



---

# CS 4300

# Computer Graphics

Prof. Harriet Fell  
Fall 2011

Lecture 5 – September 15, 2011



# Today's Topics

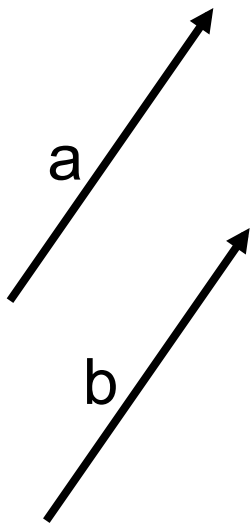
---

- Vectors – review Shirley *et al.* 2.4
- Rasters Shirley *et al.* 3.0 - 3.2.1
- Rasterizing Lines
  - Shirley *et al.*
    - 8.0 - 8.1.1
    - Implicit 2D lines pp. 30-35
    - Parametric Lines p. 41
  - Antialiasing
  - Line Attributes



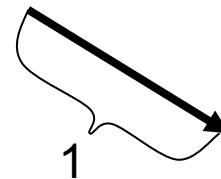
# Vectors

- A *vector* describes a length and a direction.

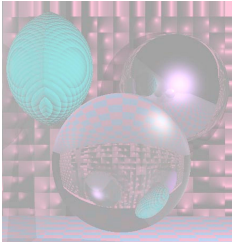


$$a = b$$

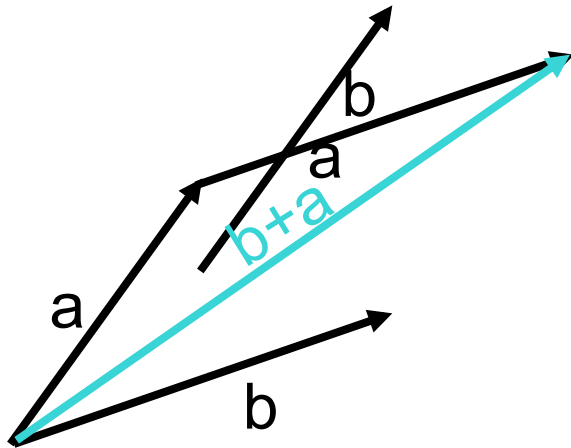
- a zero length vector



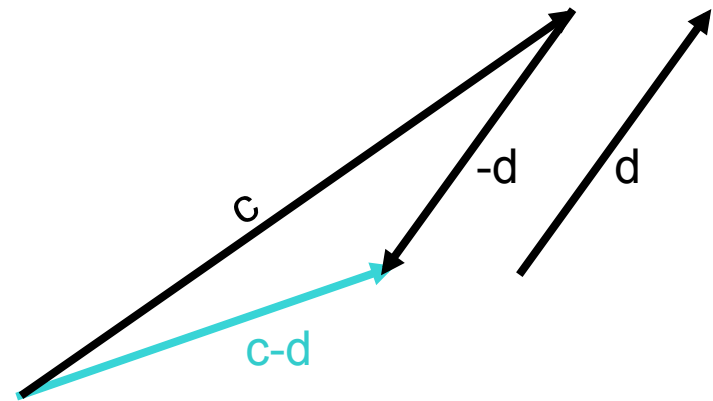
a unit vector



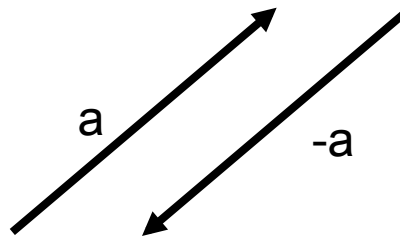
# Vector Operations



Vector Sum



Vector Difference





# Cartesian Coordinates

- Any two non-zero, non-parallel 2D vectors form a *2D basis*.
- Any 2D vector can be written uniquely as a linear combination of two 2D basis vectors.
- $\mathbf{x}$  and  $\mathbf{y}$  (or  $\mathbf{i}$  and  $\mathbf{j}$ ) denote unit vectors parallel to the  $x$ -axis and  $y$ -axis.
- $\mathbf{x}$  and  $\mathbf{y}$  form an *orthonormal* 2D basis.

$$\mathbf{a} = x_a \mathbf{x} + y_a \mathbf{y}$$

$$\mathbf{a} = (x_a, y_a) \quad \text{or} \quad \mathbf{a} = \begin{bmatrix} x_a \\ y_a \end{bmatrix}$$

$$\text{or } \mathbf{a} = (a_x, a_y)$$

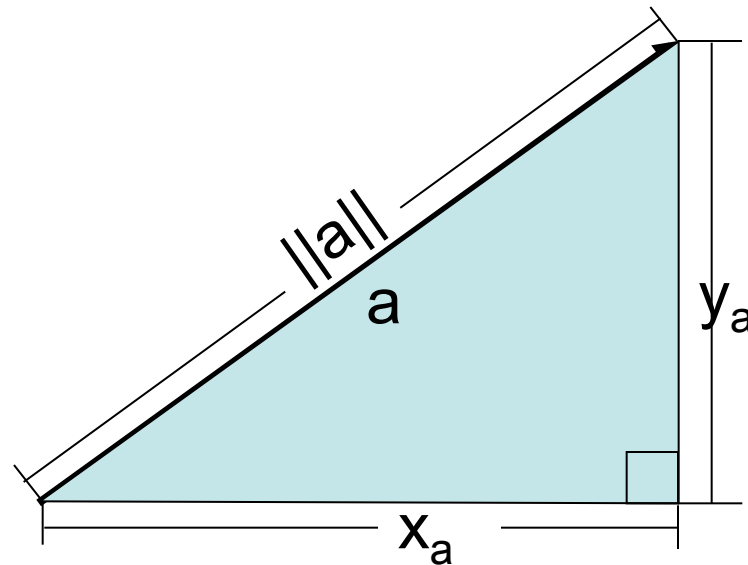
- $\mathbf{x}$ ,  $\mathbf{y}$  and  $\mathbf{z}$  form an *orthonormal* 3D basis.

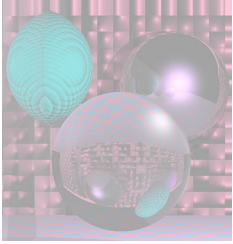


# Vector Length

Vector  $\mathbf{a} = (x_a, y_a)$

$$\text{Length}(\mathbf{a}) = \text{Norm}(\mathbf{a}) = \|\mathbf{a}\| = \sqrt{x_a^2 + y_a^2}$$





# Dot Product

## Dot Product

$$\mathbf{a} = (x_a, y_a) \quad \mathbf{b} = (x_b, y_b)$$

$$\mathbf{a} \cdot \mathbf{b} = x_a x_b + y_a y_b$$

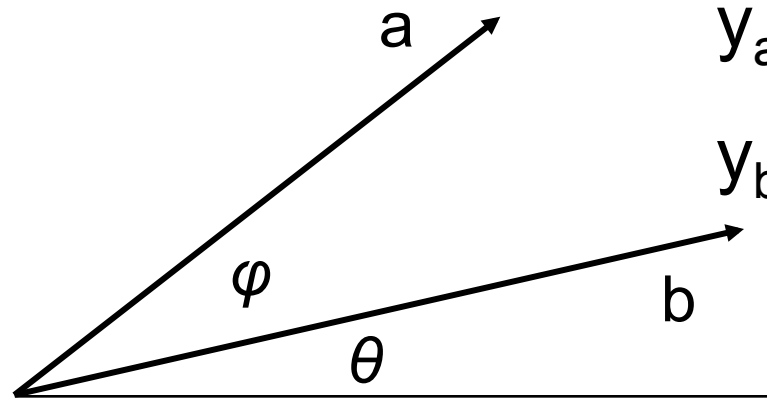
$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \cdot \|\mathbf{b}\| \cos(\varphi)$$

$$x_a = \|\mathbf{a}\| \cos(\theta + \varphi)$$

$$x_b = \|\mathbf{b}\| \cos(\theta)$$

$$y_a = \|\mathbf{a}\| \sin(\theta + \varphi)$$

$$y_b = \|\mathbf{b}\| \sin(\theta)$$





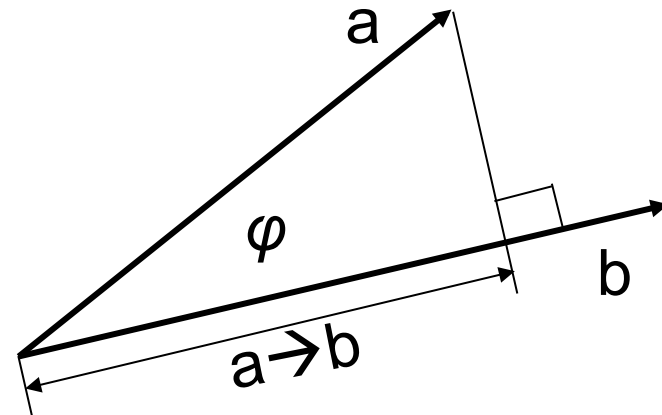
# Projection

$$\mathbf{a} = (x_a, y_a) \quad \mathbf{b} = (x_b, y_b)$$

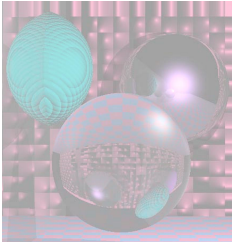
$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \cdot \|\mathbf{b}\| \cos(\varphi)$$

The length of the projection of  $\mathbf{a}$  onto  $\mathbf{b}$  is given by

$$\mathbf{a} \rightarrow \mathbf{b} = \|\mathbf{a}\| \cos(\varphi) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{b}\|}$$







# Output Devices

---

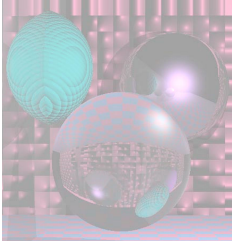
- *a raster is a rectangular array of pixels (picture elements)*
- *common raster output devices include CRT and LCD monitors, ink jet and laser printers*
- *typically considered as top-to-bottom array of left-to-right rows, because that is how CRTs are (were) typically scanned*
- *for this reason, device (e.g. on-screen) coordinate frame typically has origin in upper left, axis aims to right, and axis aims down*



# Device Resolution

---

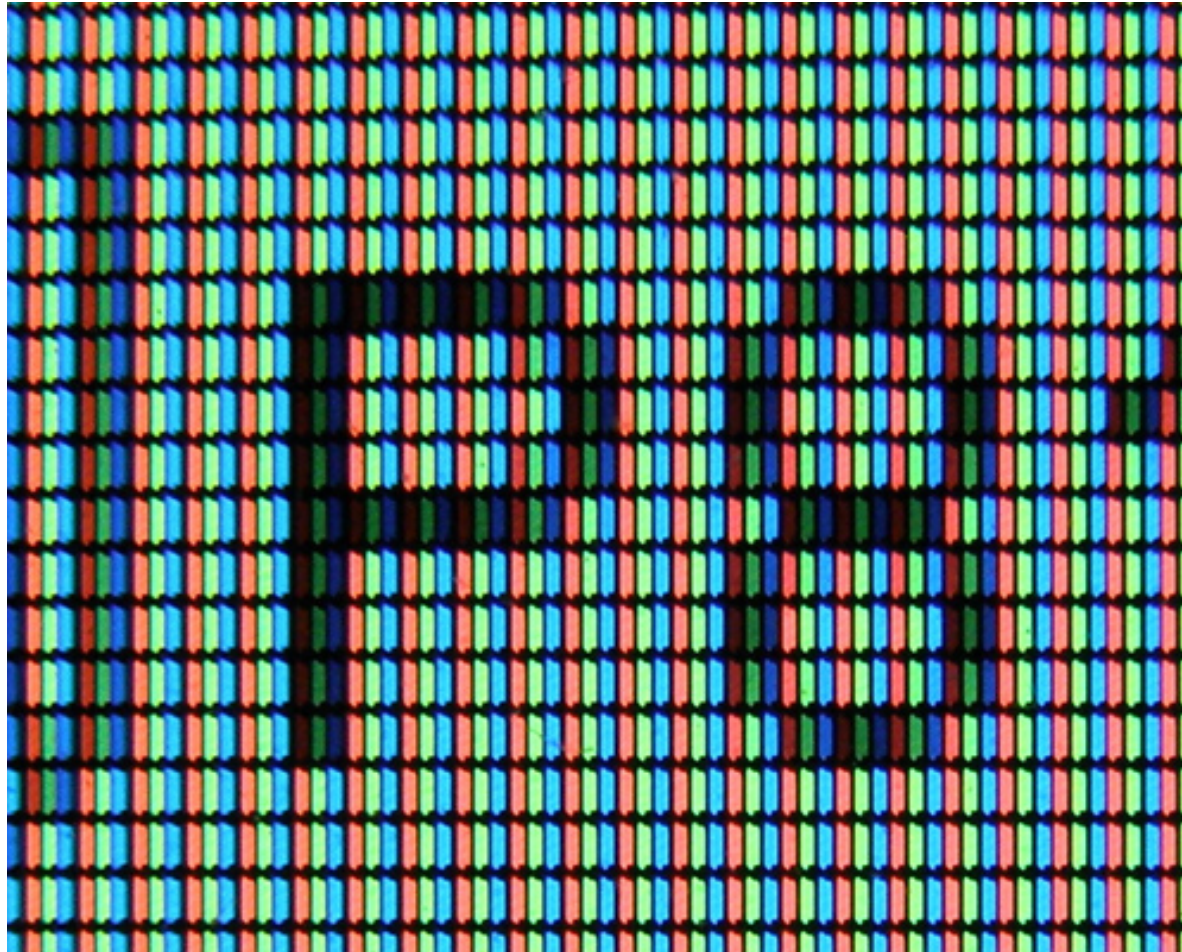
- *(native) resolution of the device is the dimensions (note this is reverse of typical way we write matrix dimensions) of its raster output hardware*
- *typical resolutions for monitors are 640x480 (VGA, the archaic but celebrated Video Graphics Array), 800x600, 1024x768, 1280x1024, 1600x1200, etc*
- *higher resolution is generally “better” because finer detail can be represented*
  - *more computation required for more pixels though, and more pixels makes the display hardware more expensive*
  - *however monitors usually can display lower or higher (within some limits) resolution images than their native resolution by scaling (we will study how to scale images later in the course)*

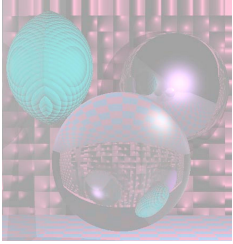


# Sub-pixel Display

<http://en.wikipedia.org/wiki/Pixel>

---





# How are Rasters Represented?

---

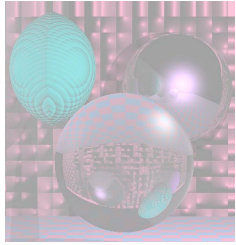
- for a *monochrome image*, each pixel corresponds to one bit (also called a *binary image*)
- typically in graphics we use at least *greyscale images*, where bits are used to represent the intensity at each pixel. The number of gray levels at each pixel is usually a multiple of 8.
- for a *color image*, compose multiple greyscale images, where each corresponds to a different color component. Three images corresponding to red, green, and blue color components are one typical arrangement. The images can be stored as independent planes or they may be interleaved.



# in-memory representation of a raster

---

- monochrome image is typically a linear array of  $r \times c \times \mathcal{B}$  bytes, where  $r$  and  $c$  are the number of rows and columns in the raster, and  $\mathcal{B}$  is the number of bytes per pixel
- value of pixel at location  $(i, j)$  is thus stored in the  $\mathcal{B}$  bytes at memory location  $(ic + j)\mathcal{B}$  relative to the beginning of the array
  - the order of bytes within the pixel value is determined by the *byte order of the computer, which may be little-endian (least significant byte first) or big-endian (most significant byte first)*.
  - *Nowadays, little-endian is more common (e.g. Intel x86). Big-endian may still be encountered on e.g. PowerPC architectures (which is what Apple used in Mac computers up to around 2006).*



# Color Image Representation

---

- for color images, either store as (typically three) separate monochrome rasters (planes), or interleave by packing all color components for a pixel into a contiguous block of memory (interleaved is more common now)
- the order of the color components, as well as the number of bits per component, is called the *pixel format*





# Common Pixel Formats

- *common pixel formats today include*
  - *24-bit RGB ( $b_r = b_g = b_b = 8$ ) (“over 16 million colors!”)*
  - *32-bit RGB (like 24 bit but with one byte of padding)*
  - *16-bit 5:6:5 RGB ( $b_r = 5, b_g = 6, b_b = 5$ ) (human eye is most sensitive to green; common for lower-quality video because it looks ok for images of real-world scenes and uses 2 bytes per pixel, reducing file size)*
- *$(ic + j)\mathcal{B}$  works with*
  - *$\mathcal{B} = \{(b_r + b_g + b_b + padding)\}/8$*
- *byte ordering (little- vs big-endian) only matters within each color component and if some  $b_r > 8$*



# Frame Buffer

---

- *In-memory raster is called a **frame buffer** when hardware is set up so that changes to memory contents drive pixel colors on the display itself. Most modern display hardware has a such a frame buffer.*
  - *in fact, generally more than one, and can switch among them*
  - *a common way to produce a smooth-looking animation is to use two buffers: the front buffer is rendered to the screen, and the back buffer is not*
  - *this is called double buffering*
  - *Each new frame of the animation is drawn onto the back buffer. Because it can take some time (hopefully not too long) to draw, this avoids seeing a “partial frame”.*
  - *once the drawing is complete, the buffers are swapped*





# Rasterization

---

- how to *render images of geometry, say line segments or triangles, onto a raster display?*
- *need to figure out what pixels to “light up” to draw the shape*
- *this is the process of rasterization*
- *will study line segment rasterization now and triangles later in the course*



# Vector Output

---

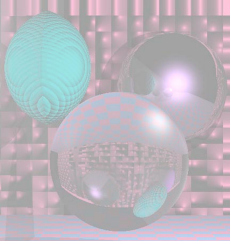
- historically, vector displays were developed first
- a CRT is made to scan line segments by steering an electron beam from start to end of each segment (can generalize to curves)
- potentially more efficient because only need to scan along the actual line segments vs always scanning a raster across the whole screen
- but hard to draw images of real-world scenes, and how to deal with color?
- nowadays, vector output is sometimes still encountered on a *pen plotter*, but even these are mostly antiques



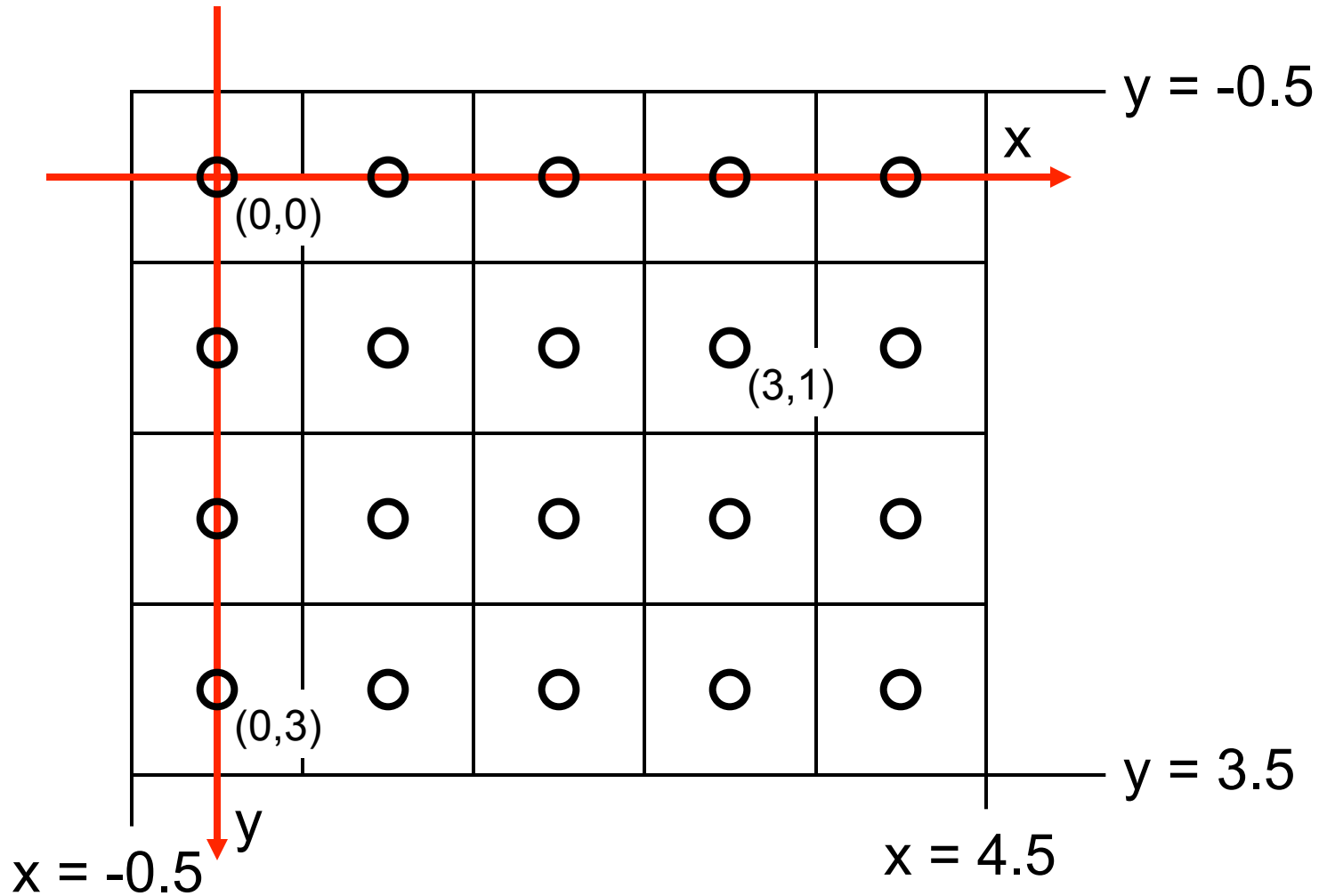
# Vector Representation

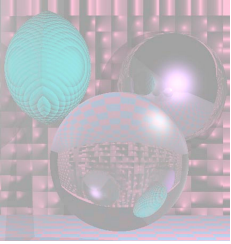
---

- *some software systems represent graphics in a vector form. PostScript, PDF (portable document format), and SVG (scalable vector graphics)*
- *in a vector format, a picture is stored not as an array of pixels, but as a list of instructions about how to draw it*
  - *vector format is “better” for some kinds of images, particularly line drawings and images (e.g. cartoons or computer art)*
- *since the actual geometry, vs a sampling of it, is stored, vector images can generally be scaled to larger or smaller sizes without any loss of quality*
  - *vector images may also require less memory to store, and may be more compressible*

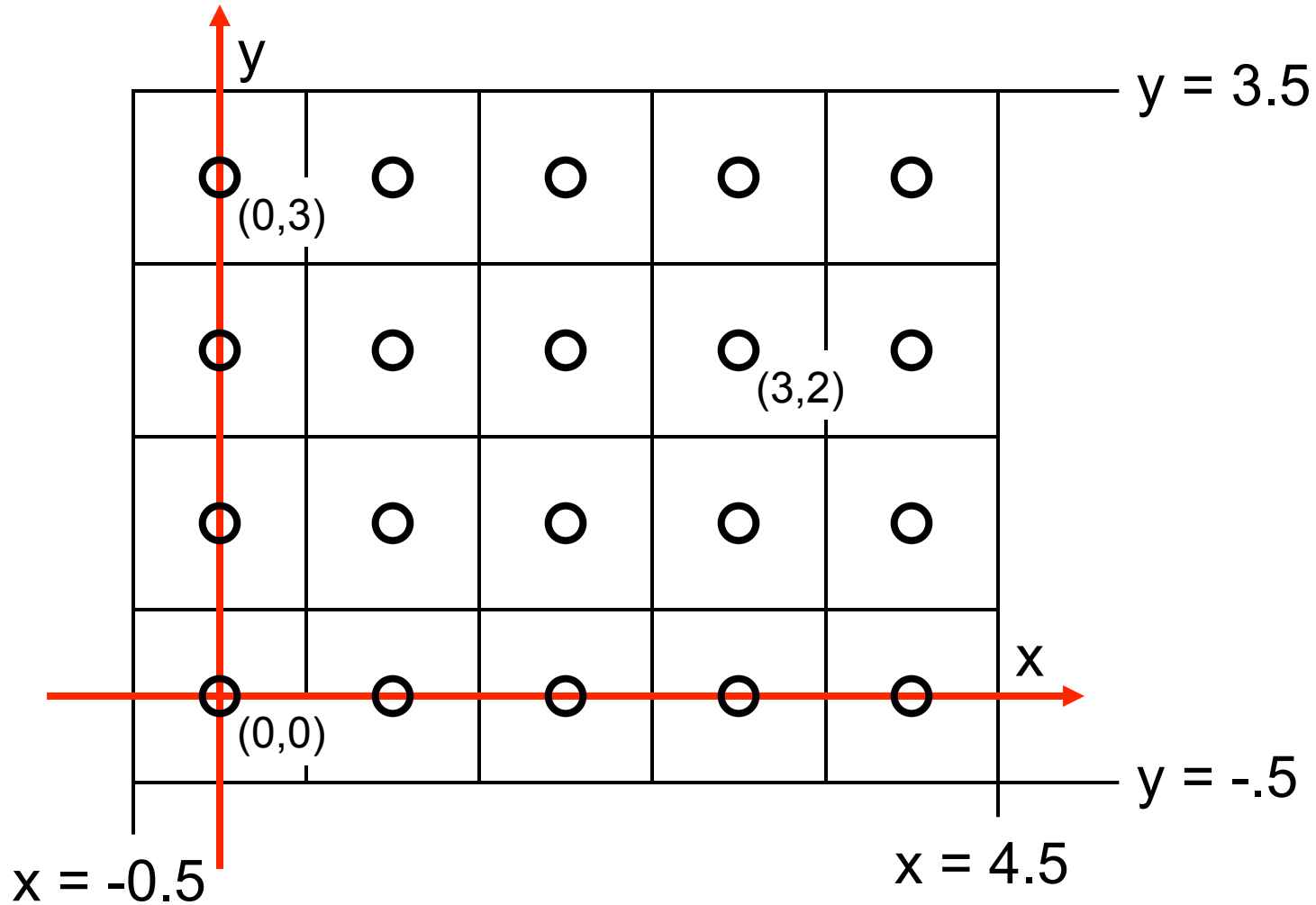


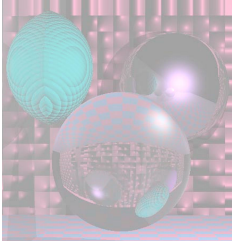
# Pixel Coordinates





# Pixel Coordinates

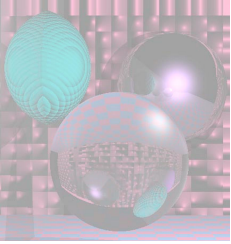




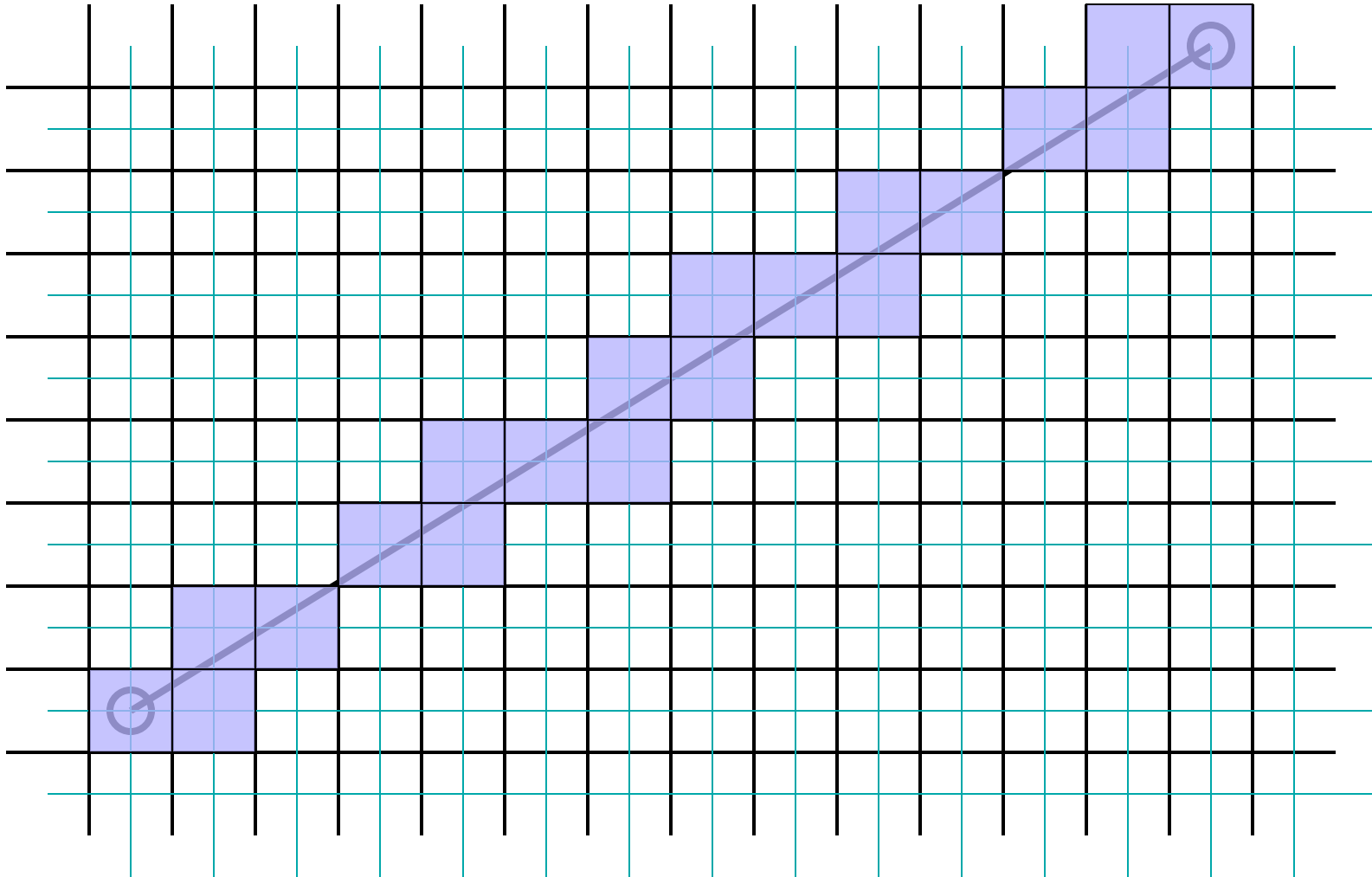
# What Makes a Good Line?

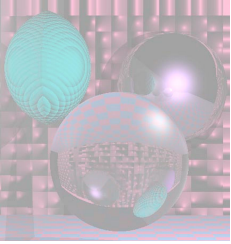
---

- Not too jaggy
- Uniform thickness along a line
- Uniform thickness of lines at different angles
- Symmetry,  $\text{Line}(P,Q) = \text{Line}(Q,P)$
- A good line algorithm should be fast.

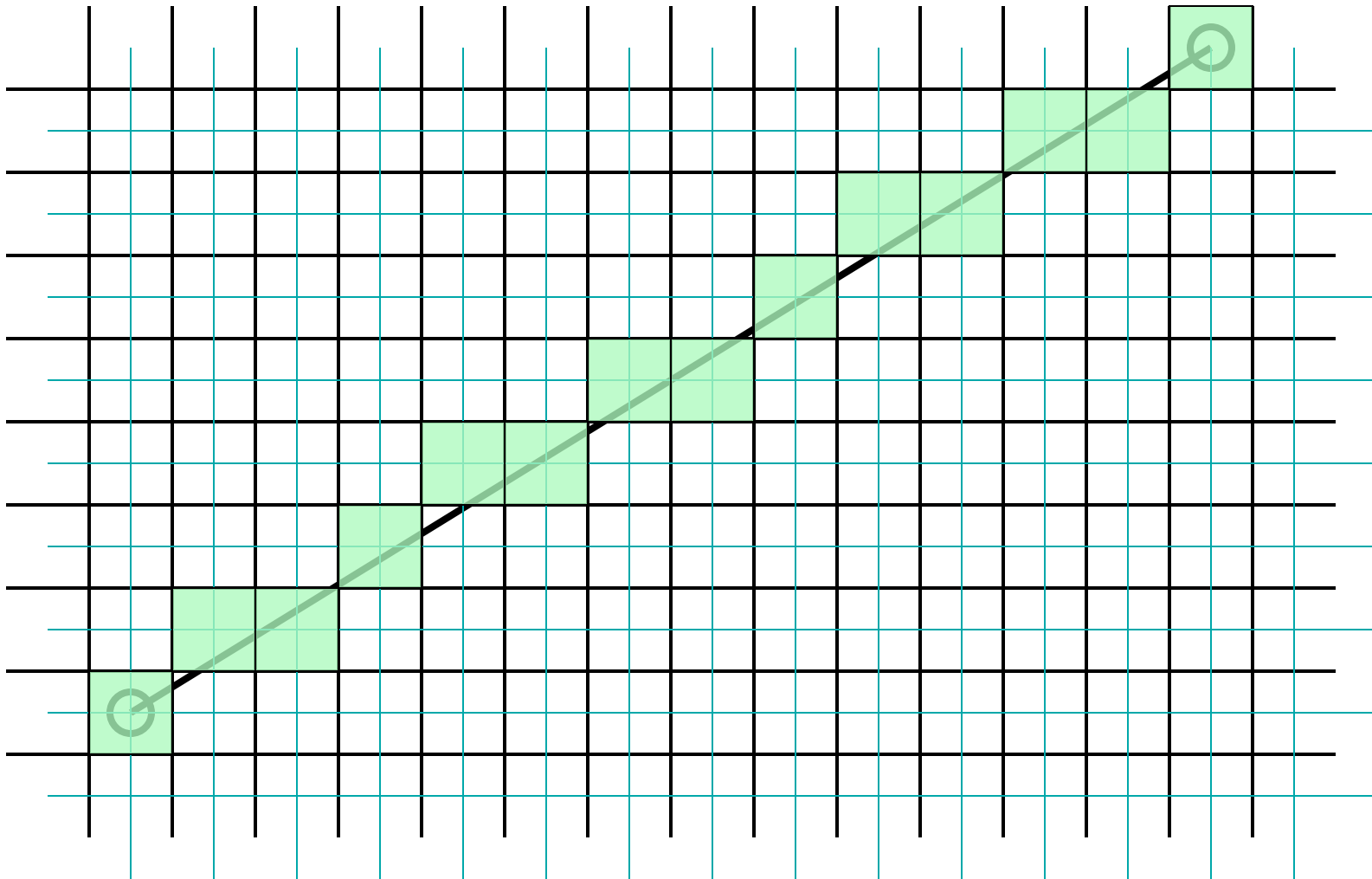


# Line Drawing





# Line Drawing







# Which Pixels Should We Color?

- Given  $P_0 = (x_0, y_0)$ ,  $P_1 = (x_1, y_1)$
- We could use the equation of the line:
  - $y = mx + b$
  - $m = (y_1 - y_0)/(x_1 - x_0)$
  - $b = y_1 - mx_1$
- And a loop
  - for**  $x = x_0$  to  $x_1$       **This calls for real multiplication**
  - $y = mx + b$             **for each pixel**
  - draw* ( $x, y$ )

**This only works if  $x_0 \leq x_1$  and  $|m| \leq 1$ .**



# Midpoint Algorithm

---

- Pitteway 1967
- Van Aiken and Nowak 1985
- Draws the same pixels as *Bresenham Algorithm* 1965.
- Uses integer arithmetic and incremental computation.
- Uses a decision function to decide on the next point
- Draws the thinnest possible line from  $(x_0, y_0)$  to  $(x_1, y_1)$  that has no gaps.
- A diagonal connection between pixels is not a gap.



# Implicit Equation of a Line

$$f(x,y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0 y_1 - x_1 y_0$$

$$f(x,y) > 0$$

$$f(x,y) = 0$$

$$f(x,y) < 0$$

$(x_0, y_0)$

$(x_1, y_1)$

We will assume  $x_0 \leq x_1$   
and that  $m = (y_1 - y_0)/(x_1 - x_0)$   
is in  $[0, 1]$ .



# Basic Form of the Algorithm

$y = y_0$

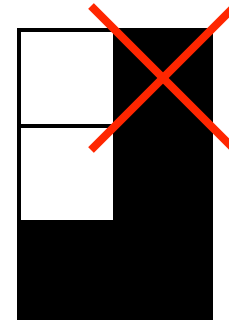
**for**  $x = x_0$  to  $x_1$  **do**

*draw* ( $x, y$ )

**if** (some condition) **then**

$y = y + 1$

We want to compute this condition efficiently.

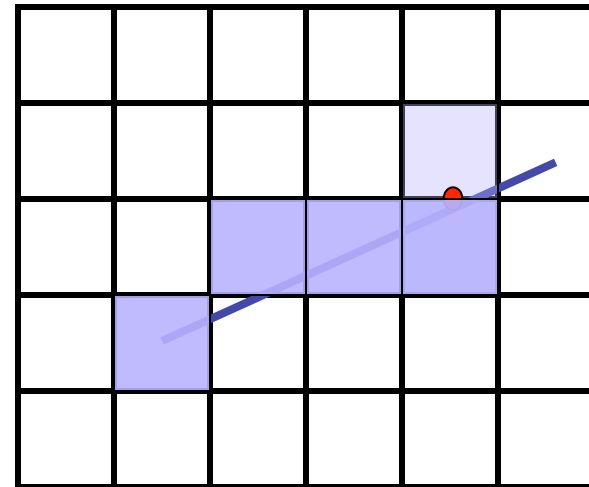
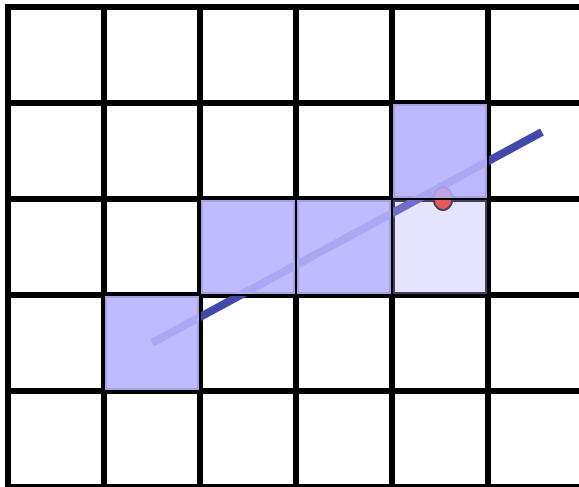


Since  $m$  is in  $[0, 1]$ , as we move from  $x$  to  $x+1$ , the  $y$  value stays the same or goes up by 1.



# Above or Below the Midpoint?

---





# Finding the Next Pixel

---

Assume we just drew  $(x, y)$ .

For the next pixel, we must decide between  
 $(x+1, y)$  and  $(x+1, y+1)$ .

The midpoint between the choices is  
 $(x+1, y+0.5)$ .

If the line passes below  $(x+1, y+0.5)$ , we  
draw the bottom pixel.

Otherwise, we draw the upper pixel.



# The Decision Function

---

```
if  $f(x+1, y+0.5) < 0$ 
```

```
// midpoint below line
```

```
 $y = y + 1$ 
```

$$f(x,y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0 y_1 - x_1 y_0$$

How do we compute  $f(x+1, y+0.5)$

incrementally?

using only integer arithmetic?



# Incremental Computation

$$f(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0 y_1 - x_1 y_0$$

$$f(x + 1, y) = f(x, y) + (y_0 - y_1)$$

$$f(x + 1, y + 1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0)$$

$$y = y_0$$

$$d = f(x_0 + 1, y + 0.5)$$

**for**  $x = x_0$  to  $x_1$  **do**

*draw*  $(x, y)$

**if**  $d < 0$  **then**

$$y = y + 1$$

$$d = d + (y_0 - y_1) + (x_1 - x_0)$$

**else**

$$d = d + (y_0 - y_1)$$





# Integer Decision Function

---

$$f(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0 y_1 - x_1 y_0$$

$$f(x_0 + 1, y + 0.5)$$

$$= (y_0 - y_1)(x_0 + 1) + (x_1 - x_0)(y + 0.5) + x_0 y_1 - x_1 y_0$$

$$2f(x_0 + 1, y + 0.5)$$

$$= 2(y_0 - y_1)(x_0 + 1) + (x_1 - x_0)(2y + 1) + 2x_0 y_1 - 2x_1 y_0$$

$2f(x, y) = 0$  if  $(x, y)$  is on the line.  
 $< 0$  if  $(x, y)$  is below the line.  
 $> 0$  if  $(x, y)$  is above the line.



# Midpoint Line Algorithm

$$y = y_0$$

$$d = 2(y_0 - y_1)(x_0 + 1) + (x_1 - x_0)(2y_0 + 1) + 2x_0 y_1 - 2x_1 y_0$$

**for**  $x = x_0$  **to**  $x_1$  **do**

*draw* ( $x, y$ )

**if**  $d < 0$  **then**

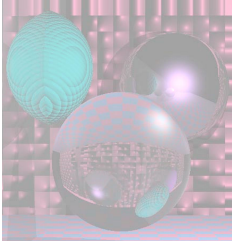
$$y = y + 1$$

$$d = d + 2(y_0 - y_1) + 2(x_1 - x_0)$$

**else**

$$d = d + 2(y_0 - y_1)$$

These are constants  
and can be computed  
before the loop.

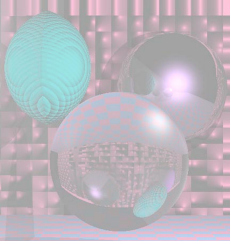


# Line Attributes

---

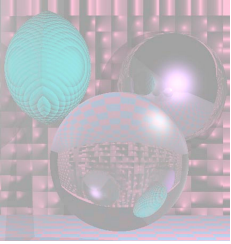
- line width
- dash patterns
- end caps: butt, round, square



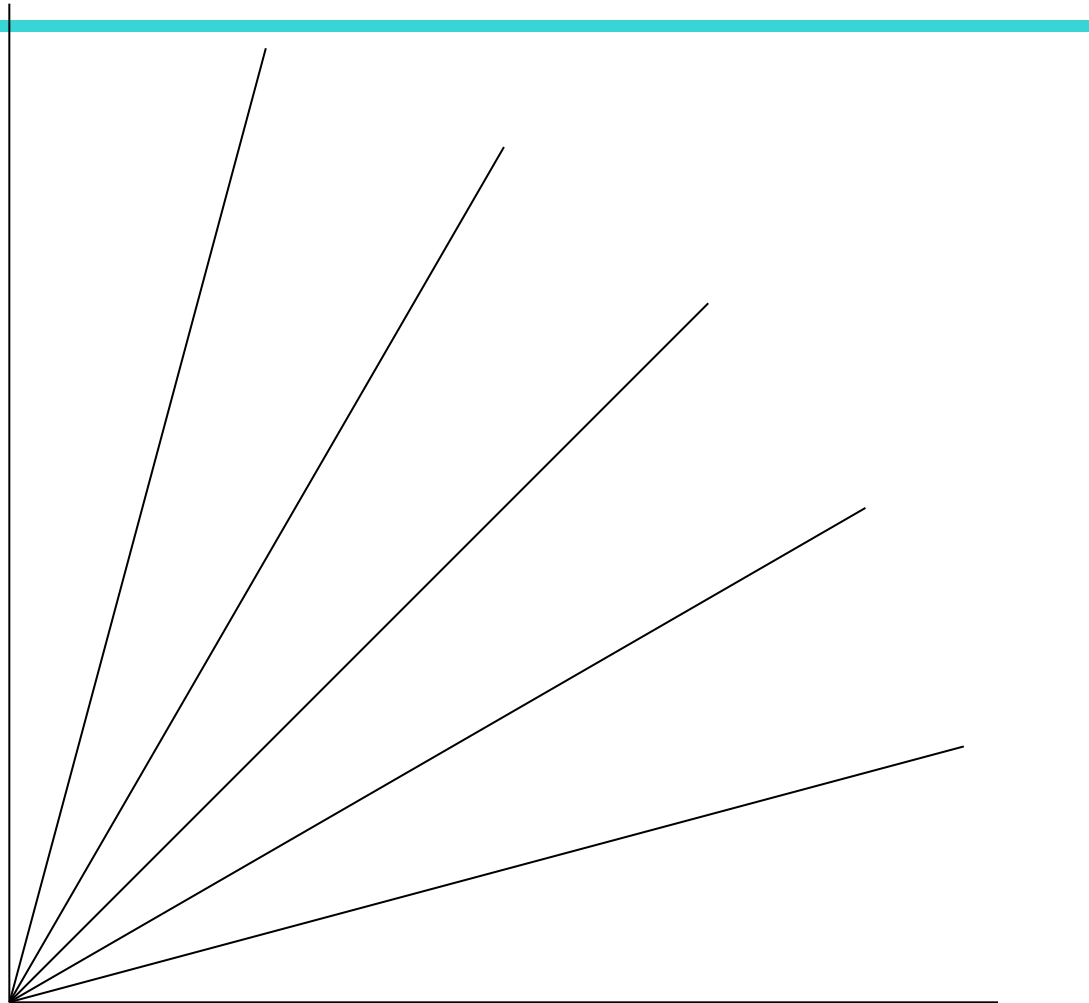


# Joins: round, bevel, miter



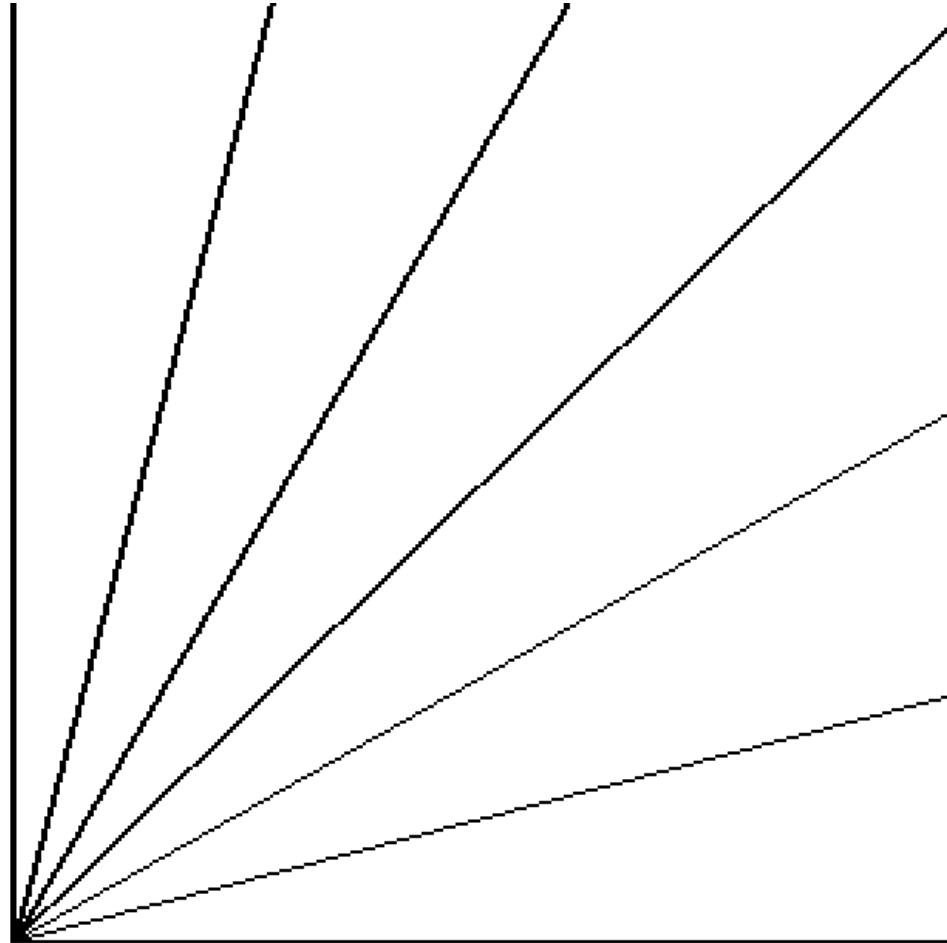


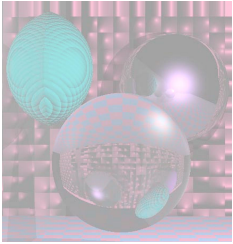
# Some Lines





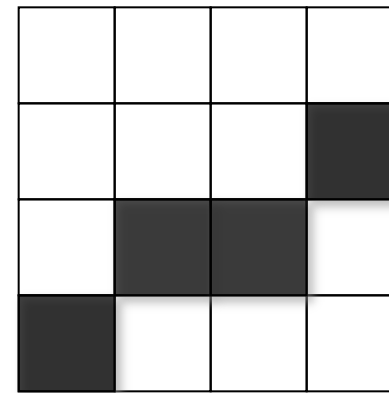
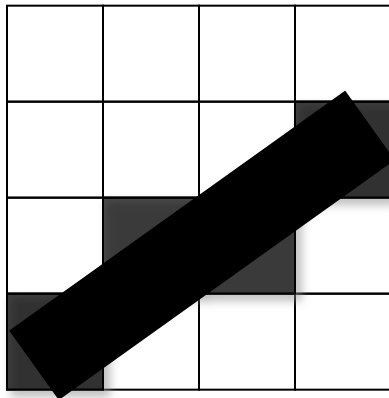
# Some Lines Magnified





# Antialiasing by Downsampling

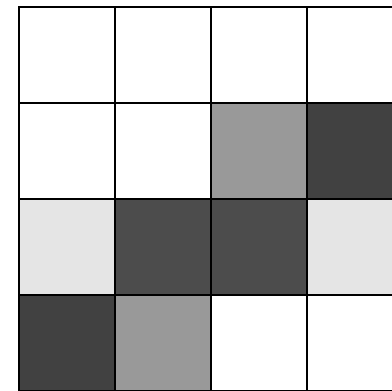
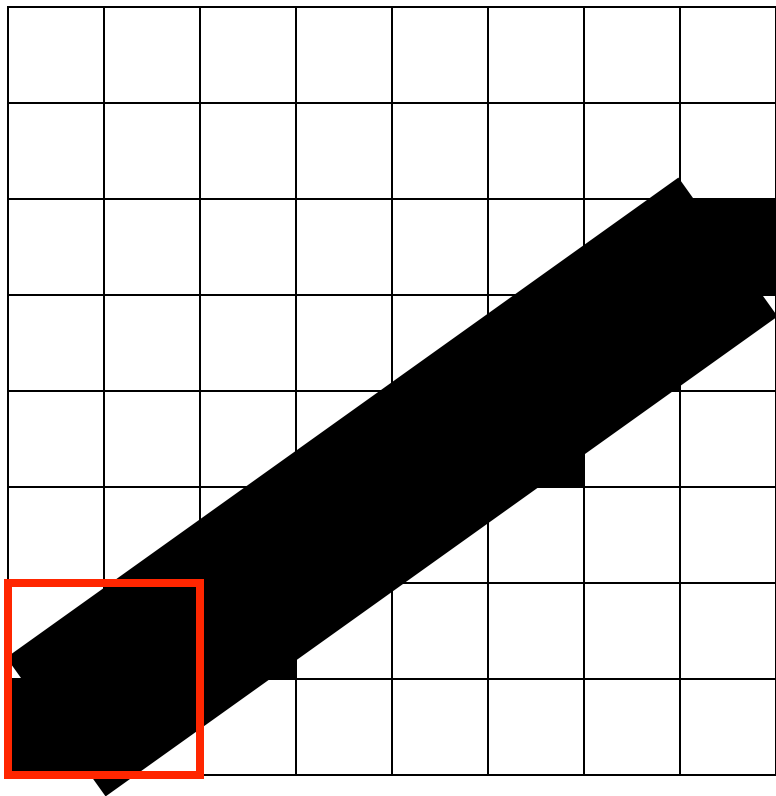
---





# Antialiasing by Downsampling

---







# Antialiasing by Downsampling

---

