing Line A and Line B with different intensities.]
Scan converting circles

- Eight way symmetry: If the point \((x,y)\) is on the circle, then we can trivially compute seven other points on this circle.
The midpoint circle scan conversion

\[ P(x_p, y_p) \]

Previous pixel  Choices for current pixel  Choices for the next step if E or SE is chosen

M  ME  M_{SE}  E  SE
The midpoint circle scan conversion

For a circle of radius R: \( F(x, y) = x^2 + y^2 - R^2 \)

As for lines, the next pixel is chosen on the basis of the decision variable \( d \), which is the value of the function at the midpoint

\[
d_{old} = F(x_p + 1, y_p - \frac{1}{2}) = (x_p+1)^2 + (y_p - \frac{1}{2})^2 - R^2
\]

If \( d_{old} < 0 \), \( E \) is chosen and we have

\[
d_{new} = F(x_p + 2, y_p - \frac{1}{2}) = (x_p+2)^2 + (y_p - \frac{1}{2})^2 - R^2
\]
resulting in \( \Delta_E = 2x_p + 3 \).

If \( d_{old} \geq 0 \), \( SE \) is chosen and the new decision variable is

\[
d_{new} = F(x_p + 2, y_p - \frac{3}{2}) = (x_p+2)^2 + (y_p - \frac{3}{2})^2 - R^2
\]
and hence \( \Delta_{SE} = 2x_p + -2y_p \cdot +5 \).
Scan converting ellipses

The same reasoning can be applied for scan converting an ellipse, given by

\[ F(x, y) = b^2 x^2 + a^2 y^2 - a^2 b^2 = 0 \]

- Division into four quadrants
- **Two regions** in the first quadrant

Tangent slope = -1
Polygon filling
Polygons

• Vertex = point in space (2D or 3D)

• Polygon = ordered list of vertices
  ▪ Each vertex is connected with the next one in the list
  ▪ The last vertex is connected with the first one
  ▪ A polygon can contain holes
  ▪ A polygon can also contain self-intersections

• Simple polygon – no holes or self-intersections
  ▪ Such simple polygons are most interesting in Computer Graphics
Examples of polygons

Convex Polygons

Simple Concave

Complex (self-intersecting) polygons
Polygons in computer graphics

• The main geometric object used for interactive graphics

• Efficient algorithms exist for polygon scanline rendering

• Efficient algorithms exist also for lighting and shading polygons as well as for texture mapping

• By using a sufficient number of polygons, we can get close to any reasonable shape
Drawing modes for polygons

• Draw lines along polygon edges
  ▪ Using e.g. midpoint line (Bresenham) algorithm
  ▪ This is called \textit{wireframe} mode

• Draw filled polygons
  ▪ \textbf{Shaded} polygons (shading modes)
    • \textbf{Flat shaded} – constant color for whole polygon
    • \textbf{Gouraud shaded} – interpolate vertex colors across the polygon
Polygon interior

• We need to fill in (color) only pixels inside a polygon

• What is “inside” of a polygon?

• Parity (odd-even) rule commonly used

• Imagine a ray from the point to infinity

• Count number of intersections $N$ with polygon edges
  ▪ If $N$ is odd, point is inside
  ▪ If $N$ is even, point is outside
Complex polygons: Star polygons

- Regular pentagram
- Regular pentagram with multiple interiors
- Concave decagon (simple polygon)

What is the inside area? - Multiple interpretations possible

Source: Tom Ruen, Wikipedia
Filling polygons: some useful definitions

**Span**: collection of adjacent pixels on a single scan line which lie inside the primitive.

**Coherence**: being *logically consistent or connected*.

**Spatial coherence**: attributes of the neighboring pixels tend to be similar. This allows us to optimise our algorithms.

**Edge coherence**: most of the edges that intersect scan line $i$ also intersect scan line $i+1$. 
Filling polygons: generalities

• Avoid drawing pixels more than once
  ▪ wastes computation
  ▪ may cause unwanted color change
  ▪ ugly artifacts can occur if two polygons share the same edge (e.g., dependence on drawing order)

• Rules for shared edges
  ▪ A shared vertical edge belongs to the rightmost of the two sharing shapes. (Therefore, the right edge of rectangles will not be filled)
  ▪ A shared non-vertical edge belongs to the upper shape. (Therefore, the top edge of rectangles will not be filled)
Polygon filling: scan line approach

• Span-filling is an important step in the whole polygon-filling algorithm, and is implemented by a three-step process:
  ▪ Find the intersections of the scan line with all edges of the polygon.
  ▪ Sort the intersections by increasing x coordinates.
  ▪ Fill in all pixels between pairs of intersections that lie interior to the polygon.

• Important questions:
  ▪ How do we find and sort the intersections efficiently?
  ▪ How do we judge whether a pixel lying inside or outside the polygon?
Polygon filling: edge coherence

• In order to calculate intersections between scan lines and edges, we must avoid the brute-force technique of testing each polygon edge for intersection with each new scan line – it is inefficient and slow.

• **Clever solution** if an edge intersects with a scan line, and the slope of the edge is \( m \), then successive scan line intersections can be found from:

\[
x_{i+1} = x_i + \frac{1}{m}
\]

where \( i \) is the scan line count.

• Given that \( \frac{1}{m} = \frac{(x_1 - x_0)}{(y_1 - y_0)} \) the floating-point arithmetic can be avoided by storing the numerator, comparing to the denominator, and incrementing \( x \) when it overflows.
Rasterisation example...
...Rasterisation example...

Scan line

(0,0)
Rasterisation example...

\[
\Delta x/\Delta y = \frac{14 - 7}{1 - 10} = -0.777
\]

\[
\Delta x/\Delta y = \frac{3 - 1}{6 - 0} = 0.333
\]
Rasterisation example...

\[ \Delta x/\Delta y = (14-7)/(1-10) = -0.777 \]

\[ \Delta x/\Delta y = (3-1)/(6-0) = 0.333 \]
Rasterisation example...

- For $P_1P_2$: $\Delta x/\Delta y = (x_2-x_1)/(y_2-y_1)$
  \[
  \mu_{\text{start}} = (3-1)/(6-0) = 0.333
  \]
- For $P_3P_4$: $\Delta x/\Delta y = (X_4-X_3)/(Y_4-Y_3)$
  \[
  \mu_{\text{stop}} = (14-7)/(1-10) = -0.777
  \]
- For the first scan line:
  \[
  X_{\text{start}} = X_2 + \mu_{\text{start}}*(-0.5) = 2.833
  \]
  \[
  X_{\text{stop}} = X_3 + \mu_{\text{stop}}*(-4.5) = 10.500
  \]
- For the next scan lines:
  \[
  X_{\text{start}} := X_{\text{start}} - \mu_{\text{start}} = 2.500, 2.166, ...
  \]
  \[
  X_{\text{stop}} := X_{\text{stop}} - \mu_{\text{stop}} = 11.278, 12.055, ...
  \]
...Rasterisation example

- Summarised
  - For a “thin” filling, round x.5 up for Xstart and down for Xstop
  - Voor a “thick” filling, round x.5 down for Xstart and up for Xstop

<table>
<thead>
<tr>
<th>Scan line</th>
<th>Xstart</th>
<th>Xstop</th>
<th>Xstart</th>
<th>Xstop</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.833</td>
<td>10.500</td>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>2.500</td>
<td>11.278</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>2.167</td>
<td>12.056</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>1.833</td>
<td>12.833</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>1.500</td>
<td>13.611</td>
<td>1</td>
<td>14</td>
</tr>
</tbody>
</table>
Rasterisation example (7)
Alternative filling algorithms
Different ways of filling a polyline

- Parity rule
- Nonzero winding rule
- Nonexterior rule
Filled by parity rule
Filling: winding rules

• Count the number of windings

• Each region gets a “winding index” \( i \)

• Possible filling rules
  - Fill with one color if \( i > 0 \) (non-zero winding fill)
  - Fill with one color if \( \text{mod}(i,2) = 0 \) (parity fill)
  - Fill with a separate color for each value of \( i \)
Count the number of windings
Count the number of windings
Nonexterior rule: any point that can be connected to the seed point by a path that does not intersect the polyline is said to be outside the polyline.
A combination of a winding rule and nonexterior rule
Clipping

- Clip a line segment at the edges of a rectangular window

- Needs to be fast and robust
  - Robust means: works for special cases too, and preferably in the same way as for the normal cases

- The method of Cohen-Sutherland (1974)
  - Very simple
  - Suitable for hardware implementation
  - Can be directly extended to 3D
Cohen-Sutherland line clipping algorithm

Idea: Encode the position of the end points with respect to left, right, bottom and top window edges and cut accordingly.
Each endpoint is assigned a 4-bit code $k = (k_1, k_2, k_3, k_4)$, $k_i \in \{0,1\}$ and

- $k_1 = 1$ if $X < XL$ (too much to the left);
- $k_2 = 1$ if $X > XR$ (too much to the right)
- $k_3 = 1$ if $Y < YB$ (too low)
- $k_4 = 1$ if $Y > YT$ (too high)

Note: $k_1$ and $k_2$ cannot be simultaneously equal to 1, same holds for $k_3$ and $k_4$ $\rightarrow$ Hence 9 possible code words (not $2^4$)
Cohen-Sutherland line clipping algorithm

\[ P(X,Y) \]

\[ k \]

- \( k_3 = 1 \) and \( k_4 = 0 \)
  - \( k = 1010 \)
  - \( k = 1000 \)

- \( k_3 = k_4 = 0 \)
  - \( k = 1001 \)
  - \( k = 0001 \)

- \( k_3 = 0 \) and \( k_4 = 1 \)
  - \( k = 0101 \)
  - \( k = 0100 \)

- \( k_3 = k_4 = 1 \)
  - \( k = 0010 \)
  - \( k = 0110 \)
Cohen-Sutherland line clipping algorithm

• Cohen-Sutherland tries to solve first simple (trivial) cases

• Assign 4-bit code words to end points: \( P_1 \rightarrow k_1, \ P_2 \rightarrow k_2 \)

Step 1: if \( k_1 = k_2 = 0 \), \( P_1P_2 \) is fully visible; otherwise go to Step 2

Step 2: Find bit per bit logic AND \( k = k_1 \land k_2 \). If \( k_1 \land k_2 \neq 0 \), \( P_1P_2 \) is fully and “trivially” non visible; otherwise go to Step 3

Step 3: find intersections with lines extending from window edges; Go to Step 1.
Cohen-Sutherland line clipping algorithm

Trivially visible \((k_1 = k_2 = 0)\) and trivially invisible \((k_1 \land k_2 \neq 0)\) examples
\( k_1 \land k_2 = 0 \), and not trivially visible can imply partially visible (subject to shortening) or invisible segment (additional testing needed)
Cohen-Sutherland line clipping algorithm

• In Step 3, $k_1 \land k_2 = 0$, but $k_1 \neq 0$ or $k_2 \neq 0$ (or both)
  ▪ Otherwise the segment would be fully visible (Step 1)

• Suppose that $k_1 \neq 0$, which means that $P_1$ is outside
  ▪ If not, switch $P_1$ and $P_2$: $X_1 \leftrightarrow X_2$; $Y_1 \leftrightarrow Y_2$; $k_1 \leftrightarrow k_2$

• We need to “bring” $P_1$ on the edge of the window
  ▪ Actually, replace $P_1$ by the intersection point of the segment $P_1P_2$ and the corresponding window edge.
  ▪ If $k_1 = 1$, find intersection with the left edge $XL$
  ▪ If $k_2 = 1$, find intersection with the the right edge $XR$
  ▪ If $k_3 = 1$, find intersection with the bottom edge $YB$
  ▪ If $k_4 = 1$ find intersection with the the top edge $YT$

• Go to Step 1 and repeat until $P_1'$ en $P_2'$ are found
Cohen-Sutherland line clipping algorithm

Search for intersections

• Intersection with XL
  ▪ \( Y_1 := Y_1 + (XL - X_1) \times (Y_2 - Y_1) / (X_2 - X_1) \)
  ▪ \( X_1 := XL \)

• Intersection with XR: same as above with XR instead of XL

• Intersection with YB
  ▪ \( X_1 := X_1 + (YB - Y_1) \times (X_2 - X_1) / (Y_2 - Y_1) \)
  ▪ \( Y_1 := YB \)

• Intersection with YT: same as above with YT instead of YB

• Denominators cannot be equal to 0 (no division by 0)
  ▪ In the first case (bringing on XL), this would mean that the segment is vertical, and too much to the left → already eliminated
Cohen-Sutherland line clipping algorithm
Cohen-Sutherland line clipping algorithm
Polygon clipping

• Polygon clipping = scissoring according to clip window

• Must deal with many different cases

• Clipping a single polygon can result in multiple polygons
Polygon clipping

• Find the parts of polygons inside the clip window
Polygon clipping

- Find the parts of polygons inside the clip window
Sutherland-Hodgman Clipping Algorithm

- Clipping to one window boundary at a time
Sutherland-Hodgman Clipping Algorithm

- Clipping to one window boundary at a time
Sutherland-Hodgman Clipping Algorithm

- Clipping to one window boundary at a time
Sutherland-Hodgman Clipping Algorithm

- Clipping to one window boundary at a time
Polygon clipping

- Clipping to one window boundary at a time
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary

Outside window

Inside window

P_0
P_1
P_2
P_3
P_4
P_5
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary
Clipping to a boundary

- Do inside test for each point in sequence
- Insert new points when cross window boundary
- Remove points outside window boundary

![Diagram showing clipping to a boundary with points P'', P', P_1, and P_2. The area outside the window is shaded.](image-url)
Inside test

Outside the clip rectangle

Inside the clip rectangle

\[
N_i \cdot (P_1 - P_E) > 0 \text{ (outside)}
\]

\[
N_i \cdot (P_2 - P_E) = 0 \text{ (on edge)}
\]

\[
N_i \cdot (P_3 - P_E) < 0 \text{ (inside)}
\]

- Define edge’s outward normal \( N_i \)
- For a point \( P_i \) test the sign of the dot product \( N_i \cdot (P_1 - P_E) \)