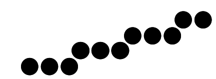
Scan-conversion

- The process of setting pixels to represent a geometric primitive
  - Only operation: setpixel(x, y, color)
- Types of geometric primitives
  - Points
  - Lines
  - Curves
  - ...
  - Triangles
  - Polygons
  - Curved surfaces

Lines

The line is the most fundamental drawing primitive, with many uses:

- Charts
- Schematics, in engineering drawings
- Illustrations
- Pencil-test animations
- Approximating curves

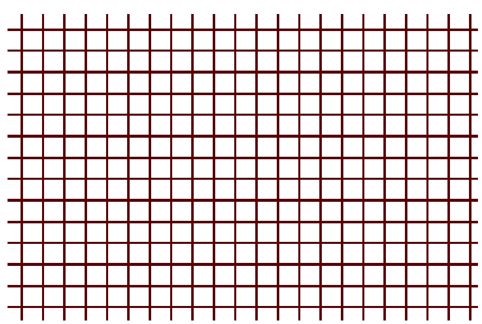


Jaggies

It is not so easy to make good lines on raster displays: "jaggies" are inherent.

Scan-converting line segments

- Given: a line segment with endpoints at pixel coordinates (x1,y1), (x2, y2)
- Find: pixels to set to approximate the line segment



Good lines

Properties we would like to have in a "good line" include:

-
-
-
-

Worst possible line-drawing algorithm

Bresenham's algorithm

Introduction:

- Example of an "incremental algorithm"
- Takes advantage of "coherence," a common theme in graphics. Other examples include:
  - Scan-converting polygons
  - Frame-to-frame coherence in animations

Bresenham's algorithm, cont.

Simplifying assumptions

- Translate one endpoint to origin
- Assume line in first "octant"
- Assume line starts and ends at pixel centers

Algorithm

- Step in *x*, painting 1 pixel/column.
- Which pixel??

Algorithm 1

exactreal *y*
int *x, dx, dy*
for *x* = 0 **to** *dx* **do**
    y = (*dy*/*dx*)*x*
    plot(*x*, *round*(*y*))
end for

<u>Problems</u>

- Uses "**exactreal**"

<u>Next step</u>

1. Apply "strength reduction"

Algorithm 2

```
exactreal y
int x, dx, dy
y = 0
plot(x, round(y))
for x = 1 to dx do
    y = y + (dy/dx)
    plot(x, round(y))
end for
```

<u>Next steps</u>

1. Expand *round*(y) to *floor*(y + .5)
2. Move .5 to initialization
3. Get rid of *floor*() with a representation change

Algorithm 3

```
exactreal y_frac
int x, dx, dy, y_int
y_int = 0
y_frac = .5
plot(x, y_int)
for x = 1 to dx do
    if y_frac + (dy/dx) >= 1 then
        y_int = y_int + 1
        y_frac = y_frac + (dy/dx) - 1
    else
        y_frac = y_frac + (dy/dx)
    end if
    plot(x, y_int)
end for
```

<u>Next step:</u> Multiply y_{frac} through by $2dx$

Algorithm 4

```
int x, dx, dy, y_int, y_frac
y_int = 0
y_frac = dx
plot(x, y_int)
for x = 1 to dx do
    if y_frac + 2dy >= 2dx then
        y_int = y_int + 1
        y_frac = y_frac + 2dy - 2dx
    else
        y_frac = y_frac + 2dy
    end if
    plot(x, y_int)
end for
```


Algorithm 4, cont.

<u>Next steps:</u>

1. Use comparison against 0 in inner loop
2. Precompute increments ***incrNE*** and ***incrE***
3. Change $\{y_{int}, y_{frac}\}$ to $\{y, p\}$

Bresenham's Algorithm

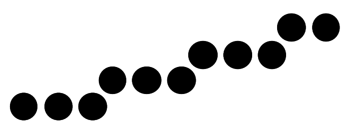
```
int x, y, p, dx, dy, incrE, incrNE
y = 0
p = 2dy - dx
incrE = 2dy
incrNE = 2dy - 2dx
plot(x, y)
for x = 1 to dx do
    if p >= 0 then
        y = y + 1
        p = p  + incrNE
    else
        p = p + incrE
    end if
    plot(x, y)
end for
```


Generalization to all octants

The previous algorithm draws lines in the first octant, illuminating every pixel whose vertical midline is intersected:

Lines in other octants are handled by symmetry:

Aliasing



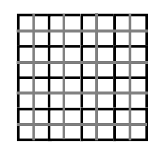
Jaggies

As mentioned earlier, Bresenham lines have a serious flaw, known as

- "the jaggies"
- "staircasing"
- "aliasing" (from sampling theory)

Remedies for this problem are called "antialiasing"

Supersampling



Four supersamples per pixel

First solution: Increase the resolution!

Gives twice as many jags, each one half as large

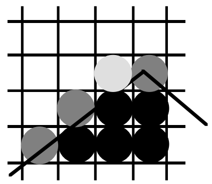
Advantages:

- Conceptually simple

Disadvantages:

- Halving the size of jags means quadrupling the memory and doubling scan conversion time
- Reduces but does not eliminate the problem

Antialiasing



Adjusting pixel intensities along an area boundry

Suppose you've got more than 1 bit/pixel.

Idea is to use multiple grey levels to soften the jaggies.

Called "antialiasing."

Unweighted area sampling

1. Assume the pixels are squares tiling the display.

2. Light up the pixels according to each square's coverage.

An easy way to implement "unweighted area sampling" is to supersample and average.

Unweighted area sampling, cont.

Properties:

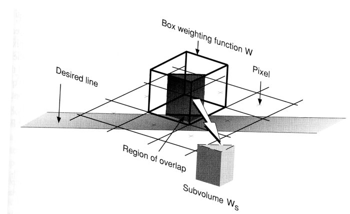
- Smooths over jaggies
- Does not eliminate them entirely
- Gives "ropey" effect
- Equal coverage gives equal contribution

Unweighted area sampling, cont.

Here's another way to look at the unweighted case:

1. Define a "weighting function" or "filter function" $W(x, y)$
2. The volume covered by a primitive under W determines the pixel contribution.

For *unweighted* area sampling, W is just a box function:



(Foley et al.) **Fig. 3.57** Box filter for square pixel.

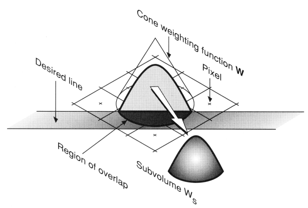
Unweighted area sampling is also called "box filtering"

Weighted area sampling

By changing *W*, we can make the contribution depend on <u>which part</u> of the pixel is covered.

Note that the weighting function can be <u>larger</u> than a single pixel.

The "cone filter" below has a linear drop off and a support radius of 1:



(Foley et al.) **Fig. 3.58** Cone filter for circular pixel with diameter of two grid units.

Another popular filter is the truncated Gaussian:

(Both give a big improvement over box filtering.)

Gupta-Sproull antialiased line drawing algorithm

Uses circularly-symmetric filters with support of radius 1.

<u>Key observations:</u>

- Volume under filter completely determined by 1 parameter -- the distance *d* from pixel center to line center.
- Each line affects a limited number of pixels.

Implementing Gupta-Sproull

The implementation of Gupta-Sproull is simple:

1. Compute table of volumes for each distance *d*.
2. Compile it in.
3. Shade 3 pixels at each step, using table-lookup on *d*.

Accomplishments of Gupta-Sproull

The Gupta-Sproull antialiased line drawing algorithm has several advantages over Bresenham's:

- Lines really appear straight!
- Lines appear to have constant width (even horizontal vs. vertical vs. diagonal)!
- Lines terminate nicely!
- Pretty fast!