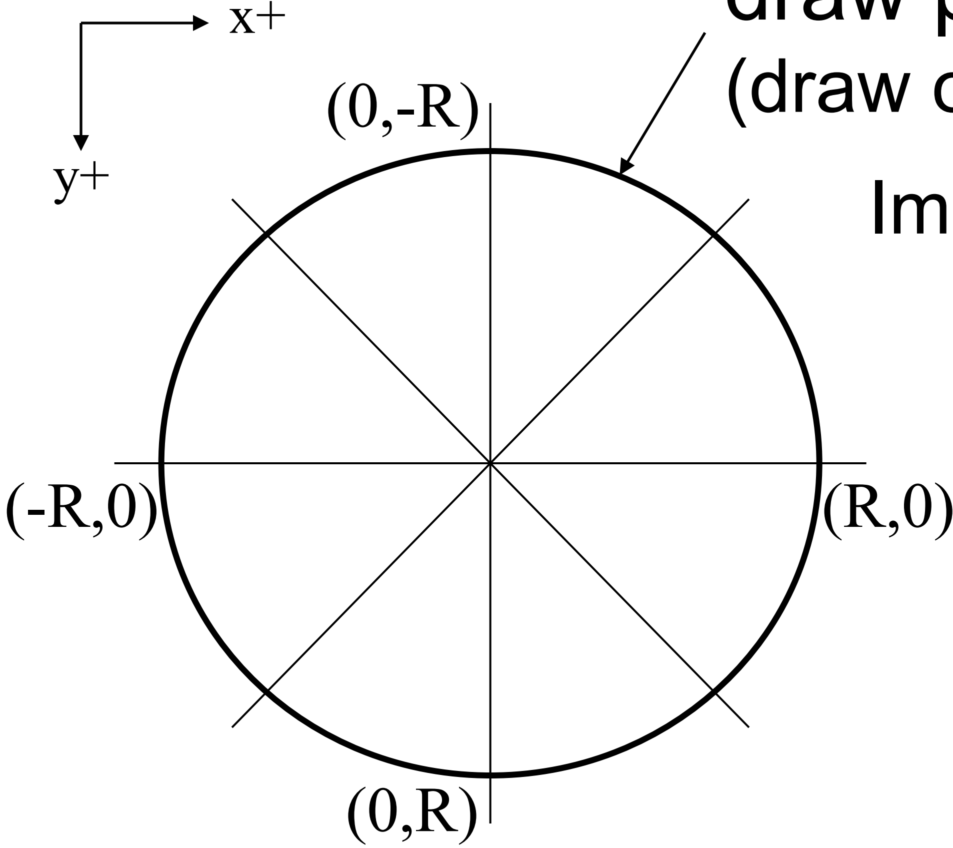
Circle Midpoint Algorithm



draw pixels in this octant
(draw others using symmetry)

Implicit function for circle

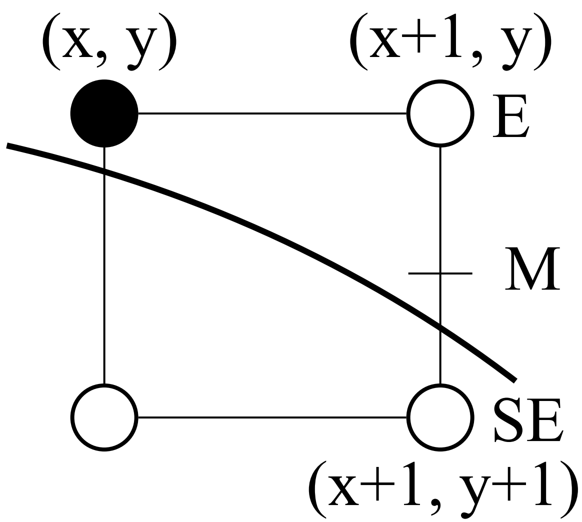
$$F(x,y) = x^2 + y^2 - R^2$$

$F(x,y) = 0$ on circle

$F(x,y) < 0$ inside

$F(x,y) > 0$ outside

Choosing the Next Pixel



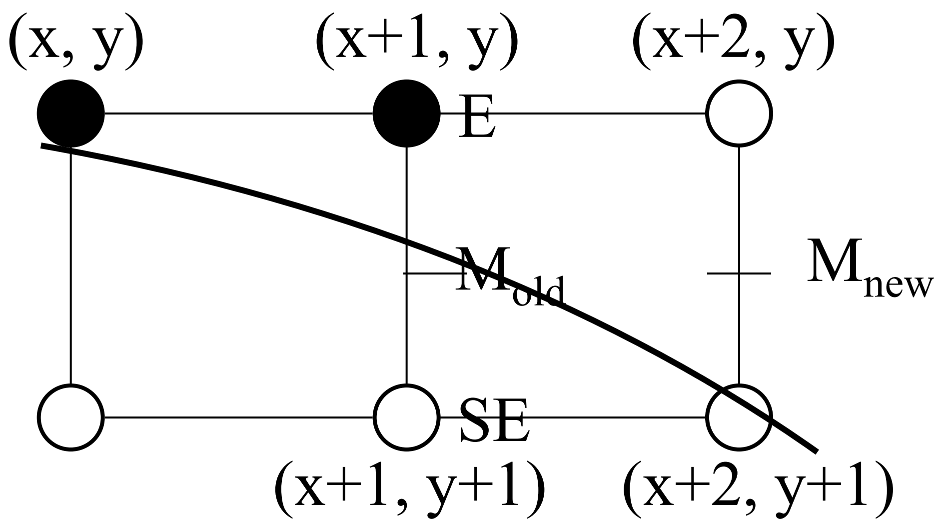
decision variable d

$$d = F(M) = F(x+1, y+1/2)$$

$F(x+1, y+1/2) > 0$ choose E

$F(x+1, y+1/2) \leq 0$ choose SE

Change of d when E is chosen



$$d_{new} = (x+2)^2 + (y+1/2)^2 - R^2$$

$$d_{old} = (x+1)^2 + (y+1/2)^2 - R^2$$

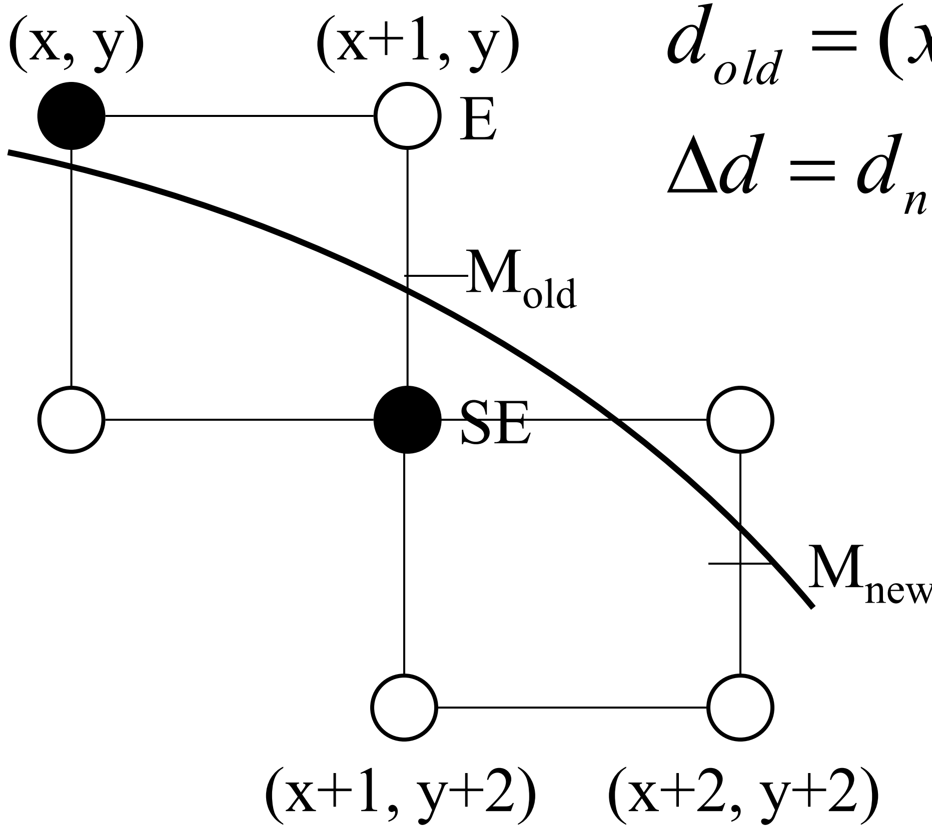
$$\Delta d = d_{new} - d_{old} = 2x+3$$

Change of d when SE is chosen

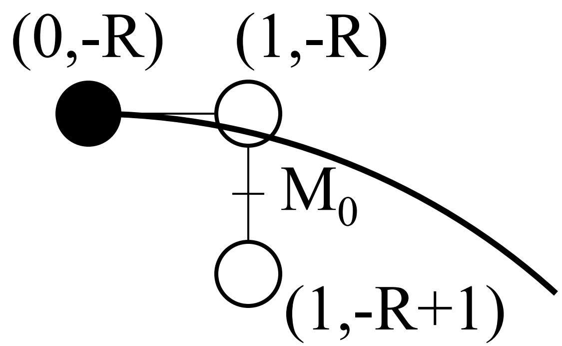
$$d_{new} = (x+2)^2 + (y+3/2)^2 - R^2$$

$$d_{old} = (x+1)^2 + (y+1/2)^2 - R^2$$

$$\Delta d = d_{new} - d_{old} = 2x + 2y + 5$$



Initial value of d



$$d_0 = F(M_0)$$

$$d_0 = F(1, -R + 1/2)$$

$$d_0 = (1)^2 + (-R + 1/2)^2 - R^2$$

$$d_0 = 5/4 - R$$

Midpoint Circle Algo

```
x = 0;
y = -R;
d = 5/4 – R;  /* real */
setPixel(x,y);
while (y > x) {
     if (d > 0) {  /* E chosen */
          d += 2*x + 3;
          x++;
     } else {      /* SE chosen */
          d += 2*(x+y) + 5;
          x++; y++;
     }
     setPixel(x,y);
}
```

New Decision Variable

- Our circle algorithm requires arithmetic with real numbers.
- Let's create a new decision variable h

  $h=d-1/4$
- Substitute $h+1/4$ for d in the code.
- Note $h > -1/4$ can be replaced with $h > 0$ since h will always have an integer value.

New Circle Algorithm

```
x = 0;
y = -R;
h = 1 - R;
setPixel(x,y);
while (y > x) {
     if (h > 0) {  /* E chosen */
          h += 2*x + 3;
          x++;
     } else {      /* SE chosen */
          h += 2*(x+y) + 5;
          x++; y++;
     }
     setPixel(x,y);
}
```

Second-Order Differences

- Note that d is incremented by a linear expression each time through the loop.
  - We can speed things up a bit by tracking how these linear expressions change.
  - Not a huge improvement since multiplication by 2 is just a left-shift by 1 (e.g. 2*x = x<<1).

2^{nd} Order Difference when E chosen

- When E chosen, we move from pixel (x,y) to (x+1,y).

$$\Delta E_{old} = 2x + 3 \qquad \Delta SE_{old} = 2(x + y) + 5$$

$$\Delta E_{new} = 2(x+1) + 3 \qquad \Delta SE_{new} = 2(x + 1 + y) + 5$$

$$\Delta E_{new} - \Delta E_{old} = 2 \qquad \Delta SE_{new} - \Delta SE_{old} = 2$$

2nd Order Difference when SE chosen

- When SE chosen, we move from pixel (x,y) to (x+1,y+1).

$$\Delta E_{old} = 2x + 3 \qquad \Delta SE_{old} = 2(x + y) + 5$$

$$\Delta E_{new} = 2(x+1) + 3 \qquad \Delta SE_{new} = 2(x+1+y+1) + 5$$

$$\Delta E_{new} - \Delta E_{old} = 2 \qquad \Delta SE_{new} - \Delta SE_{old} = 4$$

New and Improved Circle Algorithm

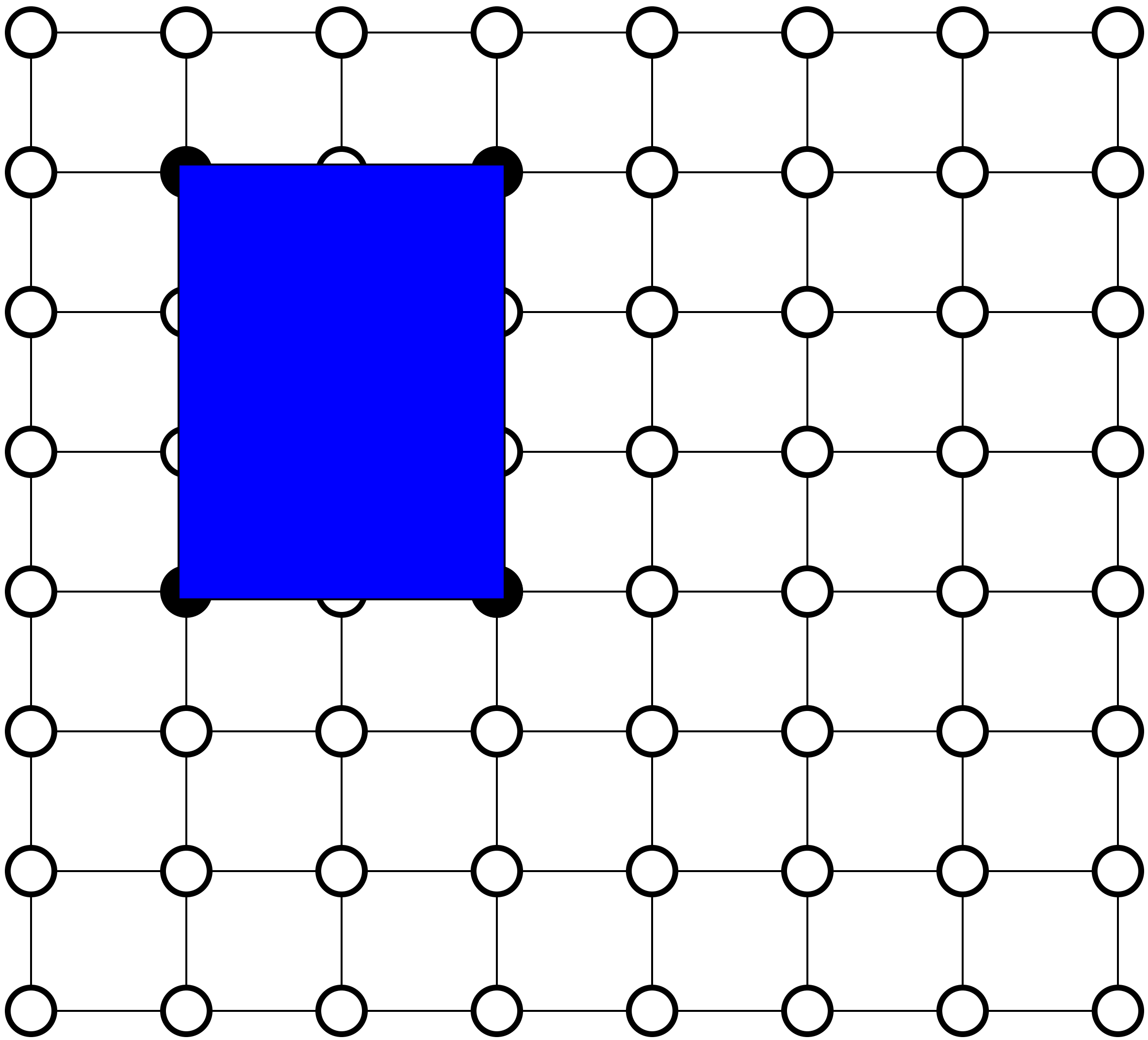
```
x = 0;  y = -R;
h = 1 – R;
dE = 3;  dSE = -2*R + 5;
setPixel(x,y);
while (y > x) {
      if (h > 0) {  /* E chosen */
            h += dE;
            dE += 2; dSE += 2;
            x++;
      } else {      /* SE chosen */
            h += dSE;
            dE += 2; dSE += 4;
            X++; y++;
      }
      setPixel(x,y);
}
```

Filling Primitives

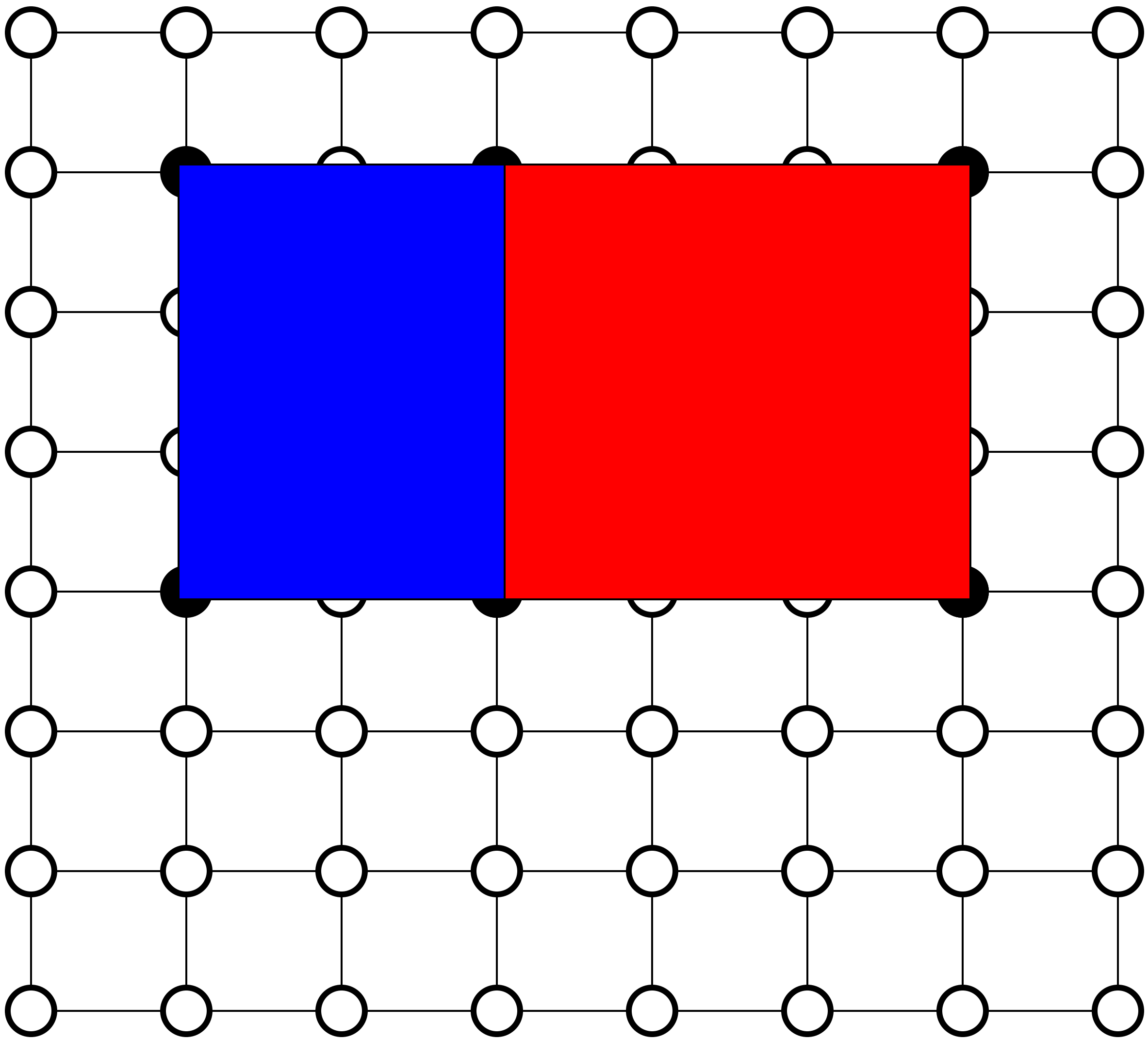
- We want to be able to fill rectangles, circles, polygons, pie-slices, etc…
- Deciding which pixels to fill is not trivial.
- We also want to fill shapes with patterns.
- We want to exploit *spatial coherence*
  - Neighboring pixels within primitive are the same.
  - e.g. span, scan-line, edge coherence

Filling Rectangles

which pixels are “inside”?



How do we handle edge pixels?



Raster Operations

- Usually you are just overwriting pixels when rasterizing a shape.

  destination pixel = source pixel

- Sometimes you want to combine the source and destination pixel in an interesting way:

  dest. pixel = source pixel XOR dest. pixel

  0101 = (1100) XOR (1001)

XOR Animation Hack

- Quick way to animate a small object (e.g. a ball) moving across the screen.
  - “Move” ball to next location
  - Draw ball using XOR
  - Draw ball again using XOR (erases ball)
  - repeat
- Does not require entire screen to be redrawn.
- A = (A XOR B) XOR B

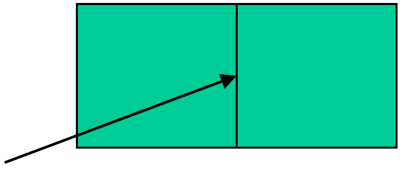
Other Ways to Combine Pixels

Name	Value written to destination
OR	S OR D
AND	S AND D
INVERT	NOT D
NOR	NOT (S OR D)
NAND	NOT (S AND D)

More pixel combining tricks later…
…back to filling primitives

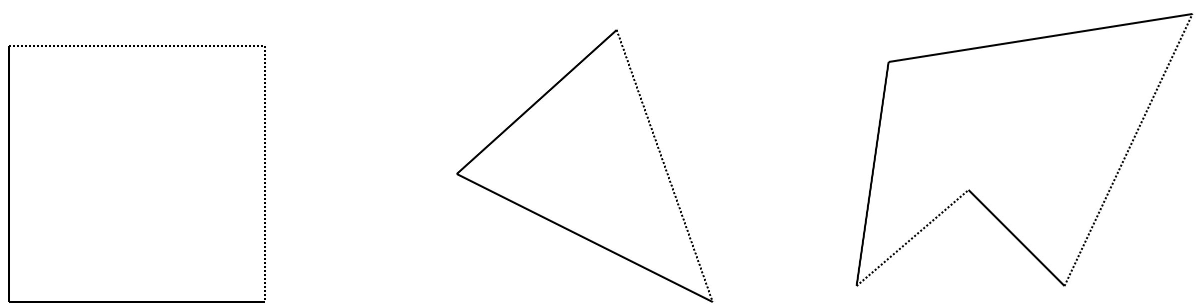
- How do we handle edge pixels?
- What if we want to tile to primitives together without creating any seams?
  - Remember, any pixels that are drawn twice in XOR mode will disappear!

don’t fill these pixels twice!



Rule for Boundary Pixels

- If a pixel lies on an edge…
  - The pixel is part of the primitive if it lies on the left boundary (or bottom boundary for horizontal edges).
  - Otherwise, the pixel is not part of the primitive.



Using Rule to Fill Adjacent Rectangles