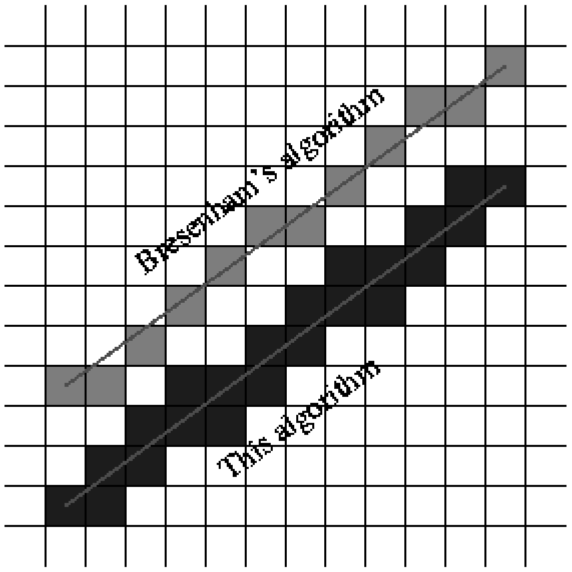
# Bresenham-based supercover line algorithm

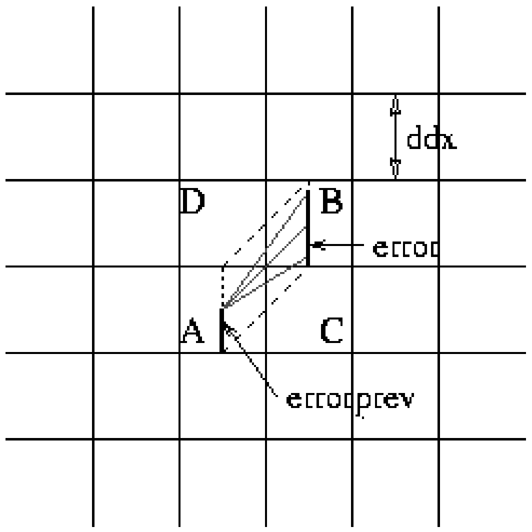
This page presents a modified version of the Bresenham's algorithm, which prints ALL the points (not only one point per axis) the ideal line contains. This line is called **supercover line** (according to Eric Andres) and this algorithm might be a particular case of DDA (**Discrete Differential Analyzer**) algorithm (according to Srikanth). It may be useful for example when you have to know if an obstacle exists between two points (in which case the points do not see each other). Figure below shows the difference between the Bresenham's algorithm (which draws a classical line) and this one (which draws a supercover line).



## Other links

Links which might be of interest are:
ligwww.epfl.ch/~srik/thesis (choose appendix_a.ps) - another algorithm (and a generalized one in 3D also) for doing the same thing. Thanks, Srikanth (address changed, but where is it now?)
Bresenham's algorithm
Linear interpolation algorithm using fixed points (allowing generally a more efficient implementation) - (thanks Alex J. Champandard for rectification)
Algorithm for arbitrary points
Grid Traversal algorithm

## Brief explanation



I suppose you already know the Bresenham's algorithm. If not, see the links above. To simplify the discussion, suppose we are in the first octant. Suppose the point is in square A. The next point (using Bresenham's algorithm) will be B or C, like in our algorithm. However, while Bresenham's checks B and C, our algorithm has to check the points B, C, and D.

If the Bresenham's algorithm does not change the y-coordinate (next point is C), this means that D will not be drawn, so we pass directly to C, and we go to the beginning again.
The other case is when Bresenham's algorithm changes the coordinate, i.e. when the next point is B. In this case, both C and D have also to be checked. As seen in the figure, we can know if a point is drawn or not by the following relation:

```
    // three cases (octant == right->right-top for directions below):
    if (error + errorprev < ddx)  // bottom square also
      POINT (y-ystep, x);
    else if (error + errorprev > ddx)  // left square also
      POINT (y, x-xstep);
    else{  // corner: bottom and left squares also
      POINT (y-ystep, x);
      POINT (y, x-xstep);
    }
```

error is the current error (in point B), while errorprev is the previous error (in point A). Remember that the error is the "distance" (non-normalized) from the ideal point to the grid line below the ideal point.

## Implementation of the algorithm

Here is the algorithm (the instructions added to the Bresenham's algorithm are in **bold**):

```
// use Bresenham-like algorithm to print a line from (y1,x1) to (y2,x2)
// The difference with Bresenham is that ALL the points of the line are
//   printed, not only one per x coordinate.
// Principles of the Bresenham's algorithm (heavily modified) were taken from:
//   http://www.intranet.ca/~sshah/waste/art7.html
void useVisionLine (int y1, int x1, int y2, int x2)
{
  int i;               // loop counter
  int ystep, xstep;    // the step on y and x axis
  int error;           // the error accumulated during the increment
  int errorprev;       // *vision the previous value of the error variable
  int y = y1, x = x1;  // the line points
int ddy, ddx;        // compulsory variables: the double values of dy and dx
int dx = x2 - x1;
int dy = y2 - y1;
POINT (y1, x1);  // first point
// NB the last point can't be here, because of its previous point (which has to be verified)
if (dy < 0){
  ystep = -1;
  dy = -dy;
}else
  ystep = 1;
if (dx < 0){
  xstep = -1;
  dx = -dx;
}else
  xstep = 1;
ddy = 2 * dy;  // work with double values for full precision
ddx = 2 * dx;
if (ddx >= ddy){  // first octant (0 <= slope <= 1)
  // compulsory initialization (even for errorprev, needed when dx==dy)
  errorprev = error = dx;  // start in the middle of the square
  for (i=0 ; i < dx ; i++){  // do not use the first point (already done)
    x += xstep;
    error += ddy;
    if (error > ddx){  // increment y if AFTER the middle ( > )
      y += ystep;
      error -= ddx;
      // three cases (octant == right->right-top for directions below):
      if (error + errorprev < ddx)  // bottom square also
        POINT (y-ystep, x);
      else if (error + errorprev > ddx)  // left square also
        POINT (y, x-xstep);
      else{  // corner: bottom and left squares also
        POINT (y-ystep, x);
        POINT (y, x-xstep);
      }
    }
    POINT (y, x);
    errorprev = error;
  }
}else{  // the same as above
  errorprev = error = dy;
  for (i=0 ; i < dy ; i++){
    y += ystep;
    error += ddx;
    if (error > ddy){
      x += xstep;
      error -= ddy;
      if (error + errorprev < ddy)
        POINT (y, x-xstep);
      else if (error + errorprev > ddy)
        POINT (y-ystep, x);
      else{
        POINT (y, x-xstep);
        POINT (y-ystep, x);
        }
      }
      POINT (y, x);
      errorprev = error;
    }
  }
  // assert ((y == y2) && (x == x2));  // the last point (y2,x2) has to be the same with the last point of the
algorithm
}
```

Here we have supposed that if the line passes through a corner, the both squares are drawn.  If you want to remove this, you can simply remove the *else* part dealing with the corner.

Note: The line is symmetric, i.e. a line from x0,y0 to x1,y1 is the same as a line from x1,y1 to x0,y0.

Written by Eugen Dedu
Last modified: June 05, 2001