Drawing a line segment

Line segments are basic graphics primitives. To efficiently display good-quality line segments is a fundamental problem in real-time computer graphics. Three methods for drawing a line segment will be discussed in this lesson, leading to Bresenham's algorithm which uses on average one integer addition per pixel to rasterize a line segment.

Input: starting point (xs, ys) and ending point (xe, ye), where xs, ys, xe, ye are integers.

Assumptions:

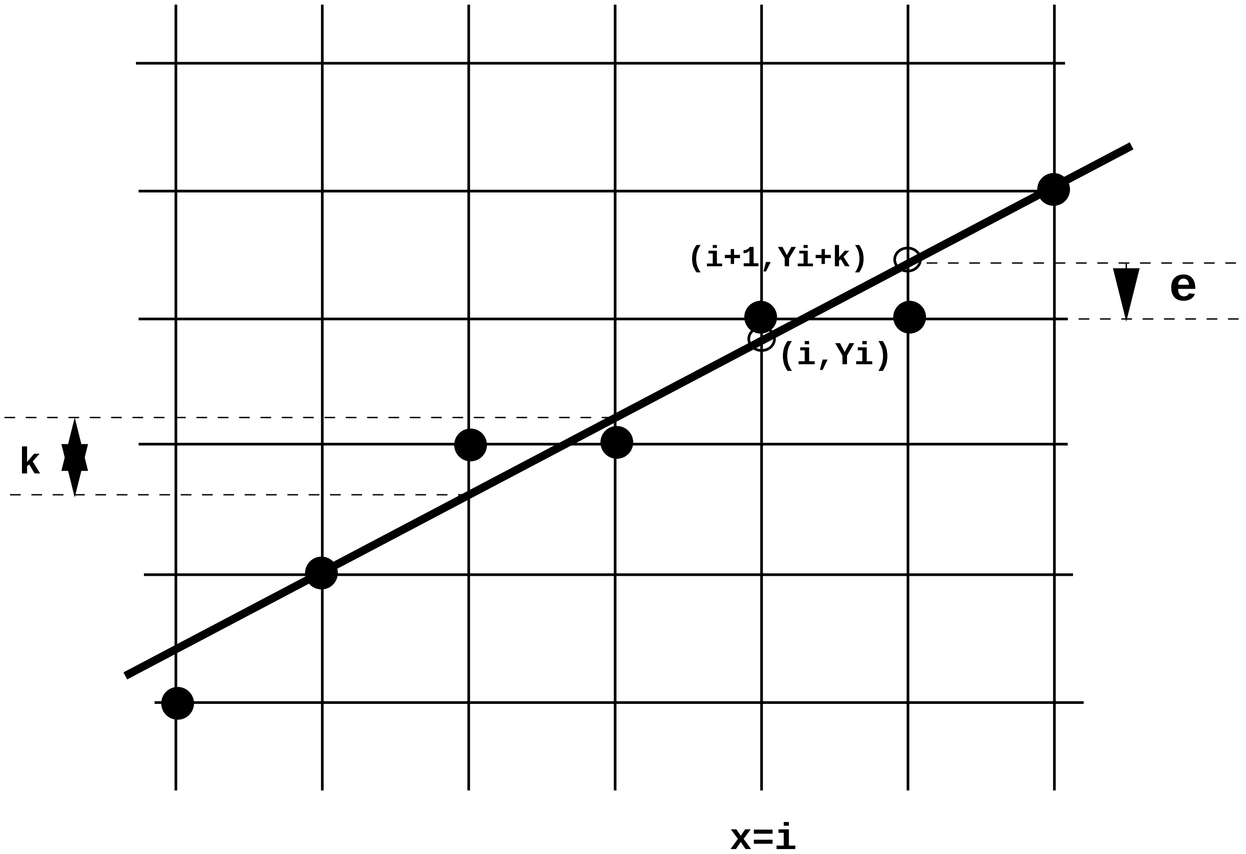
• $ye \geq ys$, $xe > xs$, and $|ye - ys| \leq |xe - xs|$. So $0 \leq k = (ye - ys)/(xe - xs) \leq 1$, k the slope. Note that any other line segment can be transformed to such a position by properly choosing the starting point or swapping x and y coordinates, if necessary.

• One pixel is to be found on each vertical line intersecting the given line segment.

• A sequence of pixels will be determined to approximate the line segment.

Method 1:

For each unit increment in x-direction, y is increased by k, the slope. If the intersection between the vertical line $x = i$ and the given line segmeny is (i, y_i), the intersection between the next vertical line $x = i + 1$ and the given line is $(i + 1, y_i + k)$. See the figure.



Raster Line Drawing

Note that, since pixel positions are needed, we must round (i, y_i), $xs \le i \le xe$, to the nearest integer point $(i, \lfloor y_i + 0.5 \rfloor)$. The pseudo code is as follows.

```
Line Drawing 1:
long x, y;
float k, yy;
        k = (ye - ys)/(xe - xs);
        yy = ys;
        for(x=xs; x<=xe; x++)
        {
                y = ftrunc(yy + 0.5);
```

```
                write_pixel(x,y);
                yy = yy + k;
        }
```

Remarks: Floating-point operations are used in this solution. Floating point operations are slower than integer operations.

Method 2:

Idea: Suppose that the distance e of (i, y_i) to the horizontal grid line right below it is recorded. Then the lower pixel should be chosen if and only if $e < 0.5$. To facilitate this test, we denote $e - 0.5$ by e instead. Thus, the lower pixel is chosen if and only if $e < 0$. Pay attention to how e is updated in each step.

```
long x, y;
float k, e;
        k = (ye - ys)/(xe - xs);
        x = xs; y = ys;
        e = -0.5;
        for(x=xs; x<=xe; x++)
        {
        if(e < 0)
                write_pixel(x,y);
        else
        {
                y = y + 1;
```

```
                write_pixel(x,y);
                e = e - 1;
        }
        e = e + k;
}
```

Remark: Floating-point operations are still used in method 2.

Method 3:

Idea: We use *program transformation* to translate method 2 into a new algorithm. The key observation is: it is the sign of e, not its value, that determines the next pixel to be selected.

Let $a = xe - xs$, $b = ye - ys$. Then $k = b/a$. All the statements in method 2 that affect the value of e are

$$e = -0.5; \quad e = e - 1; \quad e = e + \frac{b}{a}.$$

Multiplying $2a$ to both sides of these three expressions, we obtain

$$2a * e = -a; \quad 2a * e = 2a * e - 2a; \quad 2a * e = 2a * e + 2b;$$

Naming $2a * e$ by d yields

$$d = -a; \quad d = d - 2a; \quad d = d + 2b.$$

Using these three expressions to replace the original statements that are used to generate e in method 2 yields the following pseudo code.

```
long x, y, dx, dy, d;
    x = xs; y = ys;
```

```
    dx = 2*(xe - xs); /* dx = 2a */
    dy = 2*(ye - ys); /* dy = 2b */
    d = -(xe - xs); /* d = -a */
    for(x=xs; x<=xe; x++)
    {
        if(d < 0)
              write(x, y);
        else
        {
              y = y + 1;
              write_pixel(x,y);
              d = d - dx;
        }

     d = d + dy;
}
```

Remarks: This algorithm uses integer operations only. It can be further simplified by re-arranging some statements.

Bresenham's algorithm

By combining the statement `d = d - dx;` and `d = d + dy;` in the case of moving up diagonally, we have the final algorithm, which on average uses one integer addition and one sign testing per pixel.

```
long x, y, dx, dy, dy_x, d;
```

```
    x = xs; y = ys;

    dx = 2*(xe - xs); /* dx = 2a */

    dy = 2*(ye - ys); /* dy = 2b */

    dy_x = dy - dx;

    d = -(xe - xs); /* d = -a */

    for(x=xs; x<=xe; x++)

    {

        if(d < 0)

                d = d + dy;

        else

        {

                y = y + 1;

                d = d + dy_x;

        }


        write_pixel(x,y);

}
```

Questions:

Can you come up with a line drawing algorithm that is faster than Bresenham's algorithm?