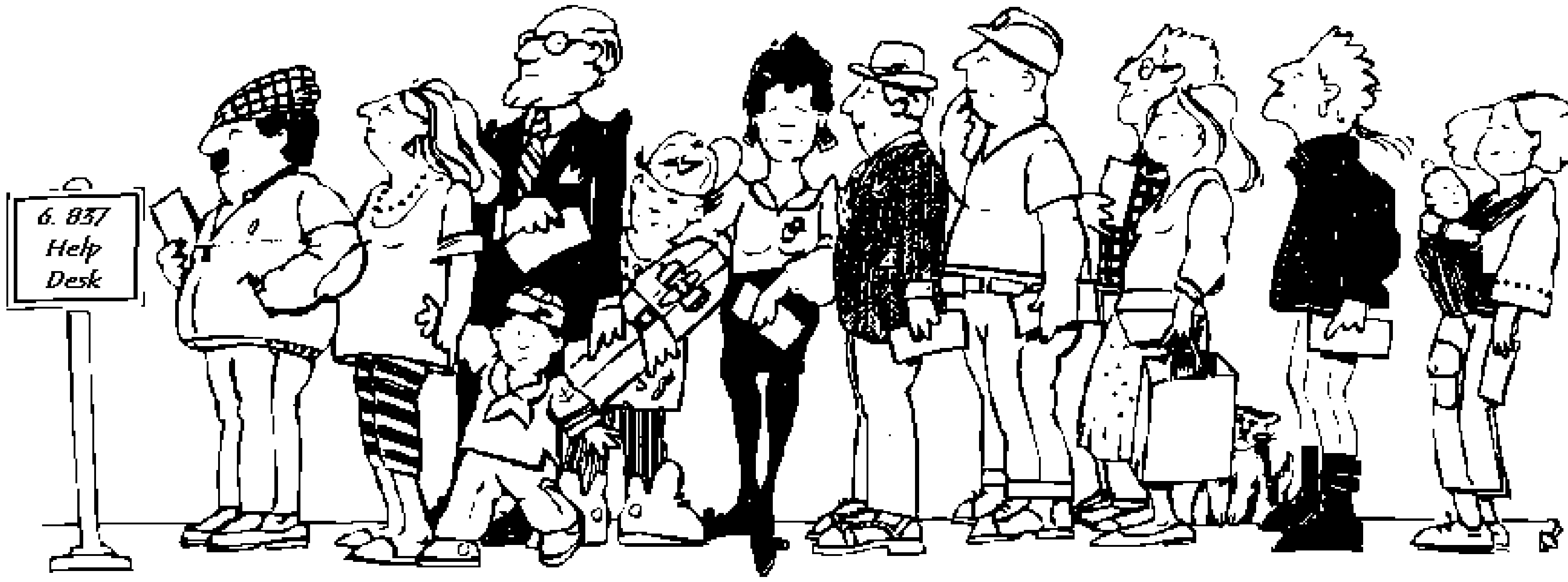
6.837 Lecture 4

Line-Drawing Algorithms

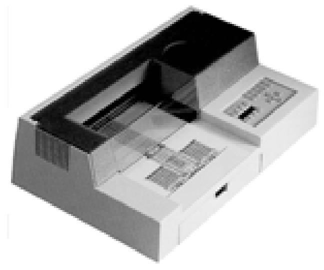
A Study in Optimization

Make it work
Make it right
Make it fast



Line-Drawing Algorithms

Our first adventure into *scan conversion*.

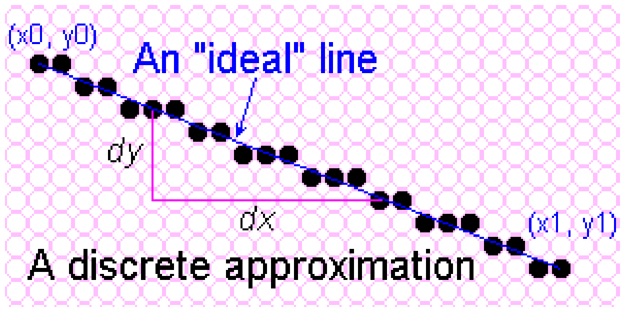


- Scan-conversion or *rasterization*
- Due to the scanning nature of raster displays
- Algorithms are fundamental to both 2-D and 3-D computer graphics
- Transforming the continuous into this discrete (sampling)
- Line drawing was easy for vector displays
- Most incremental line-drawing algorithms were first developed for pen-plotters

Most of the early scan-conversion algorithms developed for plotters can be attributed to one man, Jack Bresenham.

Quest for the *Ideal Line*

The best we can do is a discrete approximation of an ideal line.



Important line qualities:

- Continuous appearence
- Uniform thickness and brightness
- Accuracy (Turn on the pixels nearest the ideal line)
- Speed (How fast is the line generated)

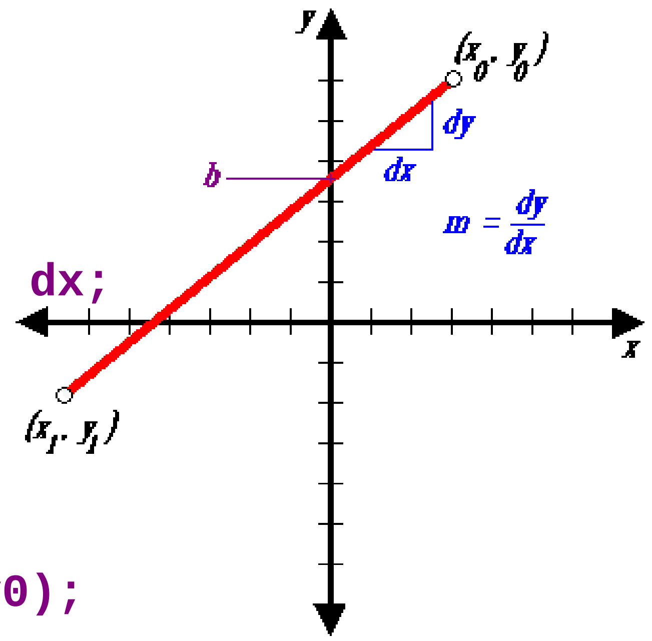
Simple Line

Based on the simple *slope-intercept algorithm* from algebra

$$y = m\,x + b$$

```
    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);
            }
        }
    }
```



lineSimple( ) Demonstration

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 described in the previous slide. An *ideal* line is then overlaid for comparison.

The *lineSimple()* method:

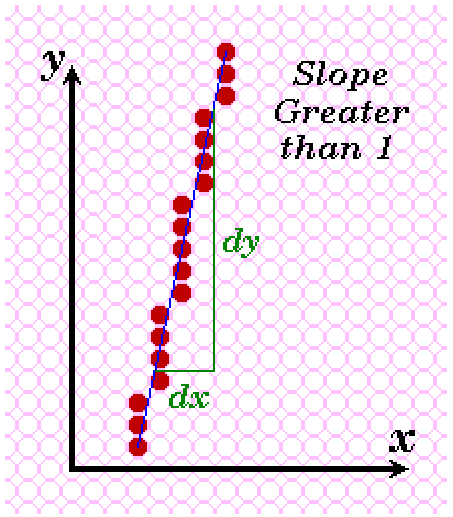
Does it work?

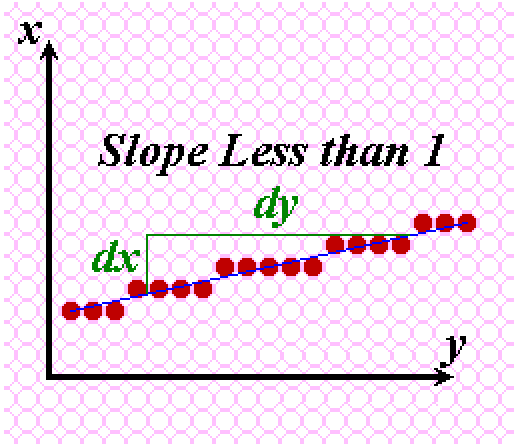
Try various slopes.

Let's Make it Work!

Problem: lineSimple( ) does not give satisfactory results for slopes > 1

Solution: symmetry





```
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);
        }
    }
}
```

lineImproved( ) Demonstration

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 *lineImproved()* method described in the previous slide. An *ideal* line is then overlaid for comparison.

The *lineImproved()* method:

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

Optimize Inner Loops

Optimize those code fragments where the algorithm spends most of its time.

Often these fragments are inside loops.

Overall code organization:

```
lineMethod( )
{
    // 1. general set up
    // 2. special case set up
    while (notdone) {
        // 3. inner loop
    }
}
```

- remove unnecessary method invocations

  replace `Math.round(m*x0 + b)`
  with `(int)(m*x0 + b + 0.5)`
  Does this always work?

- use incremental calculations

  Consider the expression

  `y = (int)(m*x + b + 0.5)`

  The value of y is known at x_0 (i.e. it is $y_0 + 0.5$)
  Future values of y can be expressed in terms of previous values with a **difference equation**:

  $$y_{i+1} = y_i + m;$$
  or
  $$y_{i+1} = y_i - m;$$

Modified Algorithm

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

```
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);
        }
    }
}
```

lineDDA( ) Demonstration

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 *lineDDA()* method described in the previous slide. An *ideal* line is then overlaid for comparison.

The *lineDDA()* method:

You should not see any difference in the lines generated by this method and the *lineImproved( )* method mentioned previously.

Was Our Objective Met?

This applet above 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 by clicking on Benchmark. In order to get more accurate timings the pattern is drawn five times (without clearing), and the final result is displayed.

To the left is a benchmarking applet

Modern compilers will often find these sorts of optimizations

Dilemma:

Is it better to retain readable code, and depend a compiler to do the optimization implicitly, or code the optimization explicitly with some loss in readability?

Low-Level Optimizations

Low-level optimizations are dubious, because
they often depend on specific machine details.

However, a set of general rules that are more-or-less consistent across machines include:

- **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).

Applications of Low-level Optimizations

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

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

Note that the incremental part of the algorthim never generates a new y that is more than one unit away from the old one (because the slope is always less than one)

$$y_{i+1} = y_i + m$$

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 >= 1) { y = y + 1; fraction -= 1; }
```

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 operation (it makes a difference on short lines), and it will normalize the tests to tests for zero.

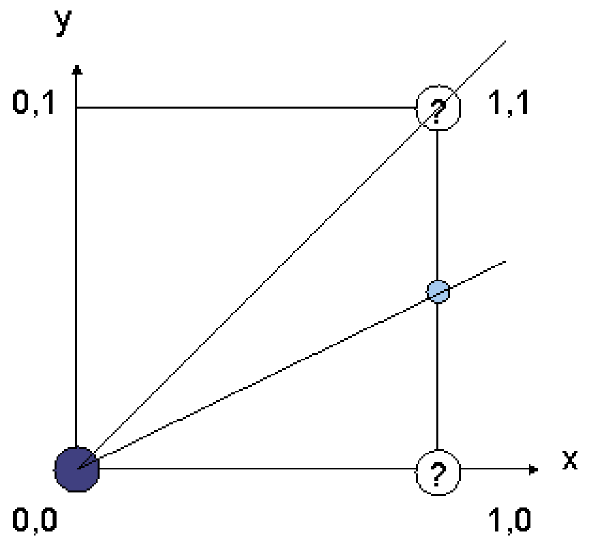
Geometric Interpretation

fraction = 0.5; // Initial

fraction += m; // Increment

if 0.5 <= m <= 1 then fraction >= 1
Plot(1,1)

if 0 < m < 0.5 then fraction < 1
Plot(1,0)



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

More Low-level Optimizations

Note that y is now an integer.
Can we represent the fraction as an integer?

After we draw the first pixel (which happens outside our main loop) the correct fraction 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        // 2*dx*(dy/dx)
```

and our test must be modified to reflect the new scaling

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

More Low-level Optimizations

This test can be made 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;
        }
```

We might as well double the values of dy and dx (this can be accomplished with either an add or a shift outside the loop).

The resulting method is known as Bresenham's line drawing algorithm

```
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);
        }
    }
}
```

lineBresenham( ) Demonstration

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 *lineBresenham()* method described in the previous slide. An *ideal* line is then overlaid for comparison.

The *lineBresenham()* method:

Does it work?

Was it worth it?

This applet above 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 by clicking on Benchmark. In order to get more accurate timings the pattern is drawn five times (without clearing), and the final result is displayed.

To the left is a benchmarking applet

Question Infrastructure

There is still a hidden multiply inside of our inner loop

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

Our next optimization of Bresenham's algorithm
eliminates even this multiply.

Faster Bresenham Algorithm

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

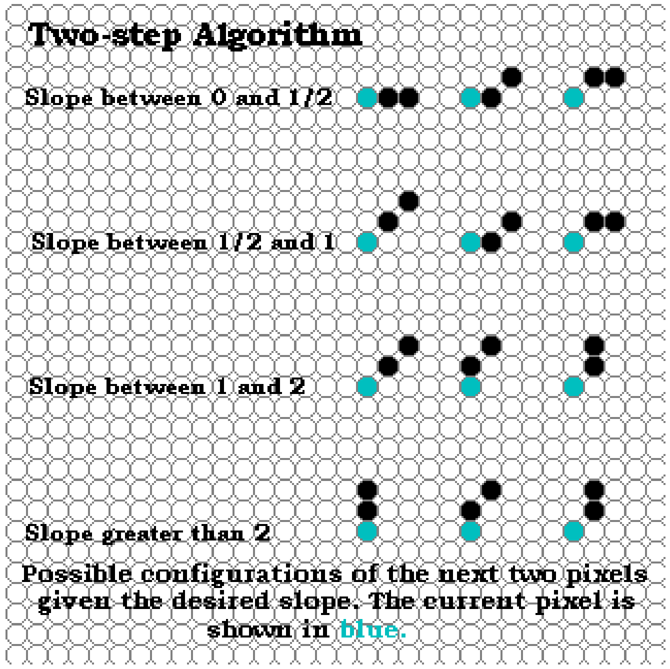
Was it worth it?

This applet above 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 by clicking on Benchmark. In order to get more accurate timings the pattern is drawn five times (without clearing), and the final result is displayed.

To the left is a benchmarking applet

Can we go even faster?

Beyond Bresenham


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.

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.

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

The code, which is here, is a bit long to show an a slide.

Was it worth it?

Hardly!

Note: Compare the speed of lineBresenham( ) to lineTwoStep( ). This comparision is most fair since I did not remove the calls to raster.setPixel() in the two-step algorithm.

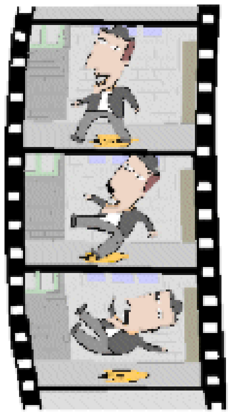
Epilogue

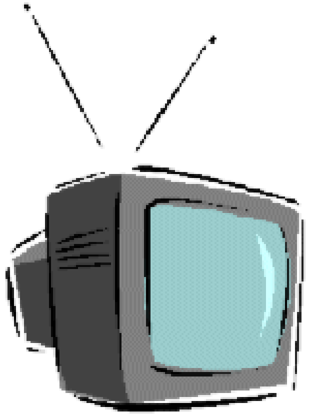
There are still many important issues associated with line drawing

Examples:

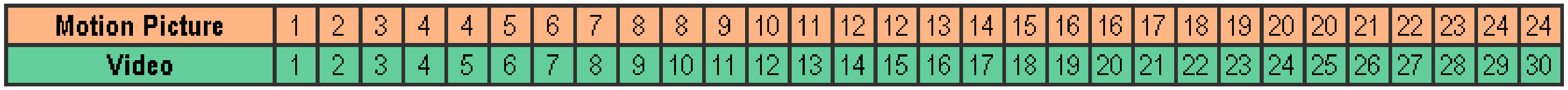
- Non-integer endpoints
  (occurs frequently when rendering 3D lines)
- Can we make lines appear less "jaggy"?
- What if a line endpoint lies outside the viewing area?
- How do you handle thick lines?
- Optimizations for connected line segments
- Lines show up in the strangest places

A Line in Sheep's Clothing





Movies developed for theaters are usually shot at 24 frames per second, while video on television is displayed at 30 frames per second. Thus, when motion pictures are transferred to video formats they must undergo a process called "3-2 pull-down", where every fourth frame is repeated thus matching the frame rates of the two media, as depicted below.



Motion Picture	1	2	3	4	4	5	6	7	8	8	9	10	11	12	12	13	14	15	16	16	17	18	19	20	20	21	22	23	24	24
Video	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

The Plot Thickens...

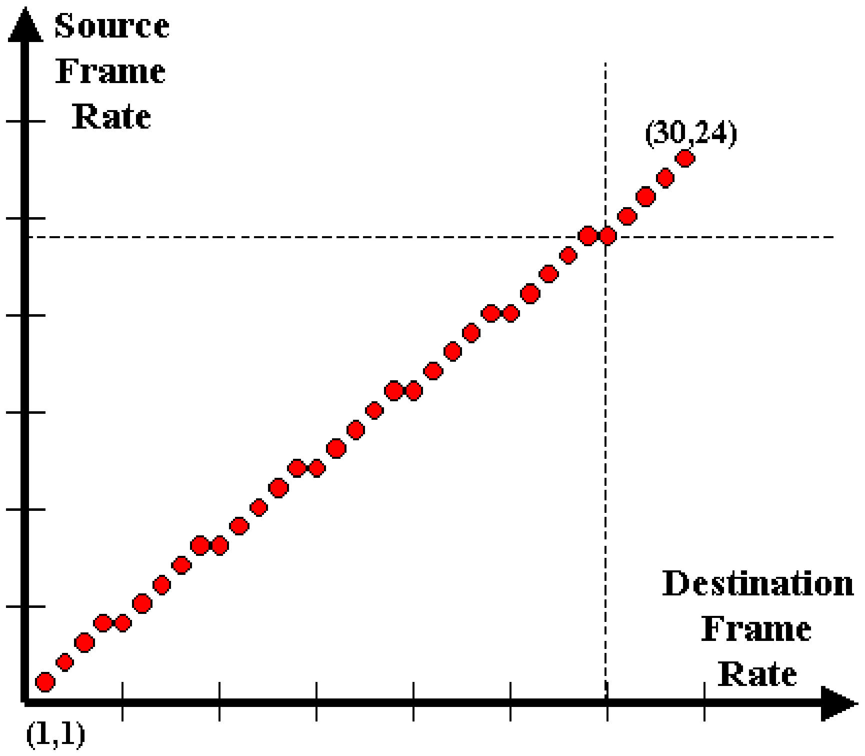
Once upon a time, there was a company with a similar, yet more complicated problem. They had developed a multimedia movie-player technology that they wished to display on a workstation's CRT screen. They also made screens with various update rates including 60, 66, 72, and 75 frames-per-second. Their movie player had to support a wide range of source materials including, but not limited to:

Source	**Frame Rate**
Motion Pictures	24 fps
NTSC Video (frames)	30 fps
NTSC Video (fields)	60 fps
PAL Video (frames)	25 fps
PAL Video (fields)	50 fps
Common Multimedia	10 fps

These source materials were to be displayed as close as possible to the correct frame rate. We were not expected to support frame rates higher than the CRT's update rate. Upon having the problem explained, someone commented, "It's just a line-drawing algorithm!"

Generalized Pull Down

The generalized pull-down problem is nothing more that drawing the best discrete approximation of the ideal line from (1, 1) to (desired frame rate, source frame rate), as shown in the following diagram for the (30 fps, 24 fps).



Moreover, it is a very special case of a line. Can you think of addition optimizations to our Bresenham line drawing code that could be applyed for the special case of pull-down?

Next Time

What is a pixel anyway?
Where do they live?
Do they have a middle?