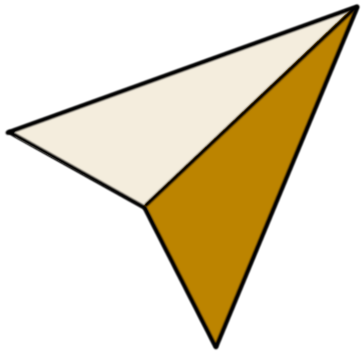


COMP4610/COMP6461

Week 2 - Basic 2D Graphics

<Print version>

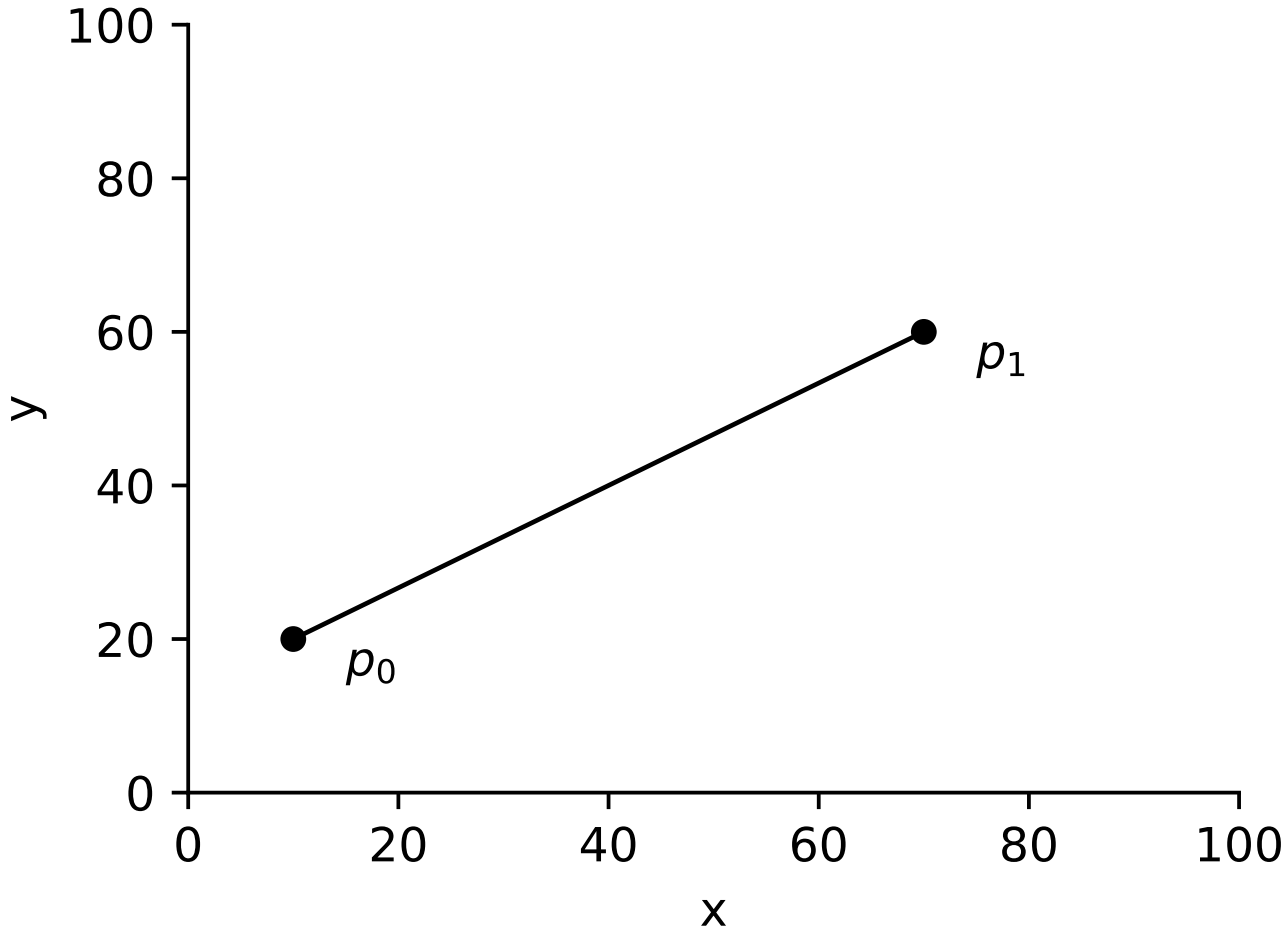
2D Graphics

2D Graphics

Line Drawing

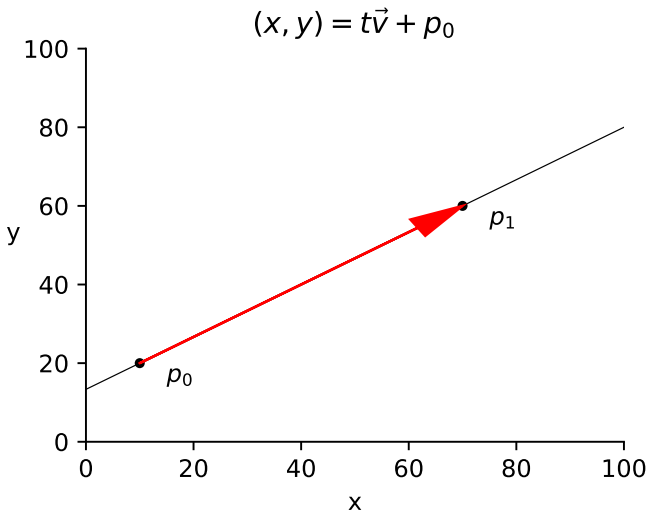
Line Drawing

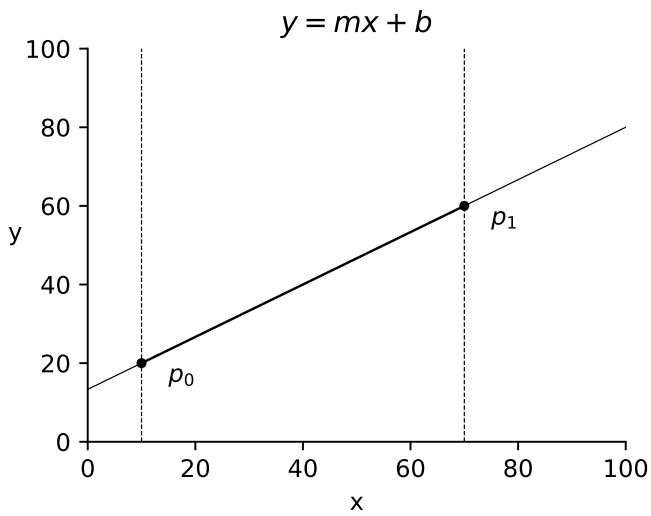
We often want to be able to draw a line segment between two points.



Line Drawing

How do you describe a line?





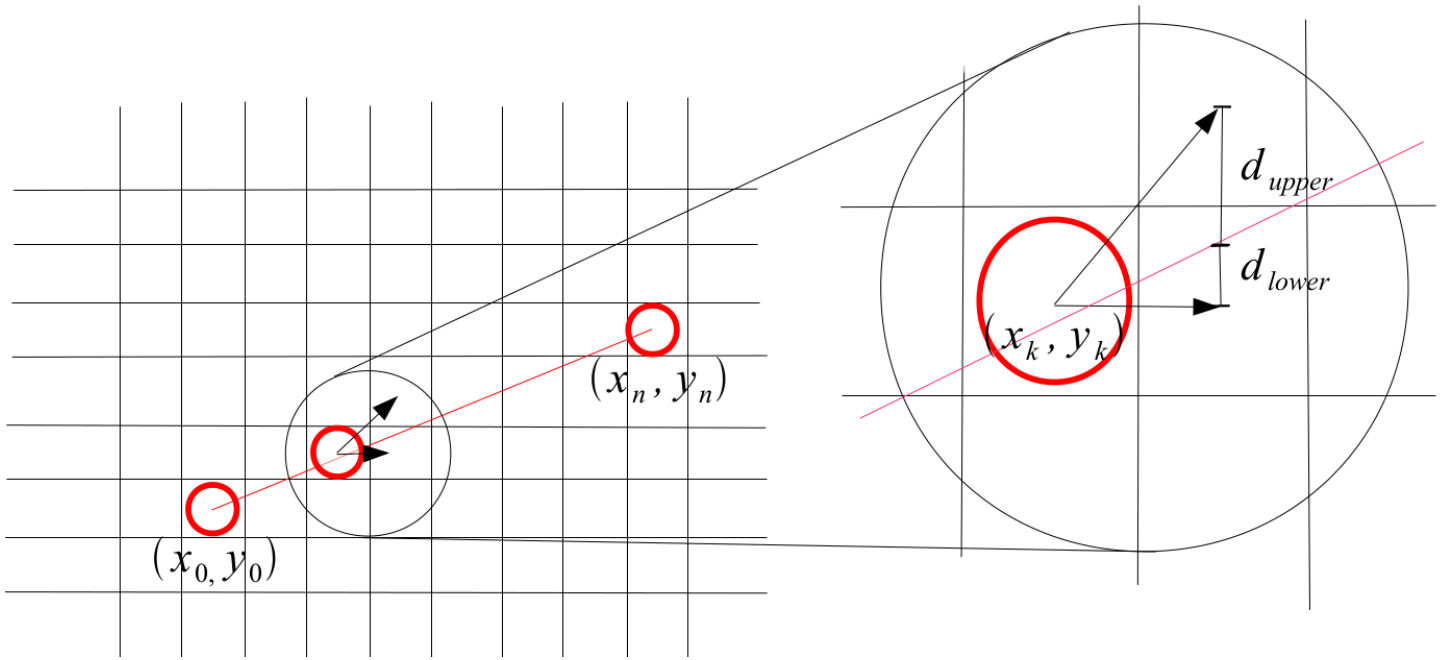
What might be the problem with the second method?

Digital Differential Analyzer DDA

```
1  private static void lineDDA(BufferedImage buff, int x1, int
       y1, int x2, int y2, int rgb) {
2      int dx = x2 - x1;  // Could be improved by multiplying
3      int dy = y2 - y1;  // values by step and then
4      float x = x1;      // using integer arithmetic.
5      float y = y1;      // Still needs 2 divisons per point.
6      int steps = Math.max(Math.abs(dx), Math.abs(dy));
7      float xinc = (float) dx / steps;
8      float yinc = (float) dy / steps;
9      buff.setRGB(Math.round(x), Math.round(y), rgb);
10     for (int i = 0; i < steps; i++) {
11         x += xinc;
12         y += yinc;
13         buff.setRGB(Math.round(x), Math.round(y), rgb);
14     }
15 }
```

Bresenham's Line Drawing Algorithm

Bresenham's Line Drawing Algorithm transforms the problem of line drawing into the decision to 'step up' or 'step right'.



How would we handle lines slopping down? Or to the left?

Bresenham's Line Drawing Algorithm

We want a way of deciding to go up or stay on the same line.

$$
\begin{aligned}
y &= m(x_k + 1) + b \\
d_{\text{lower}} &= y - y_k = m(x_k + 1) + b - y_k \\
d_{\text{upper}} &= y_k + 1 - y = y_k + 1 - m(x_k + 1) - b
\end{aligned}
$$

Let:

$$
\begin{aligned}
p_k &= \Delta x(d_{\text{lower}} - d_{\text{upper}}) \\
&= 2\Delta y\, x_k - 2\Delta x\, y_k + c \\
p_{k+1} &= 2\Delta y\, x_{k+1} - 2\Delta x\, y_{k+1} + c \\
p_{k+1} - p_k &= 2\Delta y(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k) \\
&= \begin{cases} 2\Delta y & \text{if } p_k < 0 \\ 2\Delta y - 2\Delta x & \text{otherwise} \end{cases}
\end{aligned}
$$

2D Graphics

Circles

Drawing Circles

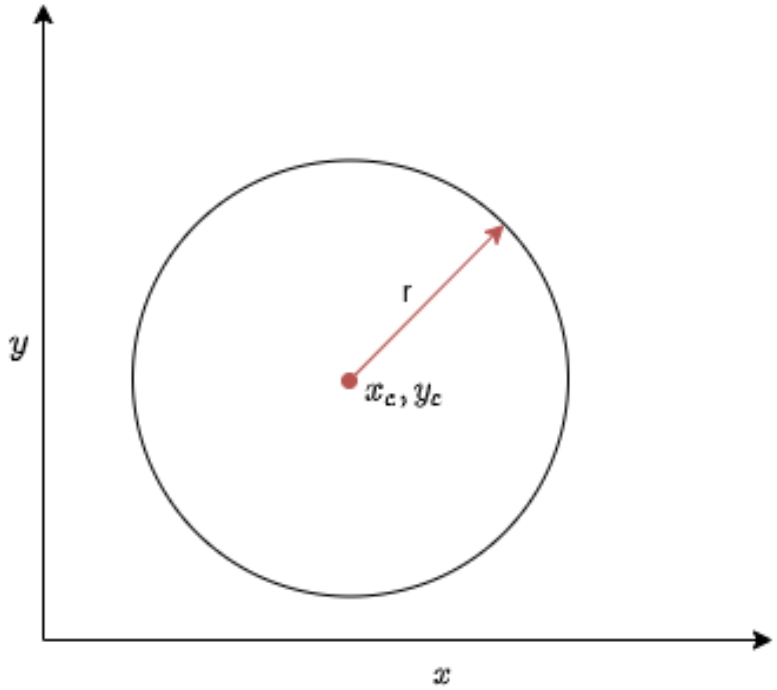
A circle can be defined by,

$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

or alternatively in parametric form,

$$x = x_c + r\cos\theta$$
$$y = y_c + r\sin\theta$$

where $\theta \in \mathbb{R}$ is an angle in *radians*.



2D Graphics

Java Graphics

Java Graphics

In some demos we will look at the following classes:

- Dimensions
- Graphics
- Color
- Font
- Image
- IconsIcon
- BufferedImage
- As well as some Graphics2D methods and associated classes.

2D Graphics

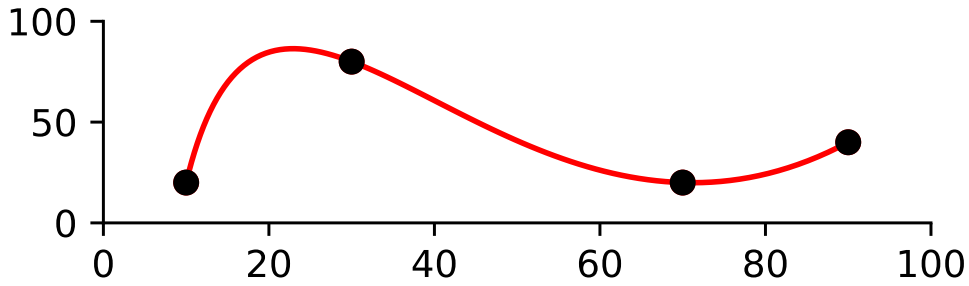
Splines

Splines

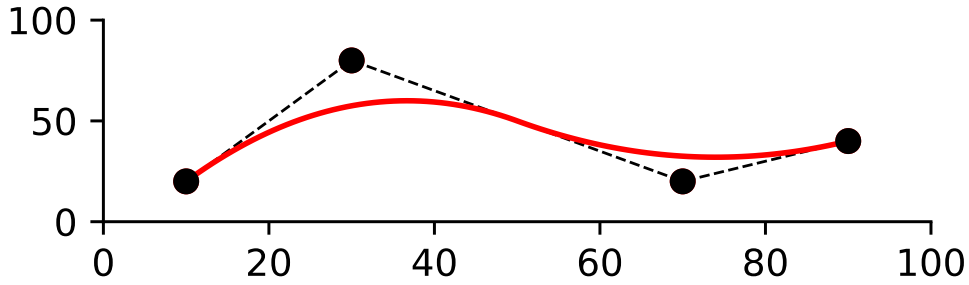
- Spline curves are *composite* curves formed with polynomial sections.
- Spline surfaces are two sets of orthogonal spline curves.
- Control points are used to describe the curve.

Control points

Interpolate: curve must pass through control points. (like a bendy ruler)



Approximate: control points guide the curve. (typically by setting the direction the curve is traveling in)



Parametric Approaches

Curves can be described using parametric functions.

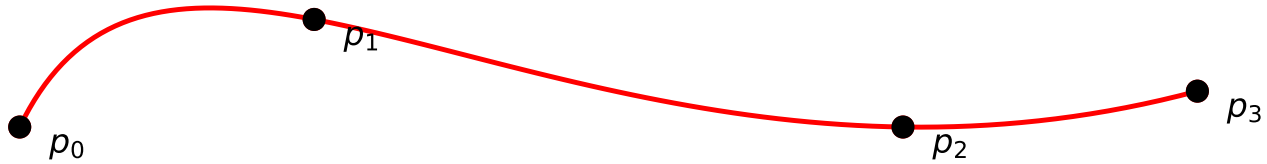
$$x = f_x(u)$$
$$y = f_y(u)$$
$$u_{\text{start}} \leq u \leq u_{\text{end}}$$

Curves can be characterised as:

- zero-order parametric continuous (location matches)
- first-order parametric continuous (slope matches)
- second-order parametric continuous (slope of slope matches)

Cubic Splines

Each piece of the spline is a cubic function.



Given $n+1$ control points we would have n different cubic functions describing the curve between the interpolated control points

$$f_{x,k}(u) = a_{x,k}u^3 + b_{x,k}u^2 + c_{x,k}u + d_{x,k}$$
$$f_{y,k}(u) = a_{y,k}u^3 + b_{y,k}u^2 + c_{y,k}u + d_{y,k}$$

for $u \in [0, 1]$ with end points

$$x_k = f_{x,k}(0.0) \qquad x_{k+1} = f_{x,k}(1.0)$$
$$y_k = f_{y,k}(0.0) \qquad y_{k+1} = f_{y,k}(1.0)$$

Natural Cubic Splines

Natural cubic splines have adjacent curve sections with the same 1st and 2nd derivatives. Assuming we have n sections and $n+1$ control points. Then in the x dimension we have:

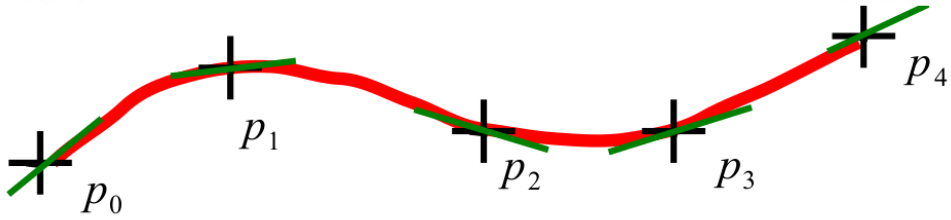
- $4n$ variables to calculate that describe the curve
- $2n$ equations from end points, n-1 equations from asserting 1st derivatives are the same at boundaries, $n-1$ equations from asserting 2nd derivatives are the same at boundaries.
- $\Rightarrow$ 2 degrees of freedom still to be determined!

Two common approaches:

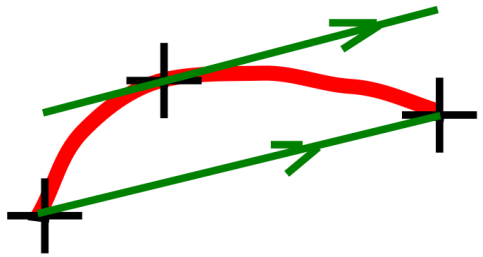
- Set 2nd derivatives to 0 at the ends, or
- add points p_{-1} and p_{n+1}.

Hermite and Cardinal Splines

Hermite Splines are piecewise cubic polynomial splines where the tangents at the interpolated control points are specified.



Cardinal Splines are like Hermite splines but the tangents are not explicitly specified rather they are calculated from adjacent control points.



Bezier Curves

Linear Bezier Curve (just a line segment)

$$B(u) = (1-u)P_0 + uP_1$$

Quadratic Bezier Curve

$$B(u) = (1-u)^2 P_0 + 2(1-u)uP_1 + u^2 P_2$$

Cubic Bezier Curve

$$B(u) = (1-u)^3 P_0 + 3(1-u)^2 uP_1 + 3(1-u)u^2 P_2 + u^3 P_3$$

Generalised Bezier Curve

$$B(u) = \sum_{i=0}^{n} (1-u)^{n-i} u^i P_i$$

for $u \in [0..1]$.

Bezier curves can joined together in a piecewise fashion to form a Bezier spline. Affine transformations on the control points transforms the curve similarly.

B-Spline Curves

B-Splines are similar to Bezier curves in that they use a number of approximate control points. One advantage of B-Splines over Bezier curves is that the degree of the polynomial can be controlled independently from the number of control points. Also changes in a control point only effects the curve in that neighbourhood. However B-Splines are more complex to program then Bezier curves.

B-Spline Curves

With n control points, and setting d to the degree of the polynomials. The points on the curve are calculated via a blending function:

$$B(u) = \sum_{i=0}^{n-1} P_i B_{i,d}(u), \qquad u \in [u_{\text{min}}, u_{\text{max}}]$$

$n + d - 1$ real values are defined. These are called the knots:

$$u_0 \leq u_1 \leq u_2 \leq ... \leq u_{n+d-2}$$

The blending function is then defined recursively:

$$B_{k,0}(u) = \begin{cases} 1 & \text{if } u \in [u_k, u_{k+1}] \\ 0 & \text{otherwise} \end{cases}$$

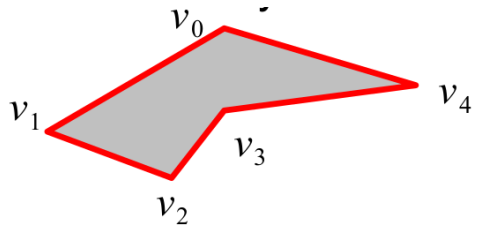
$$B_{k,d}(u) = \frac{u - u_k}{u_{k+d} - u_k} B_{k,d-1}(u) + \frac{u_{k+d+1} - u}{u_{k+d+1} - u_{k+1}} B_{k+1,d-1}(u)$$

2D Graphics

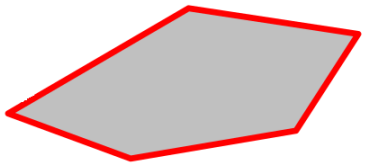
Polygons

Polygons

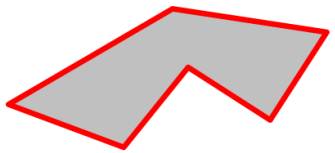
Polygons can be described by a list of vertices (normally counter-clockwise).



Polygons can be:



(a) **Convex** internal angles $< 180^\circ$

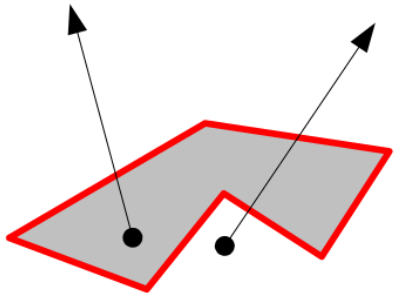


(b) **Concave** not convex

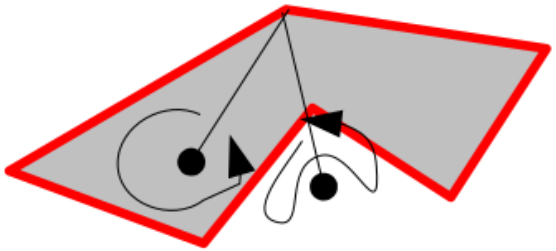
When vertices are collinear or have repeated positions, then the polygon is described as **degenerate**.
To simplify rendering, concave polygons may be broken up into convex polygons and convex polygons may be broken up into triangles.

Identifying Interior of Polygons

There are two ways of defining/identifying if a point is in the interior of a polygon or not.



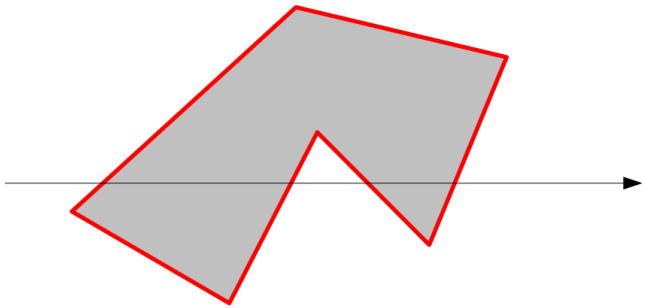
(a) **Odd-even rule** draw any line from the point in question to outside the coordinate extents of the polygon and count the number of edges of the polygon crossed. If the count is even, then the point lies outside the polygon; otherwise, if the count is odd, then the point is considered to be an interior point.



(b) **Non-zero winding number** the winding number is calculated by counting the number of times the perimeter of the polygon travels around the point in a counter-clockwise direction. If this number is non-zero, then the point is considered to be an interior point.

Filling Polygons

The odd-even rule can be used for filling polygons.



For each scan line:

- calculate intersections.
- sort in terms of their x values.
- use odd-even rule for filling.

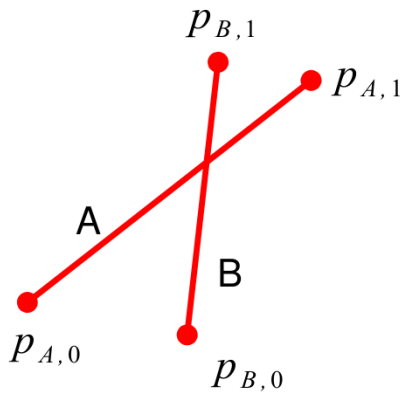
A few special cases to be careful of:

- the scan line crossing vertices, horizontal edges, fractional x value and determining interior.
- Is there a better way to do this if the polygon is convex?

Intersection of Line Segments

To determine if two line segments intersect we can use a parametric form for the line.

$$
\begin{aligned}
v_A &= p_{A,1} - p_{A,0} \\
v_B &= p_{B,1} - p_{B,0} \\
P_A(u) &= uv_A + p_{A,0} \\
P_B(u) &= tv_B + p_{B,0}
\end{aligned}
$$



We need to solve for u, t such that

$$P_A(u) = P_B(t)$$

with $u \in [0, 1]$ and $t \in [0, 1]$. This gives us two equations and 2 unknowns.

Intersection of Line Segments

The solutions are:

$$t = \frac{v_{xA}(y_B - y_A) - v_{yA}(x_B - x_A)}{v_{yA}v_{xB} - v_{xA}v_{yB}}$$
$$u = \frac{v_{xB}(y_A - y_B) - v_{yB}(x_A - x_B)}{v_{yB}v_{xA} - v_{xB}v_{yA}}$$

If $t \in [0, 1]$and $u \in [0, 1]$ then the lines intersect!