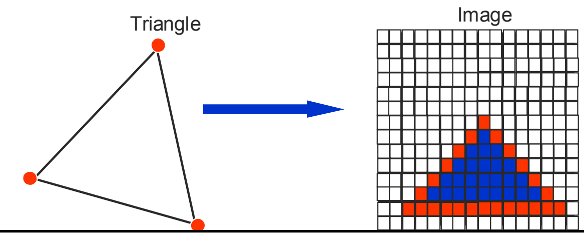
Digital Drawing - Rasterization

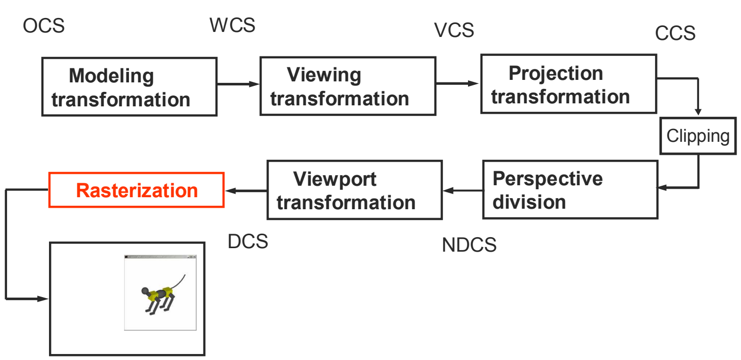
CS 174A

Rasterization of Primitives

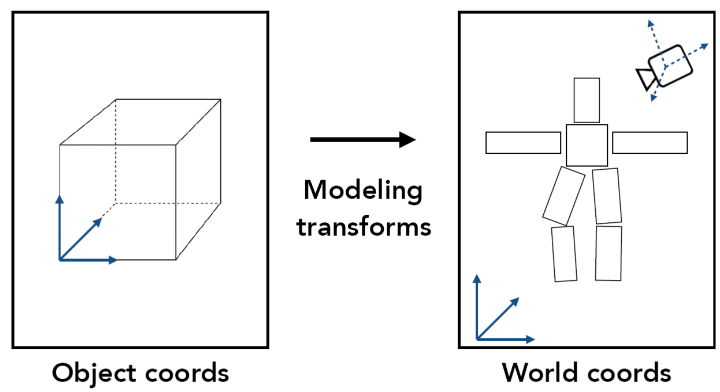
- Draw geometric primitives
  - Convert from geometric defintion of lines/obects to pixels
  - *rasterization* = selecting pixels
- Will be done frequently
  - must be fast:
    - use integer arithmetic
    - use addition instead of multiplication



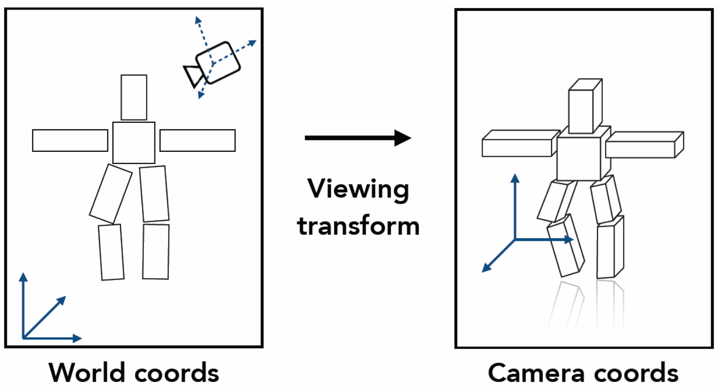
Z-Buffer Graphics Pipeline



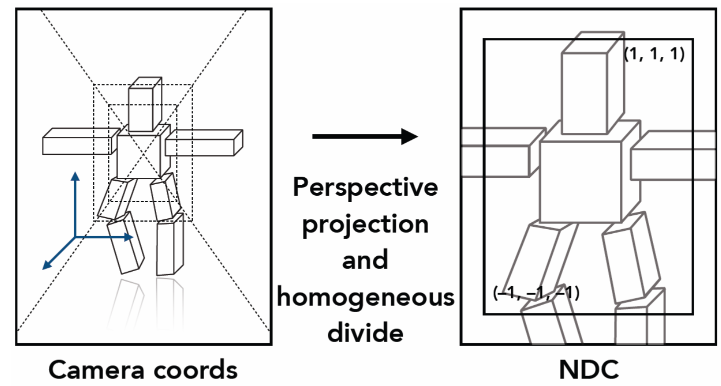
Graphics Pipeline



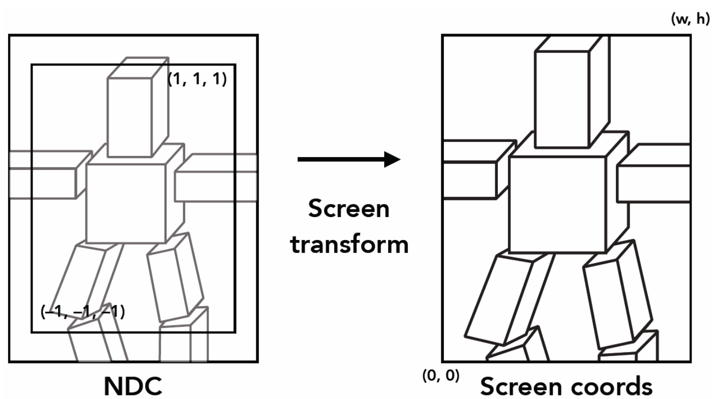
Graphics Pipeline



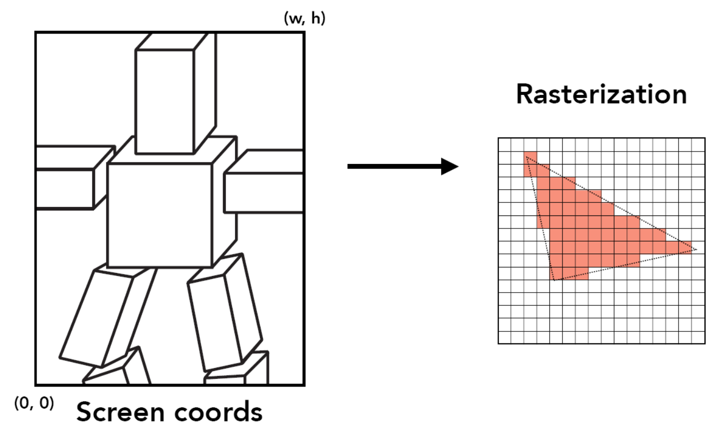
Graphics Pipeline



Graphics Pipeline



Graphics Pipeline



Lecture Outline

- Convert continuous rendering primitives into discrete fragments/pixels
- Lines
  - Bresenham
- Triangles
  - Flood Fill
  - Boundary Fill
  - Scanline

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**

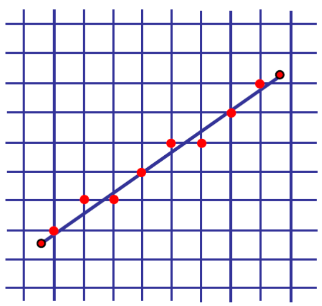
  $P = \mathbf{M}P_i$; $Q = \mathbf{M}Q_i$

  drawline(P_x/w_P, P_y/w_P, Q_x/w_Q, Q_y/w_Q)   // w_P, w_Q are 4th coords of P, Q

end for

Line Rasterization

- Given line equation fill in the pixels
  - grid points in diagram = centers of pixels



Line Rasterization

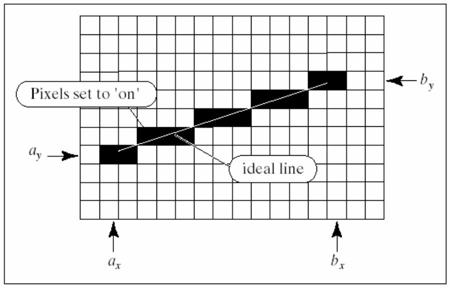
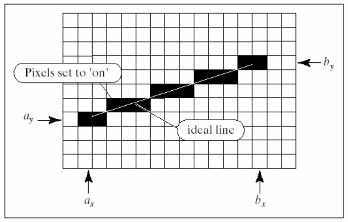


FIGURE 10.23 Drawing a straight-line-segment.

Line Rasterization

Desired properties

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



from: **Computer Graphics Using OpenGL, 2e**, by F. S. Hill

Reminder: Line Representations

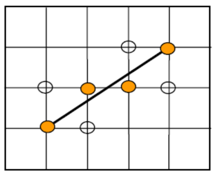
Line (in 2D)

- Explicit $$y = \frac{dy}{dx}(x - x_0) + y_0$$
- Implicit $$F(x,y) = (x - x_0)dy - (y - y_0)dx$$
  if $F(x,y) = 0$ then (x,y) is on line
  $F(x,y) > 0$ (x,y) is below line
  $F(x,y) < 0$ (x,y) is above line
- Parametric
$$x(t) = x_0 + t(x_1 - x_0)$$
$$y(t) = y_0 + t(y_1 - y_0)$$
$$t \in [0,1]$$
$$P(t) = P_0 + t(P_1 - P_0), \text{ or}$$
$$P(t) = (1-t)P_0 + tP_1$$

Implementation

- Implementation with the explicit line representation $y = \frac{dy}{dx}(x - x_0) + y_0$
- Assume $x_1 < x_2$ & line slope absolute value is ≤ 1

```
DrawLine( x1, x2, y1, y2 )
begin
float dx , dy , x, y, slope ;
dx ⇐ x2 − x1;
dy ⇐ y2 − y1;
slope ⇐ dy/dx ;
y ⇐ y1
for x from x1 to x2 do
begin
    SetPixel ( x, Round (y) );
    y = y1 + slope * (x-x1)
end ;
end ;
```

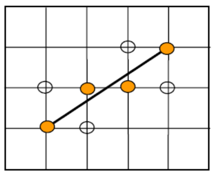


Questions:
Can this algorithm use integer arithmetic ?

Better Implementation

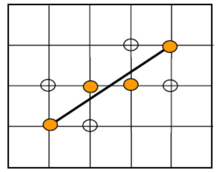
- Implementation with the explicit line representation $y = \frac{dy}{dx}(x - x_0) + y_0$
- Assume $x_1 < x_2$ & line slope absolute value is ≤ 1

```
DrawLine( x1, x2, y1, y2 )
begin
float dx , dy , x, y, slope ;
dx ⇐ x2 − x1;
dy ⇐ y2 − y1;
slope ⇐ dy/dx ;
y = y1 +.5
for x from x1 to x2 do
begin
    SetPixel ( x, Floor (y) );
    y ⇐ y + slope * (x-x1)
end ;
end ;
```



Midpoint (Brensenham) Algorithm

- Given current choice P=(x,y), how do we choose the next pixel?

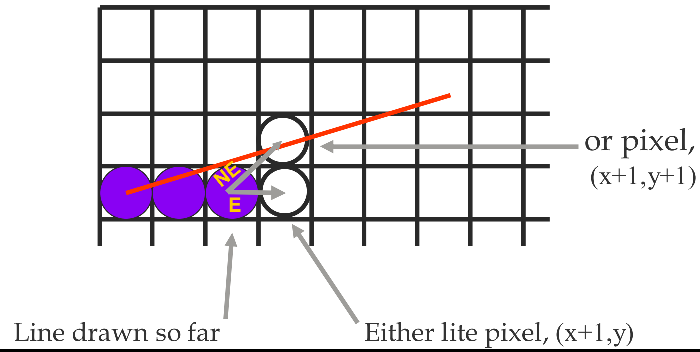


- Idea:
  - Proceed along the line incrementally
  - Have ONLY 2 choices
    - It may plot the point *(x+1,y)*, or:
    - It may plot the point *(x+1,y+1)*
  - Select one that minimizes error (distance to line), ***d***

Bresenham algorithm: core idea

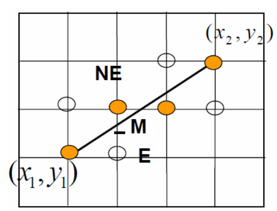
- At each step, choice between 2 pixels

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



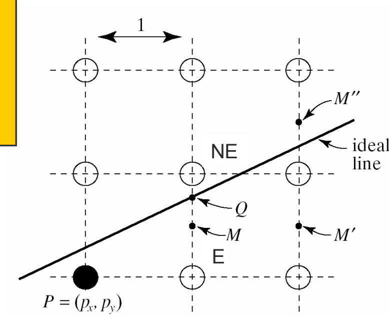
Midpoint Line Drawing

- Starting point satisfies $d(x_1,y_1)=0$
- Each step moves right (east) or upper right (northeast)
- Sign of $d(x+1,y+1/2)$ indicates if to move east or northeast
  - $d(x+1,y+1/2) <0$, move **NE**
  - $d(x+1,y+1/2) >0$, move **E**



Midpoint Algorithm (Bresenham)

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



Midpoint Algorithm

•*Distance* measure **is** the *implicit representation of the line*

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

•Given 2 points compute the A, B, C parameters of the line

$$f(x,y) = 0 = Ax + By + C$$

$$y = mx + b$$

$$y = \frac{(\Delta y)}{(\Delta x)}x + b$$

$$(\Delta x)y = (\Delta y)x + (\Delta x)b$$

$$0 = (\Delta y)x - (\Delta x)y + (\Delta x)b$$

$$0 = Ax + By + C \qquad A = \Delta y, \quad B = -\Delta x, \quad C = (\Delta x)b$$

Midpoint Algorithm (version 1)

$$f(x,y) = 0 = Ax + By + C$$

$$y = mx + b$$

$$y = \frac{(\Delta y)}{(\Delta x)}x + b$$

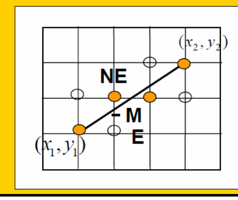
$$(\Delta x)y = (\Delta y)x + (\Delta x)b$$

$$0 = (\Delta y)x - (\Delta x)y + (\Delta x)b$$

$$0 = Ax + By + C$$

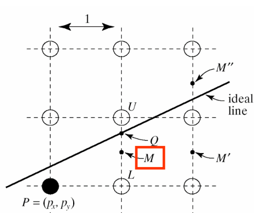
$$0 = (\Delta y)x - (\Delta x)y + (\Delta x)b$$

```
DrawLine( x1, x2, y1, y2 )
begin
int   x , y , dx , dy , d ;
x ⇐ x1 ;          y ⇐ y1 ;
dx ⇐ x2 − x1 ;    dy ⇐ y2 − y1 ;
 SetPixel   ( x , y ) ;
while  ( x < x2 )  do
    d = (2x + 2)dy - (2y + 1)dx + 2c; // 2((x + 1)dy - (y + .5)dx + c)
    if  ( d < 0 )  then
    begin
          x ⇐ x + 1 ;
    end  ;
    else  begin
          x ⇐ x + 1 ;
          y ⇐ y + 1 ;
    end  ;
    SetPixel      ( x , y ) ;
end  ;
end  ;
```



Can We Compute d in a Smart Way?

- We are at pixel (x,y) we evaluate ***d*** at M = (x+1,y+0.5) and choose E = (x+1,y) or NE = (x+1,y+1) accordingly
- Reminder: $d(x,y) = x\Delta y - y\Delta x + C$



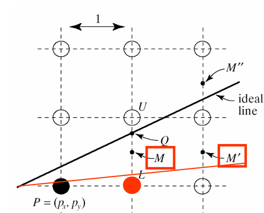
Can We Compute d in a Smart Way?

- We are at pixel (x,y), we evaluate ***d*** at M = (x+1,y+0.5) and choose **E** = (x+1,y) or **NE** = (x+1,y+1) accordingly
- Reminder: $d(x,y) = x\Delta y - y\Delta x + C$
  - If we choose E for x+1, then the next test will be at M':

$$d(x+2,y) = \qquad ' \qquad \text{rewrite in terms of d(x+1,y+0.5)}$$

$$d(x+2,y) = [(x+1)\Delta y - y\Delta x + C] + \Delta y = d(x+1, y+.5) + \Delta y$$

- $d_E = d + \Delta y$



Can We Compute *d* in a Smart Way?

- We are at pixel (x,y), we evaluate ***d*** at M = (x+1,y+0.5) and choose E = (x+1,y) or NE = (x+1,y+1) accordingly
- Reminder: ***d***(x,y) = x Δy – y Δx + c
- If we choose E =(x+1,y), the next test will be at M'=(x+2,y+1):

$$d(x+2,y)=[(x+1)\Delta y-y\Delta x+C]+\Delta y=d(x+1,y+.5)+\Delta y$$

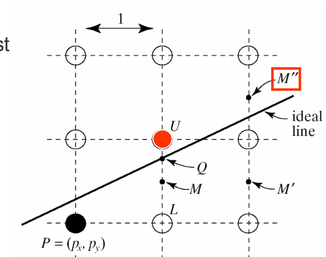
$d_E = d + \Delta y$

- If we chose NE=(x+1,y+1), then the next test will be at M''=(x+2,y+1+.5):

d(x+2,y+1+0.5) =

d(x+1,y+0.5) + Δy – Δx →

$d_{NE} = d + \Delta y - \Delta x$



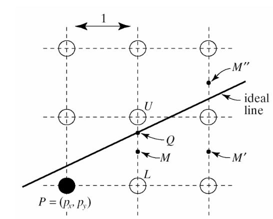
Can We Compute *d* in a Smart Way?

- We are at pixel (x,y) we evaluate ***d*** at M = (x+1,y+0.5) and choose E = (x+1,y) or NE = (x+1,y+1) accordingly
- Reminder: ***d***(x,y) = x Δy – y Δx + c
- If we chose E, then the next test will be at M':

$d_E = d + \Delta y$

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

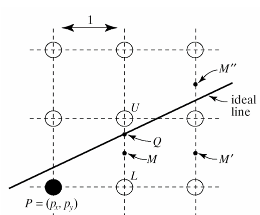
$d_{NE} = d + \Delta y - \Delta x$



Test Update

- Update

$$d_E = d + \Delta y = d + \Delta d_E \qquad (\Delta d_E = \Delta y)$$
$$d_{NE} = d + \Delta y - \Delta x = d + \Delta d_{NE} \qquad (\Delta d_{NE} = \Delta y - \Delta x)$$



- Starting value?

Line equation: $d(x,y) = x\Delta y - y\Delta x + C$

Assume line starts at pixel (x_0,y_0)

$$d_{start} = d(x_0+1, y_0+.5) \ =$$ rewrite in terms of $d(x_0,y_0)$

$$= (x_0\Delta y - y_0\Delta x + C) + \Delta y - 0.5\Delta x = \underline{d(x_0,y_0) + \Delta y - 0.5\Delta x}$$

(x_0,y_0) belongs on the line, so: $d(x_0,y_0) = 0$

Therefore:

$$d_{start} = \Delta y - .0.5\Delta x$$

Test Update - Integer Version

- Update

$$d_{start} = \Delta y - 0.5\Delta x$$
$$d_E = d + \Delta y = d + \Delta d_E$$
$$d_{NE} = d + \Delta y - \Delta x = d + \Delta d_{NE}$$

- Everything is integer except d_{start}

Multiply by 2 →
$$\boxed{\begin{aligned} d_{start} &= 2\Delta y - \Delta x \\ \Delta d_E &= 2\Delta y \\ \Delta d_{NE} &= 2(\Delta y - \Delta x) \end{aligned}}$$

Midpoint Algorithm (Bresenham)

```
DrawLine(int x1, int y1, int x2, int y2, int
  color)
  {
   …
   for (x=x1; x<=x2; x++) {
           SetPixel(x, y, color );
           if (d<0) {        // choose NE
                          …
           } else {          // choose E
                          …
           }
   }
  }
```

Midpoint Algorithm (Bresenham)

```
DrawLine(int x1, int y1, int x2, int y2){
   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);
       if (d<0) {
          d = d + dNE ;       // choose NE
          y = y + 1 ;
          }
       else {
           d = d + dE ;       // choose E
          }
   }
  }
```

Other Incremental Rasterization Algorithms

- ***The Bresenham incremental approach also works for drawing more complex geometric primitives***
- Circles
- Polynomials
- Etc.

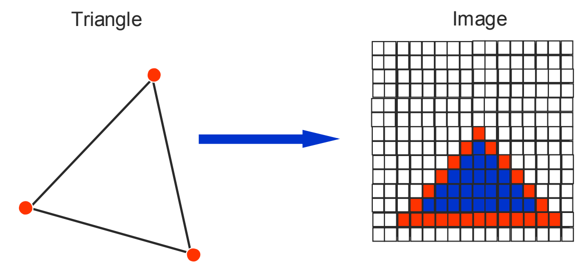
Lecture Outline

- Convert continuous rendering primitives into discrete fragments/pixels
- Lines
  - Bresenham
- Triangles
  - Flood Fill
  - Boundary Fill
  - Scanline

Rasterizing Triangles

- Basic surface representation in rendering
- Why?
- Lowest common denominator
  - Can approximate any surface with arbitrary accuracy
  - All polygons can be broken up into triangles
- Guaranteed to be:
  - Planar
  - Triangles - Convex
- Simple to render
  - Can implement in hardware

Triangle Rasterization



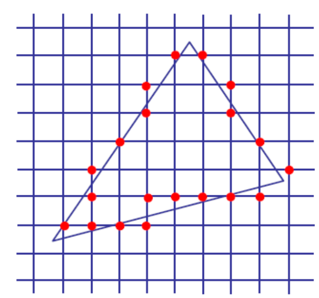
- Rasterize edges
- Optionally fill interior region

Pixel Region Filling Algorithms

- ***Rasterize boundary***
- ***Fill interior regions***
- 2D paint programs

Seed Fill Formulation

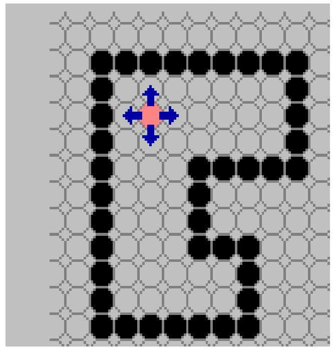
- **Input**
  - polygon P with rasterized edges
- **Problem**: Fill its interior with specified color on graphics display



Pixel Region Filling Algorithms

- Seed Fill:
  - Flood Fill
  - Boundary Fill
- Algorithm:
  1. Scan convert boundary
  2. Fill in regions

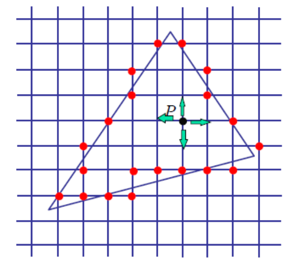
2D paint programs



Flood Fill

1. Suppose we want to color the entire area whose original color is oldColor, and replace it with fillColor.
2. Then, we start with a point in this area, and then color all surrounding points until we see a pixel that is not oldColor.

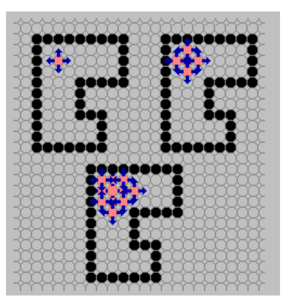
```
void floodFill(int x, int y, int fillcolor, int
oldcolor)
{
  if ((x < 0) || (x >= width)) return;
  if ((y < 0) || (y >= height))  return;
  if (getPixel(x, y) == oldcolor) {
      setPixel(x, y, fillcolor);
      floodFill(x+1, y, fillcolor, oldcolor);
      floodFill(x, y+1, fillcolor, oldcolor);
      floodFill(x-1, y, fillcolor, oldcolor);
      floodFill(x, y-1, fillcolor, oldcolor);
   }
}
```



Boundary Fill

- Suppose that the edges of the polygon has already been colored.
- Suppose that the interior of the polygon is to be colored a different color from the edge.
- Suppose we start with a pixel inside the polygon, then we color that pixel and all surrounding pixels until we meet a pixel that is already colored.

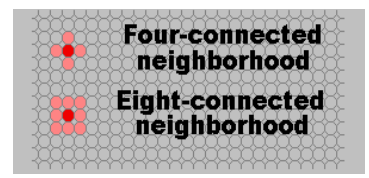
```
void boundaryFill(int x, int y, int fill, int boundary) {
    if ((x < 0) || (x >= width)) return;
    if ((y < 0) || (y >= height)) return;
    int current = getPixel(x, y);
    if ((current != boundarycolor) & (current != fillcolor)) {
       setPixel(x, y, fillcolor);
       boundaryFill(x+1,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



Adjacency

4-connected

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

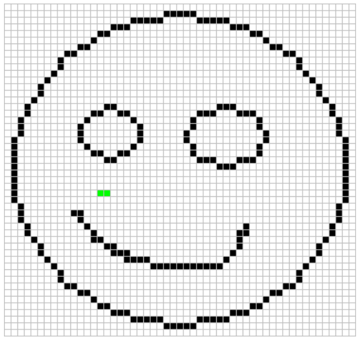


Seed Fill - Drawbacks

- How do we find a point inside?
- Pixels **visited up to 4 times** to check if already set (convince yourselves of this)
- Need per-pixel flag indicating if set already
  - clear for every polygon!

Scan 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