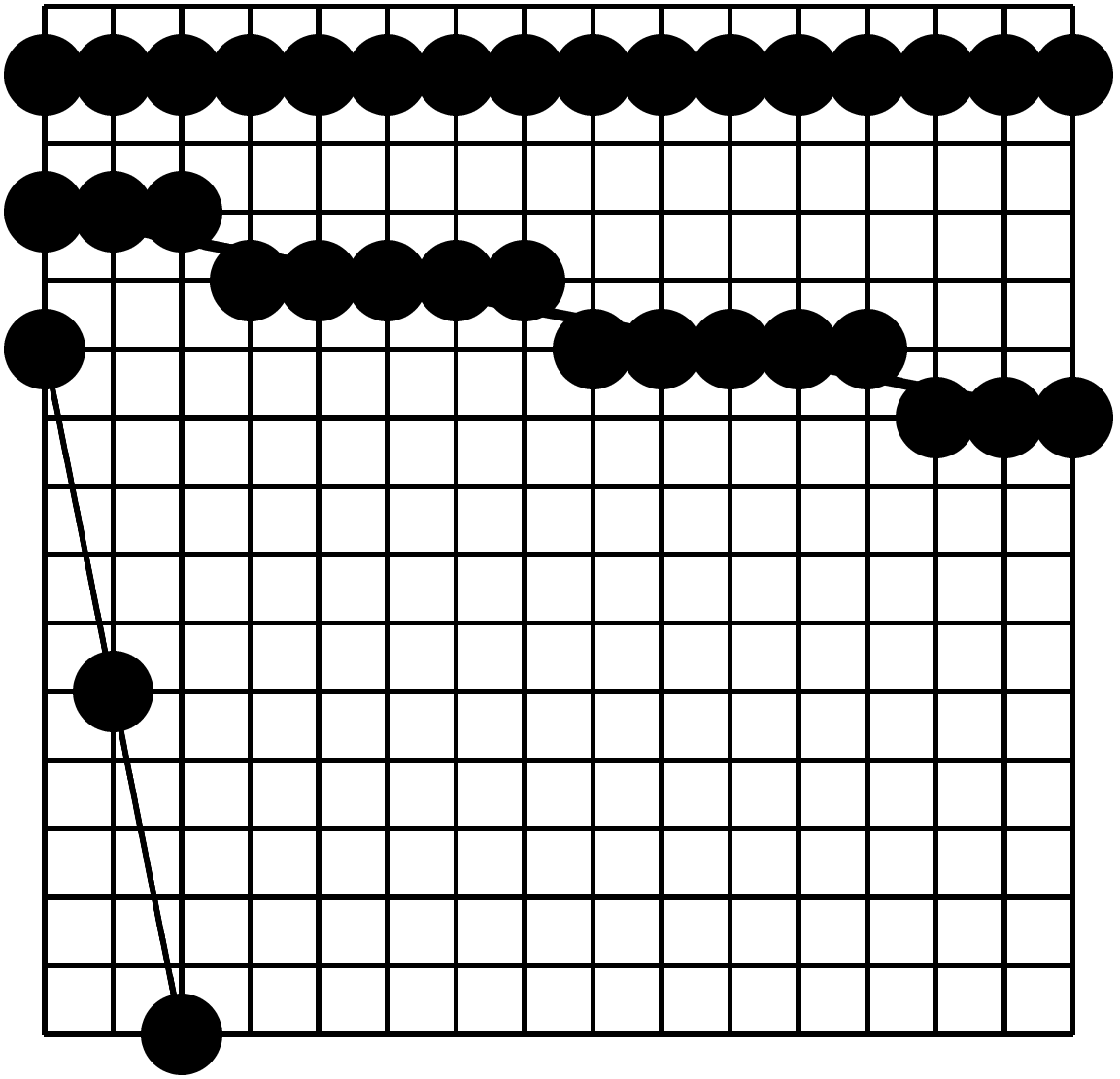
Drawing lines

Naïve line drawing algorithm

```
// Connect to grid points(x0,y0) and
// (x1,y1) by a line.
void drawLine(int x0, int y0, int x1, int y1)
{
  int x;
  double dy = y1 - y0;
  double dx = x1 - x0;
  double m  = dy / dx;
  double y  = y0;

  for (x=x0; x<=x1; x++)
  {
    drawPixel(x, round(y));
    y = y + m;   //or: y = y0 + m*(x - x0);
  }
}
```

Drawing lines



Drawing lines

For a line with an absolute slope greater than one, the roles of the x- and the y-axis should be exchanged for drawing the line.

Very often, drawing an image requires drawing a large number of lines.

Therefore, the line drawing algorithm should be as efficient as possible.

The **Bresenham** or **midpoint algorithm** does not need any floating operations and draws lines on raster graphics using only integer arithmetics.

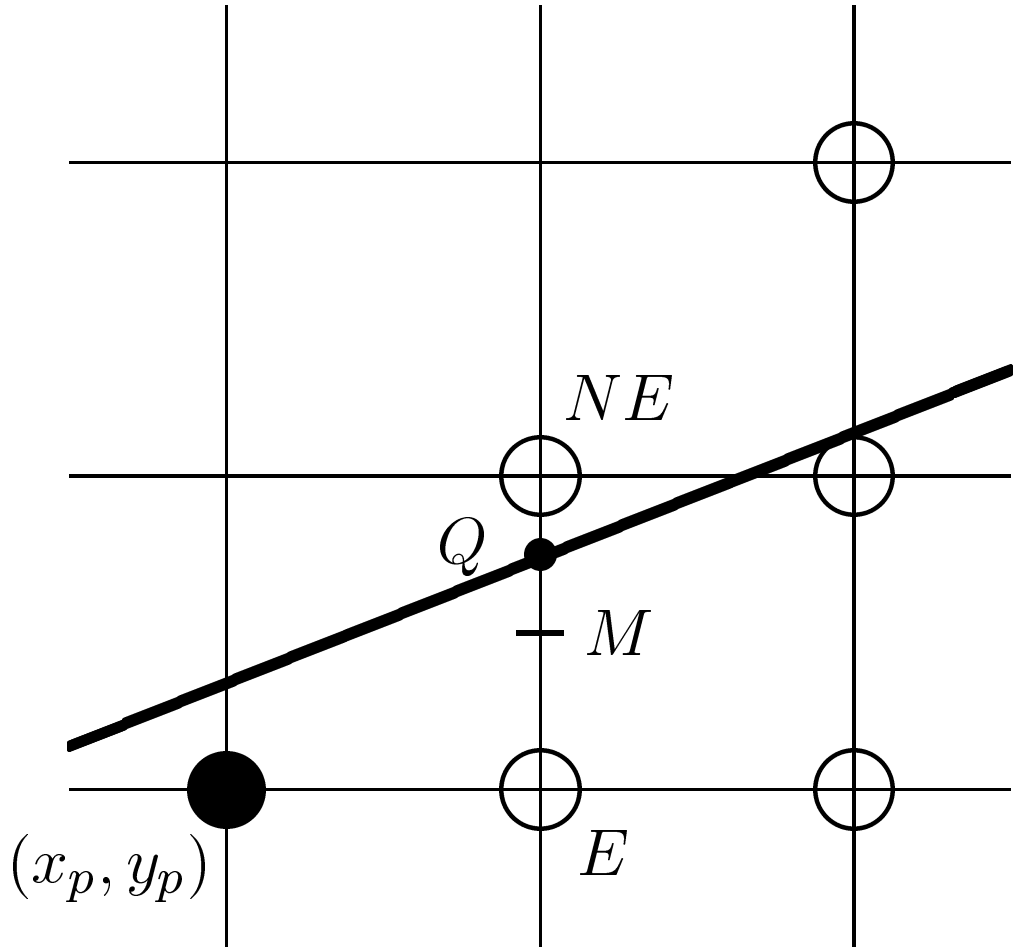
The midpoint algorithm

In the following, drawing a line with slope between 0 and 1 is considered.

If such a line is drawn pixel by pixel and the last pixel which was drawn is located at (x_p, y_p), then there are only two choices for the next pixel.

- the pixel $(x_p + 1, y_p)$ right of (x_p, y_p) or
- the pixel $(x_p + 1, y_p + 1)$ right and above of (x_p, y_p).

The midpoint algorithm



The midpoint algorithm

Considering the location of the line w.r.t. the midpoint provides the correct decision which of the two candidate pixels should be drawn next.

- If the midpoint lies below the line, the upper (NE) pixel should be drawn.
- If the midpoint lies above the line, the lower (E) pixel should be drawn.

The midpoint algorithm

Equation for the line:

$$y = f(x) = m \cdot x + b$$

Implicit form:

$$F(x, y) = Ax + By + C \qquad (a \geq 0)$$

or

$$F(x, y) = m \cdot x - y + b = 0.$$

The midpoint algorithm

- $F(x,y) = 0 \Leftrightarrow (x,y)$ lies on the line.
- $F(x,y) < 0 \Leftrightarrow (x,y)$ lies above the line.
- $F(x,y) > 0 \Leftrightarrow (x,y)$ lies below the line.

Choosing for (x,y) the coordinates of the midpoint (x_M, y_M) yields:

- $F(x_M, y_M) < 0 \Leftrightarrow$ The pixel E should be drawn.
- $F(x_M, y_M) > 0 \Leftrightarrow$ The pixel NE should be drawn.

In the case $F(x_M, y_M) = 0$ one can choose E or NE. (But the decision should be the same each time, for instance always the upper pixel.)

The midpoint algorithm

Draw a line connecting pixel (x_0, y_0) with pixel (x_1, y_1).

Equation for the line:

$$\frac{y - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0}$$

$$y = \frac{dy}{dx}x + y_0 - \frac{dy}{dx}x_0$$

where $dx = x_1 - x_0$ and $dy = y_1 - y_0$.

The midpoint algorithm

implicit form:

$$0 \;=\; \frac{dy}{dx}x - y + y_0 - \frac{dy}{dx}x_0$$

or

$$F(x,y) \;=\; dy \cdot x - dx \cdot y + C \;=\; 0$$

where

$$C \;=\; dx \cdot y_0 - dy \cdot x_0.$$

The midpoint algorithm

Inserting the midpoint $M = (x_M, y_M)$ with integer value x_M and

$$y_M = y_M^{(0)} + \frac{1}{2}$$

with integer value $y_M^{(0)}$ requires floating operations.

Multiplication by the factor 2:

The midpoint algorithm

$$\tilde{F}(x,y) \;=\; 2\cdot dy\cdot x - dx\cdot 2\cdot y + 2\cdot C \;=\; 0$$

Inserting the midpoint $M = (x_M, y_M)$ with

$$y_M \;=\; y_M^{(0)} + \frac{1}{2},$$

where $x_M, y_M^{(0)}$ are integer values yields

$$\tilde{F}(x_M,y_M) \;=\; 2\cdot dy\cdot x_M - dx\cdot (2\cdot y_M^{(0)} + 1) + 2\cdot C \;=\; 0.$$

Only integer operations are needed!

The midpoint algorithm

The midpoint algorithm used in computer graphics is further improved by

incremental computation of the values $\tilde{F}(x_M, y_M)$. $\longrightarrow$ In the loop for drawing, only integer additions, no multiplications are needed.

The midpoint algorithm

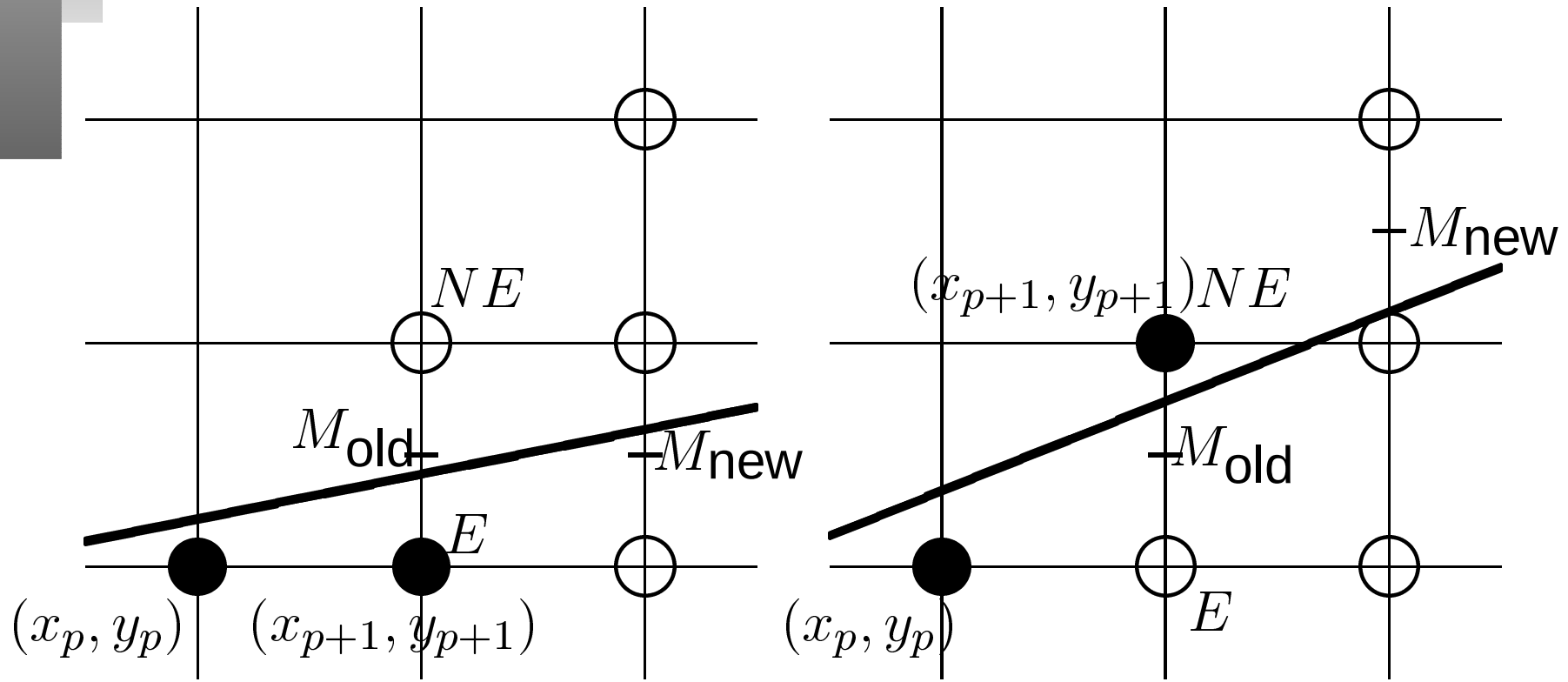
Incremental computation of the decision variable:

$$d \;=\; F(x_M, y_M) \;=\; dy \cdot x_M - dx \cdot y_M + C$$

How does d change?

The midpoint algorithm

Two cases:



The midpoint algorithm

Case 1: E, i.e. $(x_{p+1}, y_{p+1}) = (x_p + 1, y_p)$ is the pixel drawn after (x_p, y_p).

Midpoint to be considered for the next pixel (x_{p+2}, y_{p+2}):

$$M_{\text{new}} = \left(x_p + 2, y_p + \frac{1}{2}\right).$$

The midpoint algorithm

$$d_{\text{new}} = F\left(x_p + 2, y_p + \frac{1}{2}\right) = dy\cdot(x_p+2) - dx\cdot\left(y_p + \frac{1}{2}\right) + C$$

$$d_{\text{old}} = F\left(x_p + 1, y_p + \frac{1}{2}\right) = dy\cdot(x_p+1) - dx\cdot\left(y_p + \frac{1}{2}\right) + C$$

$$\Delta_E = d_{\text{new}} - d_{\text{old}} = dy.$$

The midpoint algorithm

Case 2: NE, i.e. $(x_{p+1}, y_{p+1}) = (x_p + 1, y_p + 1)$ is the pixel drawn after (x_p, y_p).

Midpoint to be considered for the next pixel:

$$M_{\text{new}} = \left(x_p + 2, y_p + \frac{3}{2}\right)$$

The midpoint algorithm

$$d_{\text{new}} = F\left(x_p + 2, y_p + \frac{3}{2}\right) = dy \cdot (x_p + 2) - dx \cdot \left(y_p + \frac{3}{2}\right) + C$$

$$d_{\text{old}} = F\left(x_p + 1, y_p + \frac{1}{2}\right) = dy \cdot (x_p + 1) - dx \cdot \left(y_p + \frac{1}{2}\right) + C$$

$$\Delta_{NE} = d_{\text{new}} - d_{\text{old}} = dy - dx$$

The midpoint algorithm

$$\Delta = \begin{cases} dy & \text{if } E \text{ was chosen,} \\ dy - dx & \text{if } NE \text{ was chosen.} \end{cases}$$

i.e.

$$\Delta = \begin{cases} dy & \text{if } d_{\text{old}} < 0, \\ dy - dx & \text{if } d_{\text{old}} > 0. \end{cases}$$

Δ is always an integer number. Therefore, the decision variable d can only change by integer values.

The midpoint algorithm

Initialisation of d: Starting pixel: (x_0, y_0)

First midpoint to be considered: $\left(x_0 + 1, y_0 + \frac{1}{2}\right)$

$$
\begin{aligned}
d_{\text{init}} &= F\left(x_0 + 1, y_0 + \frac{1}{2}\right) \\
&= dy \cdot (x_0 + 1) - dx \cdot \left(y_0 + \frac{1}{2}\right) + C \\
&= dy \cdot x_0 - dx \cdot y_0 + C + dy - \frac{dx}{2} \\
&= F(x_0, y_0) + dy - \frac{dx}{2} \\
&= dy - \frac{dx}{2}
\end{aligned}
$$

The midpoint algorithm

d_{init} is necessarily an integer number.

Instead of the decision variable d consider the decision variable $D = 2 \cdot d$:

$$D = \hat{F}(x,y) = 2 \cdot F(x,y) = 2 \cdot dy \cdot x - 2 \cdot dx \cdot y + 2 \cdot C = 0$$

D is always an integer number.

The midpoint algorithm

$$D_{\text{init}} = 2 \cdot dy - dx$$

$$D_{\text{new}} = D_{\text{old}} + \Delta \quad \text{where}$$

$$\Delta = \begin{cases} 2 \cdot dy & \text{if } D_{\text{old}} < 0, \\ 2 \cdot (dy - dx) & \text{if } D_{\text{old}} > 0. \end{cases}$$

The midpoint algorithm

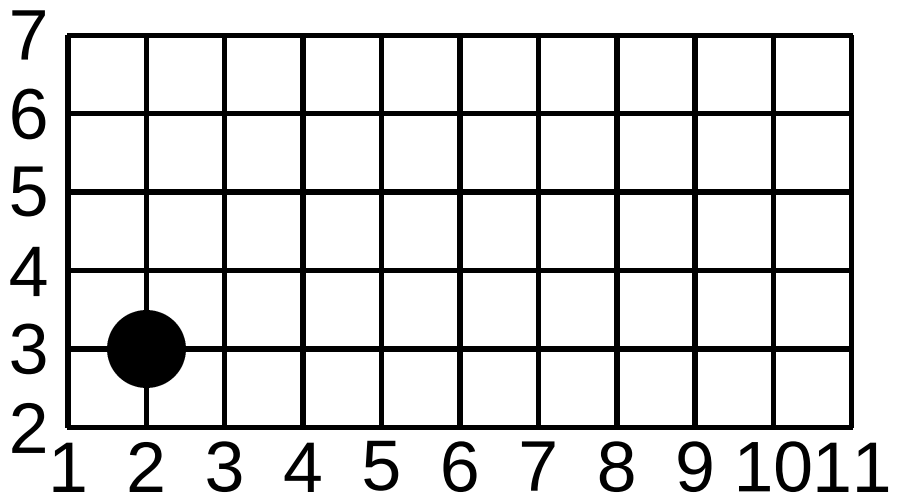
Example: A line from (2,3) to (10,6):

$$
\begin{aligned}
dx &= 10 - 2 &&= 8 \\
dy &= 6 - 3 &&= 3 \\
\\
\Delta_E &= 2 \cdot dy &&= 6 \\
\Delta_{NE} &= 2 \cdot (dy - dx) &&= -10
\end{aligned}
$$

The midpoint algorithm

$$\begin{aligned} \Delta_E &= 6 \\ \Delta_{NE} &= -10 \end{aligned}$$

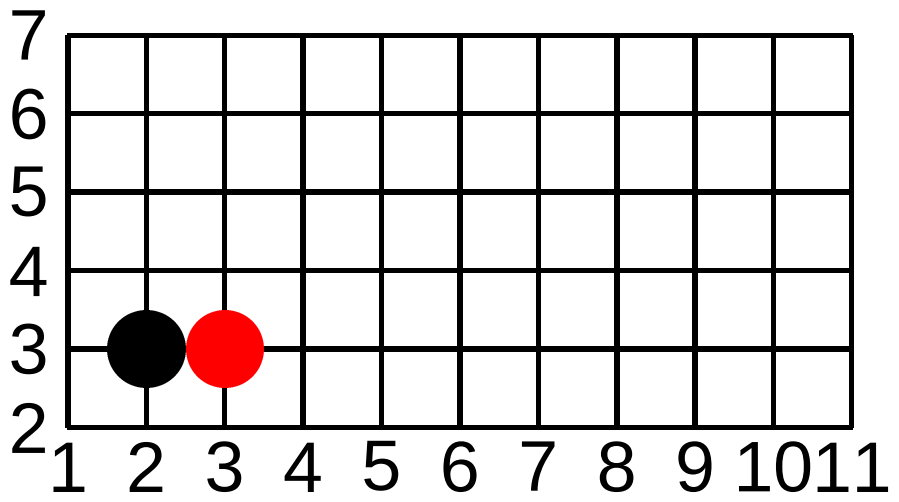
$$D_{\text{init}} = 2 \cdot dy - dx = -2 \quad (E)$$



The midpoint algorithm

$$\begin{aligned} \Delta_E &= 6 \\ \Delta_{NE} &= -10 \end{aligned}$$

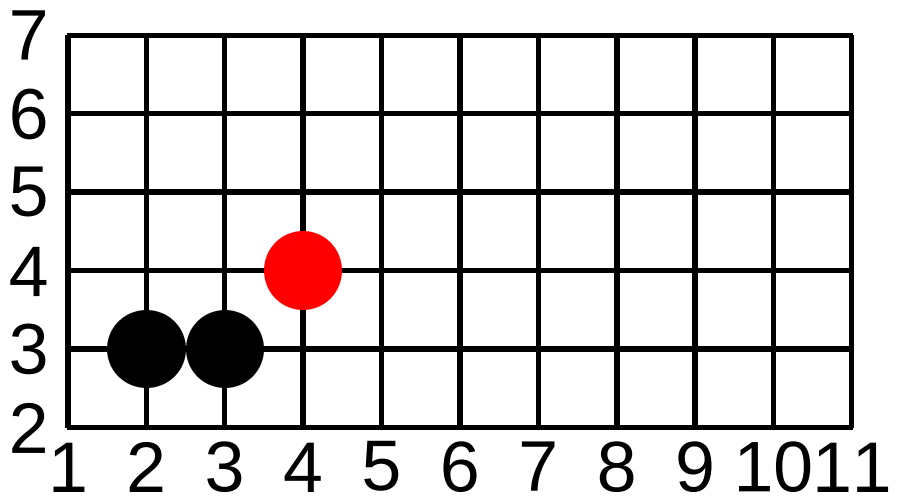
$$D_{\text{init}} = 2 \cdot dy - dx = -2 \quad (E)$$



The midpoint algorithm

$$\Delta_E = 6$$
$$\Delta_{NE} = -10$$

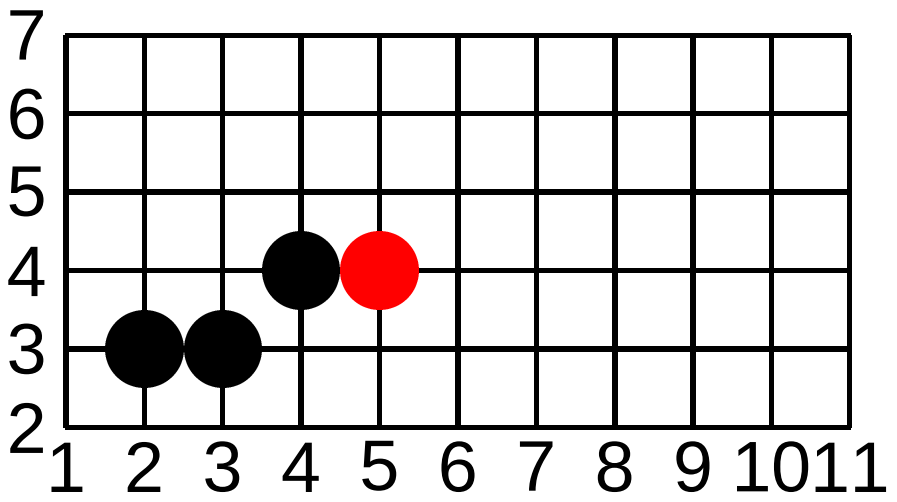
$$D_{\text{init}+1} = D_{\text{init}} + \Delta_E = 4 \quad (NE)$$



The midpoint algorithm

$$\begin{aligned} \Delta_E &= 6 \\ \Delta_{NE} &= -10 \end{aligned}$$

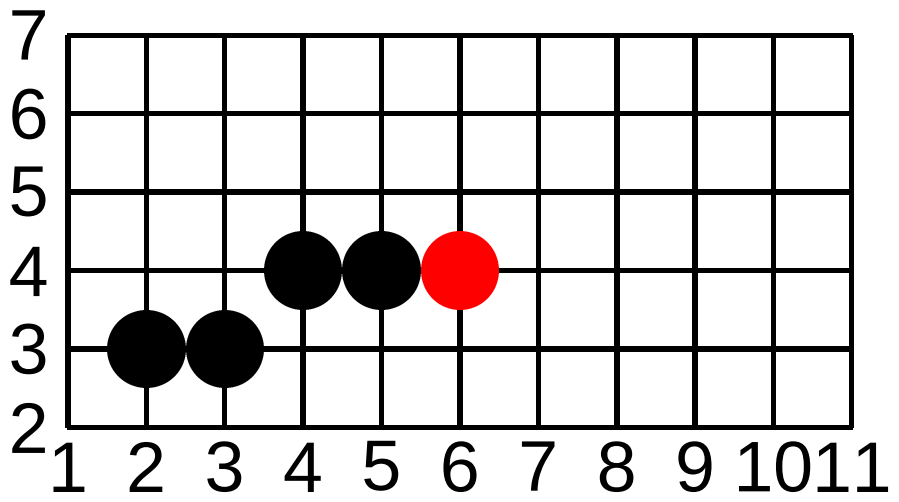
$$D_{\text{init}+2} = D_{\text{init}+1} + \Delta_{NE} = -6 \quad (E)$$



The midpoint algorithm

$$\begin{aligned} \Delta_E &= 6 \\ \Delta_{NE} &= -10 \end{aligned}$$

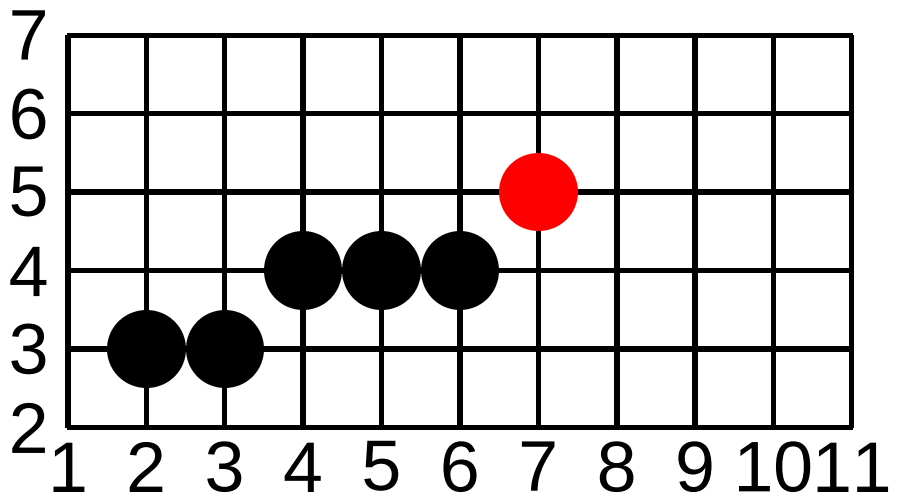
$$D_{\text{init}+3} = D_{\text{init}+2} + \Delta_E = 0 \quad (E?)$$



The midpoint algorithm

$$\begin{aligned} \Delta_E &= 6 \\ \Delta_{NE} &= -10 \end{aligned}$$

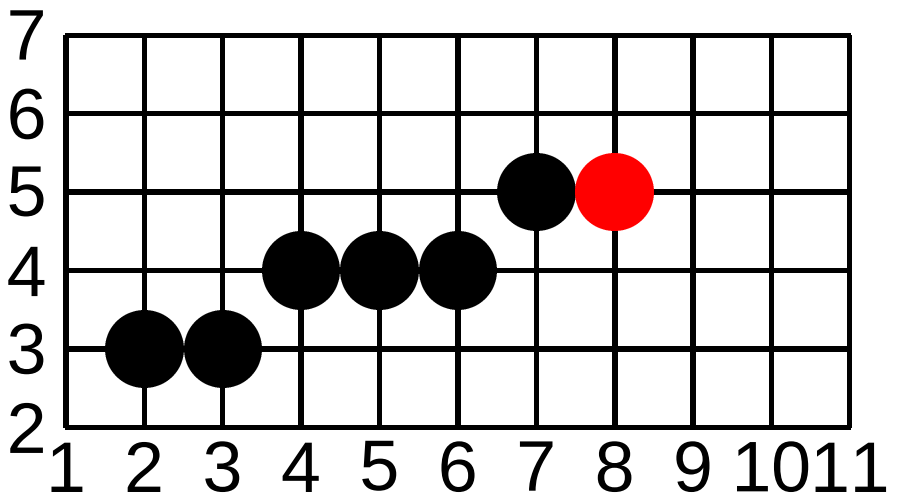
$$D_{\text{init}+4} = D_{\text{init}+3} + \Delta_E = 6 \quad (NE)$$



The midpoint algorithm

$$\Delta_E = 6$$
$$\Delta_{NE} = -10$$

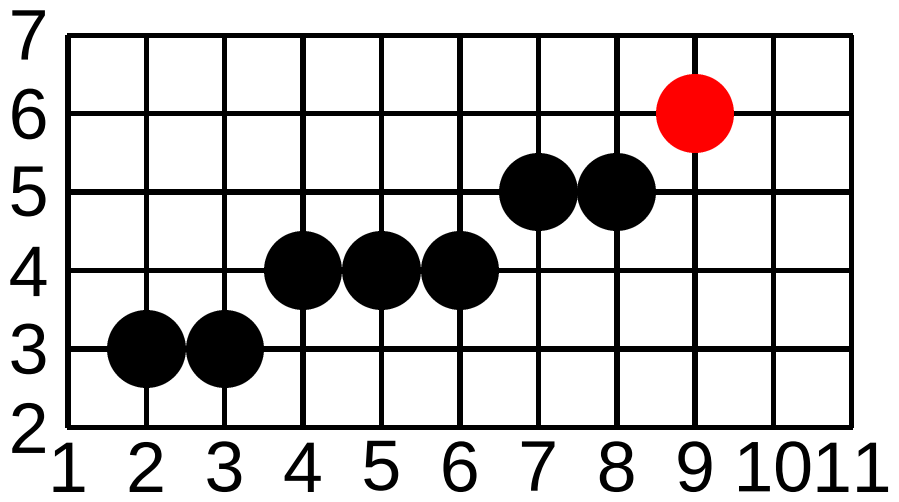
$$D_{\text{init}+5} = D_{\text{init}+4} + \Delta_{NE} = -4 \quad (E)$$



The midpoint algorithm

$$\Delta_E = 6$$
$$\Delta_{NE} = -10$$

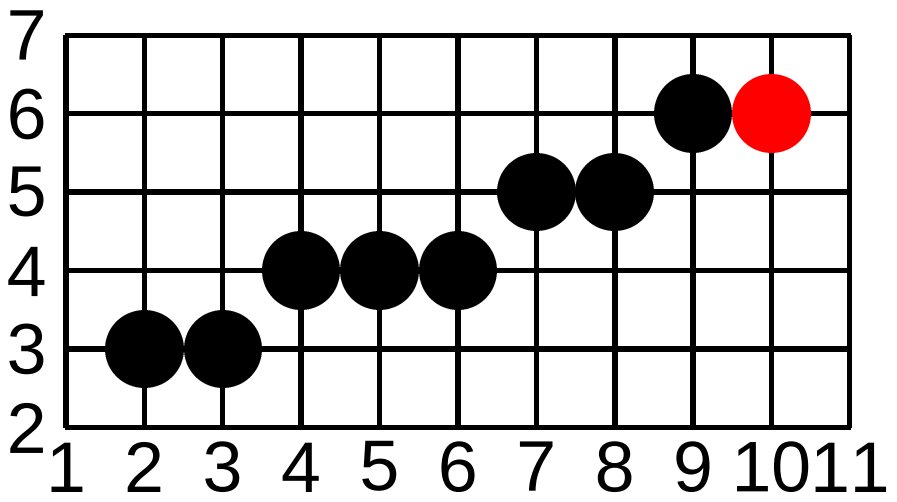
$$D_{\text{init}+6} = D_{\text{init}+7} + \Delta_E = 2 \quad (NE)$$



The midpoint algorithm

$$\Delta_E = 6$$
$$\Delta_{NE} = -10$$

$$D_{\text{init}+7} = D_{\text{init}+6} + \Delta_{NE} = -8 \quad (E)$$



The midpoint algorithm

- **Lines with an absolute slope greater than 1:**

  Change the roles of the x- and y-axis in the algorithm.

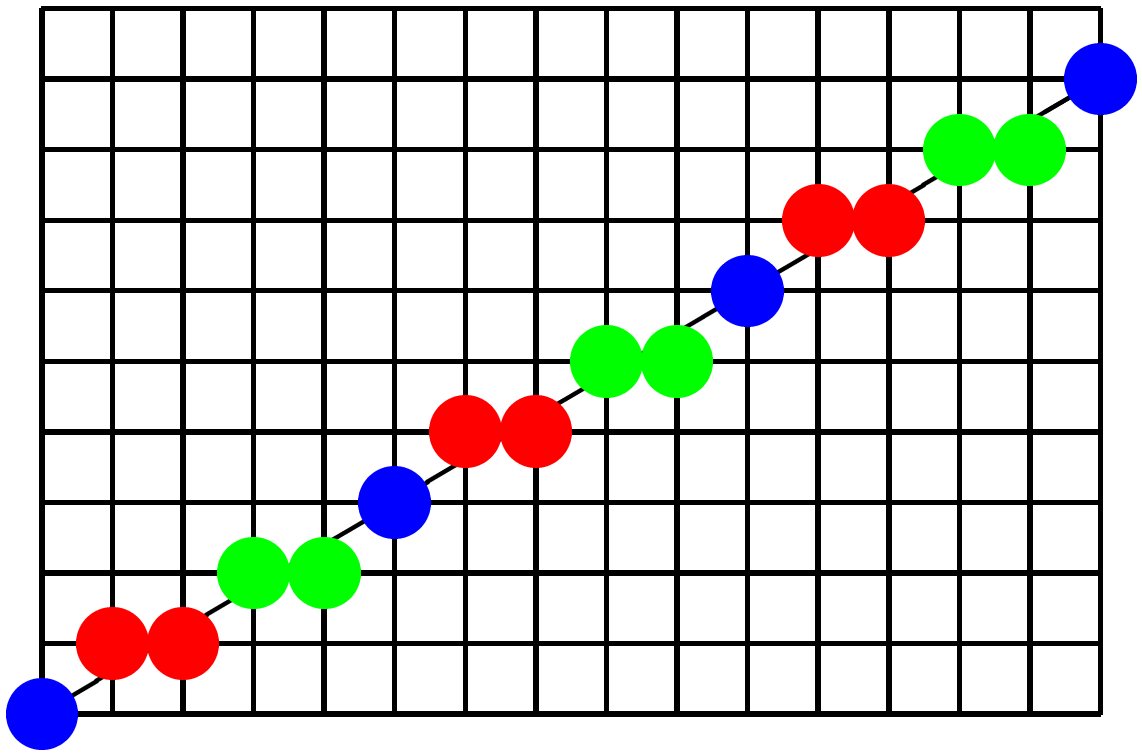
- **Lines with negative slope (between -1 and 0):**

  Carry out similar derivations as in the case of lines with positive slope. Instead of “northeastern” pixel, the t‘southeastern” must be considered.

- **Line segments whose starting and endpoint are not integer-valued:**

  Roundoff the coordinates of the starting and endpoint and connect the corresponding pixels by a line.

Structural algorithms



When drawing a line, repeated pixel patterns usually occur, for instance

D, H, D, H, D

Structural algorithms

Let D denote a diagonal (NE pixel) and H a horizontal step (E pixel). Then the line can be decribed by a (repeated) pattern of D and H steps.

Structural algorithms determine such patterns for drawing lines.

The midpoint algorithm needs $O(n)$ integer operations to draw a line of n pixels.

Structural algorithm only have logarithmic complexity (however, with more complicated operations).

Structural algorithms

Principle:

- Given: A start and endpoint (x_0, y_0) and (x_1, y_1) (of a line segment with slope between 0 and 1).
- Compute $dx = x_1 - x_0$ and $dy = y_1 - y_0$
- Apart from the starting pixel, dx pixels must be drawn. This will invoke dy diagonal and $(dx - dy)$ horizontal steps.
- Choose the sequence $H^{dx-dy}D^{dy}$ as a first approximation.
- Permutate this sequence in a suitable way to obtain the correct sequence.

Brons' algorithm

- If dx and dy (and therefore also $(dx - dy)$) have a common divisor greater than one, i.e., $g = \gcd(dx, dy) > 1$, then the pixel line can be drawn by g repetitions of a sequence of length dx/g.
- Therefore, it can be assumed without loss of generality that dx and dy have no common divisor.
- Let P and Q be two words (sequences) over the alphabet $\{D, H\}$.
- From a starting sequence $P^p Q^q$ with frequencies p and q having no common divisor and assuming without loss of generality $1 < q < p$, the integer division

$$p = k \cdot q + r, \quad 0 < r < q$$

Brons' algorithm

- leads to the permutated sequence

$$(P^k Q)^{q-r}(P^{k+1}Q)^r \quad \text{if } (q-r) > r,$$

$$(P^{k+1}Q)^r(P^k Q)^{q-r} \quad \text{if } r > (q-r).$$

- Apply the same procedure in a recursive manner to the subsequences of length r and $(q-r)$, respectively, until $r = 1$ or $(q-r) = 1$ holds.

Example

$(x_0, y_0) = (0, 0)$, $(x_1, y_1) = (82, 34)$

$dx = 82$, $dy = 34$, gcd$(dx, dy) = 2$

Therefore, $\widetilde{dx} = dx/2 = 41$, $\widetilde{dy} = dy/2 = 17$.

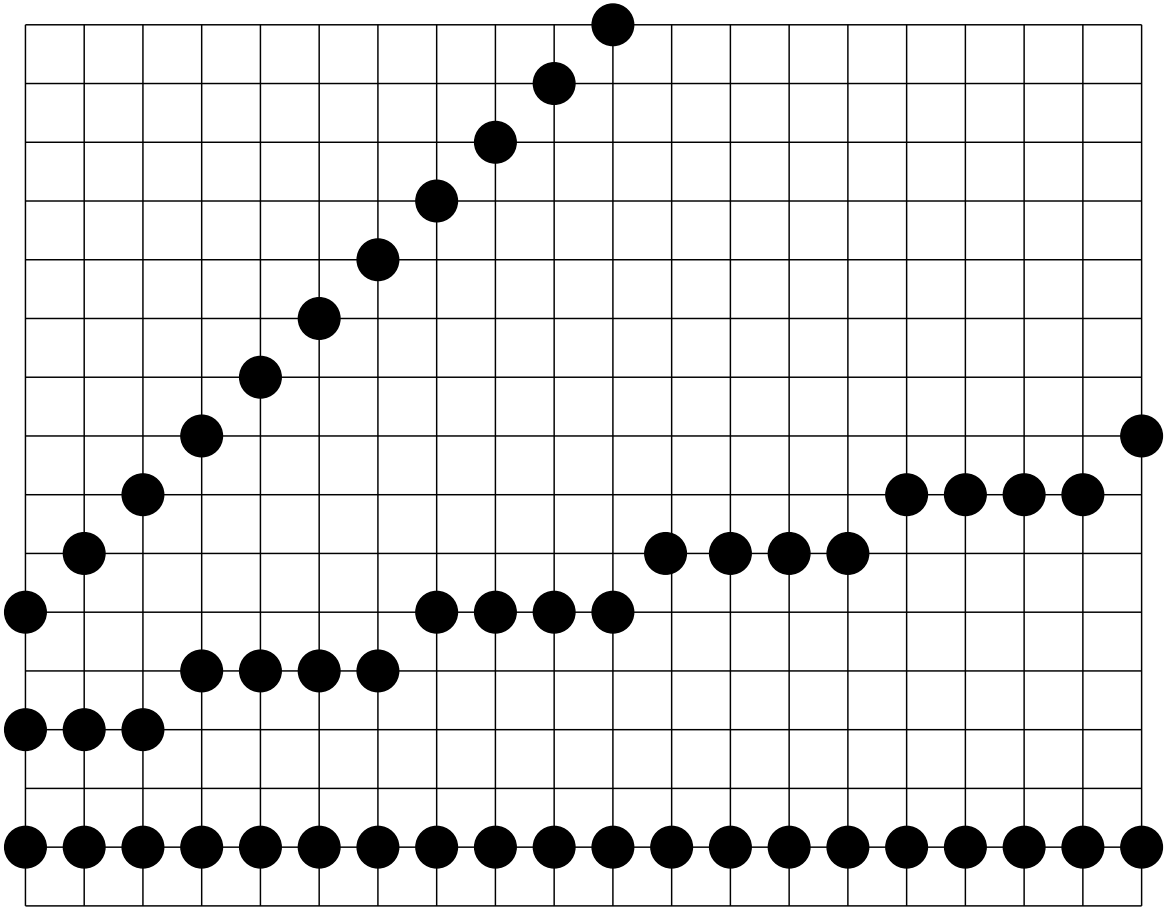
$$H^{24}D^{17} \qquad 24 = 1 \cdot 17 + 7$$

$$(HD)^{10}(H^2D)^7 \qquad 10 = 1 \cdot 7 + 3$$

$$(HDH^2D)^4((HD)^2H^2D)^3 \qquad 4 = 1 \cdot 3 + 1$$

$$(HDH^2D(HD)^2H^2D)^2((HDH^2D)^2(HD)^2((HD)^2H^2D))^1$$

Pixel densities

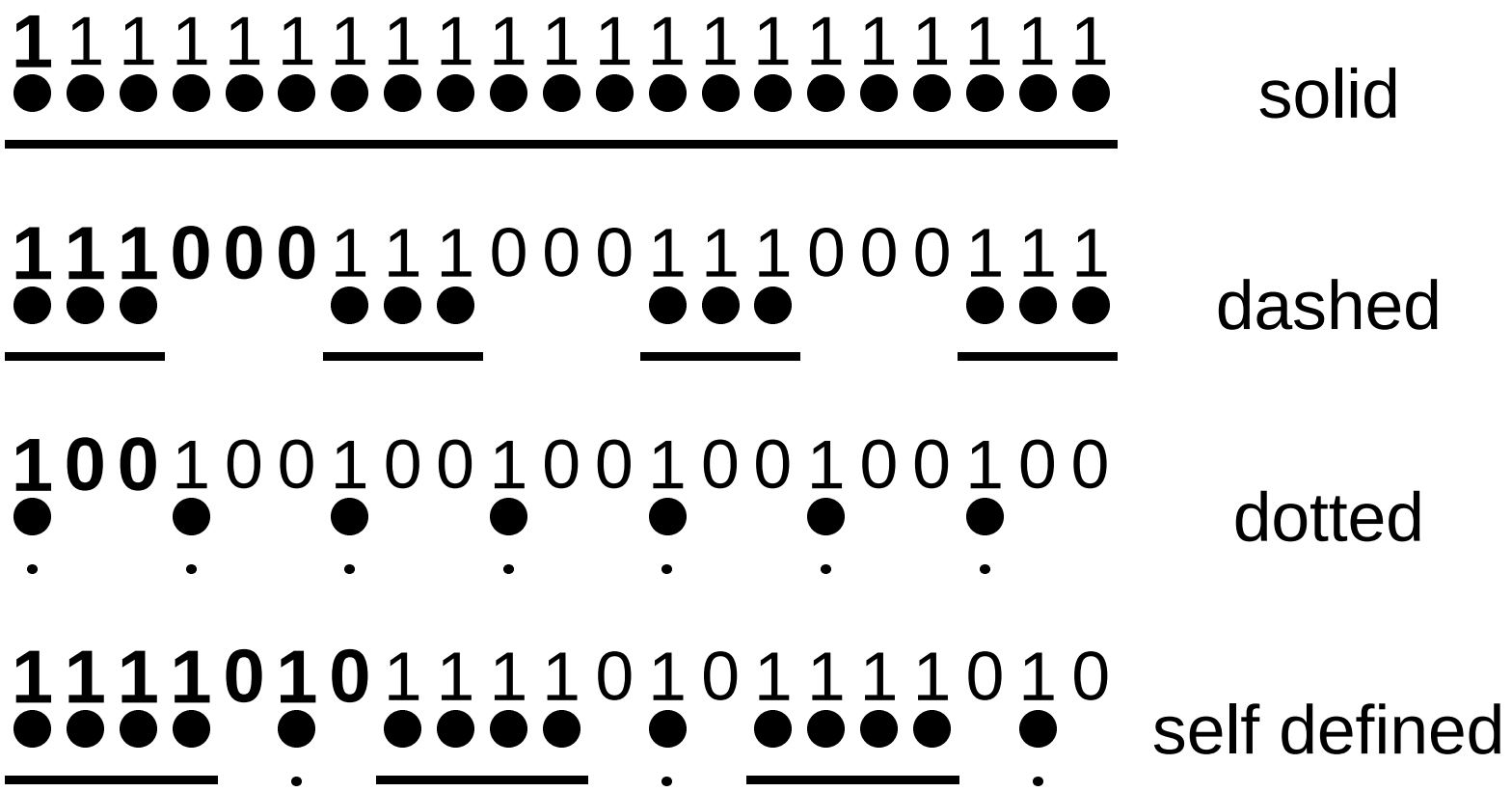


Pixel densities

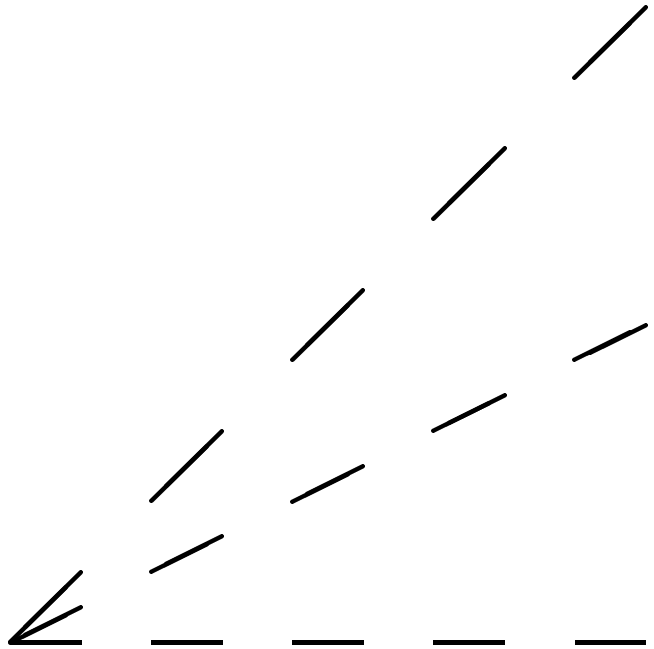
A line connecting the points $(0, 0)$ and (n, m) where $0 < m \leq n$.

m	Length of the line	Pixel density
0	n	1
$\frac{n}{4}$	$n \cdot \sqrt{1 + \frac{1}{16}}$	$\frac{1}{\sqrt{1 + \frac{1}{16}}}$
n	$n \cdot \sqrt{2}$	$\frac{1}{\sqrt{2}}$
m	$n \cdot \sqrt{1 + \left(\frac{m}{n}\right)^2}$	$\frac{1}{\sqrt{1 + \left(\frac{m}{n}\right)^2}}$

Line styles and bitmasks



Line styles and bitmasks



Different dash lengths for the same bitmask.

Line styles in Java 2D

Line thickness:

```
BasicStroke bsThickLine =
                  new BasicStroke(3.0f);
g2d.setStroke(bsThickLine);
```

Dash patterns:

```
BasicStroke bsDash =
  new BasicStroke(thickness,
                  BasicStroke.CAP_BUTT,
                  BasicStroke.JOIN_BEVEL,
                  2.0f,
                  dashPattern,dashPhase);
```

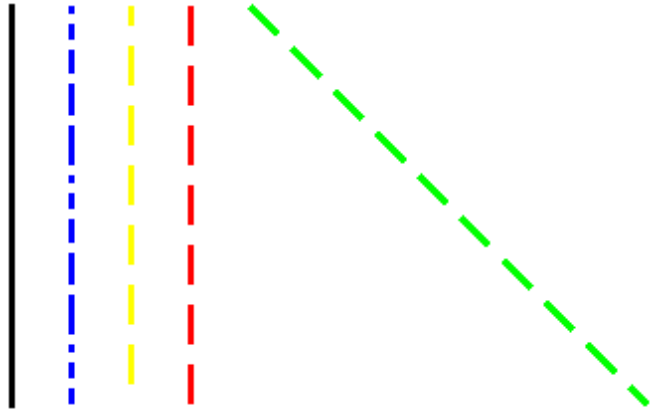
Line styles in Java 2D

- `thickness`: The thickness of the line.
- `BasicStroke.CAP_BUTT, BasicStroke.JOIN_BEVEL, 2.0f` determine how the endings of lines should look and how joins in polylines should be drawn.
- `dashPattern`: Array that defines the desired dash pattern, f.e.

```
float[] dashPattern =
        new float[]{20,10};
```

- `dashPhase`: Position in the dash pattern where drawing should begin.

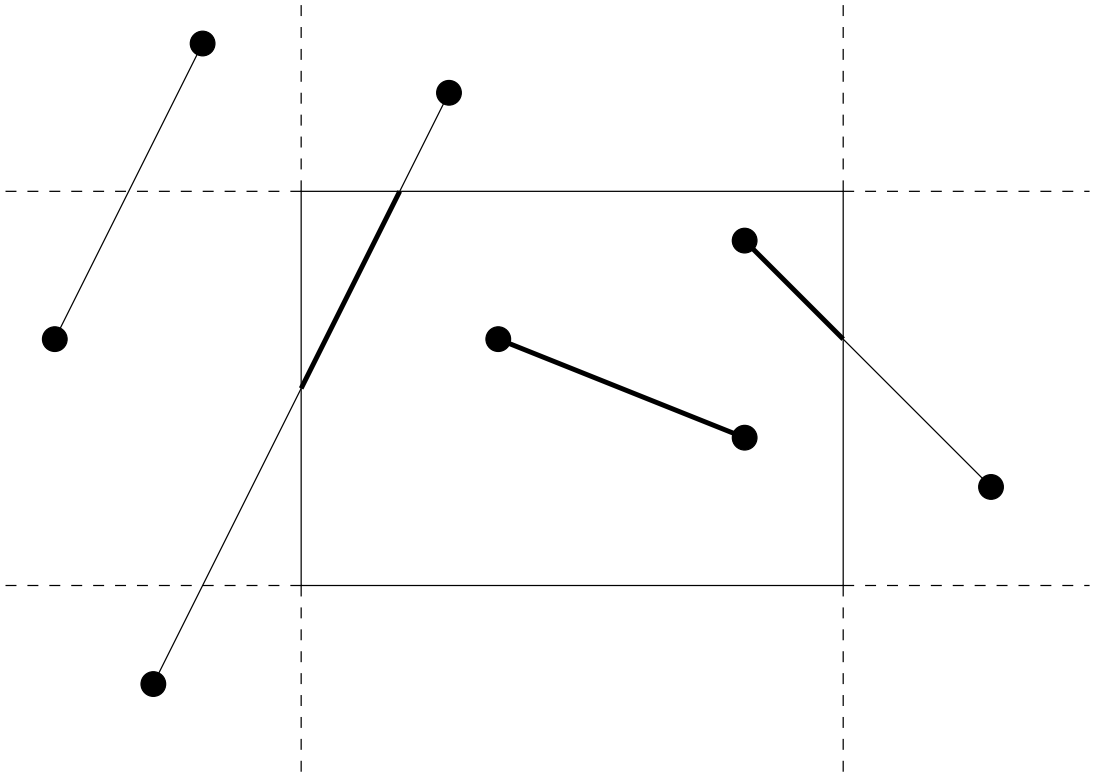
StrokingExample.java



dashPattern	dashPhase
4,5,8,5,12,5,16,5,20,5	0
20,10	10
20,10	0
20,10	0

Clipping

The task of deciding whether objects belong to the scene to be displayed or whether they can be neglected for the specific scene is called **clipping**.



Line clipping

Simple, but computationally inefficient method for line clipping:

Compute the intersection points of the line to be drawn with the four edges of the clipping rectangle.

Representation of a line segment with starting point (x_0, y_0) and endpoint (x_1, y_1) as convex combinations of these two points.

$$\mathbf{g}(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = (1-t) \cdot \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + t \cdot \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$$

where $0 \leq t \leq 1$.

Line clipping

Compute the intersection points of the line with the rectangle defined by the points $(x_{\text{min}}, y_{\text{min}})$ and $(x_{\text{max}}, y_{\text{max}})$:

Determine the intersection point with the lower edge:

$$(1-t_1)\cdot\begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + t_1\cdot\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = (1-t_2)\cdot\begin{pmatrix} x_{\text{min}} \\ y_{\text{min}} \end{pmatrix} + t_2\cdot\begin{pmatrix} x_{\text{max}} \\ y_{\text{min}} \end{pmatrix}$$

Yields solutions for t_1 and t_2. (If no or no unique solution exists, the lines are parallel.)

Line clipping

- $t_1 < 0$ and $t_2 < 0$: The intersection point lies not between the endpoints of the line segment and lies before x_{min}.
- $0 \leq t_1 \leq 1$ and $t_2 < 0$: The line segment intersects the extension of the lower edge before x_{min}.
- $t_1 > 1$ and $t_2 < 0$: The intersection point lies not between the endpoints of the line segment and lies before x_{min}.
- $t_1 < 0$ and $0 \leq t_2 \leq 1$: The intersection point of the line with the lower edge lies before (x_0, y_0).
- $0 \leq t_1 \leq 1$ and $0 \leq t_2 \leq 1$: The line segment intersects the lower edge.

Line clipping

- $t_1 > 1$ and $0 \leq t_2 \leq 1$: The intersection point of the line with the lower edge lies behind (x_1, y_1).
- $t_1 < 0$ and $t_2 > 1$: The intersection point lies not between the endpoints of the line segment and lies behind x_{max}.
- $0 \leq t_1 \leq 1$ and $t_2 > 1$: The line segment intersects the extension of the lower edge behind x_{max}.
- $t_1 > 1$ and $t_2 > 1$: The intersection point lies not between the endpoints of the line segment and lies behind x_{max}.

The same considerations must be carried out for the other edges of the rectangle.

Cohen-Sutherland clipping

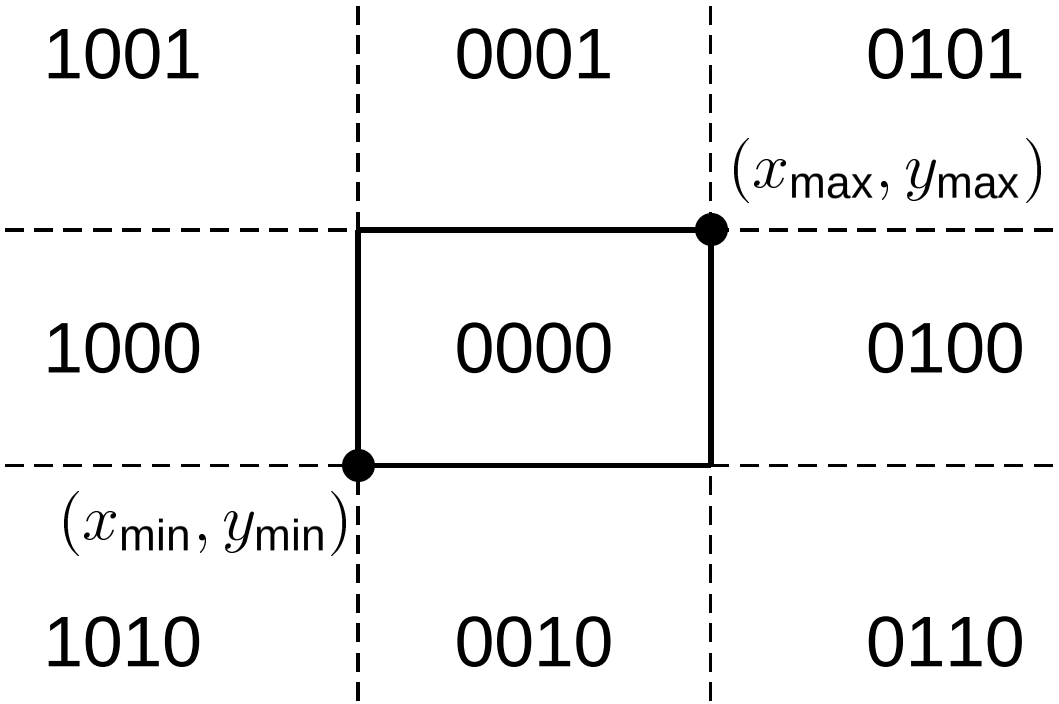
Aim: Try to avoid the computation of intersection points.

Partition the 2D-world into nine areas described by a 4-bit code:

The bit code $b_1^{(P)} b_2^{(P)} b_3^{(P)} b_4^{(P)} \in \{0,1\}^4$ is assigned to the point according to the following rules.

$$
\begin{aligned}
b_1^{(P)} = 1 &\quad\Leftrightarrow\quad x_p < x_{\text{min}} \\
b_2^{(P)} = 1 &\quad\Leftrightarrow\quad x_p > x_{\text{max}} \\
b_3^{(P)} = 1 &\quad\Leftrightarrow\quad y_p < y_{\text{min}} \\
b_4^{(P)} = 1 &\quad\Leftrightarrow\quad y_p > y_{\text{max}}
\end{aligned}
$$

Cohen-Sutherland clipping

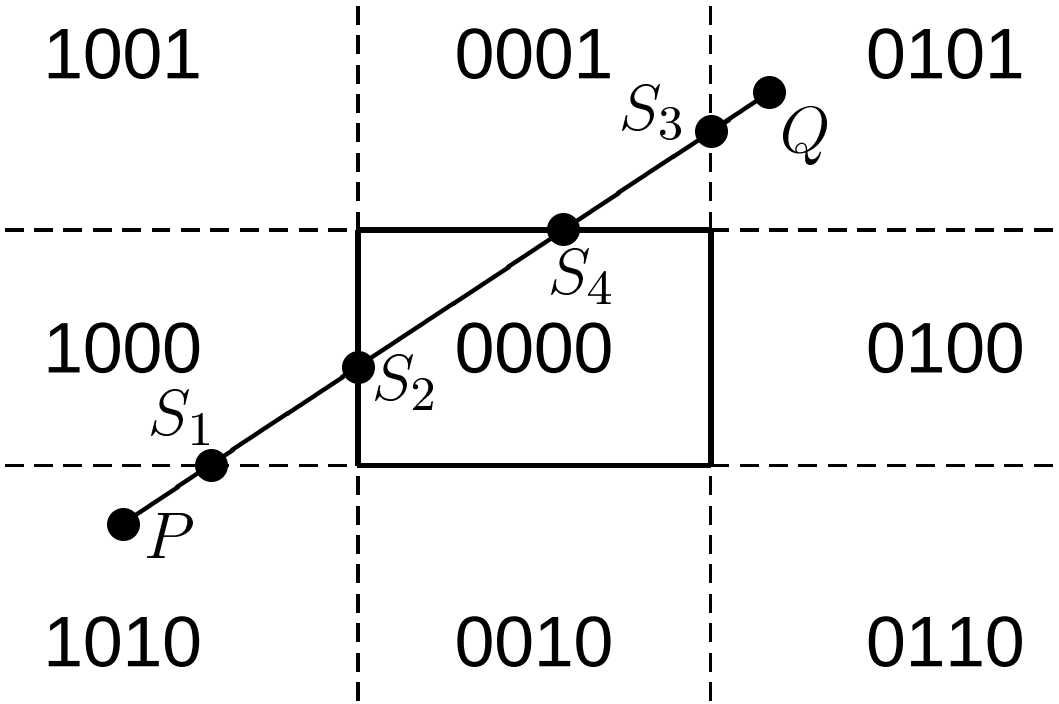


Cohen-Sutherland clipping

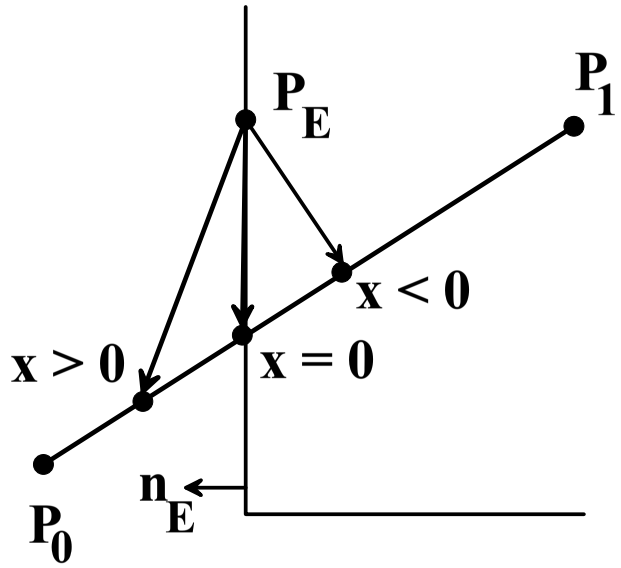
A line starting at point P and ending at Q.

- $b^{(P)} \vee b^{(Q)} = 0000 \rightarrow$ Draw the line $\overline{PQ}$. (ready)
- $b^{(P)} \wedge b^{(Q)} \neq 0000 \rightarrow$ No drawing required. (ready)
- $b^{(P)} \neq 0000$ or $b^{(Q)} \neq 0000$ must hold. Without loss of generality, assume $b^{(P)} \neq 0000$. Compute the intersection point of the line for the edge or one of the two edges responsible for the one(s) in $b^{(P)} \neq 0000$. Replace P by this intersection point and start the algorithm again with the shortened line.

Cohen-Sutherland clipping



Cyrus-Beck clipping



$$\mathbf{g}(t) \;=\; (1-t)\cdot\mathbf{p}_0 + t\cdot\mathbf{p}_1 \;=\; \mathbf{p}_0 + (\mathbf{p}_1-\mathbf{p}_0)t \qquad (t\in[0,1])$$

A vector connecting $\mathbf{p}_E$ with a point on the line from $\mathbf{p}_0$ to $\mathbf{p}_1$ can be written as

$$\mathbf{p}_0 + (\mathbf{p}_1-\mathbf{p}_0)t - \mathbf{p}_E.$$

Cyrus-Beck clipping

The intersection point of the line with the corresponding edge of the clipping rectangle must satisfy

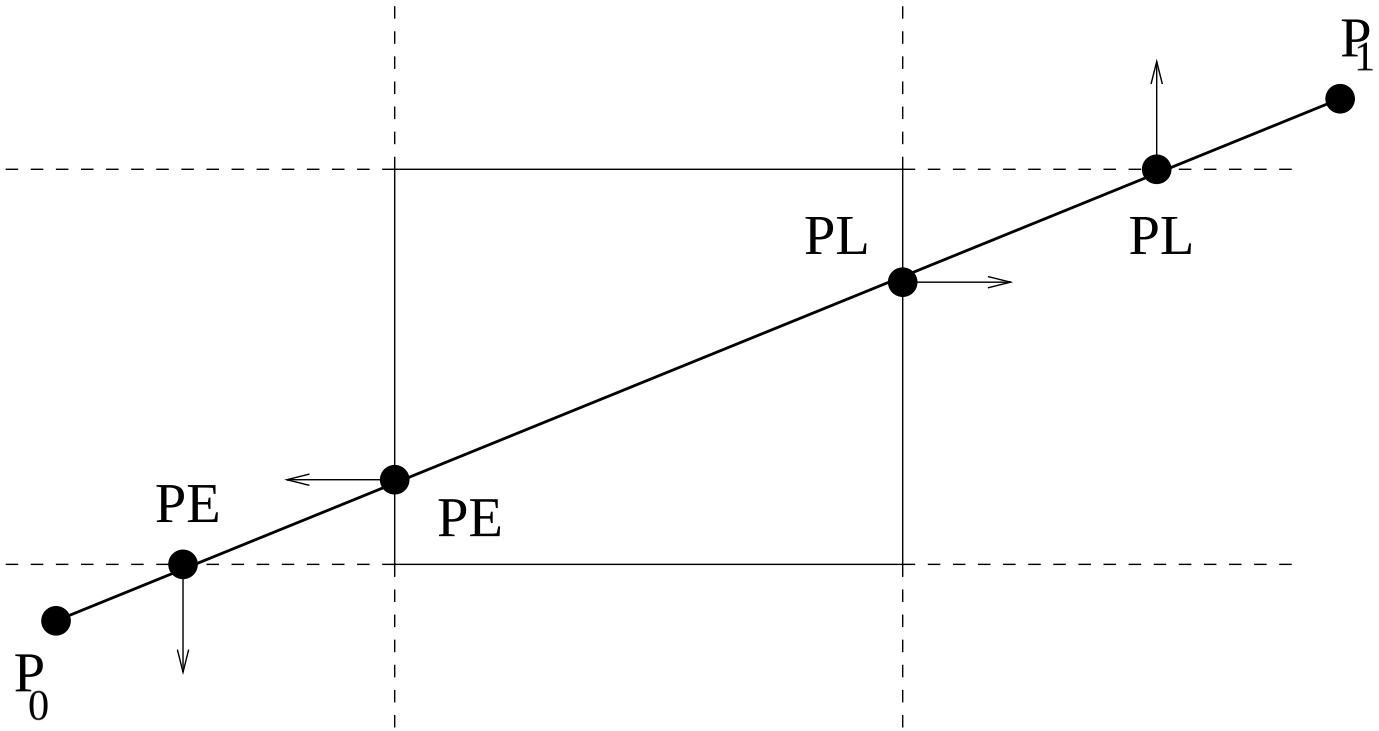
$$0 = \mathbf{n}_E^\top \cdot (\mathbf{p}_0 + (\mathbf{p}_1 - \mathbf{p}_0)t - \mathbf{p}_E) = \mathbf{n}_E^\top \cdot (\mathbf{p}_0 - \mathbf{p}_E) + \mathbf{n}_E^\top \cdot (\mathbf{p}_1 - \mathbf{p}_0)t.$$

$$t = -\frac{\mathbf{n}_E^\top \cdot (\mathbf{p}_0 - \mathbf{p}_E)}{\mathbf{n}_E^\top \cdot (\mathbf{p}_1 - \mathbf{p}_0)}$$

$t \notin [0, 1]$: No intersection point with the edge within the line segment.

Cyrus-Beck clipping

The remaining intersection points ($t \in [0, 1]$) are potential points where the line enters or leaves the clipping rectangle. These points are marked as possible entering (PE) and possible leaving points (PL).



Cyrus-Beck clipping

The angle between the line $\overline{\mathbf{p}_0\mathbf{p}_1}$ and the normal vector $\mathbf{n}$ of the corresponding edge of the clipping rectangle determines whether a point should be marked PE or PL.

- If the angle is larger than $90°$, the intersection point should be marked PE.
- If the angle is less than $90°$, the intersection point should be marked PL.

For this, it is sufficient to determine the sign of the dot product

$$\mathbf{n}^\top \cdot (\mathbf{p}_1 - \mathbf{p}_0).$$

Cyrus-Beck clipping

Determining the part of the line inside the clipping rectangle:

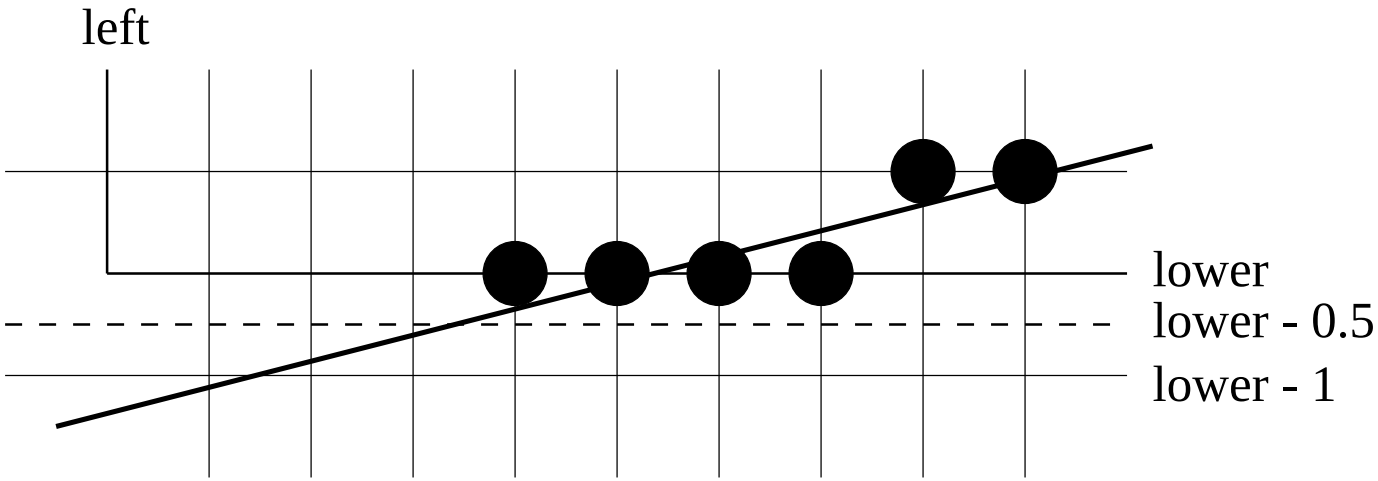
Determine the largest value t_E for a PE point and the smallest value t_L belonging to a PL point.

If $t_E < t_L$ holds, the part of the line between the points $P_0 + (P_1 - P_0)t_E$ and $P_0 + (P_1 - P_0)t_L$ must be drawn.

Otherwise the line lies outside the clipping rectangle.

Line clipping

Problems to determine the correct starting pixel:

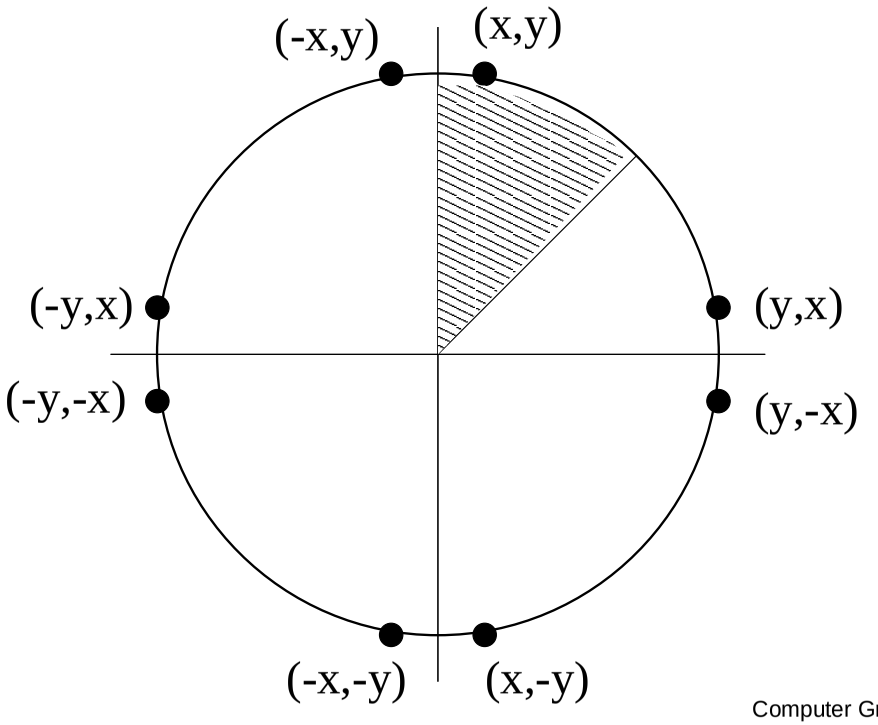


Midpoint algorithm for circles

Draw a circle with radius R around the centre (x_m, y_m).

Given that (x_m, y_m) is a pixel, i.e. $x_m, y_m \in \mathbb{N}$, it is sufficient to find an efficient algorithm for drawing a circle with centre $(0, 0)$ and to use (x_m, y_m) as an offset.

Symmetry of the circle (an eighth of the circle is sufficient):



Midpoint algorithm for circles

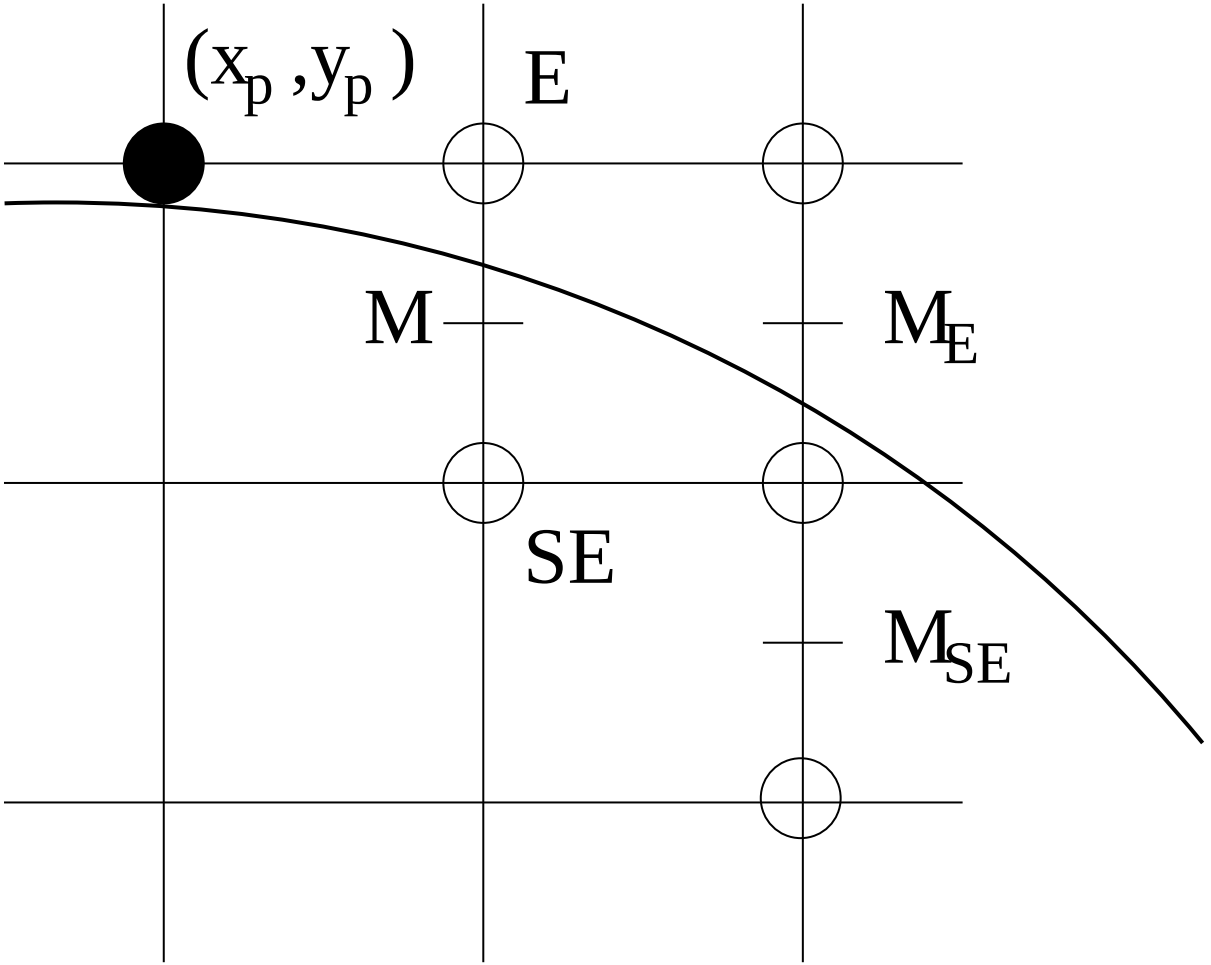
Equation for the circle $x^2 + y^2 = R^2$ in implicit form:

$$d \;=\; F(x,y) \;=\; x^2 + y^2 - R^2.$$

- $F(x,y) = 0 \;\Leftrightarrow\; (x,y)$ lies on the circle.
- $F(x,y) > 0 \;\Leftrightarrow\; (x,y)$ lies outside the circle.
- $F(x,y) < 0 \;\Leftrightarrow\; (x,y)$ lies inside the circle.

If the pixel (x_p, y_p) has been drawn in one step, the next pixel can only be the pixel E with coordinates $(x_p + 1, y_p)$ or SE with coordinates $(x_p + 1, y_p - 1)$.

Midpoint algorithm for circles



Midpoint algorithm for circles

Analogously to lines: Implicit equation for the circle as a decision variable.

Inserting the midpoint M into the equation leads to the following decisions.

- If $d > 0$ holds, then SE must be drawn.
- If $d < 0$ holds, then E must be drawn.

How much does d change in each step?

Midpoint algorithm for circles

Case 1: E, i.e. $(x_{p+1}, y_{p+1}) = (x_p + 1, y_p)$ is the pixel drawn after (x_p, y_p).

Midpoint to be considered for the next pixel (x_{p+2}, y_{p+2}):

$$M_{\text{new}} = \left(x_p + 2, y_p - \frac{1}{2}\right)$$

Midpoint algorithm for circles

$$d_{\text{new}} = F\left(x_p + 2, y_p - \frac{1}{2}\right) = (x_p + 2)^2 + \left(y_p - \frac{1}{2}\right)^2 - R^2$$

$$d_{\text{old}} = F\left(x_p + 1, y_p - \frac{1}{2}\right) = (x_p + 1)^2 + \left(y_p - \frac{1}{2}\right)^2 - R^2$$

$$\Delta_E = d_{\text{new}} - d_{\text{old}} = 2x_p + 3$$

Midpoint algorithm for circles

Case 2: SE, d.h. $(x_{p+1}, y_{p+1}) = (x_p + 1, y_p - 1)$ is the pixel drawn after (x_p, y_p).

Midpoint to be considered for the next pixel (x_{p+2}, y_{p+2}):

$$M_{\text{new}} = \left(x_p + 2, y_p - \frac{3}{2}\right)$$

Midpoint algorithm for circles

$$d_{\text{new}} = F\left(x_p + 2, y_p - \frac{3}{2}\right) = (x_p + 2)^2 + \left(y_p - \frac{3}{2}\right)^2 - R^2$$

$$d_{\text{old}} = F\left(x_p + 1, y_p - \frac{1}{2}\right) = (x_p + 1)^2 + \left(y_p - \frac{1}{2}\right)^2 - R^2$$

$$\Delta_{SE} = d_{\text{new}} - d_{\text{old}} = 2x_p - 2y_p + 5$$

Midpoint algorithm for circles

$$\Delta = \begin{cases} 2x_p + 3 & \text{if } E \text{ was chosen,} \\ 2x_p - 2y_p + 5 & \text{if } SE \text{ was chosen.} \end{cases}$$

i.e.

$$\Delta = \begin{cases} 2x_p + 3 & \text{if } d_{\text{old}} < 0, \\ 2x_p - 2y_p + 5 & \text{if } d_{\text{old}} > 0. \end{cases}$$

Δ is always an integer number. Therefore, the decision variable d can only change by integer values.

Midpoint algorithm for circles

Initialisation of d:

The first pixel to be drawn is $(0, R)$ where $R \in \mathbb{N}$.

First midpoint to be considered: $\left(1, R - \frac{1}{2}\right)$.

$$F\left(1, R - \frac{1}{2}\right) = \frac{5}{4} - R$$

Initialisation of d:

$$d = \frac{5}{4} - R$$

Midpoint algorithm for circles

Summary:

$$d_{\text{init}} = \frac{5}{4} - R$$

$$d_{\text{new}} = d_{\text{old}} + \Delta$$

$$\Delta = \begin{cases} 2x_p + 3 & \text{if } d_{\text{old}} < 0, \\ 2x_p - 2y_p + 5 & \text{if } d_{\text{old}} > 0. \end{cases}$$

Midpoint algorithm for circles

Except for the value $5/4$ at the initialisation, only integer numbers and operations occur.

Replace d by $D = d - \frac{1}{4}$.

Theoretically, instead of testing whether $d > 0$ holds, it would now be necessary to test whether $D > -\frac{1}{4}$ holds.

But since D starts with an integer value and changes only by integer values, it is sufficient to test for $D > 0$.

Midpoint algorithm for circles

$$D_{\text{init}} = 1 - R$$

$$D_{\text{new}} = D_{\text{old}} + \Delta$$

$$\Delta = \begin{cases} 2x_p + 3 & \text{if } D_{\text{old}} < 0, \\ 2x_p - 2y_p + 5 & \text{if } D_{\text{old}} > 0. \end{cases}$$

Midpoint algorithm for circles

A circle with noninteger, rational radius:

$$x^2 + y^2 = \left(\frac{m}{n}\right)^2 \quad \Leftrightarrow \quad n^2x^2 + n^2y^2 = m^2$$

Apply the same strategy as above to the implicit form

$$F(x, y) = n^2x^2 + n^2y^2 - m^2$$

Principle of the midpoint algorithm

- Given: A curve $y = f(x)$ to be drawn in the interval $[x_0, x_1]$.
- Necessary condition: Either $0 \leq f'(x) \leq 1$ or $-1 \leq f'(x) \leq 0$ is valid for the complete interval.
- Write $y = f(x)$ in a suitable implicit form $D = F(x, y) = 0$ and use D as decision variable.
- Compute the change Δ of D depending on the previously drawn pixel (E/NE or E/SE, respectively).

Principle of the midpoint algorithm

- Necessary condition: Δ is an integer value or at least rational. If Δ is a (proper) rational number, consider the decision variable nD instead of D where n is chosen such that $n\Delta$ is always an integer number.
- Compute the initial value D_{init} by inserting the first midpoint (after x_0).
- If D_{init} is not an integer number, the digits after the decimal point can be ignored if D changes only by integer values.

Drawing of arbitrary curves

The midpoint or Bresenham algorithm cannot be applied to arbitrary curves according to the restrictions required for the slope of the function.

In addition, the computational scheme must be calculated individually for each type of function.

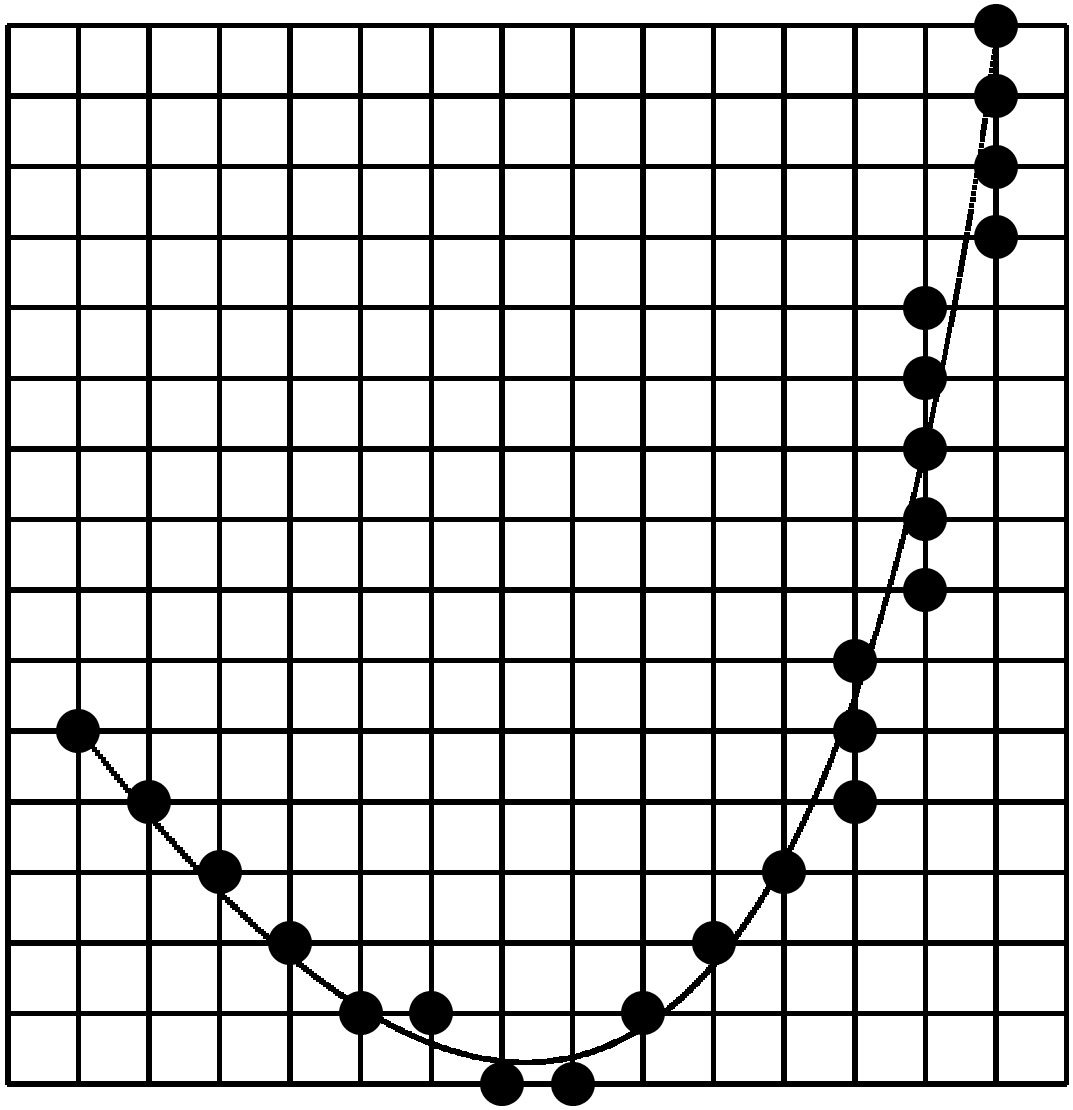
Therefore, arbitrary curves are drawn in the following way.

Given: A curve $y = f(x)$ to be drawn in the interval $[x_0, x_1]$ (where $x_0, x_1 \in \mathbb{Z}$).

Drawing of arbitrary curves

```
int yRound1, yRound2;
yRound1 = round(f(x0));
for (int x=x0; x<x1; x++)
{
  yRound2 = round(f(x+1));
  drawLine(x,yRound1,x+1,yRound2);
  yRound1 = yRound2;
}
```

Drawing of arbitrary curves



Antialiasing

Aliasing effects occur when a continuous signal is sampled in a discrete manner with a constant rate.

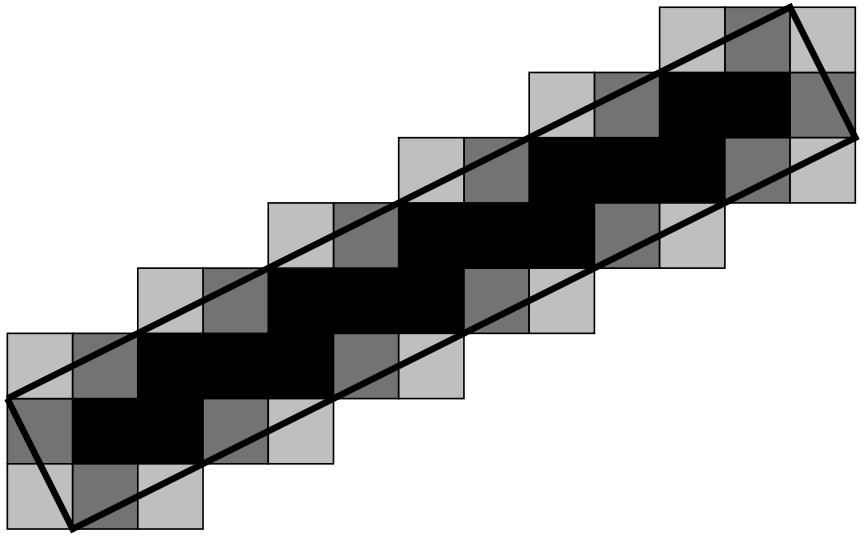
Drawing lines and curves (sampling of a continuous curve by a discrete pixel raster) can cause aliasing effects (jaggies, staircasing).

Antialiasing tries to amend these effects by the use of different grey levels or colour intensities.

Pixels are not simply drawn black or white, but with different intensities, depending how far they lie from the ideal line to be drawn or how much they are covered by the thickened line.

Unweighted Area Sampling

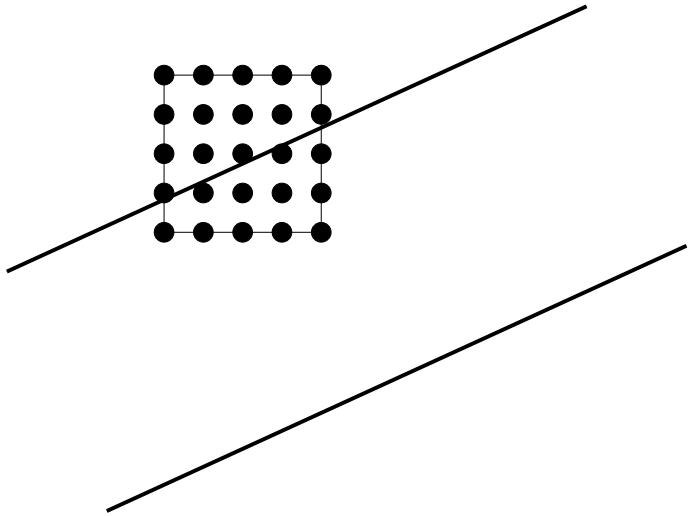
- A line is considered as (narrow, long) rectangle.
- Each pixel is associated with a small square. The intensity of the pixel is chosen proportionally to the area of the pixel's square that is covered by the rectangle that represents the line.



Unweighted Area Sampling

Simple heuristic strategy to estimate the proportion of the pixel's square that is covered by the line rectangle:

The pixel's square is covered by an imaginary refined pixel grid. The proportion of refined pixels in the square that also lie in the line rectangle gives a good estimation for the desired intensity of the pixel.

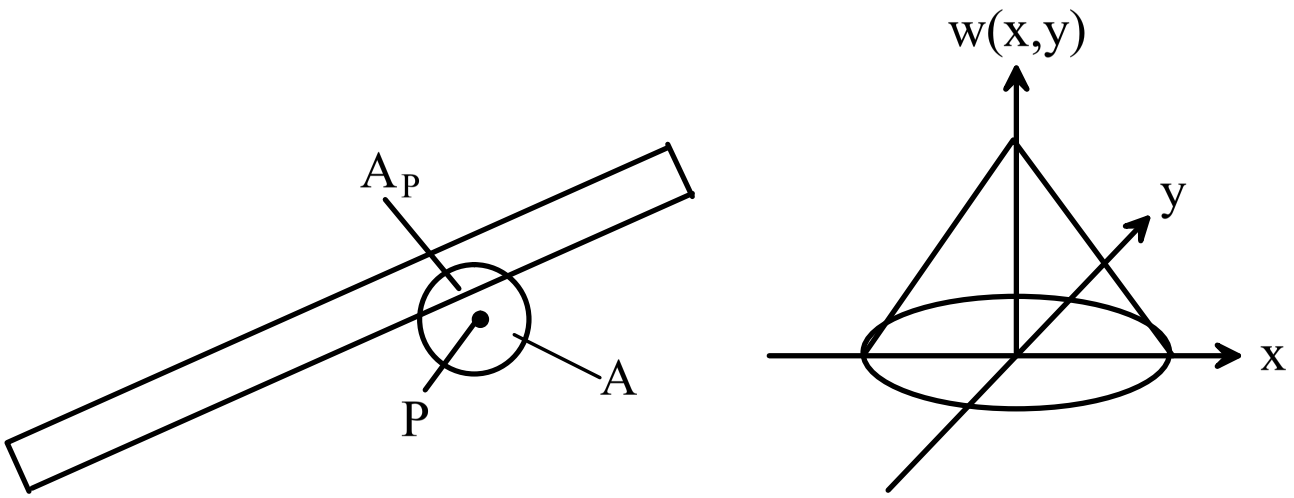


Intensity: 11/25

Weighted Area Sampling

Weighted Area Sampling: Use a weighting function $w(x, y)$:

$$\text{Intensity for pixel P} = \frac{\int_{A_P} w(x,y)dx\,dy}{\int_A w(x,y)dx\,dy}$$



Gupta-Sproull antialiasing

For suitable weighting functions, the a pixel's intensity depends only on its distance the line.

The number of displayable intensity values is usually limited, for computer screens by 256.

Antialiasing normally uses much less intensity vales.

Scanning the pixels in a suitable order, computing the discretised intensities is similar to drawing a line on a pixel raster.

- Scanning the pixels corresponds to stepping through the x-values for drawing the line.
- Determining the (discretised/rounded) intensity corresponds to computing the rounded y-value for drawing the line.

Gupta-Sproull antialiasing

The Gupta-Sproull antialiasing algorithm uses a suitable weighting function and limited number of intensity levels, so that the midpoint algorithm can be used for computing the intensity for antialiasing.

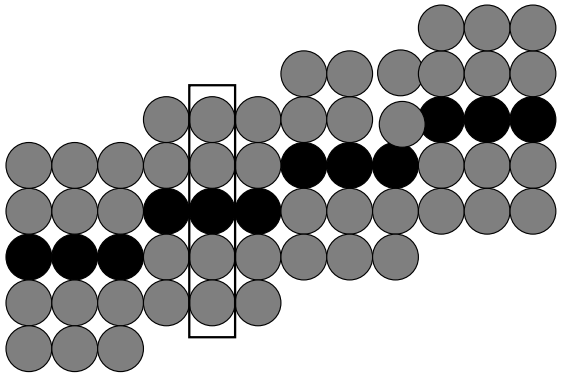
No integrals need to be solved, not even floating arithmetic is required.

Antialiasing in Java 2D:

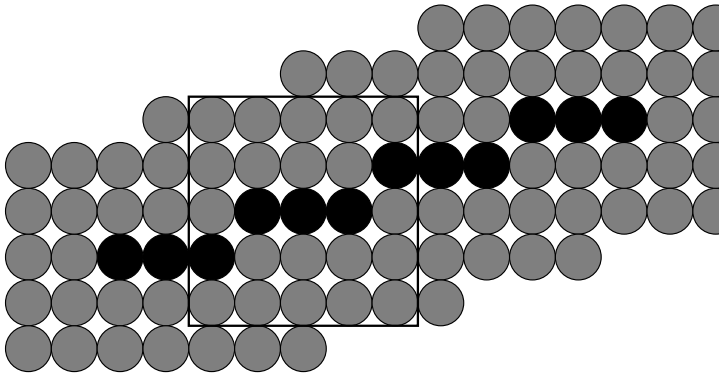
```
g2d.setRenderingHint(
        RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_ON);
```

Drawing thick lines

For devices even with today's standard resolution, lines rendered with only one pixel as their width occur extremely thin.



Pixel replication



moving pen technique

Alternative: Consider lines as filled polygons or rectangles.

Thick polylines

How should line endings of thick lines look like?

Java 2D: Drawing thick lines

```
new BasicStroke(thickness,ending,join);
```

Values for `ending`:

- `BasicStroke.CAP_BUTT`: The endings are cut off straight, orthogonal to the direction of the line.
- `BasicStroke.CAP_ROUND`: A half circle is attached to each line ending.
- `BasicStroke.CAP_SQUARE`: A rectangle is appended to the line ending, so that the line is prolongated by half of its thickness.

Java 2D: Drawing thick lines

Values for `join`:

- `BasicStroke.JOIN_MITER`: The outer edges of the line rectangles are prolonged until they meet, leading to a join with a sharp tip.
  For acute angles of the lines, the tip can be extremely long. To avoid this effect, another `float`-value can be specified, defining the maximum length of the tip. If the tip exceeds this length, then the following join mode is used.

Java 2D: Drawing thick lines

- `BasicStroke.JOIN_BEVEL`: The join is a straight cut-off, orthogonal to the middle line between the two lines to be connected.
- `BasicStroke.JOIN_ROUND`: The line endings are cut off at the join and a circle segment similar to the style `BasicStroke.JOIN_BEVEL` for line endings is attached to the join. The angle of the circle segment is chosen such that the lines form the tangents at the circle segment.