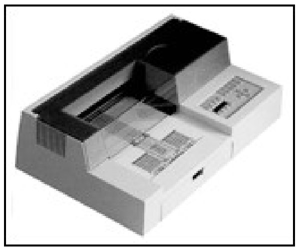
Line-Drawing Algorithms

Line drawing is our first adventure into the area of *scan conversion*. The need for scan conversion, or *rasterization*, techniques is a direct result of scanning nature of raster displays (thus the names).

Vector displays are particularly well suited for the display of lines. All that is needed on a vector display to generate a line is to supply the appropriate control voltages to the x and y deflection circuitry, and the electron beam would traverse the line illuminating the desired segment. The only inaccuracies in the lines drawn a vector display resulted from various non-linearities, such as quantization and amplifier saturation, and the various noise sources in the display circuitry.



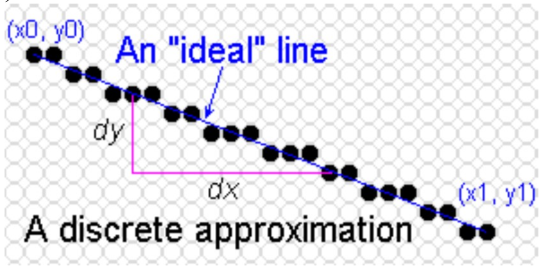
When raster displays came along the process of drawing lines became more difficult. Luckily, raster display pioneers could benefit from previous work done in the area of **digital plotter algorithms**. A pen-plotter is a hardcopy device used primarily to display engineering line drawings. Digital plotters, like raster displays, are *discretely addressable devices*, where position of the pen on a plotter is controlled by special motors called *stepper motors* that are connected to mechanical linkages that translates the motor's rotation into a linear translation. Stepper motors can precisely turn a fraction of a rotation (for example 2 degrees) when the proper controlling voltages are applied. A typical flat-bed plotter uses two of these motors, one for the x-axis and a second for the y-axis, to control the position of a pen over a sheet of paper. A selenoid is used to raise and lower the actual pen whendrawing and postioning.

The bottom line is that most of the popular line-drawing algorithms used to on computer screens (and laser and ink-jet printers for that matter) were originally developed for use on pen-plotters. Furthermore, most of this work is attributed by a single man named **Jack Bresenham** who was an IBM employee. By the way, he worked right here, in the Research Triangle Park, for a while and I am told he still maintains a residence here, but he is currently a professor at Winthrop University.

In this lecture I will gradually evolve from the basics of algebra to the famous Bresenham line-drawing algorithim (along the same lines as a famous paper by Bob Sproull), and then I'll discuss some developments that have happened since then.

Quest for the *Ideal Line*

Note that since vector-graphics displays, capable of drawing nearly perfect lines, predated raster-graphics displays, the expectations for line quality were set very high. The nature of raster-graphics display, however, only allows us to display a discrete approximation of a line, since we are restricted to only turn on discrete points, or pixels. In order to discuss, line drawing we must first consider the mathematically ideal line (or line segment).



From geometry we know that a line, or line segment, can be uniquely specified by two points. From algebra we also know that a line can be specified by a slope, usually given the name **m** and a y-axis intercept called **b**. Generally in computer graphics, a line will be specified by two endpoints. But the slope and y-intercept are often calculated as intermediate results for use by most line-drawing algorithms.

The goal of any line drawing algorithm is to construct the best possible approximation of an ideal line given the inherent limitations of a raster display. Following is a list of some of line qualities that are often considered.

- Continuous appearence
- Uniform thickness and brightness
- Are the pixels nearest the ideal line turned on
- How fast is the line generated

The first line-drawing algorithm presented is called the simple *slope-intercept algorithm*. It is a striaght forward implementation of the slope-intercept formula for a line.

```
public void lineSimple(int x0, int y0, int x1, int y1, Color color)
{
    int pix = color.getRGB();
    int dx = x1 - x0;
    int dy = y1 - y0;

    raster.setPixel(pix, x0, y0);
    if (dx != 0) {
        float m = (float) dy / (float) dx;
        float b = y0 - m*x0;
        dx = (x1 > x0) ? 1 : -1;
        while (x0 != x1) {
            x0 += dx;
            y0 = Math.round(m*x0 + b);
            raster.setPixel(pix, x0, y0);
        }
    }
}
```

The java applet above demonstrates the *lineSimple()* method. Draw a line by clicking and dragging on the pixel grid shown with the left mouse button. An *ideal* line is displayed until the left button is released. Upon release a discrete approximation of the line is drawn on the display grid using the *lineSimple()* method shown above. An *ideal* line is then overlaid for comparison.

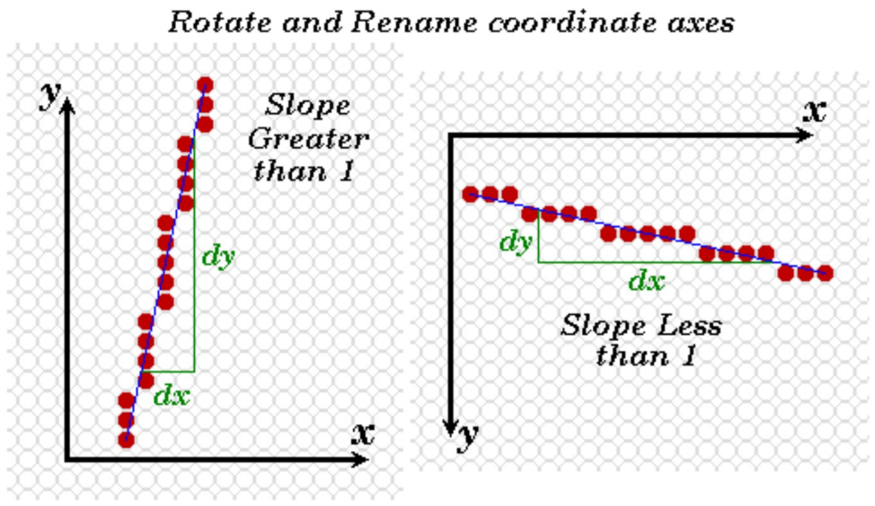
This algorithm works well when the desired line's slope is less than 1, but the lines become more and more **discontinuous** as the slope increases beyond one. Since this algorithm iterates over values of x between x0 and x1 there is a pixel drawn in each column. When the slope is greater than one then often more than one pixel must be drawn in each column for the line to appear continuous.

The progression of line drawing algorithms that we will cover is basically a case study in optimization techniques; at least that's how they will be treated here. The optimization methodology that we will cover is very general and can be applied to a wide range of algorithms.

Step 1. Make it Work!

Seldom does it make sense to begin optimizing an algorithm before satisfies its design requirements! Code is usually developed under considerable time pressures. If you start optimizing before you have a working prototype you have no fall back position.

Our simple line drawing algorithm does not provide satisfactory results for line slopes greater than 1. The solution: **symmetry**. The assigning of one coordinate axis the name *x* and the other *y* was an arbitrary choice. Notice that line slopes of greater than one under one assignment result in slopes less than one when the names are reversed.



We can modify the *lineSimple( )* routine to exploit this symmetry as follows:

```
public void lineImproved(int x0, int y0, int x1, int y1, Color color)
{
    int pix = color.getRGB();
    int dx = x1 - x0;
    int dy = y1 - y0;

    raster.setPixel(pix, x0, y0);
    if (Math.abs(dx) > Math.abs(dy)) {          // slope < 1
        float m = (float) dy / (float) dx;      // compute slope
        float b = y0 - m*x0;
        dx = (dx < 0) ? -1 : 1;
        while (x0 != x1) {
            x0 += dx;
            raster.setPixel(pix, x0, Math.round(m*x0 + b));
        }
    } else
    if (dy != 0) {                              // slope >= 1
        float m = (float) dx / (float) dy;      // compute slope
        float b = x0 - m*y0;
        dy = (dy < 0) ? -1 : 1;
        while (y0 != y1) {
            y0 += dy;
            raster.setPixel(pix, Math.round(m*y0 + b), y0);
        }
    }
}
```

The java applet above demonstrates the *lineImproved()* algorithm. Draw a line by clicking and dragging on the pixel grid shown using the left mouse button. An *ideal* line is displayed until the left button is released. Upon release a discrete approximation of the line is drawn on the display grid using the *lineImproved( )* method shown above. An *ideal* line is then overlaid for comparison.

Notice that the slope-intercept equation of the line is executed at each step of the inner loop.

Step 2. Optimize Inner Loops

The most important code fragments to consider when optimizing are those where the algorithm spends most of its time. Often these areas are within loops.

Our improved algorithm can be partitioned into two major sections. The first section is basically set-up code that is used by the second section, the central pixel-drawing while-loop. Most of the pixels are drawn in this while loop.

First optimization: **remove unnecessary method invocations**. Notice the call to the round class-method of the Math object that is executed for each pixel drawn. You might think that proper rounding is so trival a task that it hardly warrants a special method to begin with. But, it is complicated when differences between postive and negative numbers, and proper handling of fractions with a value of exactly 0.5 are considered. However, in our line drawing code we are either not going to run into these cases (i.e. we only consider positive coordinate values) or we don't care (i.e. we don't expect to see values of 0.5 very often, and when we do we'd like to see them handled consistently). So for our purposes the call to *Math.round(m*x0 + b)* could be replaced with the following *(int)(m*y0 + b + 0.5)*.

Second optimization: **use incremental calculations**. *Incremental or iterative function calculations use previous function values to compute future values.* Often expressions within loops depend on a value that is being incremented or decremented on each loop iteration. In these cases the actual function values might be more efficiently calculated by first computing an initial value of the function during the loop set up, and updating the subsequent function values using a discrete version of a differential equation called a *difference equation*.

Consider the expression,

*m*x0 + b + 0.5*,

that is computed inside the first loop. The We can initialize the first value of the function outside of the loop:

*y[0] = m*x0 + b + 0.5*

. Future values of y depend only on how the value of x changes within the loop, since all other values are constant. Thus subsequent values of y can be computed by one of the following equations:

y[i+1] = y[i] + m; // if x0 is incremented

or

y[i+1] = y[i] - m; // if x0 is decremented

We can use a single equation if we change the sign of *m*, based on whether we are incrementing or decrementing, outside of the loop.

Incorporating these changes into our previous *lineImproved( )* algorithm gives the following result:

```
public void lineDDA(int x0, int y0, int x1, int y1, Color color)
{
    int pix = color.getRGB();
    int dy = y1 - y0;
    int dx = x1 - x0;
    float t = (float) 0.5;                      // offset for rounding

    raster.setPixel(pix, x0, y0);
    if (Math.abs(dx) > Math.abs(dy)) {          // slope < 1
        float m = (float) dy / (float) dx;      // compute slope
        t += y0;
        dx = (dx < 0) ? -1 : 1;
        m *= dx;
        while (x0 != x1) {
            x0 += dx;                           // step to next x value
            t += m;                             // add slope to y value
            raster.setPixel(pix, x0, (int) t);
        }
    } else {                                    // slope >= 1
        float m = (float) dx / (float) dy;      // compute slope
        t += x0;
        dy = (dy < 0) ? -1 : 1;
        m *= dy;
        while (y0 != y1) {
            y0 += dy;                           // step to next y value
            t += m;                             // add slope to x value
            raster.setPixel(pix, (int) t, y0);
        }
        }
    }
```

This method of line drawing is called a *Digital Differential Analyzer* or DDA for short. Below is a short demonstration of the method.

The java applet above demonstrates the *lineDDA()* algorithm. Draw a line by clicking and dragging on the pixel grid shown using the left mouse button. An *ideal* line is displayed until the left button is released. Upon release a discrete approximation of the line is drawn on the display grid using the *lineDDA()* method shown above. An *ideal* line is then overlaid for comparison.

Ideally, you should be able to determine any difference in the lines generated by this method and the improved method mentioned previously. The main difference should only be speed, (I also made the method draw in a different color). When optimizing, it is important to verify at ech step that the algorithm remains intact. Next, we need to verify that our objective was met. For that purpose we can use the following benchmark applet.

This applet allows you to select from the various line drawing algorithms discussed. You can draw lines using the selected algorithm by clicking and dragging with the first mouse button. You can also time the algorithms drawing a selection of lines covering the range of slope by clickin on the **Benchmark**. In order to get more accurate timings the pattern is actually drawn five times (without clearing), and the final result is displayed. You still migh want to try each method several times to get an indication of its performance.

A parting note: Many modern optimizing compilers will automatically find the sorts of optimizations that we applied here (inlining function calls within loops and converting functions to incremental calculation). In order to check if your compiler is so clever, you'll need to do one of two things: look at the code it generated or write a test fragment to time your optimized code.

This rasies the question: *Is it better to retain readable code, and depend a compiler to do the optimation implicitly, or code the optimization explicitly with some loss in readability?* The answer is not clear cut. In general, you should make your code as readable as possible. Comments can go a long way in addressing this in optimized code. On the other hand, you should also seldom trade-off portability for elegance. Thus, if your application **depends** on a fast algorithm, you are better off coding directly rather than requiring on a compiler to do your work, because in all likelyhood, compiler versions and platforms will change more frequently than your code!

Step 3. Low-Level Optimizations

Low-level optimizations are somewhat dubious, because they depend on understanding various machine details. Because machines vary, the accuracy of these low-level assumptions can sometimes be questioned.

Typically, a set of general rules can be determined that are more-or-less consistent across machines. Here are some examples:

- **Addition** and **Subtraction** are generally faster than **Multiplication**.
- **Multiplication** is generally faster than **Division**.
- Using tables to evaluate discrete functions is faster than computing them
- Integer caluculations are faster than floating-point calculations.
- Avoid unnecessary computation by testing for various special cases.
- The intrinsic tests available to most machines are *greater than*, *less than*, *greater than or equal*, and *less than or equal* to **zero** (not an arbitrary value).

None of these rules are etched in stone. Some of these rules are becoming less and less valid as time passes. We'll address these issues in more detail later.

For our line drawing algorithm we'll investigate applying several of these optimizations. The incremental calculation effectively removed multiplications in favor of additions. Our next optimization will use three of the mentioned methods. It will remove floating-point calculations in favor of integer operations, and it will remove the single divide opertaion (it makes a difference on short lines), and it will normalize the tests to tests for zero.

Notice that the slope is always rational (a ratio of two integers).

$$m = (y1 - y0) / (x1 - x0)$$

Also note that the incremental part of the algorthim never generates a new y value that is more than one unit away from the old one, because the slope is always less than one (this assured by our improved algorithm).

$$y[i+1] = y[i] + m$$

Thus, if we maintained the only the only fractional part of y we could still draw a line by noting when this fraction exceeded one. If we initialize fraction with 0.5, then we will also handle the rounding correctly as in our DDA routine.

```
fraction += m
if (fraction[i+1] >= 1) { y = y +1; fraction -= 1; }
```

Note that the y variable is now an integer. Next we discuss how to retain the fraction as an integer. After we draw the first pixel (which happens outside our main loop) the correct fraction value is:

$$fraction = 1/2 + dy / dx$$

If we scale the fraction by 2*dx the following expression results:

$$scaledFraction = dx + 2*dy,$$

and the incremental update becomes:

$$scaledFraction += 2*dy,$$

and our test must be modified to reflect the new scaling

$$if (scaledFraction >= 2*dx) \{ ... \}.$$

This test can be made a test against a value of zero if the inital value of scaledFraction has 2*dx subtracted from it. Giving outside the loop:

$$OffsetScaledFraction = dx + 2*dy - 2*dx = 2*dy - dx,$$

and the inner loop becomes

```
OffsetScaledFraction += 2*dy
if (OffsetScaledFraction >= 0) { y = y +1; fraction -= 2*dx; }
```

The net result is that we might as well double the values of dy and dx (this can be accomplished with either an add or a shift). The result ing method is known as Bresenham's line drawing algorithm. The code is shown below.

```
public void lineBresenham(int x0, int y0, int x1, int y1, Color color)
{
    int pix = color.getRGB();
    int dy = y1 - y0;
    int dx = x1 - x0;
    int stepx, stepy;
```

```
        if (dy < 0) { dy = -dy;  stepy = -1; } else { stepy = 1; }
        if (dx < 0) { dx = -dx;  stepx = -1; } else { stepx = 1; }
        dy <<= 1;                                                  // dy is now 2*dy
        dx <<= 1;                                                  // dx is now 2*dx

        raster.setPixel(pix, x0, y0);
        if (dx > dy) {
            int fraction = dy - (dx >> 1);                         // same as 2*dy - dx
            while (x0 != x1) {
                if (fraction >= 0) {
                    y0 += stepy;
                    fraction -= dx;                                // same as fraction -= 2*dx
                }
                x0 += stepx;
                fraction += dy;                                    // same as fraction -= 2*dy
                raster.setPixel(pix, x0, y0);
            }
        } else {
            int fraction = dx - (dy >> 1);
            while (y0 != y1) {
                if (fraction >= 0) {
                    x0 += stepx;
                    fraction -= dy;
                }
                y0 += stepy;
                fraction += dx;
                raster.setPixel(pix, x0, y0);
            }
        }
    }
```

The java applet above demonstrates the *lineBresenham()* algorithm. Draw a line by clicking and dragging on the pixel grid shown using the left mouse button. An *ideal* line is displayed until the left button is released. Upon release a discrete approximation of the line is drawn on the display grid using the *lineBresenham()* method shown above. An *ideal* line is then overlaid for comparison.

Once again we should test to see that our new algorithm is indeed faster than a DDA.

This applet allows you to select from the various line drawing algorithms discussed. You can draw lines using the selected algorithm by clicking and dragging with the first mouse button. You can also time the algorithms drawing a selection of lines covering the range of slope by clickin on the **Benchmark**. In order to get more accurate timings the pattern is actually drawn five times (without clearing), and the final result is displayed. You still migh want to try each method several times to get an indication of its performance.

Step 4. Question infrastructure

There is still a hidden multiply inside of our inner loop. Remember back when we implemented the Raster object's *setPixel()* method.

```
    /**
     *  Sets a pixel to a given value
     */
    public final boolean setPixel(int pix, int x, int y)
    {
        pixel[y*width+x] = pix;
        return true;
    }
```

Notice that the setting of every pixel involves a multiply. This next version of Bresenham's algorithm eliminates even this multiply:

```
    public void lineFast(int x0, int y0, int x1, int y1, Color color)
    {
        int pix = color.getRGB();
        int dy = y1 - y0;
        int dx = x1 - x0;
        int stepx, stepy;

        if (dy < 0) { dy = -dy;  stepy = -raster.width; } else { stepy = raster.width; }
        if (dx < 0) { dx = -dx;  stepx = -1; } else { stepx = 1; }
        dy <<= 1;
        dx <<= 1;

        y0 *= raster.width;
        y1 *= raster.width;
        raster.pixel[x0+y0] = pix;
        if (dx > dy) {
            int fraction = dy - (dx >> 1);
            while (x0 != x1) {
                if (fraction >= 0) {
                    y0 += stepy;
                    fraction -= dx;
                }
                x0 += stepx;
                fraction += dy;
                raster.pixel[x0+y0] = pix;
            }
        } else {
            int fraction = dx - (dy >> 1);
            while (y0 != y1) {
                if (fraction >= 0) {
                    x0 += stepx;
                    fraction -= dy;
                }
                y0 += stepy;
                fraction += dx;
                raster.pixel[x0+y0] = pix;
            }
        }
    }
```

The java applet above demonstrates the *lineFast()* algorithm. Draw a line by clicking and dragging on the pixel grid shown using the left mouse button. An *ideal* line is displayed until the left button is released. Upon release a discrete approximation of the line is drawn on the display grid using the *lineFast()* method shown above. An *ideal* line is then overlaid for comparison.

Once again we should test to see that our new algorithm is indeed faster than a DDA.

This applet allows you to select from the various line drawing algorithms discussed. You can draw lines using the selected algorithm by clicking and dragging with the first mouse button. You can also time the algorithms drawing a selection of lines covering the range of slope by clickin on the **Benchmark**. In order to get more accurate timings the pattern is actually drawn five times (without clearing), and the final result is displayed. You still migh want to try each method several times to get an indication of its performance.

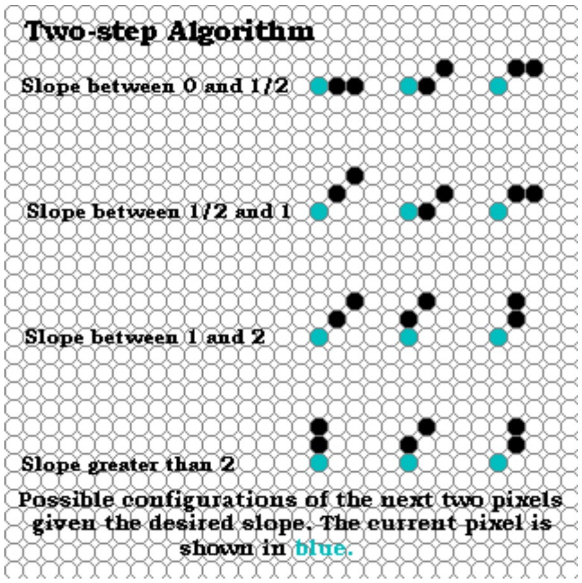
Todo: Invalid assumptions and the chaning state of computer architectures

---

Life After Bresenham

Most books would have you believe that the development of line drawing algorithms ended with Bresenham's famous algorithm. But there has been some signifcant work since then. The following 2-step algorithm, developed by Xiaolin Wu, is a good example. The interesting story of this algorithm's development is discussed in an article that appears in Graphics Gems I by Brian Wyvill.

The two-step algorithm takes the interesting approach of treating line drawing as a automaton, or finite state machine. If one looks at the possible configurations that the next two pixels of a line, it is easy to see that only a finite set of possiblities exist.


Two-step Algorithm
Slope between 0 and 1/2
Slope between 1/2 and 1
Slope between 1 and 2
Slope greater than 2
Possible configurations of the next two pixels given the desired slope. The current pixel is shown in blue.

The two-step algorithm shown here also exploits the symmetry of line-drawing by simultaneously drawn from both ends towards the midpoint.

```
public void lineTwoStep(int x0, int y0, int x1, int y1, Color color)
{
    int pix = color.getRGB();
    int dy = y1 - y0;
    int dx = x1 - x0;
    int stepx, stepy;

    if (dy < 0) { dy = -dy;  stepy = -1; } else { stepy = 1; }
    if (dx < 0) { dx = -dx;  stepx = -1; } else { stepx = 1; }

    raster.setPixel(pix, x0, y0);
    raster.setPixel(pix, x1, y1);
    if (dx > dy) {
        int length = (dx - 1) >> 2;
        int extras = (dx - 1) & 3;
        int incr2 = (dy << 2) - (dx << 1);
        if (incr2 < 0) {
            int c = dy << 1;
            int incr1 = c << 1;
            int d =  incr1 - dx;
            for (int i = 0; i < length; i++) {
                x0 += stepx;
                x1 -= stepx;
                if (d < 0) {                                                // Pattern:
                    raster.setPixel(pix, x0, y0);                   //
                    raster.setPixel(pix, x0 += stepx, y0);  //  x o o
                    raster.setPixel(pix, x1, y1);                   //
                    raster.setPixel(pix, x1 -= stepx, y1);
                    d += incr1;
                } else {
                    if (d < c) {                                                    // Pattern:
                        raster.setPixel(pix, x0, y0);                               //      o
                        raster.setPixel(pix, x0 += stepx, y0 += stepy);             //  x o
                        raster.setPixel(pix, x1, y1);                               //
                        raster.setPixel(pix, x1 -= stepx, y1 -= stepy);
                    } else {
                        raster.setPixel(pix, x0, y0 += stepy);                      // Pattern:
                        raster.setPixel(pix, x0 += stepx, y0);                      //    o o
                        raster.setPixel(pix, x1, y1 -= stepy);                      //  x
                        raster.setPixel(pix, x1 -= stepx, y1);                      //
                    }
                    d += incr2;
                }
            }
            if (extras > 0) {
                if (d < 0) {
                    raster.setPixel(pix, x0 += stepx, y0);
                    if (extras > 1) raster.setPixel(pix, x0 += stepx, y0);
                    if (extras > 2) raster.setPixel(pix, x1 -= stepx, y1);
                } else
                if (d < c) {
                    raster.setPixel(pix, x0 += stepx, y0);
                    if (extras > 1) raster.setPixel(pix, x0 += stepx, y0 += stepy);
                    if (extras > 2) raster.setPixel(pix, x1 -= stepx, y1);
                } else {
                    raster.setPixel(pix, x0 += stepx, y0 += stepy);
                    if (extras > 1) raster.setPixel(pix, x0 += stepx, y0);
                    if (extras > 2) raster.setPixel(pix, x1 -= stepx, y1 -= stepy);
                }
            }
        } else {
            int c = (dy - dx) << 1;
            int incr1 = c << 1;
            int d =  incr1 + dx;
            for (int i = 0; i < length; i++) {
                x0 += stepx;
                x1 -= stepx;
                if (d > 0) {
                    raster.setPixel(pix, x0, y0 += stepy);                  // Pattern:
                    raster.setPixel(pix, x0 += stepx, y0 += stepy);         //      o
                    raster.setPixel(pix, x1, y1 -= stepy);                  //    o
                    raster.setPixel(pix, x1 -= stepx, y1 -= stepy);         //  x
                    d += incr1;
                } else {
                    if (d < c) {
                        raster.setPixel(pix, x0, y0);                               // Pattern:
                        raster.setPixel(pix, x0 += stepx, y0 += stepy);       //      o
                        raster.setPixel(pix, x1, y1);                         //  x o
                        raster.setPixel(pix, x1 -= stepx, y1 -= stepy);       //
                    } else {
                        raster.setPixel(pix, x0, y0 += stepy);                      // Pattern:
                        raster.setPixel(pix, x0 += stepx, y0);                      //    o o
                        raster.setPixel(pix, x1, y1 -= stepy);                      //  x
                        raster.setPixel(pix, x1 -= stepx, y1);                      //
                    }
                    d += incr2;
                }
            }
            if (extras > 0) {
                if (d > 0) {
                    raster.setPixel(pix, x0 += stepx, y0 += stepy);
                    if (extras > 1) raster.setPixel(pix, x0 += stepx, y0 += stepy);
                    if (extras > 2) raster.setPixel(pix, x1 -= stepx, y1 -= stepy);
                } else
                if (d < c) {
                    raster.setPixel(pix, x0 += stepx, y0);
                    if (extras > 1) raster.setPixel(pix, x0 += stepx, y0 += stepy);
                    if (extras > 2) raster.setPixel(pix, x1 -= stepx, y1);
                } else {
                    raster.setPixel(pix, x0 += stepx, y0 += stepy);
                    if (extras > 1) raster.setPixel(pix, x0 += stepx, y0);
                    if (extras > 2) {
                        if (d > c)
                            raster.setPixel(pix, x1 -= stepx, y1 -= stepy);
                        else
                            raster.setPixel(pix, x1 -= stepx, y1);
                    }
                }
            }
        }
    } else {
        int length = (dy - 1) >> 2;
        int extras = (dy - 1) & 3;
        int incr2 = (dx << 2) - (dy << 1);
        if (incr2 < 0) {
            int c = dx << 1;
            int incr1 = c << 1;
            int d =  incr1 - dy;
            for (int i = 0; i < length; i++) {
                y0 += stepy;
                y1 -= stepy;
                if (d < 0) {
                    raster.setPixel(pix, x0, y0);
                    raster.setPixel(pix, x0, y0 += stepy);
                    raster.setPixel(pix, x1, y1);
                    raster.setPixel(pix, x1, y1 -= stepy);
                    d += incr1;
                } else {
                    if (d < c) {
                        raster.setPixel(pix, x0, y0);
                        raster.setPixel(pix, x0 += stepx, y0 += stepy);
                        raster.setPixel(pix, x1, y1);
                        raster.setPixel(pix, x1 -= stepx, y1 -= stepy);
                    } else {
                        raster.setPixel(pix, x0 += stepx, y0);
                        raster.setPixel(pix, x0, y0 += stepy);
                        raster.setPixel(pix, x1 -= stepx, y1);
                        raster.setPixel(pix, x1, y1 -= stepy);
                    }
                    d += incr2;
                }
            }
            if (extras > 0) {
                if (d < 0) {
                    raster.setPixel(pix, x0, y0 += stepy);
                    if (extras > 1) raster.setPixel(pix, x0, y0 += stepy);
                    if (extras > 2) raster.setPixel(pix, x1, y1 -= stepy);
                } else
                if (d < c) {
                    raster.setPixel(pix, stepx, y0 += stepy);
                    if (extras > 1) raster.setPixel(pix, x0 += stepx, y0 += stepy);
                    if (extras > 2) raster.setPixel(pix, x1, y1 -= stepy);
                } else {
                    raster.setPixel(pix, x0 += stepx, y0 += stepy);
                    if (extras > 1) raster.setPixel(pix, x0, y0 += stepy);
                    if (extras > 2) raster.setPixel(pix, x1 -= stepx, y1 -= stepy);
                }
            }
        } else {
            int c = (dx - dy) << 1;
            int incr1 = c << 1;
            int d =  incr1 + dy;
            for (int i = 0; i < length; i++) {
                y0 += stepy;
                y1 -= stepy;
                if (d > 0) {
                    raster.setPixel(pix, x0 += stepx, y0);
                    raster.setPixel(pix, x0 += stepx, y0 += stepy);
                    raster.setPixel(pix, x1 -= stepy, y1);
                    raster.setPixel(pix, x1 -= stepx, y1 -= stepy);
                    d += incr1;
                } else {
                    if (d < c) {
                        raster.setPixel(pix, x0, y0);
                        raster.setPixel(pix, x0 += stepx, y0 += stepy);
                        raster.setPixel(pix, x1, y1);
                        raster.setPixel(pix, x1 -= stepx, y1 -= stepy);
                    } else {
                        raster.setPixel(pix, x0 += stepx, y0);
                        raster.setPixel(pix, x0, y0 += stepy);
                        raster.setPixel(pix, x1 -= stepx, y1);
                        raster.setPixel(pix, x1, y1 -= stepy);
                    }
                    d += incr2;
                }
            }
            if (extras > 0) {
                if (d > 0) {
                    raster.setPixel(pix, x0 += stepx, y0 += stepy);
                    if (extras > 1) raster.setPixel(pix, x0 += stepx, y0 += stepy);
                    if (extras > 2) raster.setPixel(pix, x1 -= stepx, y1 -= stepy);
                } else
                if (d < c) {
                    raster.setPixel(pix, x0, y0 += stepy);
                    if (extras > 1) raster.setPixel(pix, x0 += stepx, y0 += stepy);
                    if (extras > 2) raster.setPixel(pix, x1, y1 -= stepy);
                } else {
                    raster.setPixel(pix, x0 += stepx, y0 += stepy);
                    if (extras > 1) raster.setPixel(pix, x0, y0 += stepy);
                    if (extras > 2) {
                        if (d > c)
                            raster.setPixel(pix, x1 -= stepx, y1 -= stepy);
                        else
                            raster.setPixel(pix, x1, y1 -= stepy);
                    }
                }
            }
        }
    }
}
```

The java applet above demonstrates the *lineTwoStep()* algorithm. Draw a line by clicking and dragging on the pixel grid shown using the left mouse button. An *ideal* line is displayed until the left button is released. Upon release a discrete approximation of the line is drawn on the display grid using the *lineTwoStep()* method shown above. An *ideal* line is then overlaid for comparison.

Back to outline

This page last modified Monday, September 16, 1996