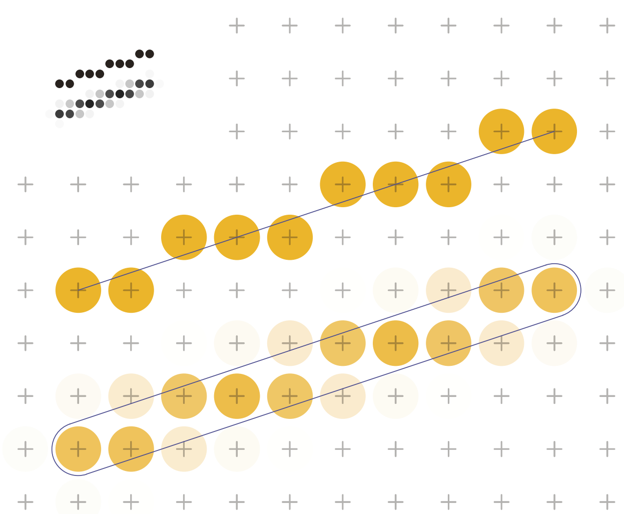
Scan Conversion

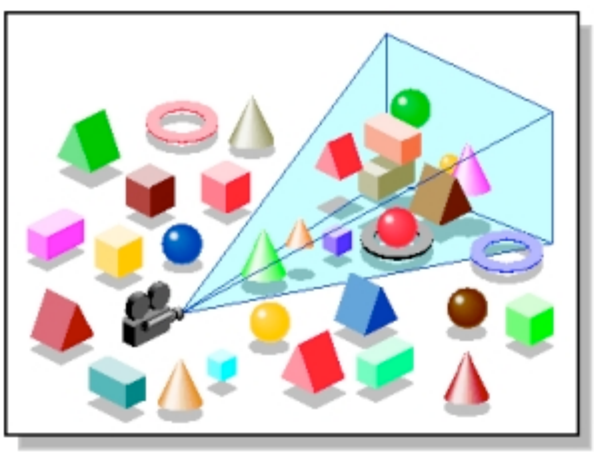


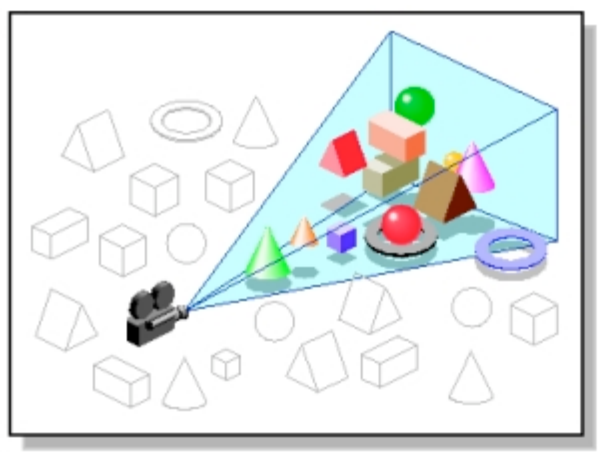
8*3
8 + 2
8*2
pow(8,3)
8/3
8/2
sqrt(8)

Recap

We know:

- How to build 3D objects
  - define 3D vertices (see P02)
- How to transform 3D vertices into 2D vertices
  - sequence of matrix multiplications (see P04)
- How to draw 2D vertices on 2D display
  - mapping from canonical space to screen (see P01)





Questions:

- How do we draw lines/polygons/faces?
- How do we clip objects?

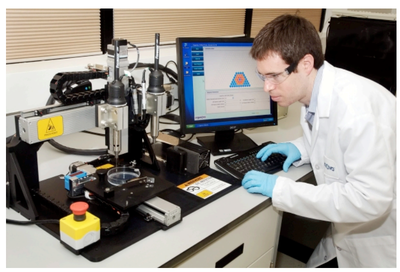
Scan Convertion

- Final step of rasterization (process of taking geometric shapes and converting them into an array of pixels stored in the frame buffer to be displayed)
- "Scan" originates from the raster-display terminology (we "scan" the display left to right, then move to the pixel line below etc)
- Takes place after clipping occurs
- All graphics packages do this at the end of the rendering pipeline
- Takes triangles and maps them to pixels on the screen
- Also takes into account other properties like lighting and shading, but we'll focus first on algorithms for line scan conversion

Motivation

- We need to understand how expensive drawing a line is (the WHY), before we discuss illumination and shading shortcuts and hacks (the HOW).

- This is how 3D printers work, too:
  http://on3dprinting.com/tag/stocks/



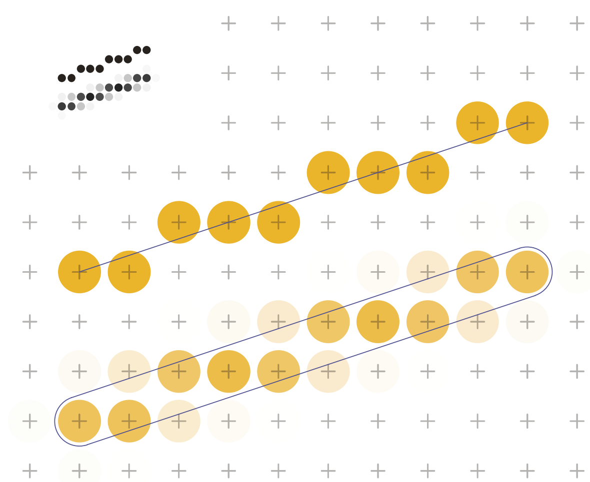
  - One day you might have to implement such an algo ☺

Problem Statement

- Consider scan-converting a line segment

- Given two points P and Q in XY plane, both with integer coordinates, determine which pixels on raster screen should be on in order to make picture of a unit-width line segment starting at P and ending at Q

- What is the cost of scan-converting a line? How do we scan-convert a line?



Finding Next Pixel:

Special case:

- Horizontal Line:

  Draw pixel *P* and increment *x* coordinate value by 1 to get next pixel.

- Vertical Line:

  Draw pixel *P* and increment *y* coordinate value by 1 to get next pixel.

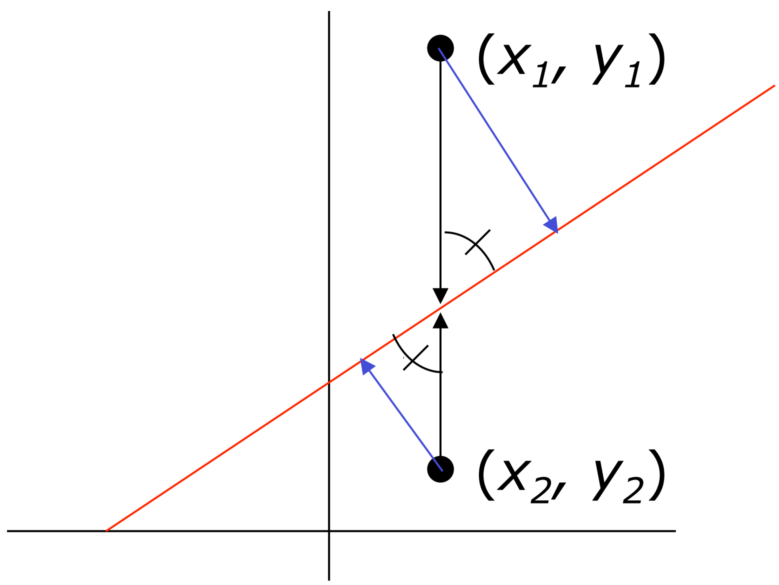
- Diagonal Line:

  Draw pixel *P* and increment both *x* and *y* coordinate by 1 to get next pixel.

- What should we do in general case?
  - Increment x coordinate by 1 and choose point closest to line.
  - But how do we measure "closest"?

Vertical Distance

- Why can we use vertical distance as measure of which point is closer?
  - because vertical distance is proportional to actual distance
  - how do we show this?
  - with similar triangles



- By similar triangles we can see that true distances to line (in blue) are directly proportional to vertical distances to line (in black) for each point
- Therefore, point with smaller vertical distance to line is closest to line

Strategy 1 - Incremental Algorithm (1/2)

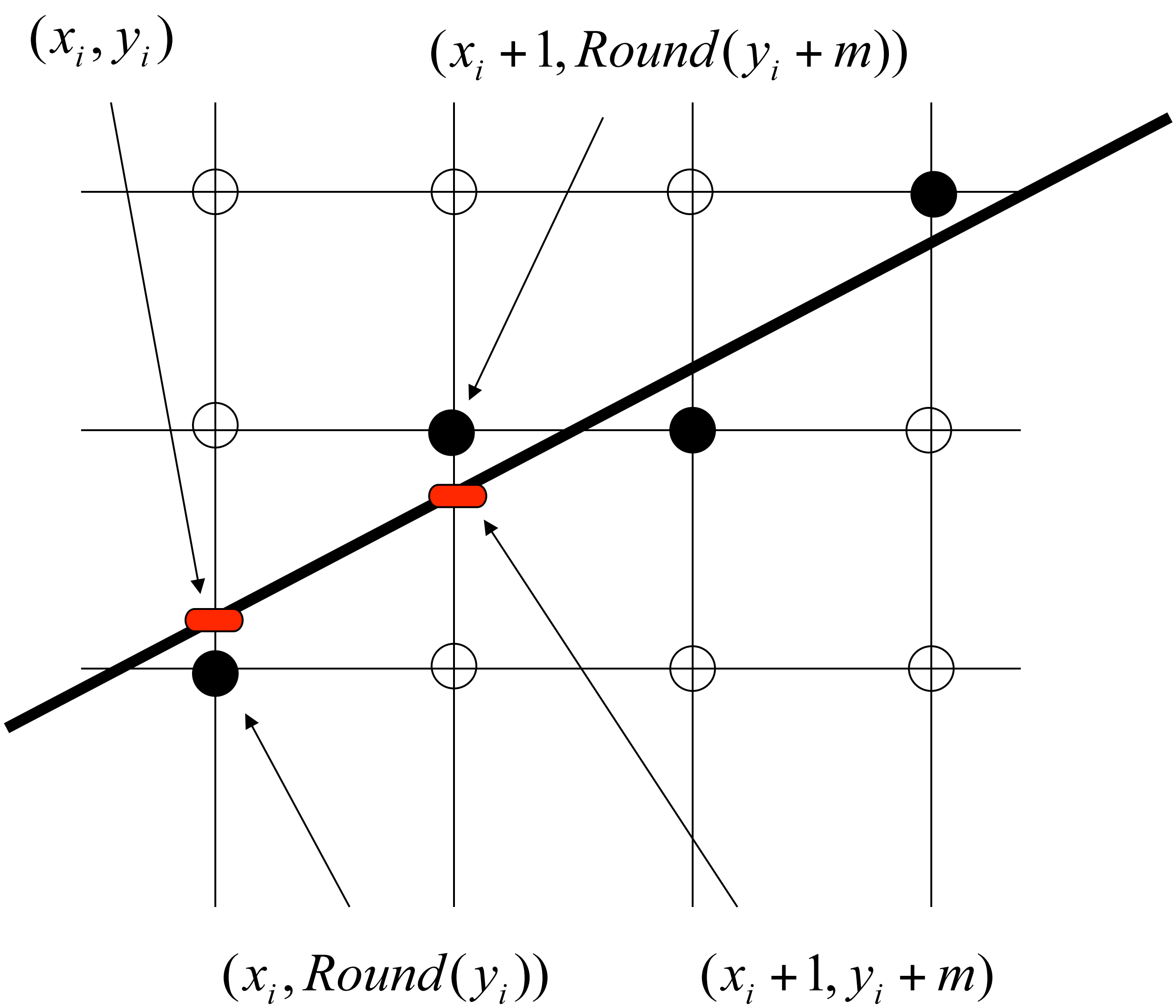
Basic Algorithm

- Find equation of line that connects two points *P* and *Q*
- Starting with leftmost point *P*, increment x_i by 1 to calculate $y_i = m*x_i + B$ where m = slope, B = y-intercept
- Draw pixel at (x_i, Round(y_i)) where Round (y_i) = Floor (0.5 + y_i)

Incremental Algorithm:

- Each iteration requires a floating-point multiplication
  - Modify algorithm to use deltas
- If $\Delta x = 1$, then $y_{i+1} = y_i + m$
- At each step, we make incremental calculations based on preceding step to find next y value

Strategy 1 - Incremental Algorithm (2/2)



Example Code

```
// Incremental Line Algorithm
// Assume x0 < x1

void Line(int x0, int y0,
          int x1, int y1) {
   int   x, y;
   float       dy = y1 – y0;
   float       dx = x1 – x0;
   float       m = dy / dx;

   y = y0;
   for (x = x0; x < x1; x++) {
     WritePixel(x, Round(y));
     y = y + m;
   }
}
```

Problem with Incremental Algorithm:

```
void Line(int x0, int y0,
          int x1, int y1) {
   int   x, y;
   float        dy = y1 - y0;
   float        dx = x1 - x0;
   float        m = dy / dx;

   y = y0;
   for (x = x0; x < x1; x++) {
     WritePixel(x, Round(y));
     y = y + m;
   }
}
```

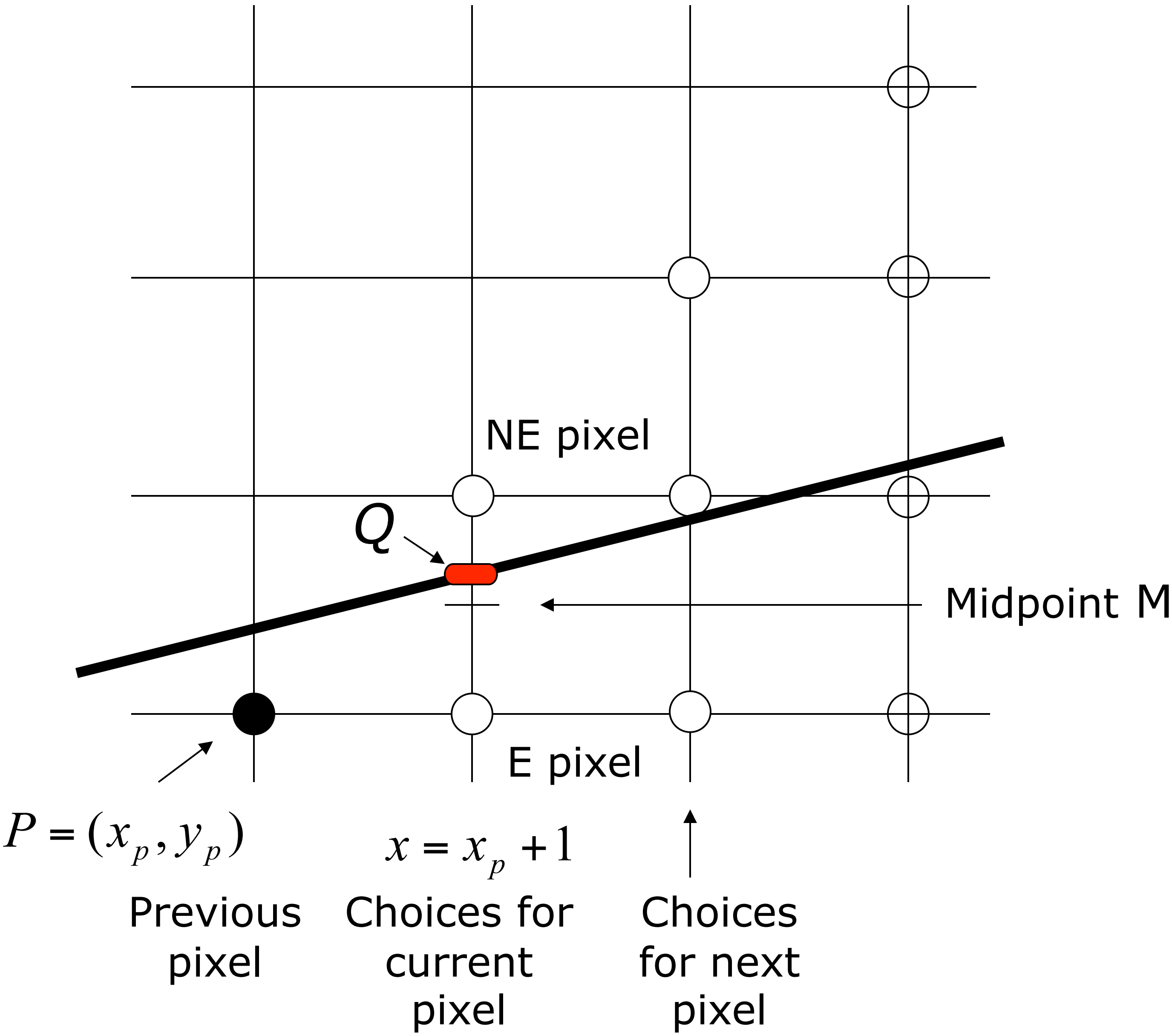
Rounding takes time

Since slope is fractional, need special case for vertical lines

Strategy 2 – Midpoint Line Algorithm (1/3)

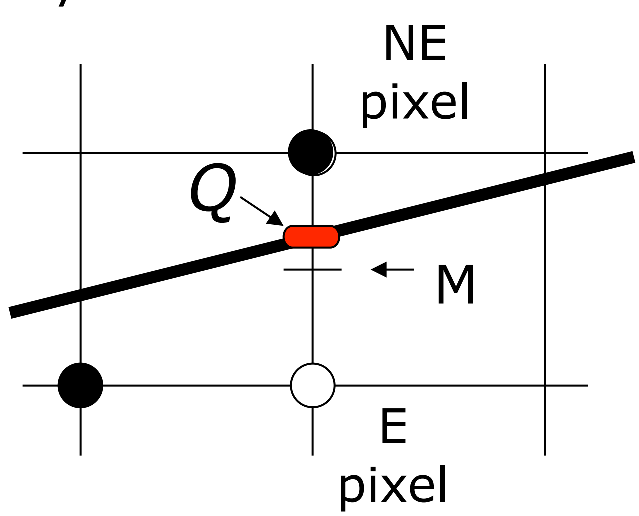
- Assume that line's slope is shallow and positive ($0 < \text{slope} < 1$); other slopes can be handled by suitable reflections about principal axes
- Call lower left endpoint (x_0, y_0) and upper right endpoint (x_1, y_1)
- Assume that we have just selected pixel P at (x_p, y_p)
- Next, we must choose between pixel to right (E pixel), or one right and one up (NE pixel)
- Let Q be intersection point of line being scan-converted and vertical line $x=x_p+1$

Strategy 2 – Midpoint Line Algorithm (2/3)



Strategy 2 – Midpoint Line Algorithm (3/3)

- Line passes between E and NE
- Point that is closer to intersection point *Q* must be chosen
- Observe on which side of line midpoint *M* lies:
  - E is closer to line if midpoint *M* lies above line, i.e., line crosses bottom half
  - NE is closer to line if midpoint *M* lies below line, i.e., line crosses top half
- Error (vertical distance between chosen pixel and actual line) is always <= ½
- Algorithm chooses NE as next pixel for line shown
- Now, need to find a way to calculate on which side of line midpoint lies



Line

Line equation as function *f(x)*:

$$y = mx + B$$

$$y = \frac{dy}{dx}x + B$$

Line equation as implicit function:

$$f(x, y) = ax + by + c = 0$$

for coefficients *a*, *b*, *c*, where *a*, *b* ≠ 0

from above,

$$y \cdot dx = dy \cdot x + B \cdot dx$$

$$dy \cdot x - y \cdot dx + B \cdot dx = 0$$

$$\therefore a = dy, b = -dx, c = B \cdot dx$$

Properties (proof by case analysis):

- $f(x_m, y_m) = 0$ when any point *M* is on line
- $f(x_m, y_m) < 0$ when any point *M* is above line
- $f(x_m, y_m) > 0$ when any point *M* is below line
- Our decision will be based on value of function at midpoint *M* at $(x_p + 1, y_p + ½)$

Decision Variable

Decision Variable *d*:

- We only need sign of $f(x_p + 1, y_p + ½)$ to see where line lies, and then pick nearest pixel
- $d = f(x_p + 1, y_p + ½)$
  - if $d > 0$ choose pixel NE
  - if $d < 0$ choose pixel E
  - if $d = 0$ choose either one consistently

How do we incrementally update *d*?

- On basis of picking E or NE, figure out location of *M* for that pixel, and corresponding value of *d* for next grid line
- We can derive *d* for the next pixel based on our current decision

If E was chosen:

Increment M by one in x direction

$$d_{new} = f(x_p + 2, y_p + ½) = a(x_p + 2) + b(y_p + ½) + c$$

$$d_{old} = a(x_p + 1) + b(y_p + ½) + c$$

- d_{new} - d_{old} is the incremental difference ΔE

$$d_{new} = d_{old} + a$$
ΔE = a = dy (2 slides back)

- We can compute value of decision variable at next step incrementally without computing *F(M)* directly

$$d_{new} = d_{old} + \Delta E = d_{old} + dy$$

- ΔE can be thought of as correction or update factor to take d_{old} to d_{new}

- It is referred to as forward difference

If NE was chosen:

Increment M by one in both *x* and *y* directions

$$d_{new} = F(x_p + 2, y_p + 3/2)$$
$$= a(x_p + 2) + b(y_p + 3/2) + c$$

- $\Delta NE = d_{new} - d_{old}$
  $d_{new} = d_{old} + a + b$
  $\Delta NE = a + b = dy - dx$

- Thus, incrementally,
  $d_{new} = d_{old} + \Delta NE = d_{old} + dy - dx$

Summary (1/2)

- At each step, algorithm chooses between 2 pixels based on sign of decision variable calculated in previous iteration.
- It then updates decision variable by adding either ΔE or ΔNE to old value depending on choice of pixel. Simple additions only!
- First pixel is first endpoint (x_0, y_0), so we can directly calculate initial value of *d* for choosing between E and NE.

Summary (2/2)

- First midpoint for first $d = d_{start}$ is at $(x_0 + 1, y_0 + ½)$

- $f(x_0 + 1, y_0 + ½)$
  $= a(x_0 + 1) + b(y_0 + ½) + c$
  $= a * x_0 + b * y_0 + c + a + b/2$
  $= \mathrm{f}(x_0, y_0) + a + b/2$

- But (x_0, y_0) is point on line and $f(x_0, y_0) = 0$

- Therefore, $d_{start} = a + b/2 = dy - dx/2$
  - use d_{start} to choose second pixel, etc.

- To eliminate fraction in d_{start} :
  - redefine f by multiplying it by 2; $f(x,y) = 2(ax + by + c)$
  - this multiplies each constant and decision variable by 2, but does not change sign

- Bresenham's line algorithm is same but doesn't generalize as nicely to circles and ellipses

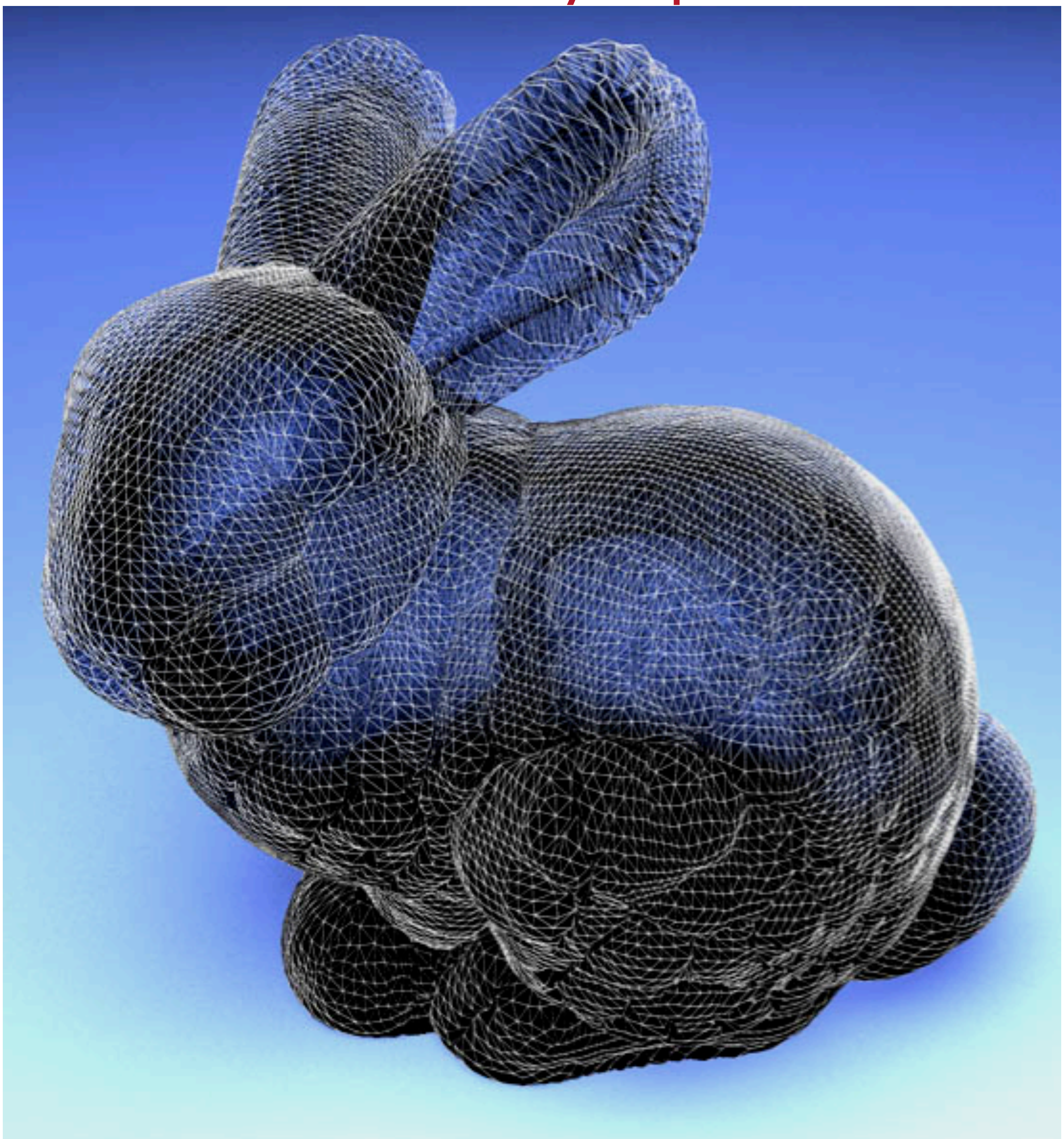
Example Code

```
void MidpointLine(int x0, int y0,
                  int x1, int y1) {
    int     dx = x1 - x0;
    int     dy = y1 - y0;
    int     d = 2 * dy - dx;
    int     incrE = 2 * dy;
    int     incrNE = 2 * (dy - dx);
    int     x = x0;
    int     y = y0;

    writePixel(x, y);

    while (x < x1) {
      if (d <= 0) {             // East Case
            d = d + incrE;
      } else {                  // Northeast Case
            d = d + incrNE;
            y++;
      }
      x++;
      writePixel(x, y);
    }       /* while */
}                 /* MidpointLine */
```

How Many Ops?



Scan Converting Circles

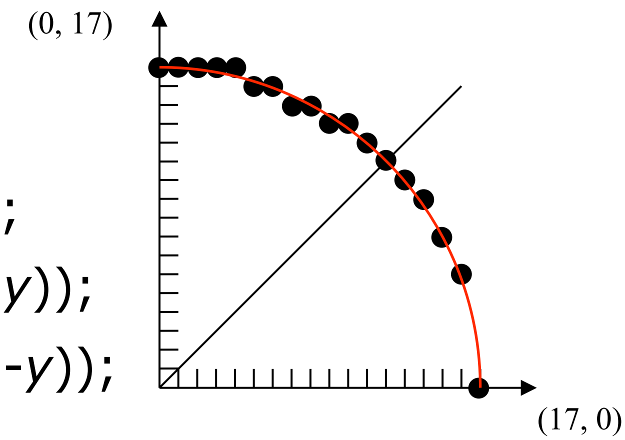
Version 1: <u>really</u> bad

For x = – R to R

    y = sqrt(R * R – x * x);

    Pixel (round(x), round(y));

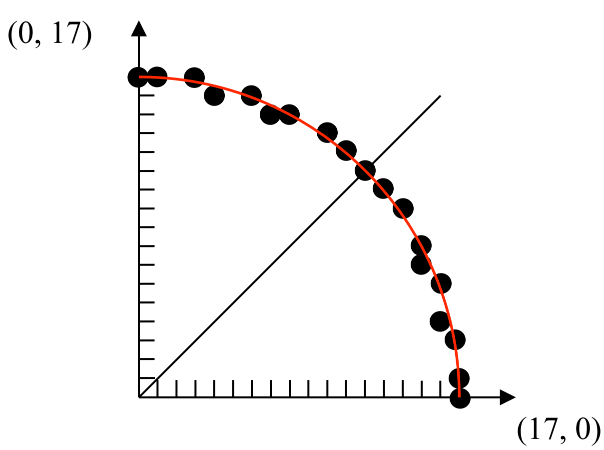
    Pixel (round(x), round(-y));



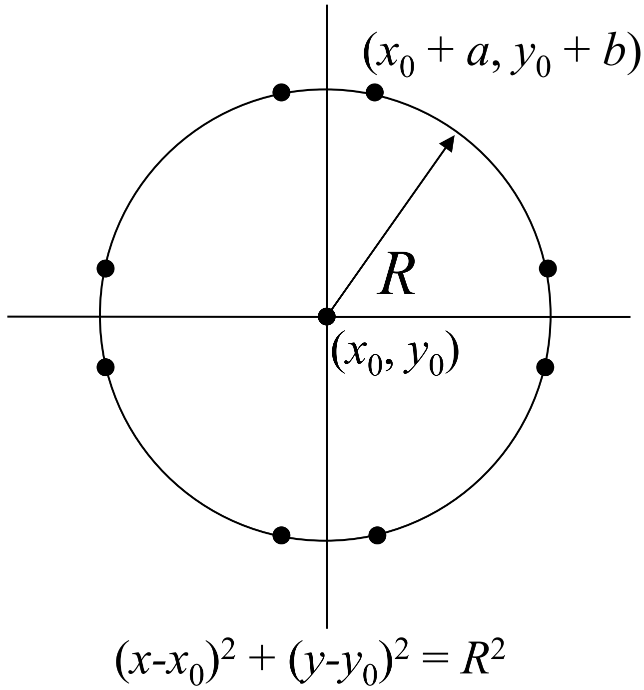
Version 2: slightly less bad

For x = 0 to 360

    Pixel (round (R • cos(x)), round(R • sin(x)));



Version 3 — Use Symmetry

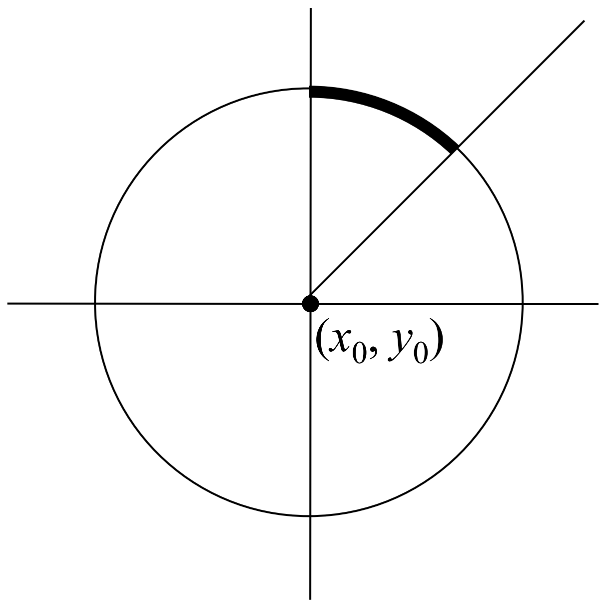


- Symmetry: If $(x_0 + a, y_0 + b)$ is on circle
  - also $(x_0 \pm a, y_0 \pm b)$ and $(x_0 \pm b, y_0 \pm a)$;
    hence 8-way symmetry.

- Reduce the problem to finding the pixels for 1/8 of the circle

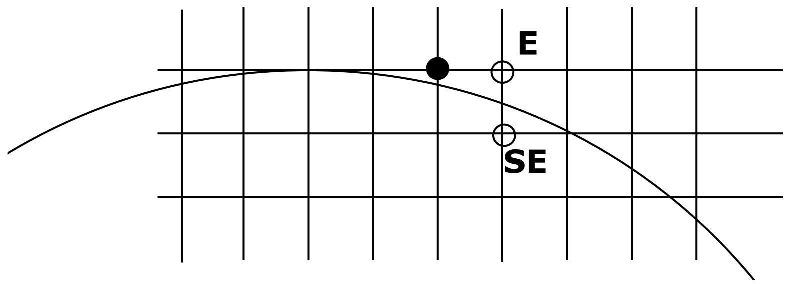
Using the Symmetry

- Scan top right 1/8 of circle of radius R



- Circle starts at $(x_0, y_0 + R)$
- Let's use another incremental algorithm with decision variable evaluated at midpoint

Sketch of Incremental Algorithm




```
x = x0; y = y0 + R; Pixel(x, y);
for (x = x0+1; (x – x0) > (y – y0); x++) {
   if (decision_var < 0) {
     /* move east */
     update decision_var;
   }
   else {
     /* move south east */
     update decision_var;
     y--;
   }
   Pixel(x, y);
}
```

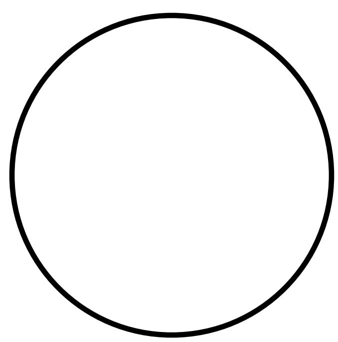
- Note: can replace all occurrences of x_0, y_0 with 0, 0 and Pixel ($x_0 + x$, $y_0 + y$) with Pixel (x, y)
- Essentially: shift of coordinates

Midpoint Eighth Circle Algorithm

```
MEC (R) /* 1/8th of a circle w/ radius R */
{
    int x = 0, y = R;
    int delta_E, delta_SE;
    float decision;
    delta_E = 2*x + 3;
    delta_SE = 2(x-y) + 5;
    decision = (x+1)*(x+1) + (y + 0.5)*(y + 0.5) -R*R;
    Pixel(x, y);
    while( y > x ) {
        if (decision > 0) {/* Move east */
            decision += delta_E;
            delta_E += 2; delta_SE += 2; /*Update delta*/
        }
        else {/* Move SE */
            y--;
            decision += delta_SE;
            delta_E += 2; delta_SE += 4; /*Update delta*/
        }
        x++;
        Pixel(x, y);
    }
}
```

Analysis

- Uses floats!
- 1 test, 3 or 4 additions per pixel
- Initialization can be improved
- Multiply everything by 4 → No Floats!
  - Makes the components even, but sign of decision variable remains same

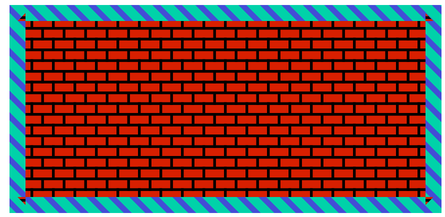


Questions

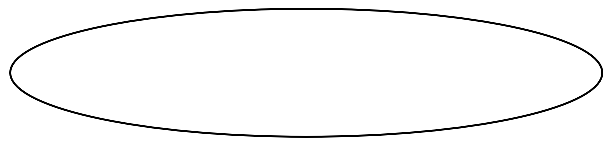
- Why is ***y* > *x*** the right stopping criterion?
- What if it were an ellipse?

Other Scan Conversion Problems

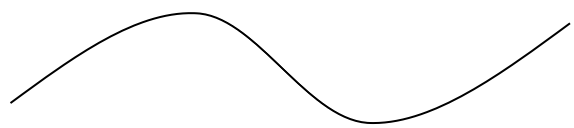
- Patterned primitives



- Aligned Ellipses
  - Only 4-fold symmetry

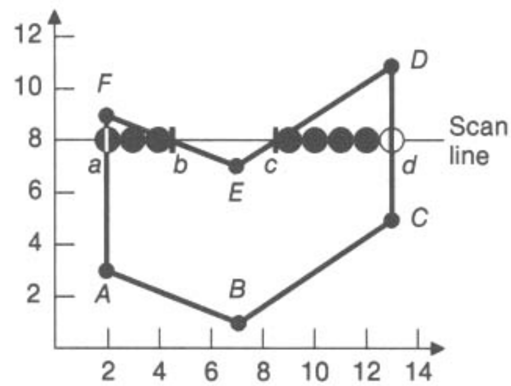


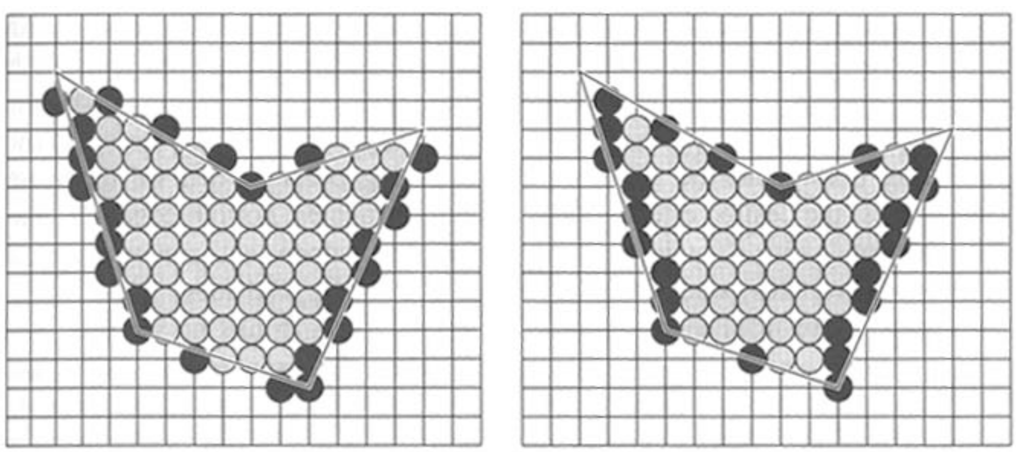
- Non-integer primitives
  - Initialization is harder
  - Endpoints are hard, too
    - making Line (*P*,*Q*) and Line (*Q*,*R*) join properly is a good test
  - Symmetry is lost



- General conics
  - Very hard--the octant-changing test is tougher, the difference computations are tougher, etc.
    - do it only if you have to.

Generic Polygons





What is the difference between these two solutions? Under which circumstances is the right one "better"?

See Balsa demo:
http://www.youtube.com/watch?v=GXi32vnA-2A

Summary

- Scan-conversion has a steep cost
- Silly incremental algorithm; cost
- Mid-point line algo; cost reduction
- Extension to circles and curves; cost
- Extension to polygons; cost

- The cost of scan conversion limits the number of edges we can draw while maintaining interactive rendering rates
- This cost is such a burden, the scan-conversion is delegated to the GPU, relieving the main CPU of some computation
- If your program slows down, consider reducing the level of detail (tessellation) in your geometric shapes
- We'll see the hacky type of shortcuts we'll need to take for shading.