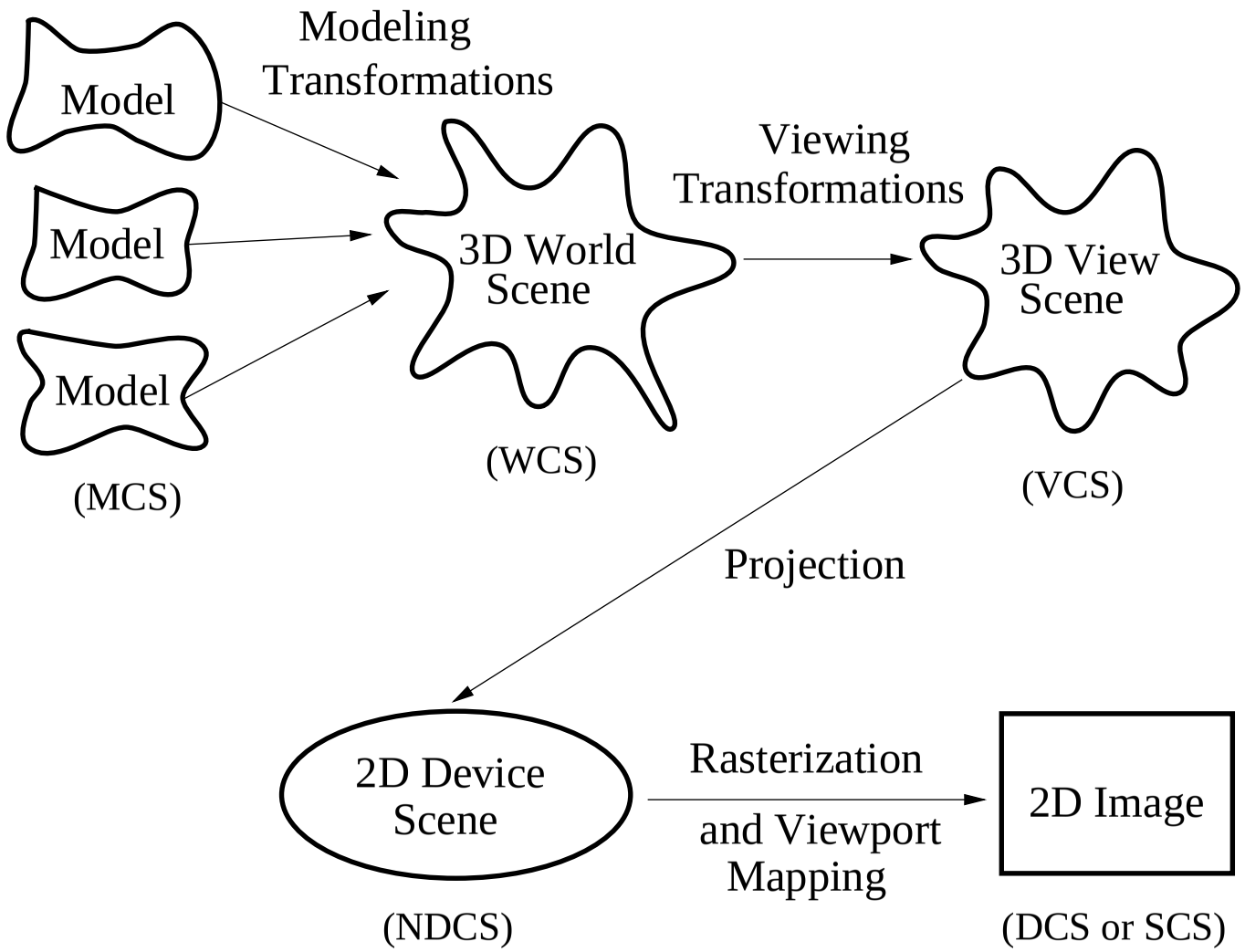
Graphics Rendering Pipeline



- Coordinate Systems
  - MCS: Modeling Coordinate System
  - WCS: World Coordinate System
  - VCS: Viewer Coordinate System
  - NDCS: Normalized Device Coordinate System
  - DCS or SCS: Device Coordinate System or, equivalently, Screen Coordinate System

  Keeping the coordinate systems straight is an important key to understanding a rendering system.
- Pipeline stages: Refine the scene step by step:
  - Convert primitives in the MCS to primitives in the DCS.
  - Add derived information: shading, texture, shadows.
  - Remove invisible primitives.
  - Convert primitives in the DCS to *pixels* in a *raster image*.
- Transformations: Coordinate system conversions can be represented with matrix-vector multiplications.

Rendering Primitives

Models are typically composed of a large number of *geometric primitives*. The *only* rendering primitives typically supported in hardware are

- Points (single pixels)
- Line segments
- Polygons (usually restricted to *convex polygons*).

Modeling primitives include these, but also

- Piecewise polynomial (spline) curves
- Piecewise polynomial (spline) surfaces
- Implicit surfaces (quadrics, blobbies, etc)
- Other...

A software renderer may support these modeling primitives directly, or they may be converted into polygonal or linear approximations for hardware rendering.

Algorithms

A number of basic algorithms are needed:

- Transformation: convert representations of primitives from one coordinate system to another.
- Clipping/Hidden Surface Removal: Remove primitives and parts of primitives that are not visible on the display.
- Rasterization: Convert a projected screen-space primitive to a set of pixels.

Later, we will look at some more advanced algorithms:

- Picking: Select a 3D object by clicking an input device over a pixel location.
- Shading and Illumination: Simulate the interaction of light with a scene.
- Texturing and Environment Mapping: Enhancing the realism
- Animation: simulate movement by rendering a sequence of frames.

Application Programming Interfaces

- Application Programming Interfaces (APIs) provide access to rendering hardware:
  - Xlib: 2D rasterization.
  - PostScript: 2D transformations, 2D rasterization
  - Phigs+, GL, OpenGL: 3D pipeline
- APIs hide which parts of the rendering are actually implemented in hardware by simulating the missing pieces in software, usually at a loss in performance.
- For 3D interactive applications, we might modify the scene or a model directly or just the viewing information.
- After each modification, usually the images needs to be regenerated.
- We need to consider how to interface to input devices in an asynchronous and device independent fashion. APIs have also been defined for this task; we will be using X11 through Glut, OpenGL optimizer, GTwidgets

Device Independence

In this module, we

- Consider display devices for computer graphics:
  - calligraphic devices
  - raster devices
  - CRTs
  - direct vs. pseudocolor frame buffers
- Discuss the problem of device independence:
  - window-to-viewport mapping
  - normalized device coordinates

Calligraphic and Raster Devices

- Calligraphic display devices draw polygon and line segments directly:
  - plotters
  - direct beam control CRTs
  - laser light projection systems
- Raster display devices represent an image as a regular grid of *samples*.
  - Each sample is usually called a *pixel* or, less commonly, a *pel*.
  - Both are short for *picture element*.
  - Rendering requires *rasterization algorithms* to quickly determine a sampled representation of geometric primitives.

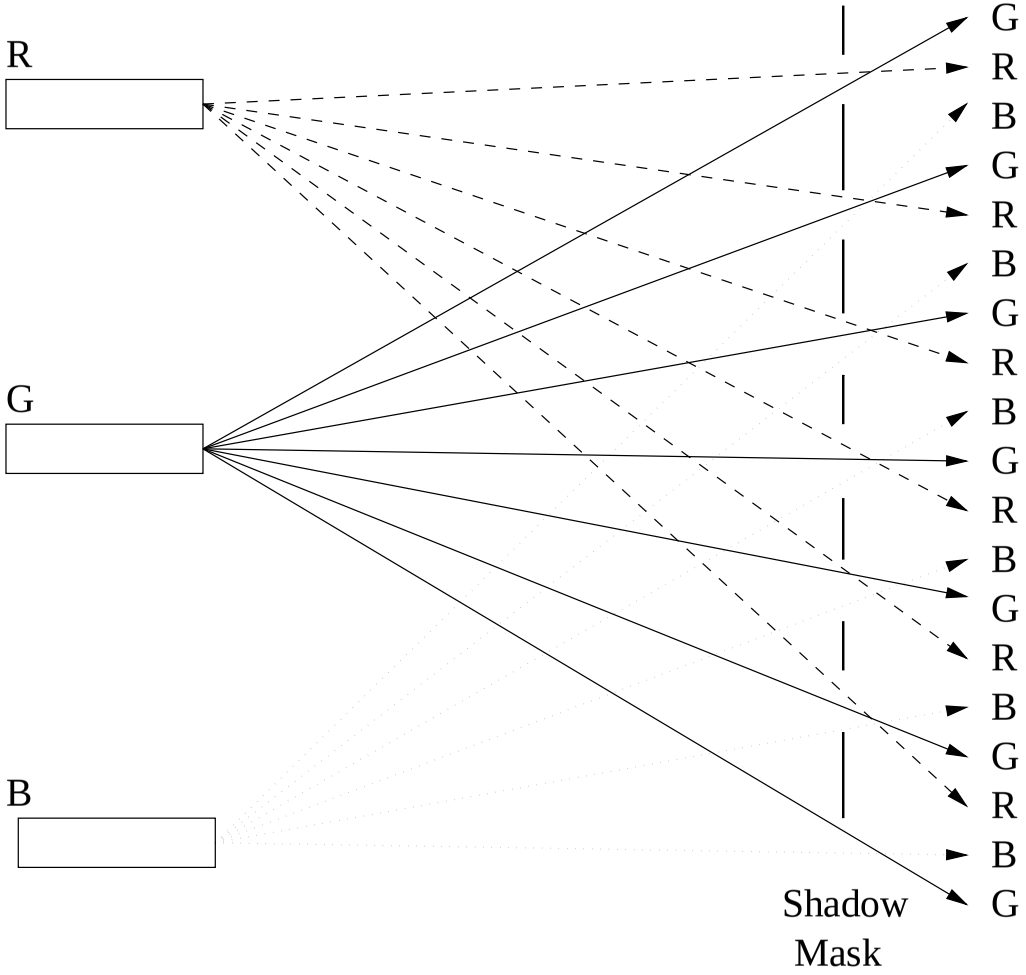
How a Monitor Works

- Raster Cathode Ray Tubes (CRTs) are the most common display device today.
  - capable of high resolution
  - good color fidelity
  - high contrast (100:1)
  - high update rates

  An electron beam is continually scanned in a regular pattern of horizontal *scanlines*.
- *Raster images* are stored in a *frame buffer*.
- *Frame buffers* are composed of *VRAM* (video RAM).
- VRAM is dual-ported memory capable of
  - Random access.
  - Simultaneous high-speed serial output: A built-in *serial shift register* can output an entire scanline at a high rate synchronized to a *pixel clock*.

  At each pixel location in a scanline, the intensity of the electron beam is modified by the pixel value being shifted synchronously out of the VRAM.

- Color CRTs have three different colors of phosphor and three independent electron guns.
- *Shadow masks* only allow each gun to irradiate one color of phosphor.

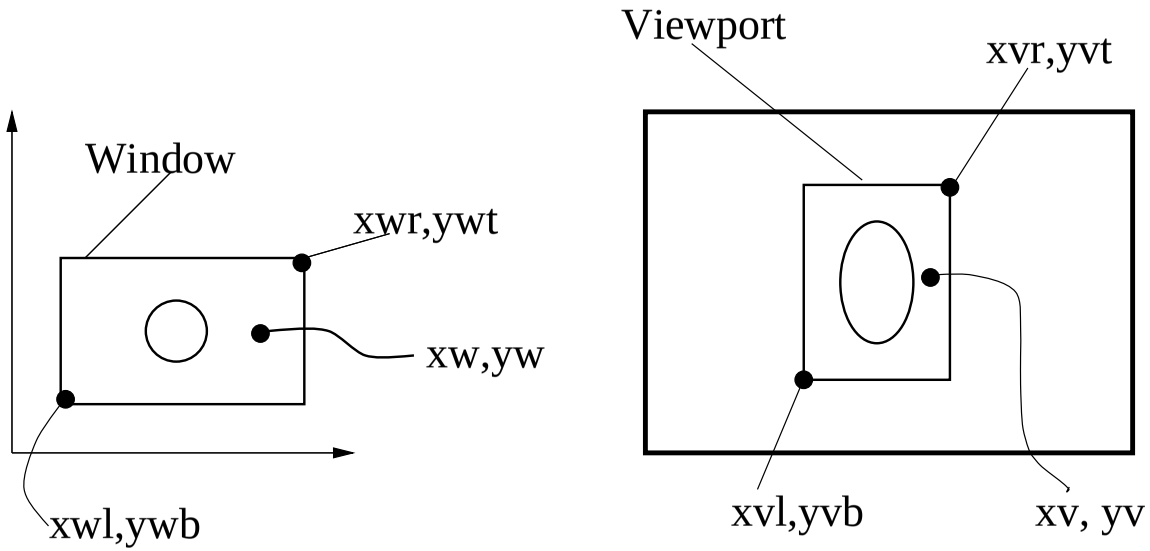


- Color is specified either

– directly, using three independent intensity channels, or
– indirectly, using a *color lookup table* (LUT).
  In the latter case, a *color index* is stored in the frame buffer.

Window to Viewport Mapping

- Start with 3D scene, but eventually project to 2D scene.
- 2D scene is infinite plane. Device has a finite visible rectangle. What do we do?
- Answer: map rectangular region of 2D device scene to device.
  - Window: rectangular region of interest in scene.
  - Viewport: rectangular region on device.
  - Usually, both rectangles are aligned with the coordinate axes.



- Window point (x_w, Y_w) maps to viewport point (x_v, y_v).
  - Window has corners (x_{wl}, y_{wb}) and (x_{wr}, y_{wt});
    Viewport has corners (x_{vl}, y_{vb}) and (x_{vr}, y_{vt});
  - Length and height of the window are L_w and H_w
    Length and height of the viewport are L_v and H_v
- Proportionally map each of the coordinates according to:

$$\frac{\Delta x_w}{L_w} = \frac{\Delta x_v}{L_v}$$

$$\frac{\Delta y_w}{H_w} = \frac{\Delta y_v}{H_v}$$

- To map x_w to x_v:

$$\frac{x_w - x_{wl}}{L_w} = \frac{x_v - x_{vl}}{L_v}$$

$$\Rightarrow x_v = \frac{L_v}{L_w}(x_w - x_{wl}) + x_{vl}$$

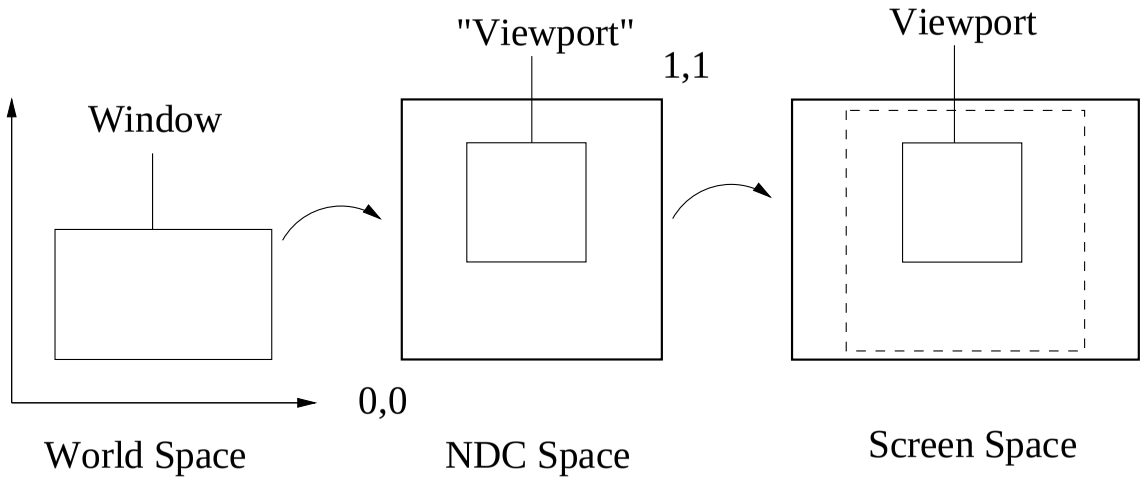
  and similarly for y_v.

- If $H_w/L_w \neq H_v/L_v$ the image will be distorted.
  These quantities are called the *aspect ratios* of the window and viewport.

Normalized Device Coordinates

- Where do we specify our viewport?
- Could specify it in device coordinates, BUT, suppose we want to run our program on several different hardware platforms or on different graphic devices.
- Two common conventions for DCS:
  - Origin in the lower left corner, with x to the right and y upward.
  - Origin in the top left corner, with x to the right and y downward.
- Many different resolutions for graphics display devices:
  - Workstations commonly have 1280 $\times$ 1024 frame buffers.
  - A PostScript page is 612 $\times$ 792 points, but 2550 $\times$ 3300 pixels at 300dpi.
  - And so on...
- Aspect ratios may vary...

- If we map directly from WCS to a DCS, then changing our device requires rewriting this mapping (among other changes).
- Instead, use Normalized Device Coordinates (NDC) as an intermediate coordinate system that gets mapped to the device layer.
- Will consider using only a square portion of the device.
  Windows in WCS will be mapped to viewports that are specified within a unit square in NDC space.
- Map viewports from NDC coordinates to the screen.



Pixels

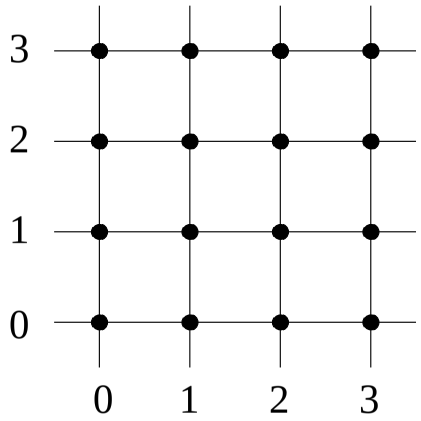
- *Pixel:* Intensity or color sample.
- *Raster Image:* Rectangular grid of pixels.
- *Rasterization:* Conversion of a primitive's geometric representation into
  - A set of pixels.
  - An intensity or color for each pixel (*shading, antialiasing*).
- For now, we will assume that the background is white and we need only change the color of selected pixels to black.

Pixel Grids

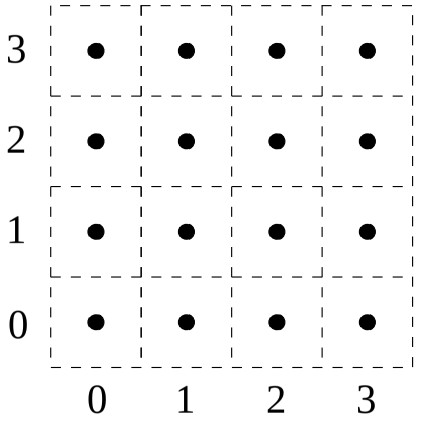
- *Pixel Centers:* Address pixels by integer coordinates (i, j)
- *Pixel Center Grid:* Set of lines passing through pixel centers.
- *Pixel Domains:* Rectangular semi-open areas surrounding each pixel center:

$$P_{i,j} = (i - 1/2, i + 1/2) \times (j - 1/2, j + 1/2)$$

- *Pixel Domain Grid:* Set of lines formed by domain boundaries.



Pixel center grid



Pixel domain grid

Specifications and Representations

Each rendering primitive (point, line segment, polygon, etc.) needs both

- A geometric specification, usually "calligraphic."
- A pixel (rasterized) representation.

Standard device-level geometric specifications include:

Point: $A = (x_A, y_A) \in \mathbb{R}^2$.

Line Segment: $\ell(AB)$ specified with two points, A and B. The line segment $\ell(AB)$ is the set of all collinear points between point A and point B.

Polygon: Polygon $\mathcal{P}(A_1 A_2 \ldots A_n)$ specified with an ordered list of points $A_1 A_2 \ldots A_n$. A polygon is a region of the plane with a piecewise linear boundary; we connect A_n to A_1.

This "list of points" specification is flawed... a more precise definition will be given later.

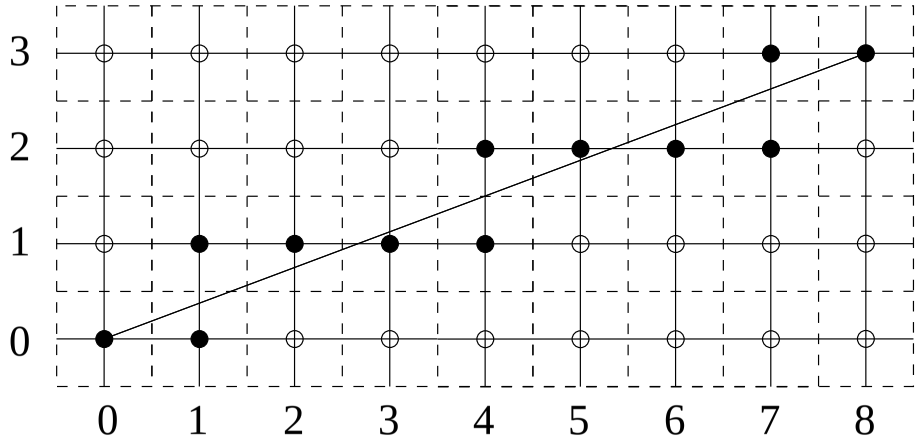
Line Segments

- Let $\ell(AB) = \{P \in \mathbf{R}^2 | P = (1-t)A + tB, t \in [0,1]\}$
- Problem: Given a line segment $\ell(AB)$ specified by two points A and B,
- Decide: Which pixels to illuminate to represent $\ell(AB)$.
- Desired properties: Rasterization of line segment should
  1. Appear as straight as possible;
  2. Include pixels whose domains contain A and B;
  3. Have relatively constant intensity (i.e., all parts should be the same brightness);
  4. Have an intensity per unit length that is independent of slope;
  5. Be symmetric;
  6. Be generated efficiently.

Line Segment Representations

1. Given AB, choose a set of pixels $L_1(AB)$ given by

$$L_1(AB) = \{(i, j) \in \mathbb{Z}^2 | \ell(AB) \cap P_{i,j}\}$$

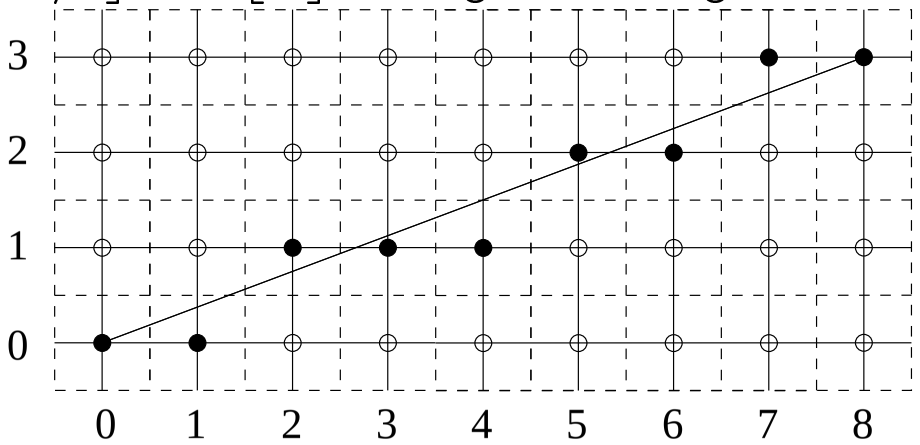


Unfortunately, this results in a very blotchy, uneven looking line.

2. Given AB, choose a set of pixels $L_2(AB)$ given by

$$L_2(AB) = \begin{cases} |x_B - x_A| \geq |y_B - y_A| \longrightarrow \\ \quad \{(i,j) \in \mathbb{Z}^2 | (i,j) = (i,[y]), (i,y) \in \ell(AB), y \in \mathbb{R}\} \\ \quad \cup([x_A],[y_A]) \cup ([x_B],[y_B]). \\ |x_B - x_A| < |y_B - y_A| \longrightarrow \\ \quad \{(i,j) \in \mathbb{Z}^2 | (i,j) = ([x],j), (x,j), \in \ell(AB), x \in \mathbb{R}\} \\ \quad \cup([x_A],[y_A]) \cup ([x_B],[y_B]). \end{cases}$$

Where $[z] = \lfloor z + 1/2 \rfloor$, and $\lfloor w \rfloor$ is the greatest integer less than or equal to w.



Line Equation Algorithm

Based on the line equation $y = mx + b$, we can derive:

```
LineEquation (int xA, yA, xB, yB)
    float m, b;
    int xi, dx;

    m = (yB - yA)/(xB - xA);
    b = yA - m*xA;
    if ( xB - xA > 0 ) then dx=1;
                       else dx = -1;
    for xi = xA to xB step dx do
         y = m*xi + b;
         WritePixel( xi, [y] );
endfor
```

Problems:

- One pixel per column so lines of slope > 1 have gaps

- Vertical lines cause divide by zero

To fix these problems, we need to use $x = m^{-1}(y - b)$ when $m > 1$.

Discrete Differential Analyzer

Observation: Roles of x and y are symmetric...

- Change roles of x and y if $|y_B - y_A| > |x_B - x_A|$

Observation: Multiplication of m inside loop...

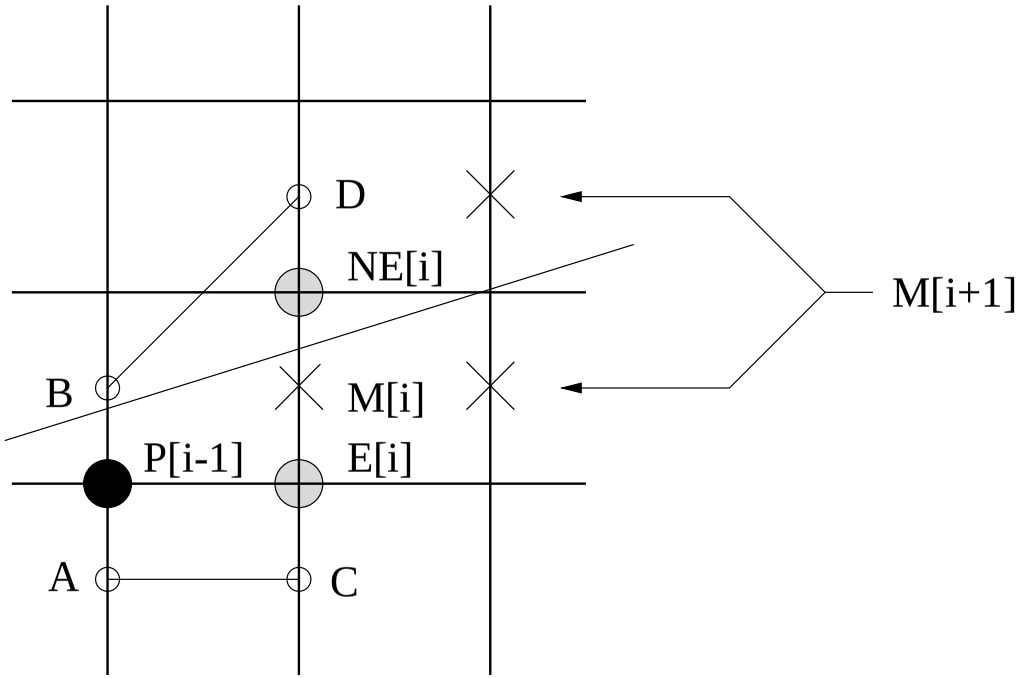
- The value of m is constant for all iterations.
- Can reduce computations inside loop:
  y_{i+1} and be computed incrementally from y_i

$$y_{i+1} = m(x_i + 1) + b = y_i + m$$

```
DDA (int xA, yA, xB, yB)
    int length, dx, dy, i;
    float x,y,xinc,yinc;

    dx = xB - xA;
    dy = yB - yA;

    length = max ( |dx| > |dy| );

    xinc = dx/length; # either xinc or yinc is -1 or 1
    yinc = dy/length;

    x = xA; y = yA;
    for i=0 to length do
        WritePixel( [x], [y] );
        x += xinc;
        y += yinc;
    endfor
```

Bresenham's Algorithm

- Completely integer;
- Will assume (at first) that x_A, y_A, x_B, y_B are also integer.
- Only addition, subtraction, and shift in inner loop.
- Originally for a pen plotter.
- "Optimal" in that it picks pixels closest to line, i.e., $L_2(AB)$.
- Assumes $0 \leq (y_B - y_A) \leq (x_B - x_A) \leq 1$ (i.e., slopes between 0 and 1).
- Use reflections and endpoint reversal to get other slopes: 8 cases.



- Suppose we know at step $i-1$ that pixel $(x_i, y_i) = P_{i-1}$ was chosen.
  Thus, the line passed between points A and B.
- Slope between 0 and 1 $\Rightarrow$
  line must pass between points C and D at next step $\Rightarrow$
  $E_i = (x_i + 1, y_i)$ and $N\ E_i = (x_i + 1, y_i + 1)$ are only choices for next pixel.
- If M_i above line, choose E_i;

- If M_i below line, choose N E_i.

- Implicit representations for line:

$$y = \frac{\Delta y}{\Delta x}x + b$$
$$F(x,y) = \underbrace{(2\Delta y)}_{Q}\,x + \underbrace{(-2\Delta x)}_{R}\,y + \underbrace{2\Delta x b}_{S} = 0$$

  where

$$\begin{aligned} \Delta x &= x_B - x_A \\ \Delta y &= y_B - y_A \\ b &= y_A - \frac{\Delta y}{\Delta x}x_A \Rightarrow S = 2\Delta x y_A - 2\Delta y x_A \end{aligned}$$

  Note that

  1. $F(x,y) < 0 \Rightarrow (x,y)$ *above* line.
  2. $F(x,y) > 0 \Rightarrow (x,y)$ *below* line.
  3. Q, R, S are all integers.

- The mystery factor of 2 will be explained later.

- Look at $F(M_i)$. Remember, F is 0 if the point is on the line:
  - $F(M_i) < 0 \Rightarrow M_i$ *above* line $\Rightarrow$ choose $P_i = E_i$.
  - $F(M_i) > 0 \Rightarrow M_i$ *below* line $\Rightarrow$ choose $P_i = N\ E_i$.
  - $F(M_i) = 0 \Rightarrow$ arbitrary choice, consider choice of pixel domains...
- We'll use $d_i = F(M_i)$ as an *decision variable*.
- Can compute d_i incrementally with integer arithmetic.

- At each step of algorithm, we *know* P_{i-1} and d_i...
- Want to *choose* P_i and *compute* d_{i+1}
- Note that

$$\begin{aligned} d_i &= F(M_i) = F(x_{i-1} + 1, y_{i-1} + 1/2) \\ &= Q \cdot (x_{i-1} + 1) + R \cdot (y_{i-1} + 1/2) + S \end{aligned}$$

- If E_i is chosen then

$$\begin{aligned} d_{i+1} &= F(x_{i-1} + 2, y_{i-1} + 1/2) \\ &= Q \cdot (x_{i-1} + 2) + R \cdot (y_{i-1} + 1/2) + S \\ &= d_i + Q \end{aligned}$$

- If N E_i is chosen then

$$\begin{aligned} d_{i+1} &= F(x_{i-1} + 2, y_{i-1} + 1/2 + 1) \\ &= Q \cdot (x_{i-1} + 2) + R \cdot (y_{i-1} + 1/2 + 1) + S \\ &= d_i + Q + R \end{aligned}$$

- Initially, we have

$$\begin{aligned} d_1 &= F(x_A + 1, y_A + 1/2) \\ &= Q_{x_A} + R_{y_A} + S + Q + R/2 \\ &= F(x_A, y_A) + Q + R/2 \\ &= Q + R/2 \end{aligned}$$

- Note that $F(x_A, y_A) = 0$ since $(x_A, y_A) \in \ell(AB)$.
- Why the mysterious factor of 2?
  It makes everything integer.

```
Bresenham (int xA, yA, xB, yB)
    int d, dx, dy, xi, yi
    int incE, incNE

    dx = xB - xA
    dy = yB - yA
    incE = dy<<1                    /* Q */
    incNE = incE - dx<<1;           /* Q + R */
    d = incE - dx                   /* Q + R/2 */
    xi = xA; yi = yA
    WritePixel( xi, yi )
    while ( xi < xB )
        xi++
        if ( d < 0 ) then /* choose E */
             d += incE
        else /* choose NE */
             d += incNE
             yi++
        endif
```

```
    WritePixel( xi, yi )
endwhile
```

- Some asymmetries (choice when $==$).
- Did we meet our goals?
  1. Straight as possible: yes, but depends on metric.
  2. Correct termination.
  3. Even distribution of intensity: yes, more or less, but:
  4. Intensity varies as function of slope.
     - Can't do better without gray scale.
     - Worst case: diagonal compared to horizontal (same number of pixels, but $\sqrt{2}$ longer line).
  5. Careful coding required to achieve some form of symmetry.
  6. Fast! (if integer math fast ...)
- Interaction with clipping?
- Subpixel positioning of endpoints?
- Variations that look ahead more than one pixel at once...
- Variations that compute from both end of the line at once...
- Similar algorithms for circles, ellipses, ...
  (8 fold symmetry for circles)