COMP30019 Graphics and Interaction
Scan Converting Polygons and Lines

Adrian Pearce

Department of Computer Science and Software Engineering
University of Melbourne

The University of Melbourne
Lecture outline

Introduction
Scan conversion
Scan-line algorithm
Edge coherence
Bresenham’s algorithm
Edge coherence (revisited)
Z buffering
High Performance Visible Surface Detection
Scan conversion

*How are polygons and lines drawn onto a digital image?*

Aim: understanding polygonal filling and line drawing algorithms.

Reading:
- Foley Sections 3.2 Scan converting lines and Section 3.5 Filling polygons.

Additional reading:
- 3.3 Scan converting circles.
Inside or outside a polygon?

Filling polygons requires a way of determining whether a point is inside or outside a polygon.

See if you can think up a technique for determining whether you are inside or outside a two-dimensional polygon from a completely arbitrary point (imagine you have no way of knowing whether you are inside or out initially)?

Hint: imagine you are standing at a point \( q \) and watching another point \( p \) that can reside at any vertex of polygon \( P \).
Filling polygons using winding numbers

Filling polygons requires a way of determining whether a point is inside or outside a polygon.

One technique is known as *winding numbers*.

Imagine you are standing at a point $q$ while watching another point $p$ traverse a polygon $P$ counterclockwise (the point moves from vertex to vertex around the perimeter of the polygon).

If $q$ is inside, you would turn a full circle, $2\pi$ radians, if $q$ is outside the sum of your total angular turn will be zero 0.
The \textit{winding number} of \( q \) with respect to \( P \) is the number of revolutions point \( p \) makes around polygon \( P \): the total signed angular turn divided by \( 2\pi \).

For polygon \( P \), exterior points \( q \) have winding number 0: a total angular turn of 0.
Angle $\theta$ can be determined from dot or cross project

\[ v_i \cdot v_{i+1} = |v_i||v_{i+1}| \cos \theta_i \]

and

\[ v_i \times v_{i+1} = |v_i||v_{i+1}| \sin \theta_i \]

Angle $\theta_i$ is the angle subtended by $e$ from $q$
The odd-even test

The odd-even test—follow an arbitrary ray or along scan line and count the number of times a boundary is crossed, either an odd or even number of times.
Describe each of the two coloured lines and which direction they are going in and what happens to their parity.

notice for the top most line starting outside and skimming the top of the polygon, it stays outside because the single pixel is a special case of a horizontal line (more in next slides).
Selection of *inside* model

Let’s now compare the winding-number model to the odd-even test model according to our desidera for issues in numerical methods (numerical precision and computational complexity).

- Winding numbers work in case when $q$ lies on the perimeter of $P$, but the odd-even case needs to treat these as special case (e.g. when horizontal scan line intersects a horizontal polygonal edge).

- However, winding numbers require floating point numbers that is the winding angle will not always be 0 or $2\pi$ because of round-off error.
In terms of computational complexity, both algorithms are of order $O(n)$ in the worst case, where $n$ is the number of polygon vertices for winding numbers of pixels for the odd-even test.

- However, the winding-number algorithm depends critically on floating-point computations and trigonometric computations, which means it is significantly slower on standard hardware (up to 20 times slower according to research by Haines in 1992).

- For filling polygons in images, the odd-even test is superior as it only requires a parity bit and can be easily implemented by stepping along the scan lines of image rows and columns.
Scan-line filling

The scan-line method is an efficient way of filling-in polygons, based on the odd-even test by crossing segments.

Segments are sorted to allow for an orderly progression that determines whether inside or outside.
Figure 1.14 (Rowe), using scan line to determine the interior points of a polygon

Question: does scan-line filling work for non-simple and multiple-boundary polygons?
The first intersection occurs when the scan line crosses edge $AB$, so the parity becomes odd at that point.

The second intersection occurs with edge $AF$, so the parity becomes even.

We can therefore say that points along the scan line between edges $AB$ and $AF$ are interior, points between edges $AF$ and $EF$ are exterior,

and points between $EF$ and $DE$ are interior.

Scan-line filing does work for both non-simple and multiple boundary polygons (provided you start scanning from outside).
Basic scan-line algorithm

1. Find intersections of scan lines with edges of polygon(s)
2. Sort the intersections by increasing $x$ coordinate
3. fill in-between pixel extrema using odd-even parity rule
Pixel extrema

(a) Span extrema
(b) Other pixels in the span
Special cases

1. Given an intersection with an arbitrary, fractional $x$ value, how do we determine which pixel on either side of the intersection is interior?

2. How do we deal with intersections at integer pixel coordinates?

3. How do we deal with shared vertices?

4. How do we deal with horizontal edges?
Special case solutions

1. If inside and approaching fraction intersection to the right round $x$ coordinate down; else if outside round up.

2. If leftmost pixel (has integer coordinate) define as interior; else if right most coordinate (has integer coordinate) define to be exterior.

3. Count the $y_{min}$ vertex of an edge in parity but not the $y_{max}$ vertex (therefore vertex $y_{max}$ is only drawn if it is the $y_{min}$ vertex for the adjacent edge).

For example vertex A in 3.14, is counted one in the parity calculation because it is the $y_{min}$ vertex for edge FA but the $y_{max}$ vertex for edge AB (Thus, both edges and spans are treated as intervals that are closed at their minimum value and open at their maximum value).
Horizontal edges
The bucket-sorted edge table right for the polygon on the left (Foley Figures 3.14 and 3.18).
Active edge table for scan line 9 (above) and 10 (below) (Foley Figures 3.14 and 3.19).
Need to fill discreet pixel locations, but don’t want to have to check all edges from all polygons for large scenes - would take too much computationally!

Therefore sort edge locations based on x and y coordinates and use a hash-table like approach of retrieving them, relative to what location you are presently scanning (or filling).

Foley’s Figure 3.18 is shown along side figure 3.14 to indicate the associated bucket-sorted edge table generated (ahead of actual filling operation) for this polygon.
Scan-line polygon filling algorithm

1. Create a data structure for edges, containing the minimum and maximum $y$ values ($y_{min}$ and $y_{max}$, respectively) for each edge, and the $x$ value corresponding to $y_{min}$, called $x_{min}$.

2. Create an *edge table* in which the edges are listed, sorted in ascending order by their $y_{min}$ value. Omit any horizontal edges from the table.

3. Set scan line value $y_{scan}$ to the minimum $y_{min}$ value for all edges (that is, to the lowest $y$ value reached by the polygon).

4. Create an *active edge table*, containing all the edges currently intersected by the scan line.

5. Add to the active edge table (and delete from the master edge table) any edges whose $y_{min}$ value is equal to the current $y_{scan}$.
Scan-line polygon filling algorithm

6 Sort the active edge table in ascending order on \( x \).

7 Traverse active edge table, taking pairs of edges and drawing all pixels between \( x \) value for the first edge in a pair and the \( x \) value in the second edge in the same pair.

8 Move to the next scan line by incrementing \( y_{\text{scan}} \) by 1.

9 Delete any edges from the active edge table if their \( y_{\text{max}} \) value is equal to \( y_{\text{scan}} \).

10 Determine where scan line intersects edge (see edge coherence slide in next lecture).

11 While either master edge table or the active edge table has edges, goto step 5.
Define a data structure for representing an edge. It must be remembered that $x_{\min}$ is not necessarily the minimum $x$ value on the edge—it is the $x$ value that corresponds to the point with minimum $y$ value.

The *edge table* lists edges, as represented by the data structures defined in step 1. In practice, an array or linked list is used for the edge table. (Omitting horizontal edges, like in previous diagram, treat as a special outside case).

This is easily done by taking the $y_{\min}$ value from the first entry in the edge table, since the table was sorted on $y_{\min}$. We begin scanning the polygon at this point (only start scanning at top of polygon).

Like the original edge table, this second active edge table is usually implemented as an array or linked list (only need to do this once).
Edges are therefore distributed between active edge table and master edge table (loops from step 11, so do this at beginning of each scan line).

Sorting the edges in the active edge table means they are listed in the order that they are intersected by the scan line (as it travels left to right).

Traversing active edge table—For example, with the scan line positioned as shown in Fig 1.14, pixels along this line between edges $AB$ and $AF$ would be drawn from the first pair of edges in the active edge table, then pixels between $EF$ and $ED$. (Note that the active edge table will always contain an even number of edges).

Move to next scan line.
Delete edges from active edge table—Note that we do not draw the pixel corresponding to the $y_{\text{max}}$ value before deleting the edge—this is consistent with the rule ed in determining the parity of a pixel.

Determine edge coherence

Repeat algorithm by going back to step 5 and repeating, while either master edge table or the active edge table has edges.
Edge coherence

- Many edges that intersect scan line \( i \) also intersect scan line \( i + 1 \)
- Can incrementally calculate intersection \( x_{i+1} = x_i + \frac{a}{m} \), where \( m \) is the slope of the edge
- Can avoid fractional arithmetic by computing an integer decision variable and checking only its sign to decide when to increment \( x \) (as we will see later in lecture)
Scan converting lines

Involves *turning on* the correct pixels, either in display or in an image. Assume line segment is described by its start point \( \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \) and end point \( \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \).
First attempt

Choose function $y = mx + b$, and use the following basic iterative algorithm that steps through the range of $x$ values $[x_0..x_1]$:

- For each $x$ compute corresponding (real) $y$ value from the line equation.
- Round that $y$ value to the nearest pixel (grid) position and switch on, or set pixel at that $\begin{bmatrix} x \\ y \end{bmatrix}$ position.
Example: Drawing $y = 7x + 2$ (Rowe fig 1.4).

First, the problem that floating point arithmetic is computationally complex therefore slow to compute, particularly since this needs to be done very frequently, and

Second, the problem of discretisation in determining which pixel to turn on, one or both?
The problem: a line with slope $|m| > 1$ will have gaps in it.

Solution:

$|m| > 1$ draw $x$ as function of $y$

$|m| \leq 1$ draw $y$ as function of $x$

(as above)
Refinement

Reduce line generation to one very special case, like $0 \leq m \leq 1$
Handle all other cases as symmetries of the special case, by
- parameterised subroutines
- explicit transformations
- duplicating code

Leads to kind of line generation algorithm where step is always 0 or 1.
Further refinement: Bresenham’s algorithm and variants—only integer arithmetic, only addition/ subtraction.
Bresenham’s algorithm

Bresenham’s insight was to step horizontally, accumulating error until error gets big enough to force a step upwards.

- For integer coordinates all quantities are ratios with $2$ or $\Delta x = x_1 - x_0$ in the denominator, multiplying everything by $2\Delta x$, makes everything integer.

Essential insight:

- For integer coordinates, all quantities are ratios with $2$ or $\Delta x = x_1 - x_0$ in the denominator.
- Multiplying everything by $2\Delta x$, makes everything integer.

Step horizontally, accumulating error, until error gets big enough to force a step upwards.
At each step, choose between two pixels based on the sign of the decision variable calculated in the previous iteration:

- then it updates the decision variable by adding the error to the old value, depending on the choice of pixel.
The recursive problem

Assuming starting point \((x_0, y_0)\) is provided, the recursive problem is given \((x_k, y_k)\) to find \(y\) value for subscript position \(x_k + 1\),

\[
y = m(x_k + 1) + b
\]

\(y\) is either \(y_k\) or \(y_k + 1\), assuming that algorithm is drawing line from left to right as in first case (positive slope \(< 1\)).

- Initially, \(y = mx_k + b\), next row we get
- \(y = m(x_k + 1) + b\) and
- either \(y = y + k\) or \(y = y_k + 1\).
The sign of the difference $d_1 - d_2$ determines which pixel to choose: if difference is negative then $y_k$ is closer to actual line $(y)$, if it is positive $y_k + 1$ is closer.

How are non-integer calculations eliminated?
Bresenham requires that the endpoints of line segments \((x_0, y_0)\) and \((x_n, y_n)\) are provided, therefore the slope \(m\) can be expressed as a ratio of integers, \(m = \frac{\Delta y}{\Delta x}\), substitution gives

\[
\begin{align*}
   d_1 - d_2 &= 2m(x_k + 1) + 2b - 2y_k - 1 \\
   \Delta x(d_1 - d_2) &= 2\Delta yx_k - 2\Delta xy_k + [2\Delta y + \Delta x(2b - 1)]
\end{align*}
\]

As \(\Delta x\) is always 1 and \([2\Delta y + \Delta x(2b - 1)]\) does not depend on pixel index \(k\), we can write

\[C = 2\Delta y + \Delta x(2b - 1)\]

The pixel choice parameter \(p_k\) can now be defined as

\[p_k = \Delta x(d_1 - d_2) = 2\Delta yx_k - 2\Delta xy_k + C\]

If \(p_k\) is positive, we choose \(y_k + 1\); if negative, \(y_k\). Now how do we get rid of \(C\)?
The answer is to use recursion for calculating a series of $p_k$ values, the pixel choice at the next location is determined by $p_{k+1}$, and $C$ can be eliminated by subtraction, as follows

$$p_k = \Delta x(d_1 - d_2) = 2\Delta yx_k - 2\Delta xy_k + C$$
$$p_{k+1} = 2\Delta yx_{k+1} - 2\Delta xy_{k+1} + C$$
$$p_{k+1} - p_k = 2\Delta y(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k)$$

and since algorithm steps along $x$ axis, $x_{k+1} = x + k + 1$, can recursively define a relationship for choosing pixels as

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k)$$

Note that all quantities in this equation are integers.
For initialisation, a starting value for the first pixel, \( p_0 \), is required, using equation (see earlier)

\[ \Delta x(d_1 - d_2) = 2\Delta yx_k - 2\Delta xy_k + [2\Delta y + \Delta x(2b - 1)] \]

and using starting point of the line \((x_0, y_0)\) and substituting the following for \( b \)

\[ b = y_0 - mx_0 = y_0 - (\Delta y/\Delta x)x_0 \]

we obtain

\[ p_0 = 2\Delta y - \Delta x \]
Bresenham’s algorithm

1. Provide endpoints \((x_0, y_0)\) and \((x_n, y_n)\) of line.
2. Calculate constants \(2\Delta y\) and \(2\Delta y - 2\Delta x\), where \(\Delta y = y_n - y_0\) and \(\Delta x = x_n - x_0\).
3. Plot the origin \((x_0, y_0)\).
4. Calculate the pixel choice parameter \(p_k\), and
   - if \(p_k < 0\), plot \((x_k + 1, y_k)\) and
     \[ p_{k+1} = p_k + 2\Delta y \]
   - otherwise plot \((x_k + 1, y_k + 1)\) and
     \[ p_{k+1} = p_k + 2\Delta y - 2\Delta x \]
5. Repeat step 4, until \(x_n\) is reached.
Implementation of Bresenham’s algorithm

Implementation of Bresenham’s algorithm for drawing lines based on mathematical equation \( y = mx + b \) for either

(i) positive slope \(< 1\) (only this case covered here)

(ii) negative slope \(> -1\), or

(iii) positive slope \(\geq 1\).

For the discrete case, subscript \( k \) is added to pixel \( x_k \), the equation for a line then becomes

\[
y = mx_k + b
\]

however, in general \( y \) will not be an integer.
// Bresenham’s algorithm for line drawing

void drawBresenhamLine(Graphics g, Point start, Point end) {
    // Implements case for slope < 1
    if (m_slopeLT1) {
        x = m_x0 + 1; y = m_y0;
        while (x < m_x1) {
            if (m_param < 0) {
                m_param += m_twoDeltaY;
            } else {
                m_param += m_twoDYDX;
            }
            y += m_slopeSign;
        }
        drawPixel(g, x, y);
    } else {
        m_param += m_twoDYDX;
        y += m_slopeSign;
    }
    drawPixel(g, x, y);
}

void drawBresenhamLine(Graphics g, Point start, Point end) {
    // Implements case for slope < 1
    if (m_slopeLT1) {
        x = m_x0 + 1; y = m_y0;
        while (x < m_x1) {
            if (m_param < 0) {
                m_param += m_twoDeltaY;
            } else {
                m_param += m_twoDYDX;
            }
            y += m_slopeSign;
        }
        drawPixel(g, x, y);
    } else {
        m_param += m_twoDYDX;
        y += m_slopeSign;
    }
    drawPixel(g, x, y);
}


Edge coherence (revisited)

In principle, finding the intersection of the point between a scan line and a polygon edge just involves substituting $y = y_{scan}$ into equation for a line and solving for $s$. However, it suffers from same problem as drawing lines, in that it involves floating point arithmetic.

It turns out that there is a more efficient algorithm which allows us to calculate the $x$ value on a scan line directly from the $x$ value on the previous scan line (like Bresenham’s algorithm), based on the edge coherence property of a polygon.
The idea behind coherence is that most geometric figures have regions where properties are relatively consistent.

- In edge coherence, this means that non-horizontal edge in a polygon will be intersected by more than one scan line and therefore should be able to generate intersection points iteratively, rather than using the line equation at each step.
- Suggests a recursive algorithm closely related to Bresenham’s line drawing algorithm.
Edge coherence algorithm

1. When a scan line is added to the active edge table, initialise $x$ to $x_{\text{min}}$ and the increment to 0. Calculate and store $\Delta x$ and $\Delta y$.
2. When a scan line is increased by 1, add $\Delta x$ to the increment.
3. If the magnitude of the slope is greater than or equal to 1, compare the magnitude of the increment with $\Delta y$. If it is less than $\Delta y$, leave $x$ as it is. Otherwise, increase $x$ by 1, and subtract (add) $\Delta y$ to the increment if the increment is positive (negative).
4. If magnitude of slope is less than 1, determine number of pixels to increase $x$ by dividing magnitude of increment by $\Delta y$ (ignoring remainder). The remainder (positive if increment is positive, and negative if it is negative) becomes new value of the increment.
Z Buffer

As well as pixel frame buffer, keep a parallel Z buffer, which at each point records the depth of corresponding pixel.

The Z buffer is initialised to some representation of “infinite” depth.

The frame buffer initialised to “background” colour then pixel writes take pixel value, \(x\), \(y\), and \(Z\). Updates only take place at \((x, y)\) only if new depth is closer.
Z Buffer

Depth must be in camera coordinates, any order-preserving function of true depth will do (pseudo-depth is OK).

Z buffer must have sufficient bits to resolve depth finely enough and may use some bits of frame buffer, if enough bits are available, and display lookup tables can be manipulated.

Can use buffer with higher spatial resolution, and block-average down for antialiasing.
The z-buffer drawing (a) first and (b) second polygons

- Assume looking down the z axis.
- Depends on spatial resolution (must have sufficient bits to resolve depth finely enough).

Adrian Pearce
University of Melbourne
Interpolation of z values along edges for scan lines

\[ z_a = z_1 - \frac{(z_1 - z_2)}{y_1 - y_2} \]

\[ z_b = z_1 - \frac{(z_1 - z_3)}{y_1 - y_3} \]

\[ z_p = z_b - \frac{(z_b - z_a)}{x_b - x_a} \]
Pros and cons of the z buffer

+ simple to implement.
+ implementable in hardware (generally is).
+ works for all kinds of objects (not just polyhedra).
  - extra storage needed for Z buffer.
  - many accesses to Z buffer and frame buffer—bad for display through a communications link.
  - limited to space and depth resolution of buffers.
Properties of Z-buffers

- for each polygon or object in scene (piecewise Z buffering).
- (z-buffer) for each scan line (single-scan-line Z buffering)
- If shading computation is time-consuming, a rough front-to-back depth sort of objects to display closest one first improves efficiency (can use radix sort for this, at resolution of z-buffer).
- Don’t calculate shading for hidden surfaces — important if using Phong illumination model as we will see later.
Rendering pipeline for z-buffer (revisited)

(Foley Figure 14.41)
Rendering pipeline for z-buffer - the steps

See page 521 of Foley:

- **db traversal & modelling transformation**: transform all (relatively defined) polygons or polygonal meshes to their correct location.
- **Trivial accept/reject**: Entirely outside & back face culling
- **Lighting**: calculate intensity (for vertices only: need to do in 3D before perspective projection)
- **Viewing transformation**: perspective transformation (e.g. perspective foreshortening).
- **Clipping**: clip to viewport (involves creation of new vertices on border).
- **Map to viewport**: Change of coordinate systems (Divide by W: based on homogeneous coordinate system).
- **Rasterisation**: scan-line drawing polygons (including z-buffering & shading interpolation)
Using the Phong illumination model, lighting cannot be calculated early in the pipeline why? (Foley Figure 14.42),
Rendering pipeline for z-buffer and Phong shading

See pages 522-523 of Foley:

- **Viewing transformation:** Need to calculate lighting *after* viewing transformation, why? because Phong illumination is dependent on viewer orientation.
- **Clipping:** because Phong shading interpolates surface normals, rather than intensities, the vertices cannot be lit early in the pipeline
- **Rasterisation:** lighting equation must be calculated for every pixel.
High performance visible surface determination

Many methods and variations of visible surface determination are possible, which performs best depends on hardware available, and nature of graphics drawn, E.g.

- for a scene made of many small polygons, Z buffering might be best,
- for a scene made of a few large polygons, binary space partition techniques (BSPs) might be better.

The Z-buffer idea can also be used for a variety of tasks, such as

- piecewise Z buffering (for example polygon-by-polygon), or
- single-scan-line Z buffering (pixel-by-pixel).
Summary

- Polygons can be filled using a *scan-line* approach based on the *odd-even* to determine whether inside or outside.
- Efficient scan-line algorithms rely on bucket (hash) sorting edges and maintaining *active-edge* tables. This allows only edges of relevant polygons to be checked.
- Efficient line drawing algorithms typically utilise the edge coherence property to calculate the value of the pixel on the current scan line from the value on previous scan line.
- Bresenham’s insight was how to rely only on integer operations, thus avoiding floating point arithmetic.
Summary

- Edge-coherence properties can be utilised in polygon filling algorithms, to decide whether $x$ values in the active-edge table need incrementing (Can use same approach for drawing circles and ellipses)
- The Rendering pipeline, based on z-buffering, is the basis of Hidden surface removal OpenGL.
- Various techniques exist for high performance visible surface detection, including binary space partition (BSP) and piecewise (polygon-by-polygon) Z buffering and single-scane-line (pixel-by-pixel) buffering.