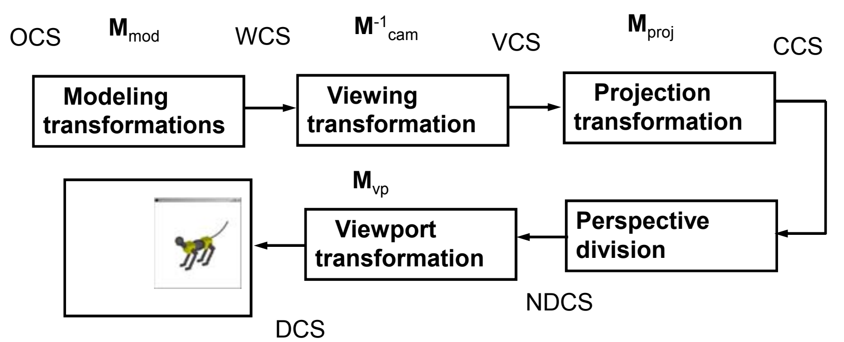
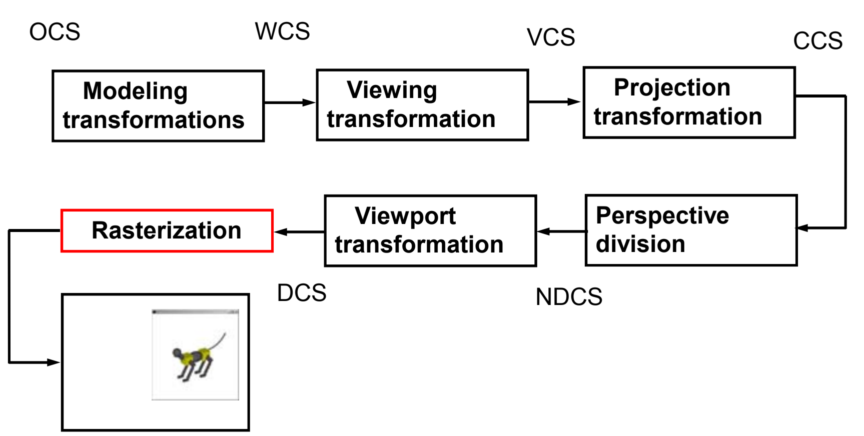
Reminder:
Z-Buffer Graphics Pipeline



Z-Buffer Graphics Pipeline



Line Rasterization
Reminder: Line Rendering Algorithm

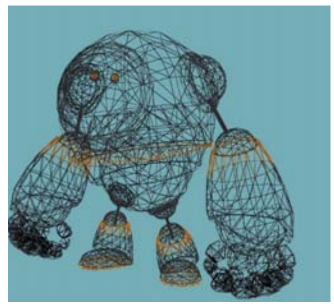
Compute $\mathbf{M} = \mathbf{M}_{vp}\,\mathbf{M}_{proj}\,\mathbf{M}^{-1}{}_{cam}\,\mathbf{M}_{mod}$

for each line segment *i* between points P_i and Q_i **do**

$P = \mathbf{M}P_i$; $Q = \mathbf{M}Q_i$ // w_P, w_Q are 4th coords of P, Q

drawline(P_x/w_P, P_y/w_P, Q_x/w_Q, Q_y/w_Q)

end for



Line Rasterization

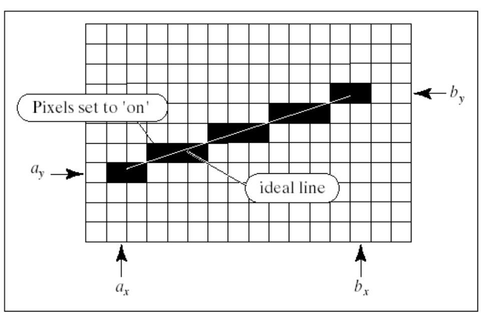


FIGURE 10.23 Drawing a straight-line-segment.

Line Rasterization

Desired properties

- Straight
- Pass through end points
- Smooth
- Independent of end point order
- Uniform brightness
- Brightness independent of slope
- Efficiency!

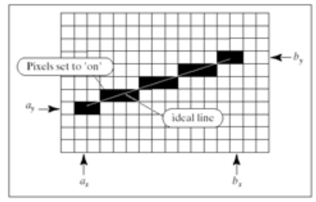
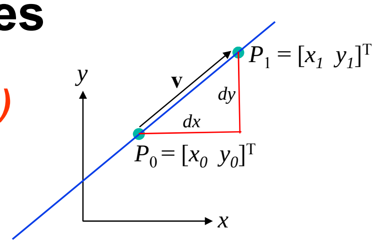


FIGURE 10.23 Drawing a straight-line-segment.

Reminder: Lines



Representations of a line (in 2D)

- Explicit $y = \alpha x + \beta$

  $$y = m(x - x_0) + y_0; \qquad m = \frac{dy}{dx} = \frac{y_1 - y_0}{x_1 - x_0}$$

- Implicit $f(x,y) = (x - x_0)dy - (y - y_0)dx$

  if $f(x,y) = 0$ then (x,y) is **on** the line

  $f(x,y) > 0$ then (x,y) is **below** the line

  $f(x,y) < 0$ then (x,y) is **above** the line

- Parametric $x(t) = x_0 + t(x_1 - x_0)$

  $y(t) = y_0 + t(y_1 - y_0)$

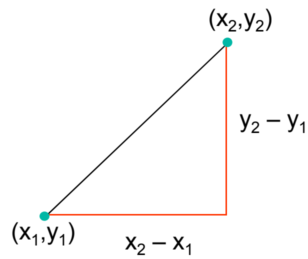
  $t \in [0,1]$ for line segment, or $t \in [-\infty, \infty]$ for infinite line

  $P(t) = P_0 + t(P_1 - P_0)$ or $P(t) = P_0 + t\,\mathbf{v}$

  $P(t) = (1 - t)P_0 + tP_1$

Straightforward Implementation

Line between two points




```
DrawLine(int x1,int y1, int x2,int y2)
  {
       int x;
       float y;
       for (x=x1; x<=x2; x++) {
             y = y1 +  (x-x1)*(y2-y1)/(x2-x1)
             SetPixel(x, Round(y) );
       }
  }
```

More Efficient Implementation

How can we improve this algorithm?

```
DrawLine(int x1,int y1, int x2,int y2)
  {
       int x;
       float y;
       for (x=x1; x<=x2; x++) {
             y = y1 +  (x-x1)*(y2-y1)/(x2-x1)
             SetPixel(x, Round(y) );
       }
  }
```

More Efficient Implementation

```
DrawLine(int x1,int y1, int x2,int y2)
  {
      int x;
      int dx = x2-x1;
      int dy = y2-y1;
      float y;
      float m = dy/(float)dx;
      for (x=x1; x<=x2; x++) {
            y = y1 + m*(x-x1) ;
            SetPixel(x, Round(y) );
      }
  }
```

Even More Efficient Implementation

```
DrawLine(int x1,int y1, int x2,int y2)
  {
      int x;
      int dx = x2-x1;
      int dy = y2-y1;
      float y;
      float m = dy/(float)dx;
      y = y1 + 0.5 ;
      for (x=x1; x<=x2; x++) {
            SetPixel(x, Floor(y) );
            y = y + m ;
      }
  }
```

(Bresenham) Midpoint Algorithm

Line in the first quadrant (0 < slope < 45 deg)

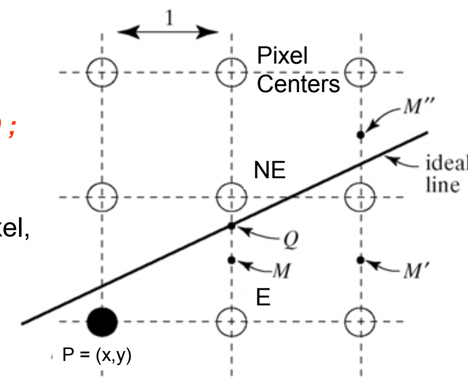
Implicit form of line:

$F(x,y) = x\,dy - y\,dx + c$,

Note: $dx = x2 - x1$; $dy = y2 - y1$

$dx, dy > 0$ and $dy/dx \leq 1.0$;

- Current pixel choice P = (x,y)
- How do we choose the next pixel,
  P′ = (x+1,y′) ?



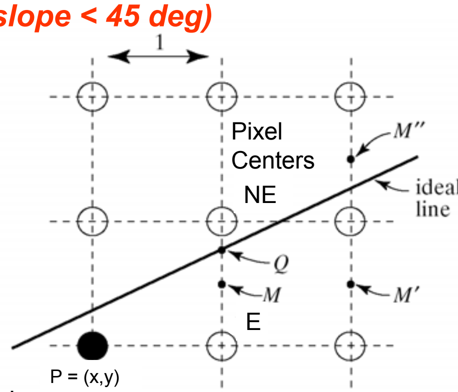
(Bresenham) Midpoint Algorithm

Line in the first quadrant (0 < slope < 45 deg)

Implicit form of line:

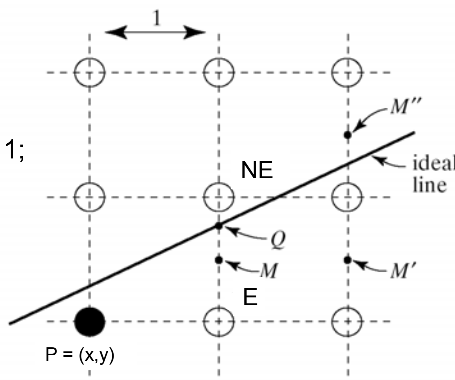
$F(x,y) = x\,dy - y\,dx + c$

- Current pixel choice P = (x,y)
- How do we choose the next pixel,
  P′ = (x+1,y′) ? Test F(M)
  If(F(x+1,y+0.5) > 0)
  M is below line: choose NE pixel
  else
  M is on or above line: choose E pixel



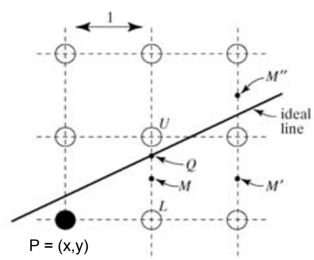
(Bresenham) Midpoint Algorithm

```
DrawLine(int x1, int y1, int x2, int y2,)
  {
      int x, y;
      y = y1;
      for (x=x1; x<=x2; x++) {
        SetPixel(x, y);
        if (F(x+1,y+0.5) > 0)  y = y + 1;
      }
  }
```



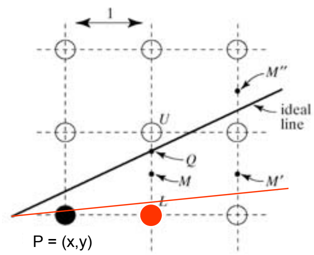
Can We Compute F in a Smart Way?

- We are at pixel (x,y) we evaluate F at M = (x+1,y+0.5) and choose E = (x+1,y) or NE = (x+1,y+1) accordingly
- Reminder: F(x,y) = x dy – y dx + c



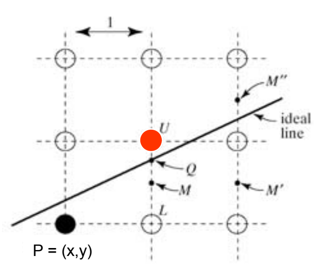
Can We Compute F in a Smart Way?

- We are at pixel (x,y) we evaluate F at M = (x+1,y+0.5) and choose E = (x+1,y) or NE = (x+1,y+1) accordingly
- Reminder: F(x,y) = x dy – y dx + c
- If we choose E for x+1, then the next test will be at M′:
  F(x+2,y+0.5) = [(x+1)dy + 1dy] – (y+0.5)dx + c
  = F(x+1,y+0.5) + dy
  So, F_E = F + dy



Can We Compute F in a Smart Way?

- We are at pixel (x,y) we evaluate F at M = (x+1,y+0.5) and choose E = (x+1,y) or NE = (x+1,y+1) accordingly
- Reminder: F(x,y) = x dy – y dx + c
- If we choose E for x+1, the next test will be at M′:
  F(x+2,y+0.5) = [(x+1)dy + dy] - (y+0.5)dx + c
  = F(x+1,y+0.5) + dy
  So, F_E = F + dy
- If we chose NE, then the next test will be at M″:
  F(x+2,y+1.5) =
  F(x+1,y+0.5) + dy – dx
  So, F_{NE} = F + dy – dx



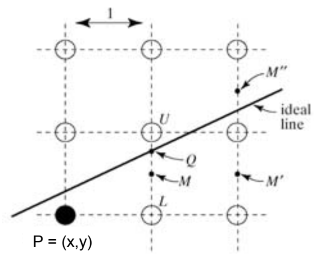
Can We Compute F in a Smart Way?

- We are at pixel (x,y) we evaluate F at M = (x+1,y+0.5) and E = (x+1,y) or NE = (x+1,y+1) accordingly
- Reminder: F(x,y) = x dy – y dx + c
- If we chose E for x+1, then the next test will be at M′:

$F_E = F + dy$

- If we chose NE, then the next test will be at M″:

$F_{NE} = F + dy – dx$



Test Update

Update

$F_E = F + dy = F + dF_E$ ($dF_E = dy$)

$F_{NE} = F + dy – dx = F + dF_{NE}$ ($dF_{NE} = dy – dx$)

What is the starting value?

Reminder: F(x,y) = x dy – y dx + c

Assume line starts at pixel (x_1,y_1)

$F_{start} = F(x_1+1,y_1+0.5)$

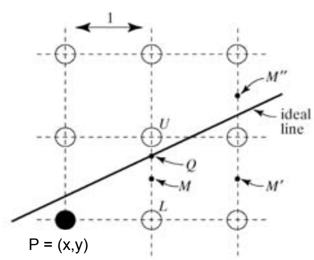
$= (x_1+1)dy – (y_1+0.5)dx + c$

$= (x_1dy – y_1dx + c) + dy – 0.5dx$

$= F(x_1,y_1) + dy – 0.5dx.$

But (x_1,y_1) is on the line, so $F(x_1,y_1) = 0$

Therefore, $F_{start} = dy – 0.5dx$



Test Update (Integer Version)

Update

$F_{start} = dy – 0.5dx$

$F_E = F + (dy) = F + dF_E$

$F_{NE} = F + (dy – dx) = F + dF_{NE}$

Everything is integer except F_{start}

Multiply by 2 → $\boxed{\begin{aligned} F_{start} &= 2dy – dx \\ dF_E &= 2(dy) \\ dF_{NE} &= 2(dy – dx) \end{aligned}}$

(Bresenham) Midpoint Algorithm

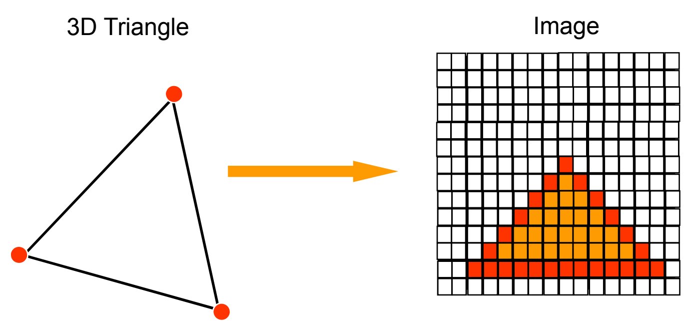
```
DrawLine(int x1, int y1, int x2, int y2,   int red, int green, int blue)
    {
        int x, y, dx, dy, d, dE, dNE;
        dx = x2-x1;
        dy = y2-y1;
        d = 2*dy-dx;    // initialize d
        dE = 2*dy;
        dNE = 2*(dy-dx);
        y = y1;
        for (x=x1; x<=x2; x++) {
                SetPixel(x, y, red, green, blue);
                if (d > 0) {                // choose NE pixel
                        d = d + dNE;
                        y = y + 1;
                } else {                    // choose E pixel
                        d = d + dE;
                    }
        }
    }
```

Other Incremental Rasterization Algorithms

The Bresenham incremental approach also works for drawing more complex geometric primitives

- Circles
- Polynomials
- Etc.

Triangle Rasterization



- Rasterize edges
- Optionally fill interior region

Pixel Region Filling Algorithms

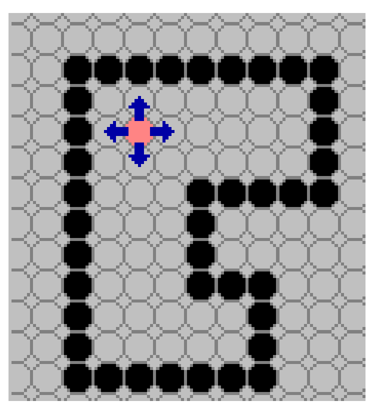
Rasterize boundary

Fill interior regions

2D paint programs

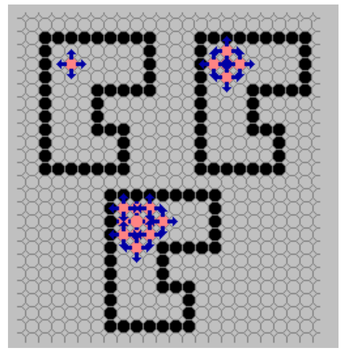
Flood Fill

```
public void floodFill(int x, int y, int fill, int old)
{
    if ((x < 0) || (x >= width)) return;
    if ((y < 0) || (y >= height))  return;
    if (getPixel(x, y) == old) {
        setPixel(x, y, fill);
        floodFill(x+1, y, fill, old);
        floodFill(x, y+1, fill, old);
        floodFill(x–1, y, fill, old);
        floodFill(x, y–1, fill, old);
    }
}
```



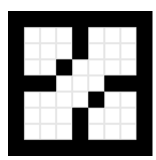
Boundary Fill

```
boundaryFill(int x, int y, int fill, int boundary) {
        if ((x < 0) || (x >= width)) return;
        if ((y < 0) || (y >= height)) return;
        int current = getPixel(x, y);
        if ((current != boundary) & (current != fill)) {
            setPixel(x, y, fill);
            boundaryFill(x+1, y, fill,boundary);
            boundaryFill(x, y+1, fill, boundary);
            boundaryFill(x–1, y, fill, boundary);
            boundaryFill(x, y–1, fill, boundary);
        }
}
```



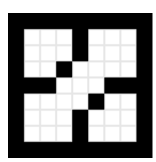
Adjacency

4-connected



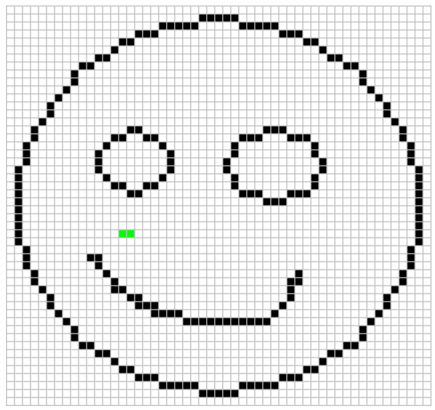
8-connected

- Will leak through diagonal boundaries
- Can be used to color boundaries



Scanline Fill

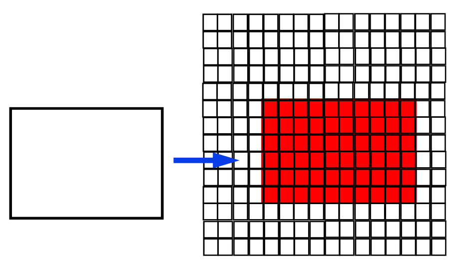
For more info, see "Flood fill" in Wikipedia



Polygon Rasterization

Scan conversion

Shade pixels lying within a closed polygon **efficiently**



Algorithm

- For each row of pixels define a *scanline* through their centers
- Intersect each scanline with all edges
- Sort intersections in x
- Calculate parity of intersections to determine 'interior' / 'exterior'
- Fill the 'interior' pixels
- Exploit coherence of intersections between scanlines