# Bresenham’s algorithm

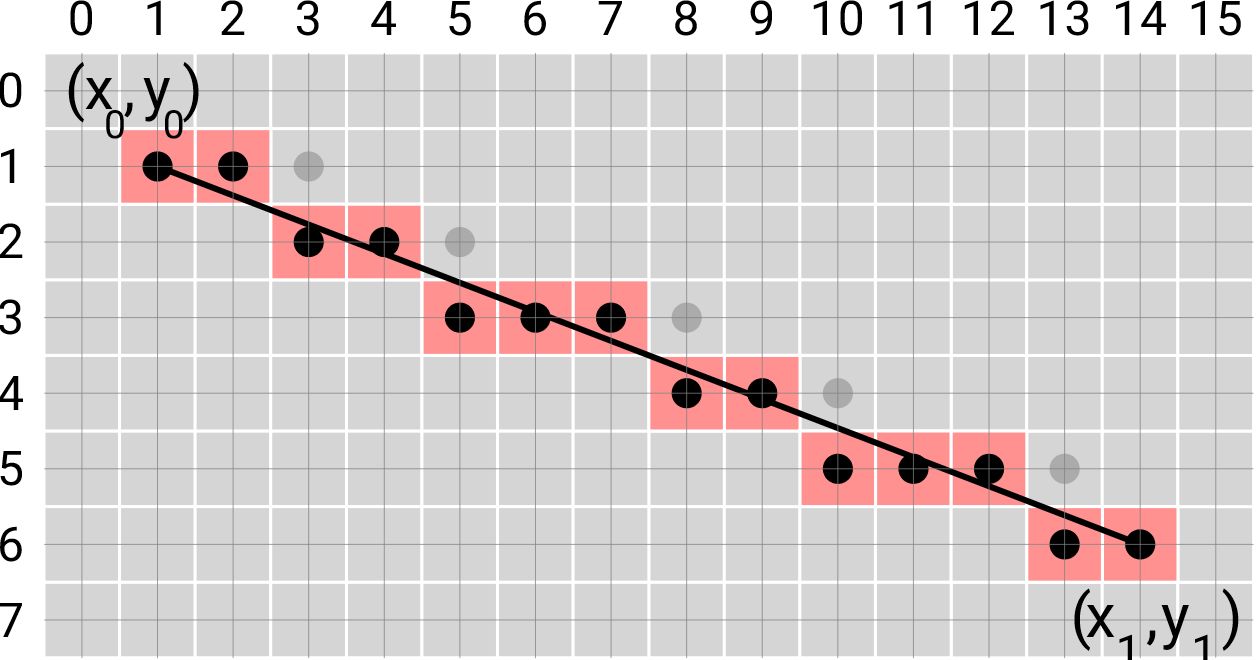
In this article, the theory behind Bresenham’s algorithm is explained and translated to an implementation in the c language. Much of this text is derived from the great article on <http://kt8216.unixcab.org/murphy/index.html>

In this document, I’m trying to explain the theory but transfer it to usable c code as quickly as possible.

## Principle, valid for the first octant (dx>0, dy≥0, dx≥dy)

In the coordinate system below, the upper left pixel represents coordinate (0,0), as in most computer screens. The coordinates refer to the centres of pixels.

Suppose we would want to draw a line from (x0,y0) to (x1,y1):



The formula for the exact line is:

y=y0+(x-x0)\*(dy/dx)

Suppose a pixel is drawn at pixel (xp,yp), which may not be exactly on the line. The error in vertical direction between the pixel and the line is then:

error=yp-(y0+(xp-x0)\*dy/dx)

Initially, at point (x0,y0), the error is 0. If the x-coordinate moves to the right, but the y-coordinate does not, the error can also be increased at every pixel moving to the right:

error=error+dy/dx;

If the error exceeds 0.5 pixels, it is better to increase yp, so the error is reduced:

void DrawLineFirstOctant(x0,y0,x1,y1) {

// Valid for first octant: dx>=0, dy>=0, dx>=dy

int xp=x0;

int yp=y0;

int dx=x1-x0;

int dy=y1-y0;

float error=0;

while(x1>xp) {

pixel(xp,yp);

xp+=1;

error=error+dy/dx;

if (error>0.5) {

yp=yp+1;

error=error-1;

}

}

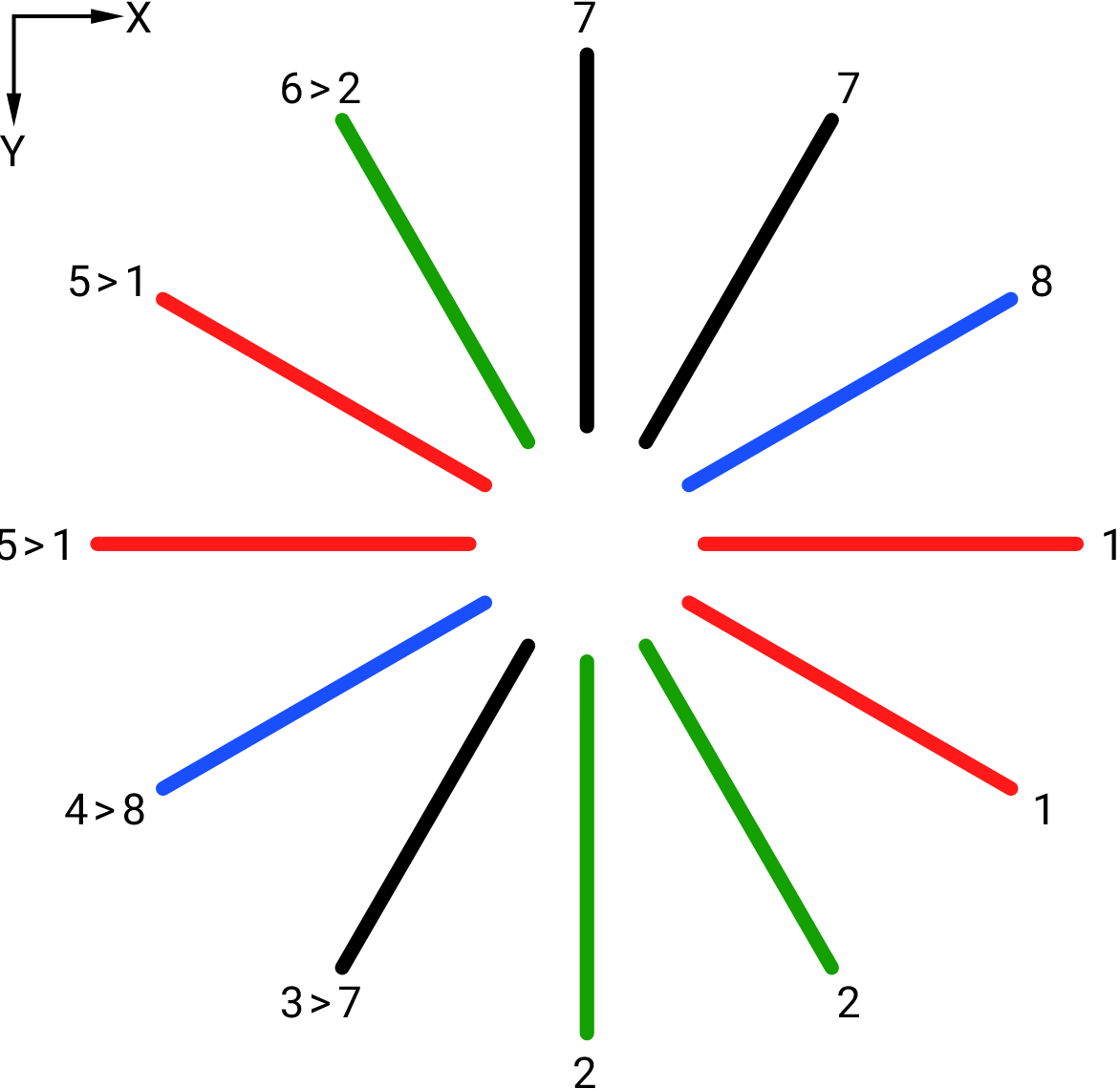
}

The error is now represented as a floating point variable, and the function therefore also uses floating point operations, which slows down the routine. Floating point operations can be mitigated by multiplying all error operations by 2\*dx:

|  |  |
| --- | --- |
| Original routine, including floats:  void DrawLineFirstOctant(x0,y0,x1,y1) {  // Valid for first octant:  // dx>0, dy>=0, dx>=dy  int xp=x0;  int yp=y0;  int dx=x1-x0;  int dy=y1-y0;  **float error=0;**  while(x1>=xp) {  pixel(xp,yp);  xp+=1;  error=error+dy/dx;  if (error>0.5) {  yp=yp+1;  error=error-1;  }  }  } | Multiplying the error operations by 2\*dx:  void DrawLineFirstOctant(x0,y0,x1,y1) {  // Valid for first octant:  // dx>0, dy>=0, dx>=dy  int xp=x0;  int yp=y0;  int dx=x1-x0;  int dy=y1-y0;  **int error=0;**  while(x1>=xp) {  pixel(xp,yp);  xp+=1;  **error=error+2\*dy;**  **if (error>dx) {**  yp=yp+1;  **error=error-2\*dx;**  }  }  } |
|  |  |

The routine on the right hand side is known as ‘Bresenhams algorithm’, developed in 1962 by IBM researcher Jack Bresenham. It is very fast and effective and is still used today in many graphics applications.

This can also be solved for the other octants.



Lines in the left hand quadrants (x1<x0) in octants 3 to 6 can be converted to their respective opponent by swapping start- and endpoints.

For the remaining octants, the decision tree is:

|  |  |  |
| --- | --- | --- |
| y0<=y1 | x1-x0>y1-y0 | Octant 1 |
| x1-x0<=y1-y0 | Octant 2 |
| y0>y1 | x1-x0<=y1-y0 | Octant 7 |
| x1-x0>y1-y0 | Octant 8 |

This can be implemented as a single function as follows:

void Bresenham(int16\_t x0,int16\_t y0,int16\_t x1,int16\_t y1, uint16\_t color) {

int16\_t xp=x0;

int16\_t yp=y0;

int16\_t error=0;

int16\_t dx, dy;

if (x0>x1) {

// Reverse the begin- and endpoint to get in the right octant

Bresenham(x1,y1,x0,y0,color);

}

else {

// Solve for the remaining 4 octants

dx=x1-x0;

if (y0<=y1) {

dy=y1-y0;

// Octant 1 and 2

if ( dx>=dy ) {

// Octant 1: x0<=x1, y0<=y1, dx>=dy

while(x1>xp) {

pixel(xp, yp, color);

xp+=1;

error=error+2\*dy;

if (error>dx) {

yp=yp+1;

error=error-2\*dx;

}

}

}

else {

// Octant 2: x0<=x1, y0<=y1, dx<dy

while(y1>yp) {

pixel(xp, yp, color);

yp+=1;

error=error+2\*dx;

if (error>dy) {

xp=xp+1;

error=error-2\*dy;

}

}

}

}

else {

dy=y0-y1;

if (dx>dy) {

// Octant 8: x0<=x1, y0>y1, dx>dy

while(x1>xp) {

pixel(xp, yp, color);

xp+=1;

error=error+2\*dy;

if (error>dx) {

yp=yp-1;

error=error-2\*dx;

}

}

}

else {

// Octant 7: x0<=x1, y0>y1, dx<=dy

while(y1<yp) {

pixel(xp, yp, color);

yp-=1;

error=error+2\*dx;

if (error>dy) {

xp=xp+1;

error=error-2\*dy;

}

}

}

}

}

}

This function efficiently draws a single pixel straight line between points.

Bresenham’s algorithm has many derived algorithms (many of which were not derived by Bresenham) such as algorithms that draw:

* circles,
* cubic and quadratic bezier curves,
* lines with arbitrary thickness,
* native anti-aliased versions of those.

# Lines with arbitrary thickness

In 1978, another IBM researcher, A.S. Murphy, published an article titled “[Line Thickening by Modification to Bresenham's Algorithm](http://homepages.enterprise.net/murphy/thickline/index.html)”.