Université Bretagne Sud

Introduction à l’Informatique Graphique

Lecture 1. Drawing 2D Primitives

Caroline Larboulette

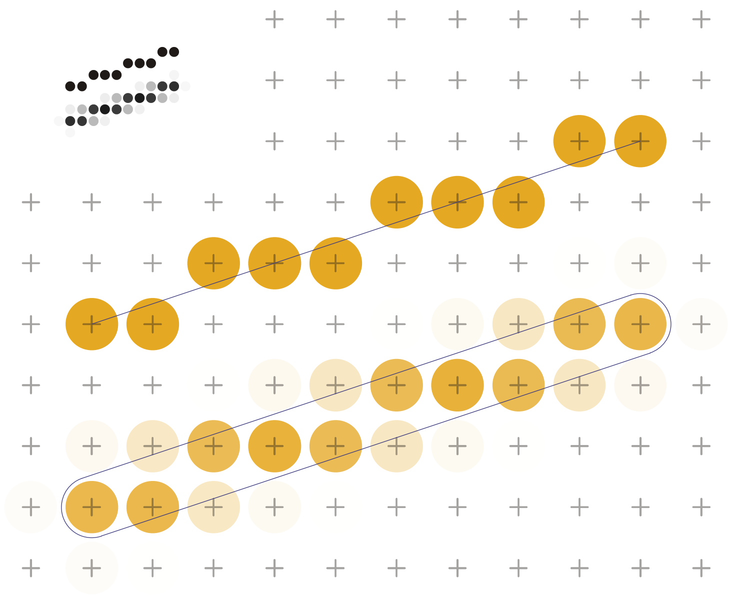
Motivation

Drawing 2D primitives

- Models are mathematical descriptions of geometric elements called primitives
  - lines and segments
  - polygons: quads (2 triangles), triangles, …
  - circles
  - polyhedrons
  - polygonal meshes : connected triangles

Overview
Drawing 2D primitives

1. Scan Conversion
   - Lines
   - Circles
2. Filling Polygons
3. Clipping
4. Generating Characters
5. Antialiasing



1. Scan Conversion

Definitions
Rasterization

- Raster screen (or image) is a screen (or image) discretised in pixels
- Rasterization is the process of taking geometric shapes (defined by vertices and their coordinates) and converting them into an array of pixels stored in the framebuffer to be displayed (b&w or color)

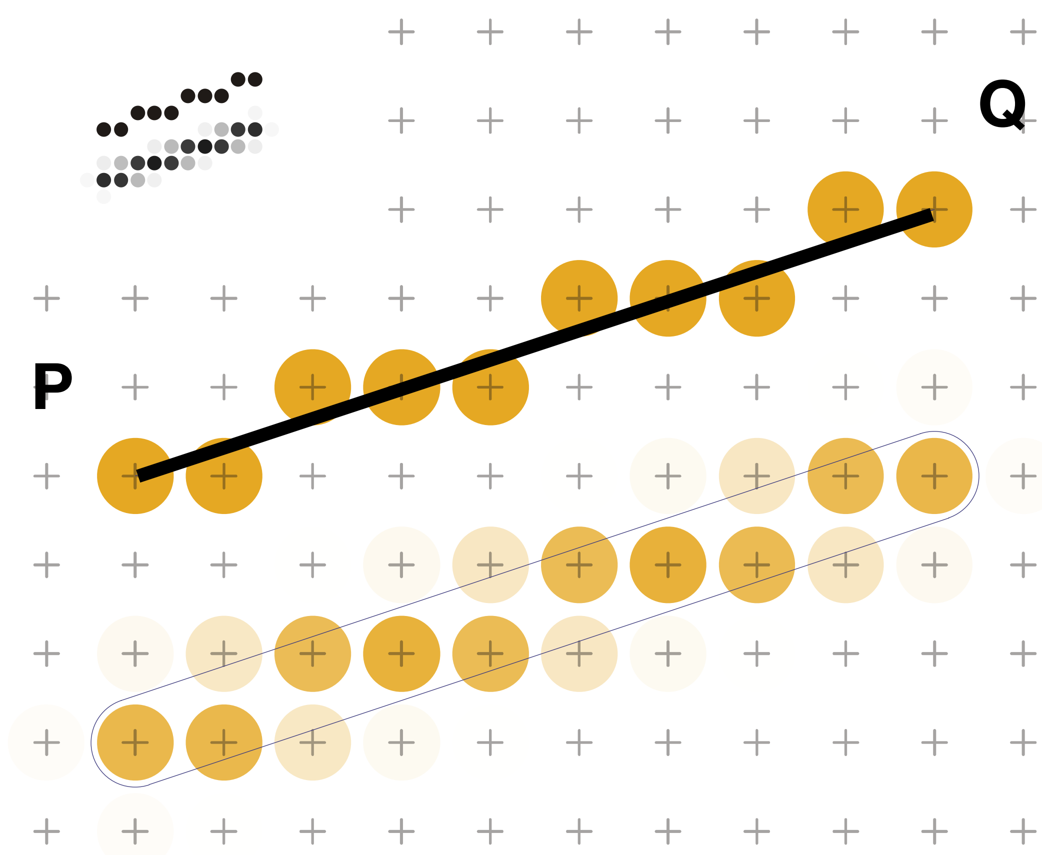
Definitions

Scan Conversion

- Scan conversion is the final step of rasterization (end of the rendering pipeline)
- Takes place after clipping
- Takes triangles (or higher-order primitives) and maps them to pixels on screen
- For 3D rendering also takes into account other processes, like lighting and shading

Scan Converting Lines

- Line Drawing
  - Draw a line on a raster screen between 2 points (P,Q)

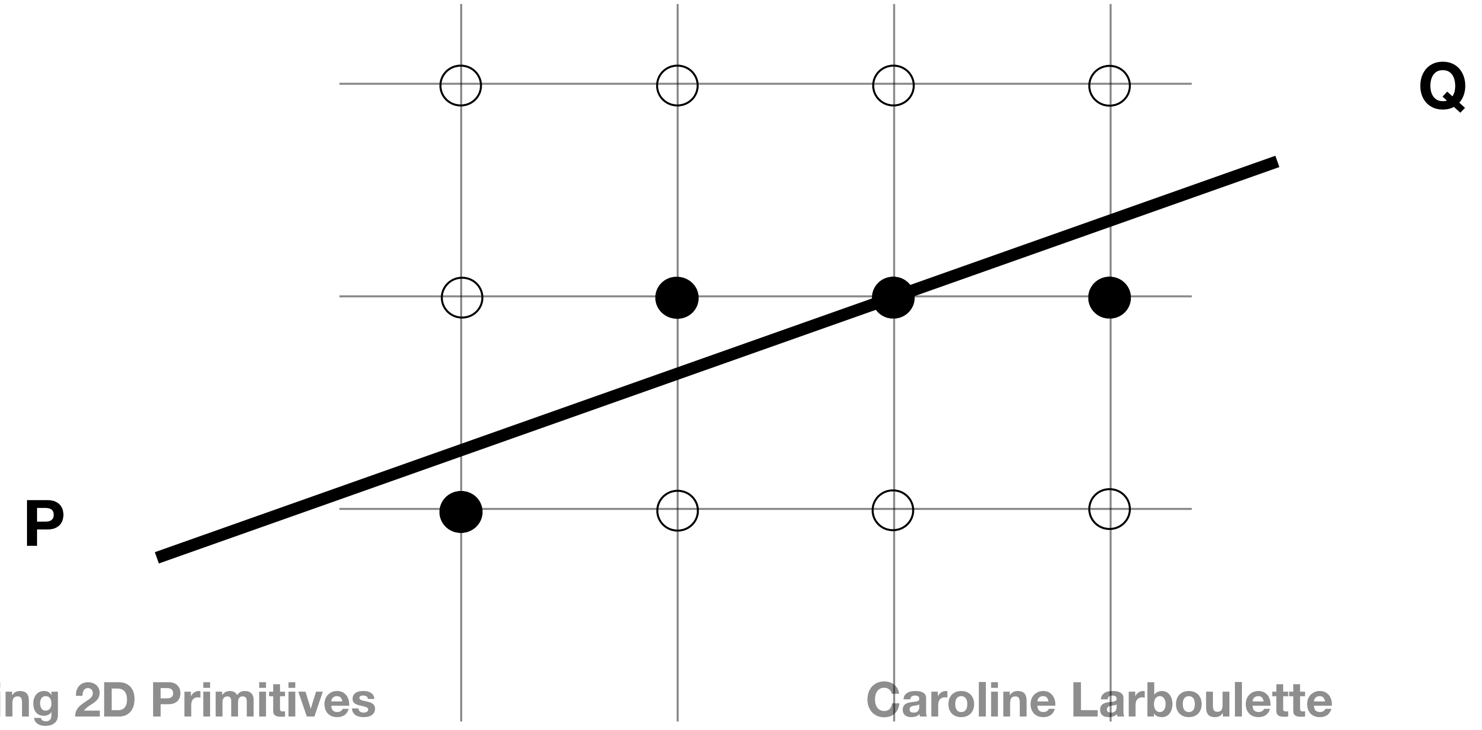


Scan Converting Lines

- Why is it a difficult task ?
  - What is “drawing” on a raster display ?
  - What is a “line” in raster world ?
  - Efficiency and appearance are both important !

Scan Converting Lines
Problem Statement

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

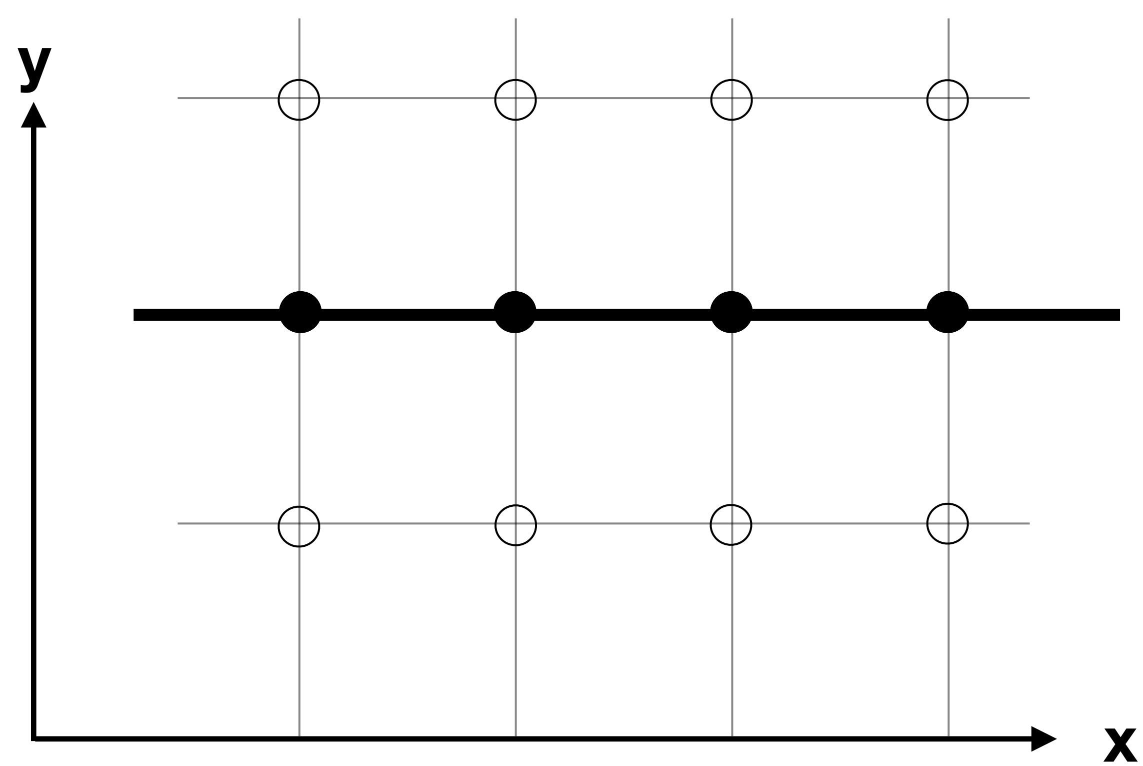


Scan Converting Lines

Special Cases

Scan Converting Lines
Special Cases

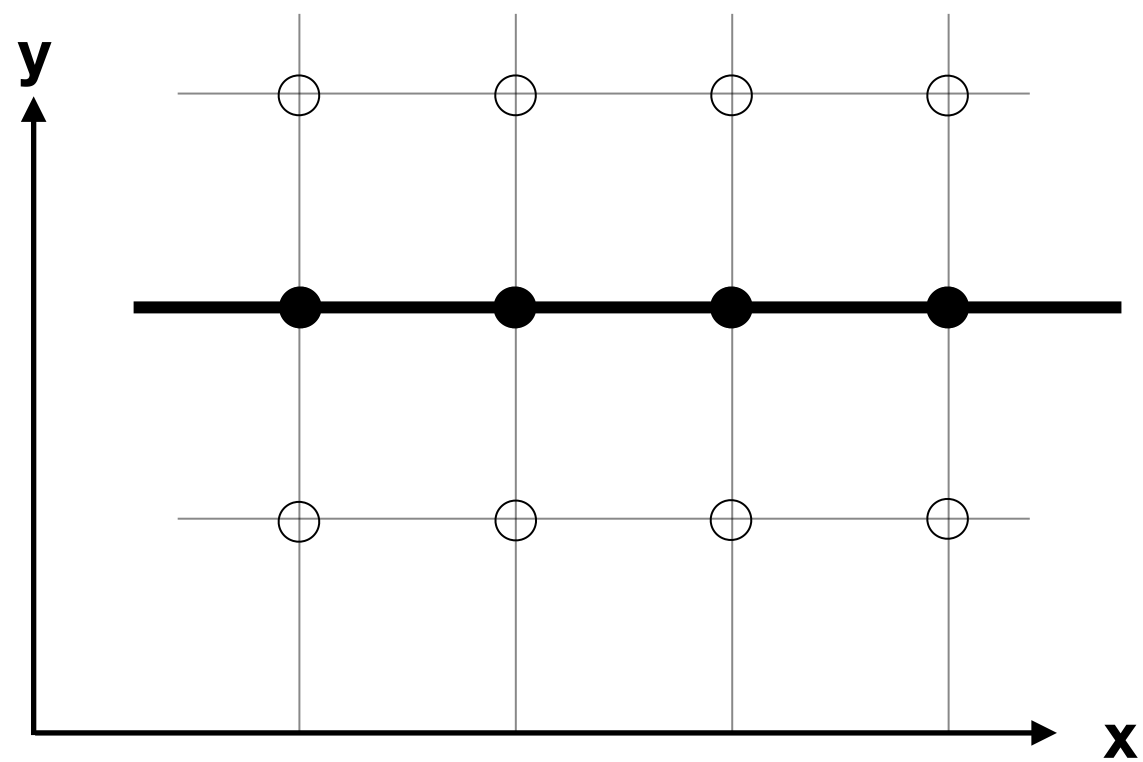
- Horizontal line



Scan Converting Lines

Special Cases

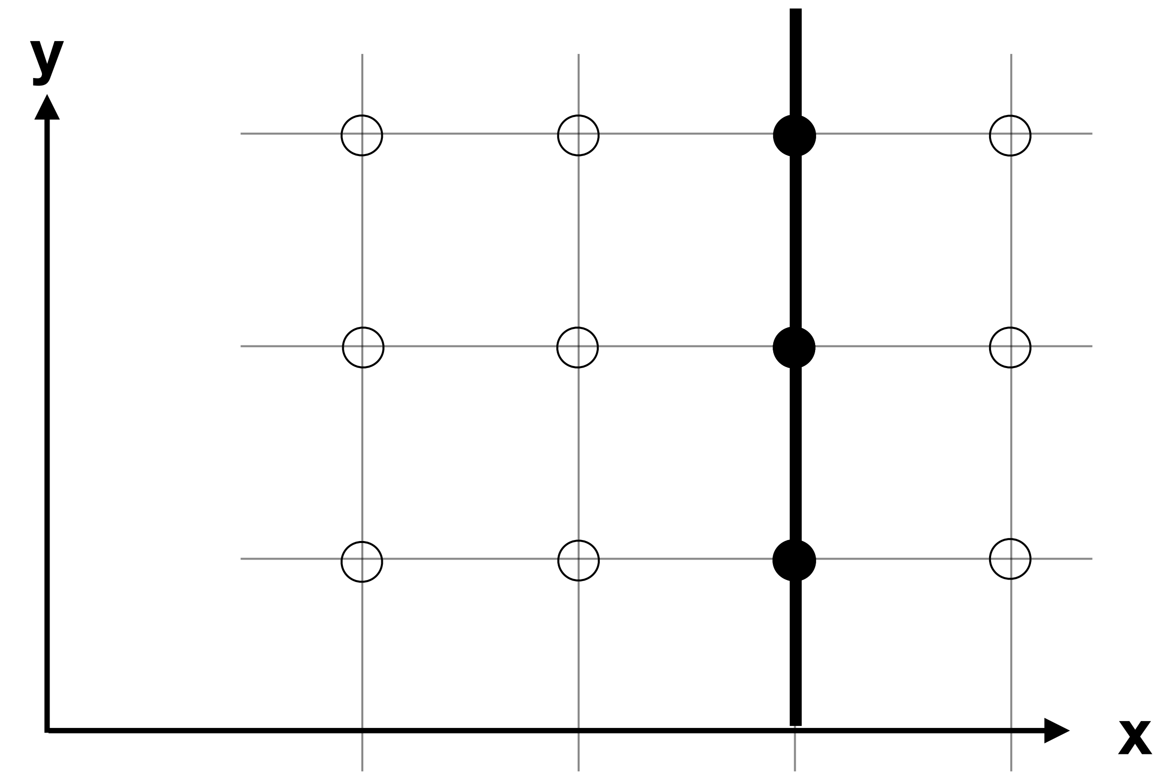
- Horizontal line
  - Draw pixel P and increment x-coordinate value by 1 to get next pixel



Scan Converting Lines

Special Cases

- Horizontal line
  - Draw pixel P and increment x-coordinate value by 1 to get next pixel
- Vertical line



Scan Converting Lines
Special Cases

- 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

Scan Converting Lines
Special Cases

- 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

Scan Converting Lines

Special Cases

- 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 values by 1 to get next pixel

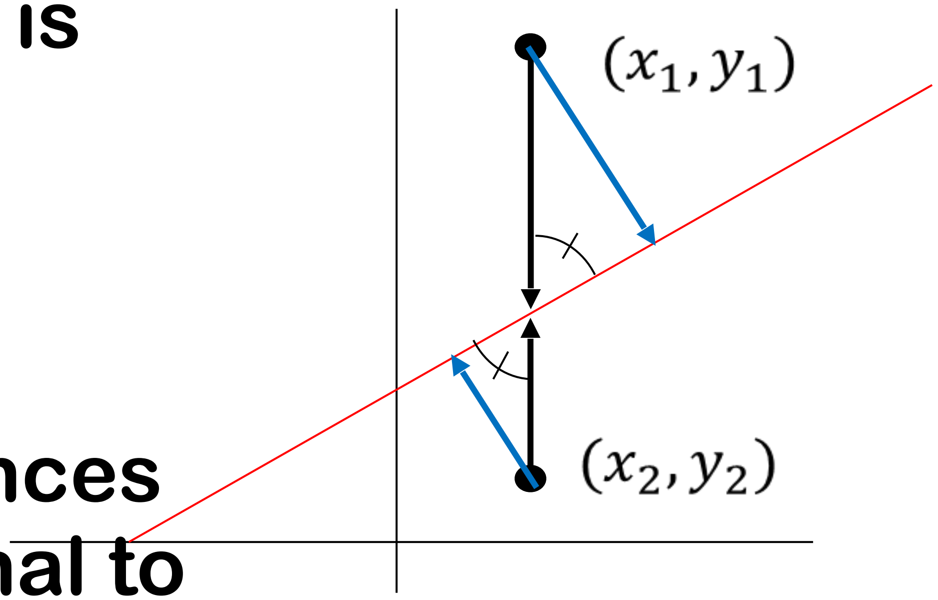
Scan Converting Lines

General Case

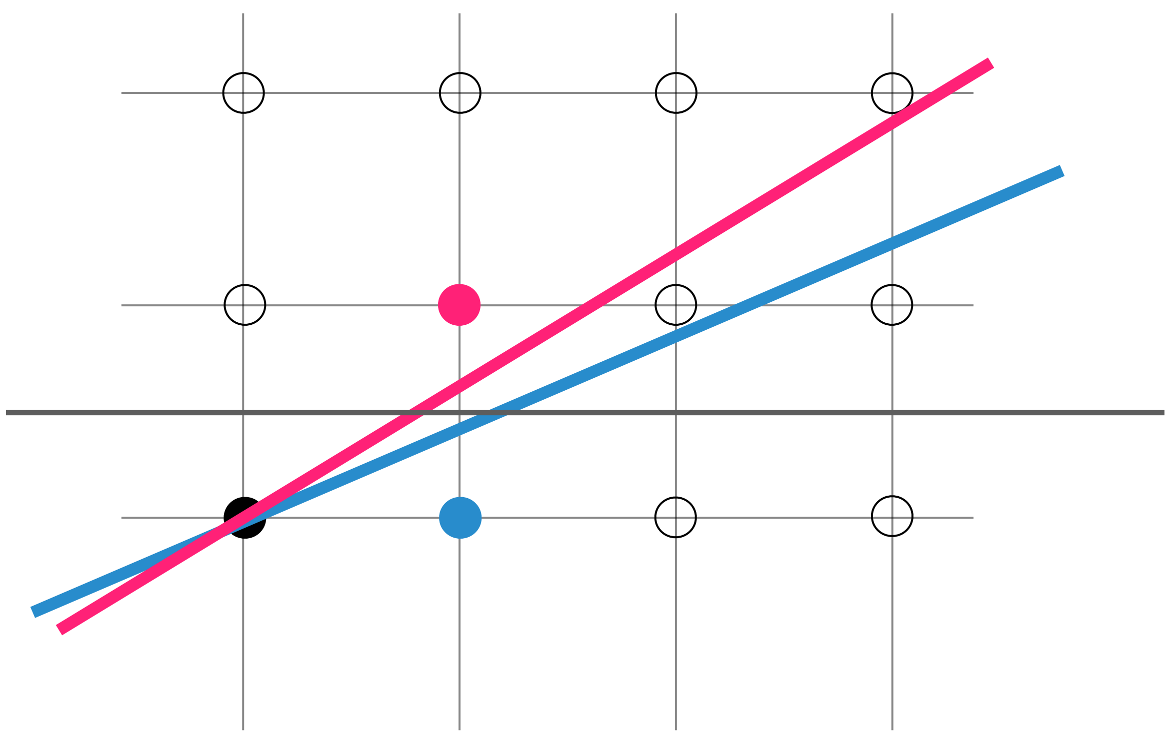
- For slopes m <= 1, increment x-coordinate by 1 and choose pixel on or closest to line.
- For slopes m > 1, increment y-coordinate by 1…
- But how do we measure “closest”?

Vertical Distance

- Why can we use vertical distance as a measure of which point (pixel center) is closer?
  - … because vertical distance is proportional to actual distance
- Similar triangles show 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
- **floor**$(0.5 + y_i)$



Vertical Distance



floor$(0.5 + y_i)$

Scan Converting Lines

1. Basic Algorithm

- Find equation of line that connects 2 points P and Q
- Starting with leftmost point, increment x_i by 1 to calculate $y_i = m \cdot x_i + b$ (m = slope, b = y intercept)
- y_i is a float, round it up to get an int (floor(0.5+ y_i))
- Draw pixel at (x_i, round(y_i))

Scan Converting Lines
1. Basic Algorithm

- Problem: each iteration requires a floating point multiplication, an addition and a floor operation
- Too slow !

Scan Converting Lines
2. Incremental Algorithm

- **Use incremental rather than direct computation**

$$y_i = m \cdot x_i + b$$

$$y_{i+1} = m \cdot x_{i+1} + b$$

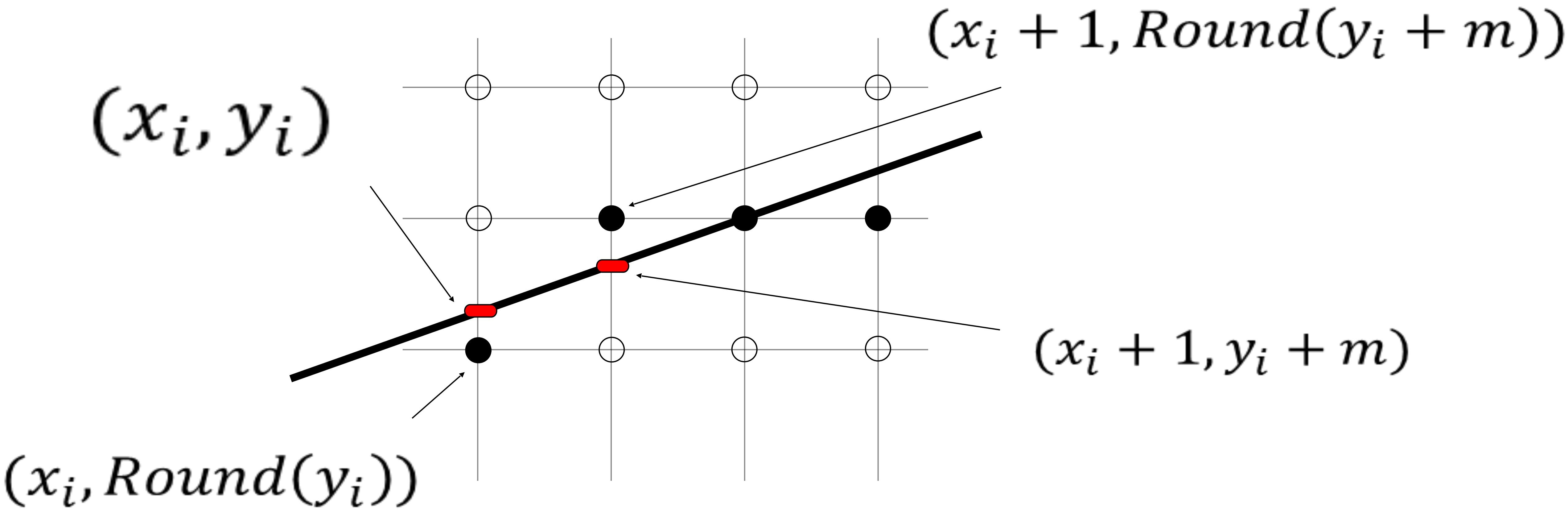
$$y_{i+1} = y_i + m \cdot (x_{i+1} - x_i)$$

But $\Delta x = x_{i+1} - x_i =, 1$ **thus** $y_{i+1} = y_i + m$

- **At each step, we make incremental calculation based on preceding step**

Scan Converting Lines

2. Incremental Algorithm



Scan Converting Lines

2. Incremental Algorithm

```
void Line(int x0, int y0, int x1, int y1)
{
    int   x;
    float 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 += m;
    }
}
```

Since slope is fractional, need special case for vertical lines (dx = 0)

Rounding takes time !

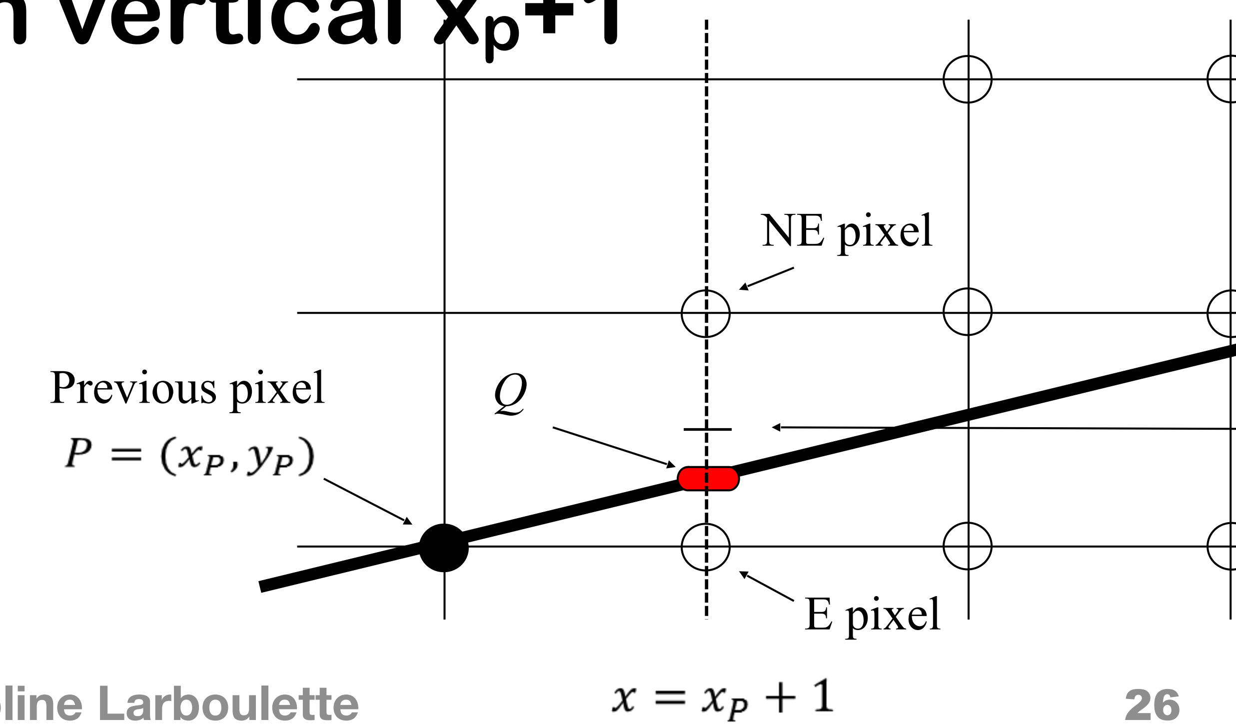
Scan Converting Lines
2. Incremental Algorithm

- Problem: floor operation takes time
- Numerical drift after too many iterations (not a real problem as segments are often short) : y and m are floats

Scan Converting Lines

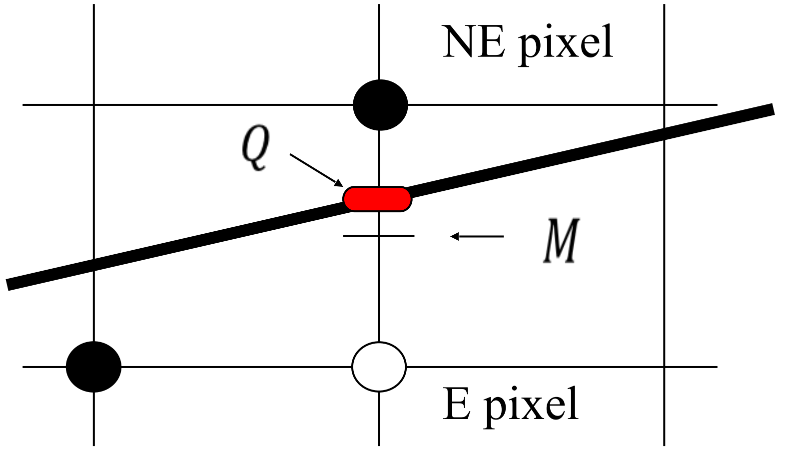
3. Midpoint Line Algorithm

- For line slope shallow and positive ($0 < m < 1$)
- Assume we have just selected pixel at x_p, y_p
- Must choose between pixel to right (E pixel) or the one right and up (NE pixel)
- Q: intersection of line with vertical x_p+1



Scan Converting Lines

3. Midpoint Line Algorithm

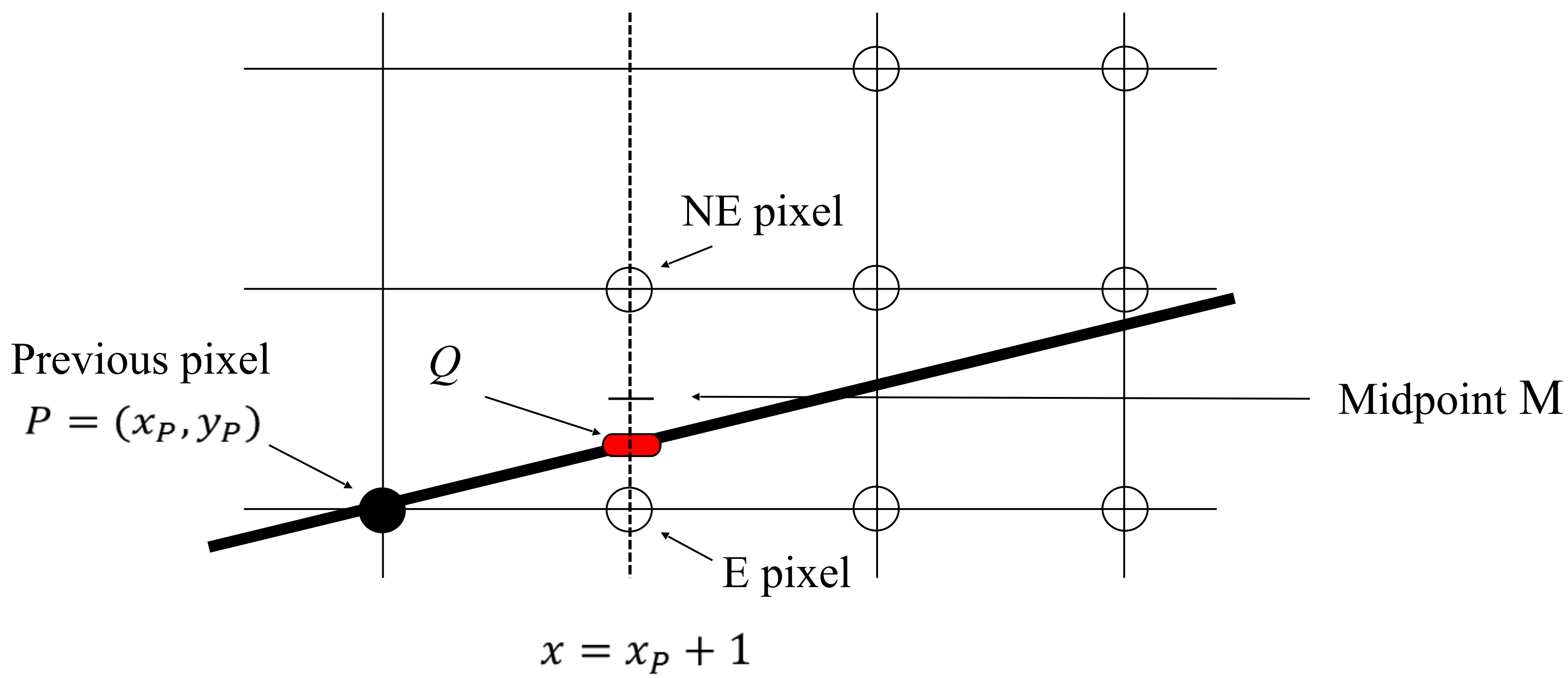


- Line passes between E and NE
- Point closer to intersection point Q
- M is midpoint between E and NE, $M(x_p + 1, y_p + 0.5)$
  - E is closer to line if M is above line
  - NE is closer to line id M is below line
- Error (vertical distance between chosen pixel and line) is always <= 0.5

Scan Converting Lines

3. Midpoint Line Algorithm

- How do we calculate on which side of line M lies ?



Line Equations

- Slope-intercept form $f(x) = y = m \cdot x + b$
- Point-slope form $y - y_0 = m(x - x_0)$
- Implicit form $f(x, y) = ax + by + c = 0$
  - Avoids infinite slopes
  - Provides symmetry between x and y

$$f(x) = y = m \cdot x + B$$

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

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

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

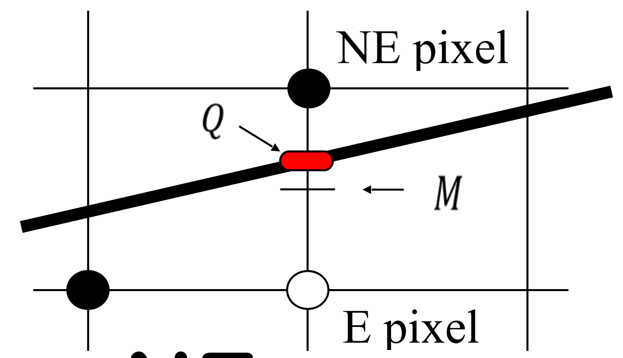
$$a = dy$$
$$b = -dx$$
$$c = dx \cdot B$$

Line Equations

- Properties of the implicit form
  - $f(x_i, y_i) = 0$ when any point i is on the line
  - $f(x_i, y_i) < 0$ when any point i is above the line
  - $f(x_i, y_i) > 0$ when any point i is below the line
- Hence decision based on value of function at midpoint $i = M(x_p + 1, y_p + 0.5)$

Scan Converting Lines

3. Midpoint Line Algorithm



- Let d be the decision variable

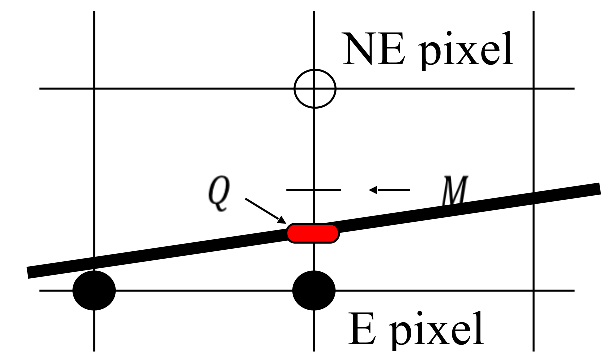
$$d = f(M) = f(x_p + 1, y_p + 0.5)$$

- if d > 0 , line is above midpoint, choose NE
- if d < 0, line is below midpoint, choose E
- if d = 0, line is on midpoint, choose either one consistently (E by default)
- Problem: how to incrementally update d ?
    - On the basis of choosing E or NE, we can derive d for next pixel

Scan Converting Lines

3. Midpoint Line Algorithm

- Incrementing decision variable if E was chosen
- Increment M by 1 in x-direction



$$d_{old} = a(x_p + 1) + b(y_p + 0.5) + c$$

$$d_{new} = f(x_p + 2, y_p + 0.5)$$

$$d_{new} = a(x_p + 2) + b(y_p + 0.5) + c$$

$$d_{new} = d_{old} + a = d_{old} + dy$$

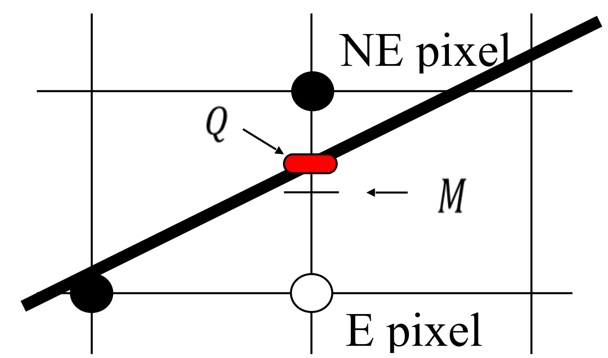
- $\Delta E = d_{new} - d_{old} = dy$ is referred to as forward difference (it's a correction factor)

Scan Converting Lines

3. Midpoint Line Algorithm

- Incrementing decision variable if NE was chosen
- Increment M by 1 in x-direction

  and 1 in y-direction



$$d_{old} = a(x_p + 1) + b(y_p + 0.5) + c$$

$$d_{new} = f(x_p + 2, y_p + 1.5)$$

$$d_{new} = a(x_p + 2) + b(y_p + 1.5) + c$$

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

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

Scan Converting Lines

3. Midpoint Line Algorithm

- Loop
  - At each iteration, choose between pixels E or NE based on sign of variable d computed in previous iteration
  - Update d by adding ΔE or ΔNE depending on the decision taken
- Init
  - First pixel is first endpoint (x_0, y_0)
  - First midpoint is at $(x_0 + 1, y_0 + 0.5)$

Scan Converting Lines

3. Midpoint Line Algorithm

- **Init (continued)**
  - **First** $d = f(x_0 + 1, y_0 + 0.5) = a(x_0 + 1) + b(y_0 + 0.5) + c$

    $$d = a \cdot x_0 + b \cdot y_0 + a + \frac{b}{2} + c = f(x_0, y_0) + a + \frac{b}{2}$$
  - **But by definition** (x_0, y_0) **is on line, so** $f(x_0, y_0) = 0$

    $$d = a + \frac{b}{2} = dy - \frac{dx}{2}$$
- **To eliminate fraction in** d**, redefine** $f(x, y)$ **by multiplying by 2**
  - **Each constant is also multiplied by 2**

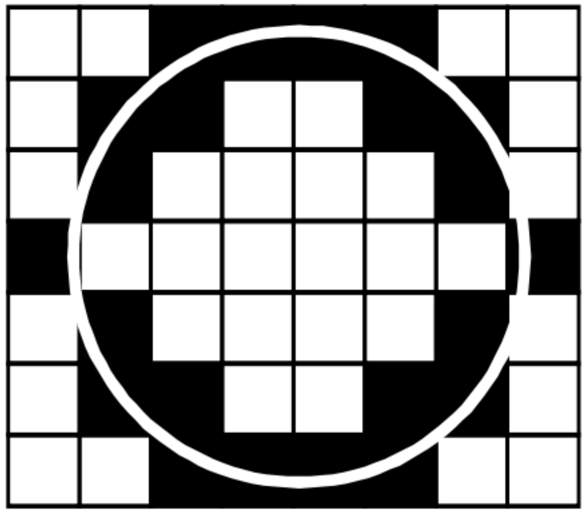
Scan Converting Lines

3. Midpoint Line Algorithm

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

    while (x < x1)
    {
        if (d <= 0) d += incrE;          // East Case
        else { d += incrNE; ++y; }       // NorthEast Case
        ++x;
        WritePixel(x, y);
    }
}
```

Scan Converting Circles

Circle Equations



- Explicit equation: $R^2 = x^2 + y^2$

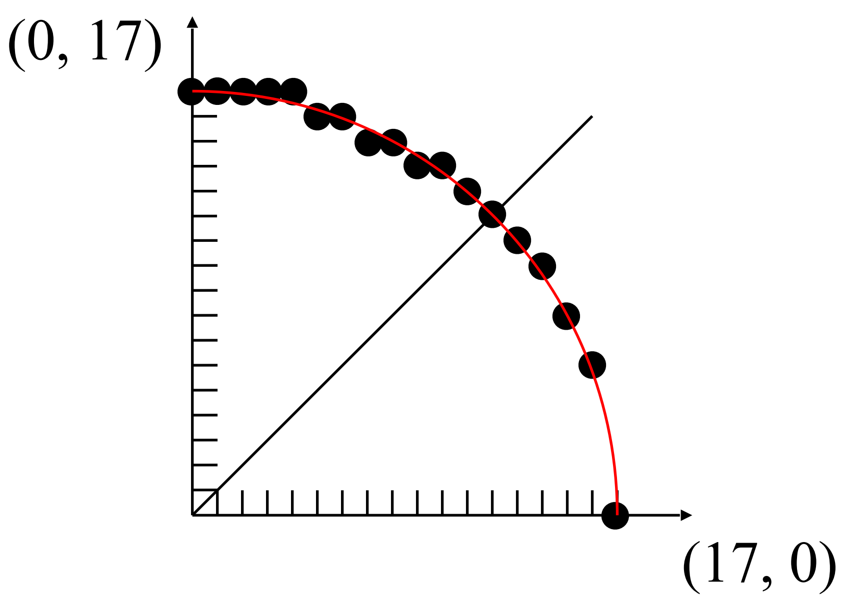
$$y = \sqrt{(R^2 - x^2)}$$

- cercle(center, R):
  - x = R * cos(alpha) + xcenter
  - y = R * sin(alpha) + ycenter
  - alpha: angle from 0 to 2Π
- Implicit equation: $f(x,y) = x^2 + y^2 - R^2 = 0$
  - $f(x,y) = 0$ on circle
  - $f(x,y) < 0$ inside
  - $f(x,y) > 0$ outside

Scan Converting Circles
1. Basic Algorithm

- **Using explicit equation** $y = \sqrt{(R^2 - x^2)}$

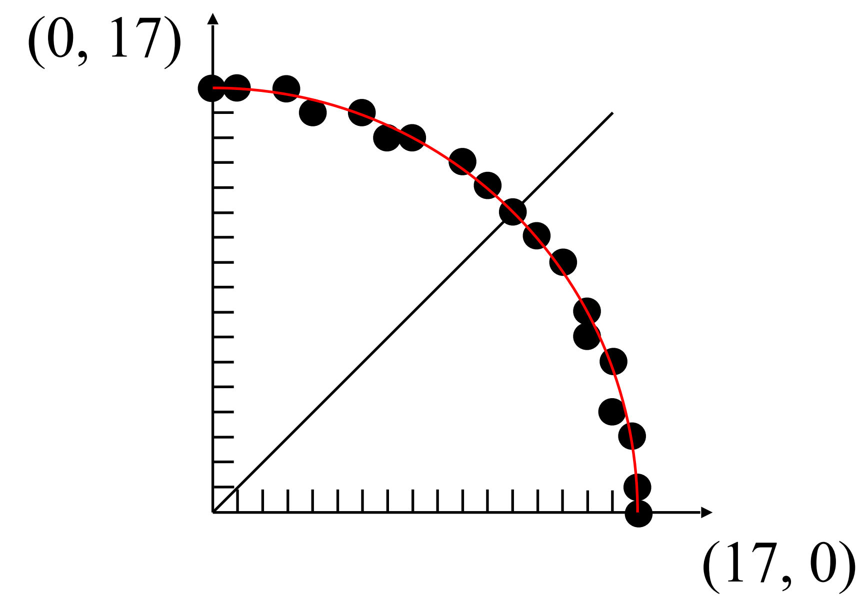
```
for (x from -R to +R)
{
    y = sqrt(R*R - x*x);
    WritePixel(round(x), round(y));
    WritePixel(round(x), round(-y));
}
```



- **Really inefficient !**

Scan Converting Circles

2. Basic Algorithm version 2



- Coordinates in polar form

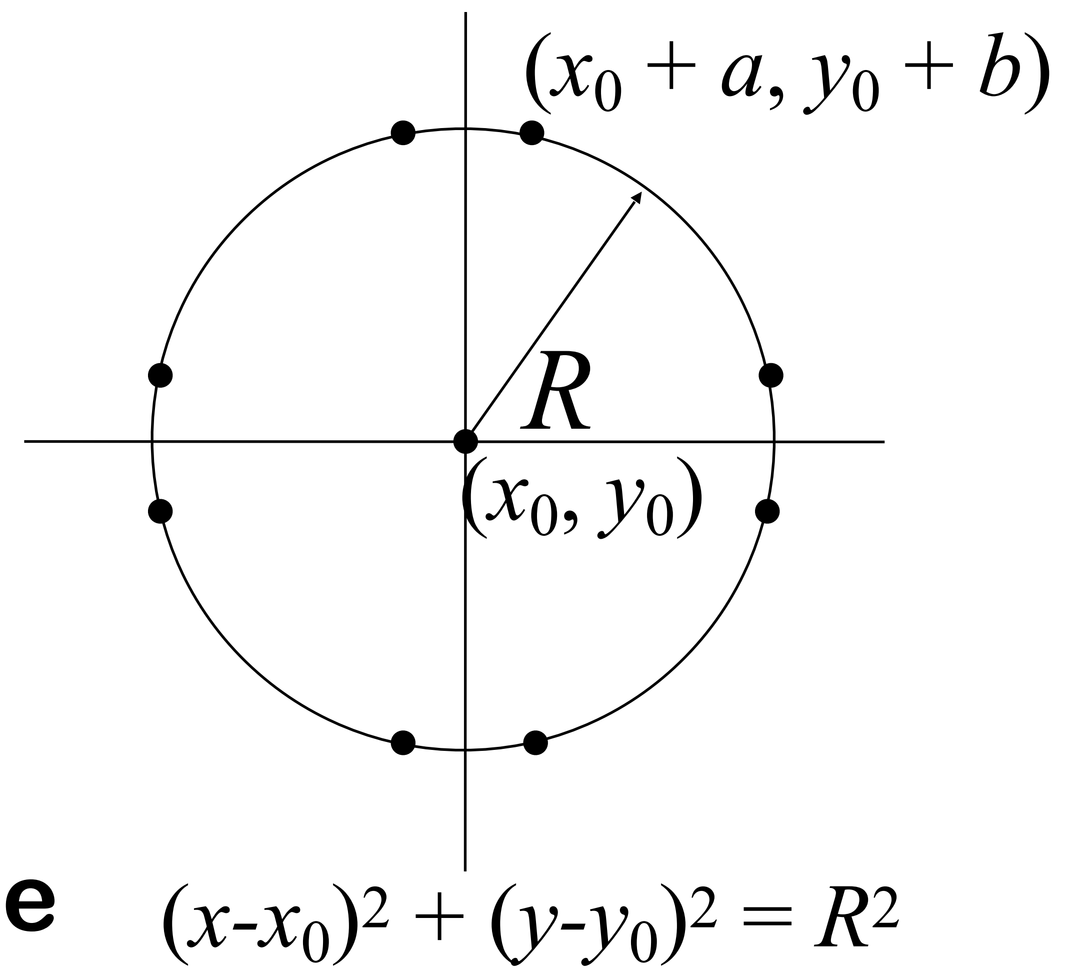
```
for (x from 0 to 360)
{
    WritePixel(round(R.cos(x)), round(R.sin(x)));
}
```

- Slightly less bad but still inefficient !

Scan Converting Circles

Using Symmetry

- If $(x_0 + a, y_0 + b)$ is on circle centered at (x_0, y_0)
  - $(x_0 \pm a, y_0 \pm b)$ is on circle
  - $(x_0 \pm b, y_0 \pm a)$ is on circle
  - 8-way symmetry
- Reduces the problem to finding the pixels for 1/8 of the circle



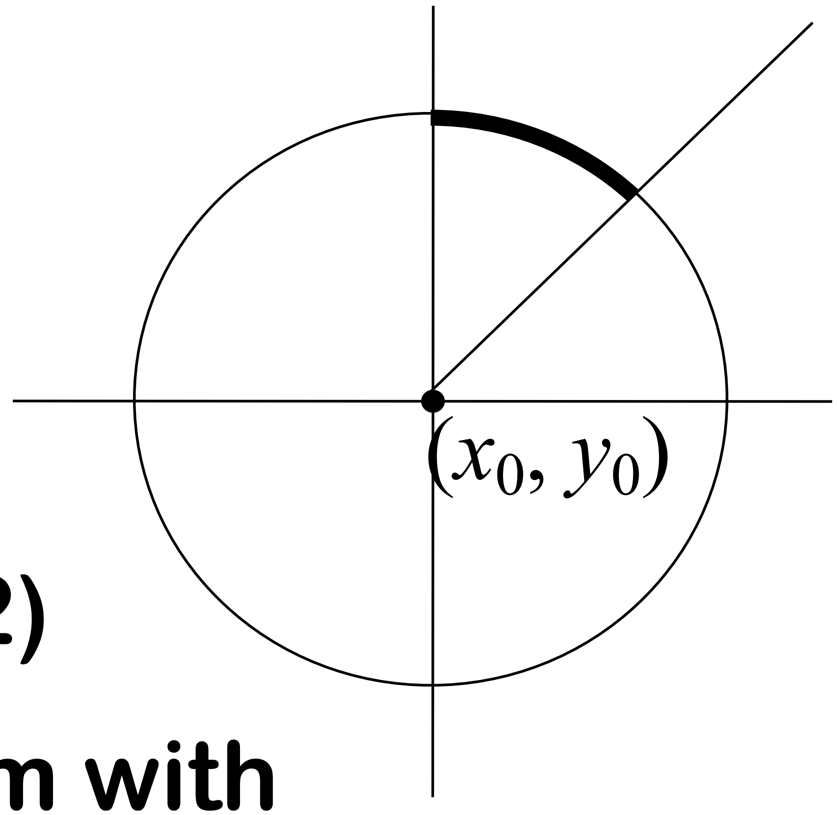
Scan Converting Circles

Using Symmetry

```
x = x0 + a;
y = y0 + b;

void CirclePoints(float x, float y)
{
    WritePixel(x, y);
    WritePixel(x, -y);
    WritePixel(-x, y);
    WritePixel(-x, -y);
    WritePixel(y, x);
    WritePixel(y, -x);
    WritePixel(-y, x);
    WritePixel(-y, -x);
}
```

Special case: x = y !

Scan Converting Circles
Using Symmetry

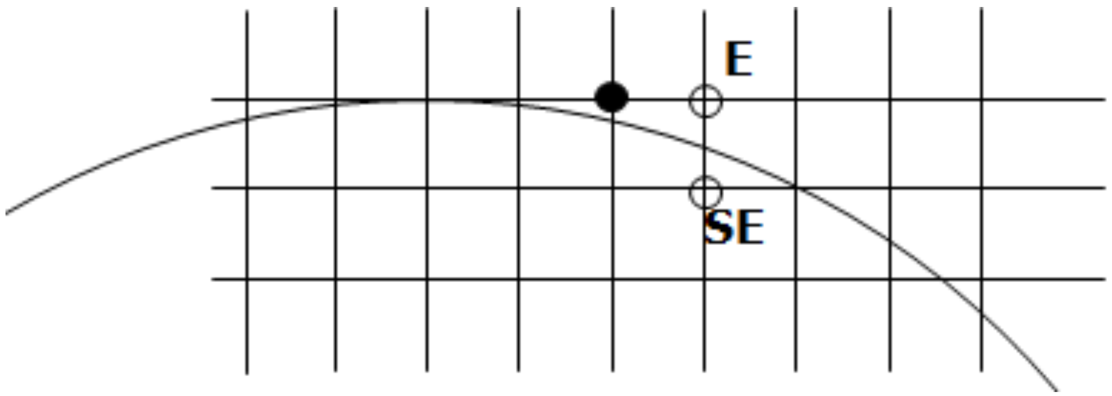


- Scan top 1/8 of circle of radius R
- Start at $(x_0, y_0 + R)$
- Loop from x = 0 to x = y = R / sqrt(2)
- Goal: use an incremental algorithm with decision variable evaluated at midpoint

Scan Converting Circles

3. Incremental Algorithm

```
x = x0, y = y0 + R; WritePixel(x, y);

for (x = x + 1; (x - x0) > (y - y0); x++) {
    if (decision_var < 0) {
        // move east
        update decision variable
    } else {
        // move south east
        update decision variable
        y--;
    }

    WritePixel(x, y);
}
```



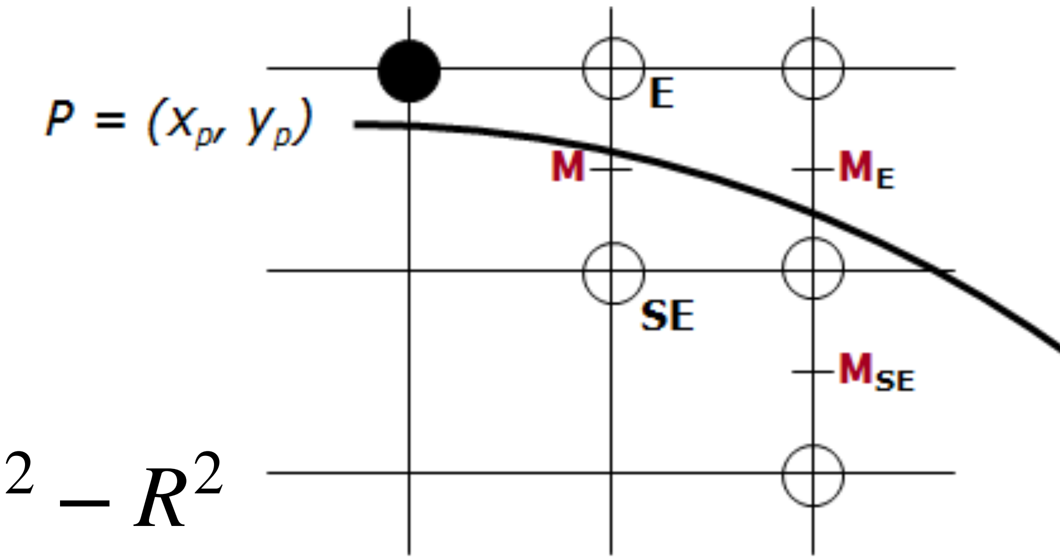
Note: can replace all occurrences of x_0, y_0 with 0, shifting coordinates by $(-x_0, -y_0)$

Scan Converting Circles
3. Incremental Algorithm

- Decision variable
  - Move E if positive
  - Move SE if negative
- Use implicit equation of circle
- Compute f at midpoint
- If E, next pixel is at x+1, y
- if SE, next pixel is at x+1, y-1

Scan Converting Circles

3. Incremental Algorithm



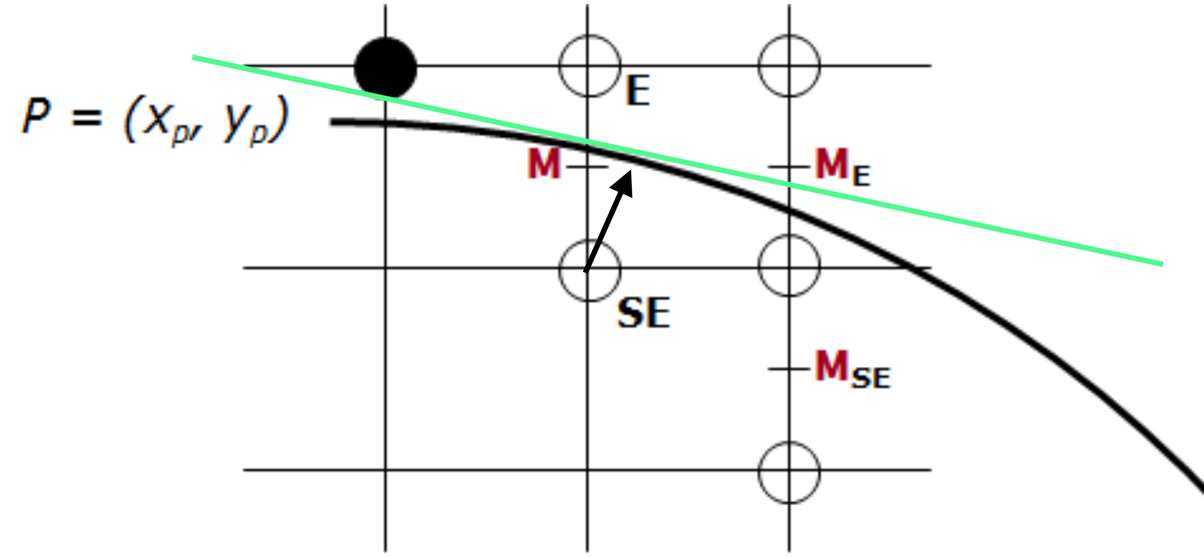
- **Midpoint is at M(x+1, y-0.5)**

$$f(M) = f(x+1, y-0.5) = (x+1)^2 + (y-0.5)^2 - R^2$$

  - **If f(M) > 0, midpoint is inside the circle, choose E**
  - **if f(M) < 0, midpoint is outside the circle, choose SE**

- **Note that it implies we use a vertical distance decision which is a linear approximation of the radial distance decision**

The right decision variable?



- Decision based on vertical distance
- Ok for lines, since d and d_{vert} are proportional
- For circles, not true:

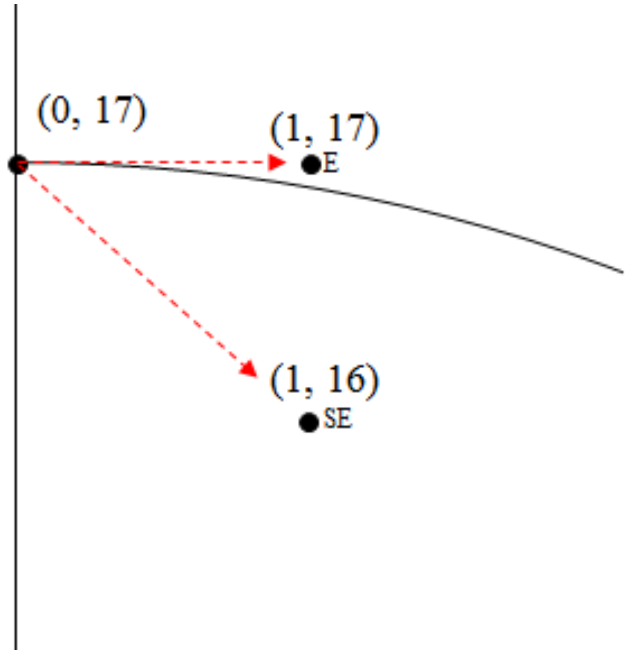
$$d\big((x+1,y),Circ\big) = \sqrt{(x+1)^2 + y^2} - R$$
$$d\big((x+1,y-1),Circ\big) = \sqrt{(x+1)^2 + (y-1)^2} - R$$

- Which d is closer to zero? (i.e., which value below is closest to R?):

$$\sqrt{(x+1)^2 + y^2} \text{ or } \sqrt{(x+1)^2 + (y-1)^2}$$

Alternate Phrasing (1/3)

- We could ask instead: "Is $(x+1)^2 + y^2$ or $(x+1)^2 + (y-1)^2$ closer to R^2?"
- The two values in equation above differ by:
- $[(x+1)^2 + y^2] - [(x+1)^2 + (y-1)^2] = 2y - 1$



$f_E = 1^2 + 17^2 = 290$
$f_{SE} = 1^2 + 16^2 = 257$

$f_E - f_{SE} = 290 - 257 = 33$
$2y - 1 = 2(17) - 1 = 33$

Alternate Phrasing (2/3)

- The second value, which is always less, is *closer* if its difference from R^2 is less than: $\frac{1}{2}(2y-1)$

i.e., if $\quad R^2 - [(x+1)^2 + (y-1)^2] < \frac{1}{2}(2y-1)$

then
$$0 < y - \frac{1}{2} + (x+1)^2 + (y-1)^2 - \mathrm{R}^2$$
$$0 < (x+1)^2 + y^2 - 2y + 1 + y - \frac{1}{2} - R^2$$
$$0 < (x+1)^2 + y^2 - y + \frac{1}{2} - R^2$$
$$0 < (x+1)^2 + (y - \frac{1}{2})^2 + \frac{1}{4} - R^2$$

Alternate Phrasing (3/3)

- The ***radial distance decision*** is whether

$$d_1 = (x+1)^2 + \left(y - \frac{1}{2}\right)^2 + \frac{1}{4} - R^2$$

  is positive or negative.

- The ***vertical distance decision*** is whether

$$d_2 = (x+1)^2 + \left(y - \tfrac{1}{2}\right)^2 - R^2$$

  is positive or negative; d_1 and d_2 are ¼ apart.

- The integer d_1 is positive only if $d_2 +$ ¼ is positive (except special case where $d_2 = 0$: remember you're using integers).

Scan Converting Circles

3. Incremental Algorithm

- Incremental computation of f(M)

$$f(M) = f(x+1, y-0.5) = (x+1)^2 + (y-0.5)^2 - R^2$$

- If moving E

$$\Delta E = f(x+1, y) - f(x, y) = 2x + 3$$

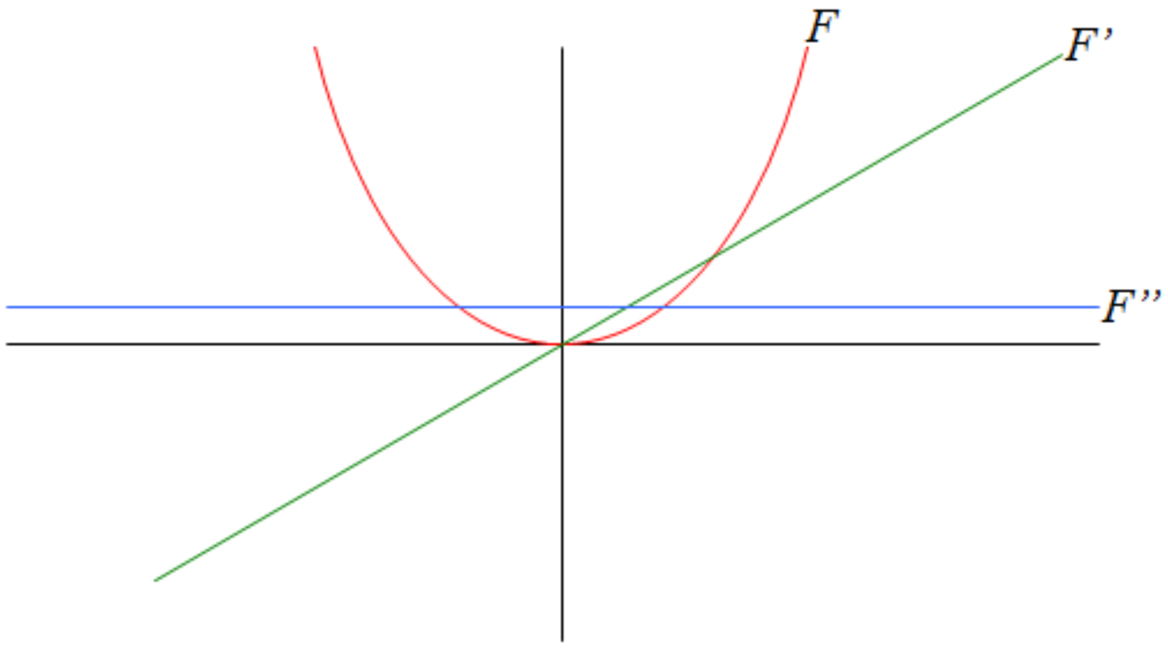
- If moving SE

$$\Delta SE = f(x+1, y-1) - f(x, y) = 2x - 2y + 5$$

- Difference with line: now deltaE and delta SE need to be updated as they depend on x, y

Incremental Computation (2/2)

- If we move E, update d = f(M) by adding $2x + 3$
- If we move SE, update d by adding $2x - 2y + 5$
- Forward differences of a 1st degree polynomial are constants and those of a 2nd degree polynomial are 1st degree polynomials
  - this "first order forward difference," like a partial derivative, is one degree lower



Second Differences (1/2)

- The function $\Delta_{\mathrm{E}}(x, y) = 2x + 3$ is linear, hence amenable to incremental computation:

$$\Delta_{\mathrm{E}}(x+1, y) - \Delta_{\mathrm{E}}(x, y) = 2$$ **East**

$$\Delta_{\mathrm{E}}(x+1, y-1) - \Delta_{\mathrm{E}}(x, y) = 2$$ **South East**

- Similarly

$$\Delta_{\mathrm{SE}}(x+1, y) - \Delta_{\mathrm{SE}}(x, y) = 2$$ **East**

$$\Delta_{\mathrm{SE}}(x+1, y-1) - \Delta_{\mathrm{SE}}(x, y) = 4$$ **South East**

Second Differences (2/2)

- For any step, can compute new $\Delta_{\mathrm{E}}(x, y)$ from old $\Delta_{\mathrm{E}}(x, y)$ by adding appropriate second constant increment – update delta terms as we move. This is also true of $\Delta_{\mathrm{SE}}(x, y)$.
- Having drawn pixel (a, b), decide location of new pixel at $(a+1, b)$ or $(a\ +\ 1, b\ –\ 1)$, using previously computed $\Delta(a, b)$
- Having drawn new pixel, must update $\Delta(a, b)$ for next iteration; need to find either $\Delta(a\ +\ 1, b)$ *or* $\Delta(a\ +\ 1, b\ –\ 1)$ depending on pixel choice
- Must add $\Delta_E(a, b)$ or $\Delta_{SE}(a, b)$ to $\Delta(a, b)$
- So we…
  - Look at d to decide which to draw next, update x and y
  - Update d using $\Delta_E(a, b)$ *or* $\Delta_{SE}(a, b)$
  - Update each of $\Delta_E(a, b)$ *and* $\Delta_{SE}(a, b)$ for future use
  - Draw pixel

Midpoint Eighth Circle Algorithm

```
MidpointEighthCircle(R)
{ /* 1/8th of a circle w/ radius R */
    int x = 0, y = R;
    int deltaE   = 2 * x + 3;
    int deltaSE  = 2 * (x - y) + 5;
    float decision = 5.0/4 – R;        //f(1,R-0.5)
    WritePixel(x, y);

    while ( y > x )
    {
        if (decision > 0)
        { // Move East
            x++; WritePixel(x, y);
            decision += deltaE;
            deltaE += 2; deltaSE += 2; // Update deltas
        } else
        { // Move SouthEast
            y--; x++; WritePixel(x, y);
            decision += deltaSE;
            deltaE += 2; deltaSE += 4; // Update deltas
        }
    }
}
```

Midpoint Circle Algorithm

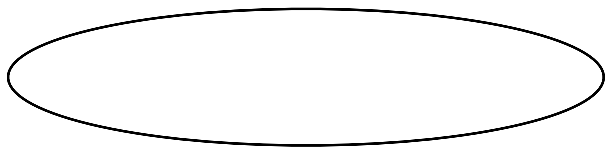
```
MidpointCircle(R)
{ /* the entire circle with radius R */
    int x = 0, y = R;
    int deltaE   = 2 * x + 3;
    int deltaSE  = 2 * (x - y) + 5;
    float decision = 5.0/4 – R;
    CirclePoints(x, y);

    while ( y > x )
    {
        if (decision > 0)
        { // Move East
            x++; CirclePoints(x, y);
            decision += deltaE;
            deltaE += 2; deltaSE += 2; // Update deltas
        } else
        { // Move SouthEast
            y--; x++; CirclePoints(x, y);
            decision += deltaSE;
            deltaE += 2; deltaSE += 4; // Update deltas
        }
    }
}
```

Analysis

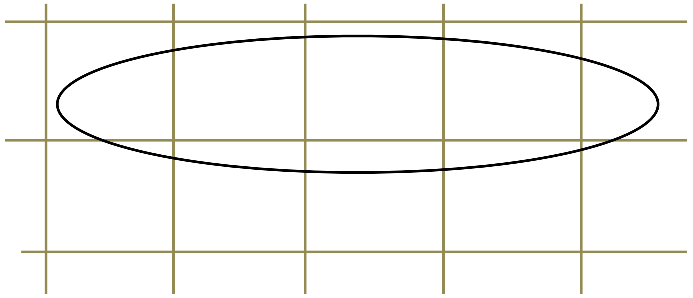
- Uses floats!
- 1 test, 3 or 4 additions per pixel
- Initialization can be improved
- Multiply everything by 4: No Floats!
  - Makes the components even, but sign of decision variable remains same

Other Scan Conversion Problems

- Aligned Ellipses

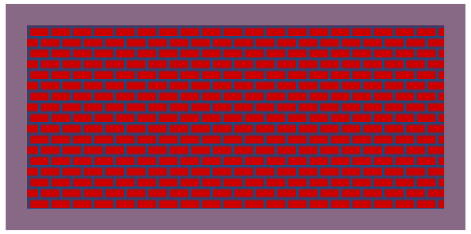


- Non-integer primitives



- General conics

- Patterned primitives

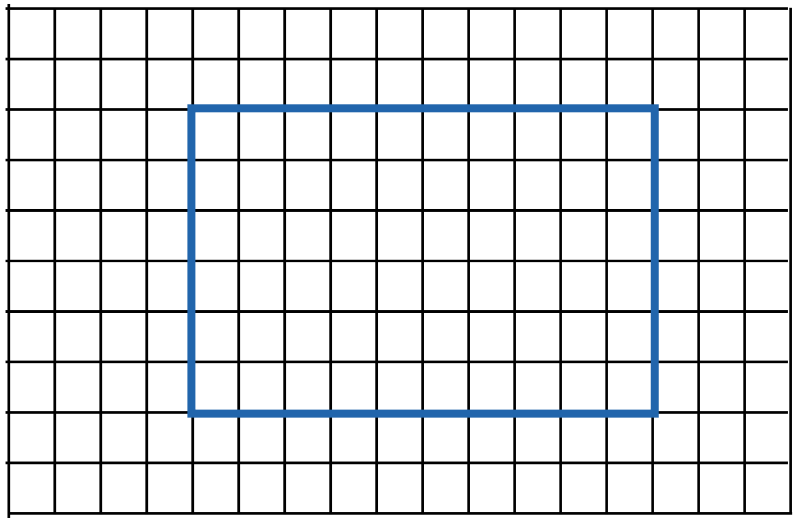


2. Filling Polygons

Filling Rectangles

- From left to right: color the adjacent pixels between x_{min} and x_{max}

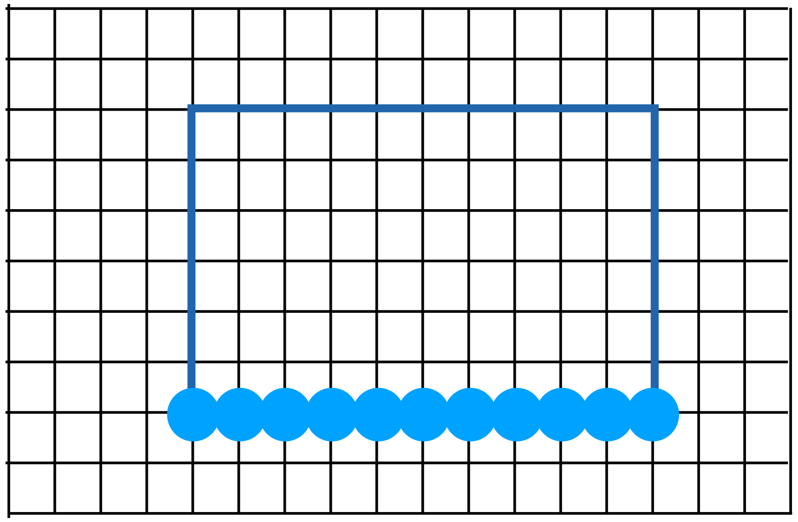
```
for ( y from ymin to ymax )
{
  for ( x from xmin to xmax )
  {
    WritePixel(x, y);
  }
}
```



Filling Rectangles

- From left to right: color the adjacent pixels between x_{min} and x_{max}

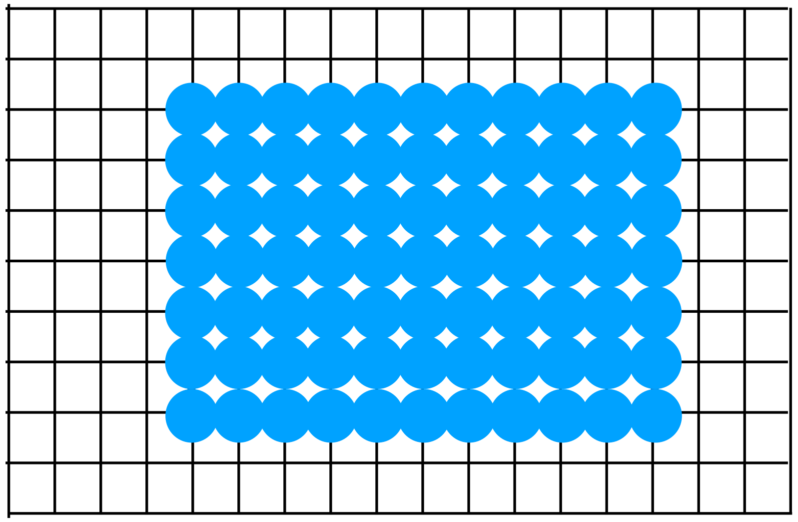
```
for ( y from ymin to ymax )
{
  for ( x from xmin to xmax )
  {
    WritePixel(x, y);
  }
}
```



Filling Rectangles

- From left to right: color the adjacent pixels between x_{min} and x_{max}

```
for ( y from ymin to ymax )
{
  for ( x from xmin to xmax )
  {
    WritePixel(x, y);
  }
}
```



Filling Rectangles

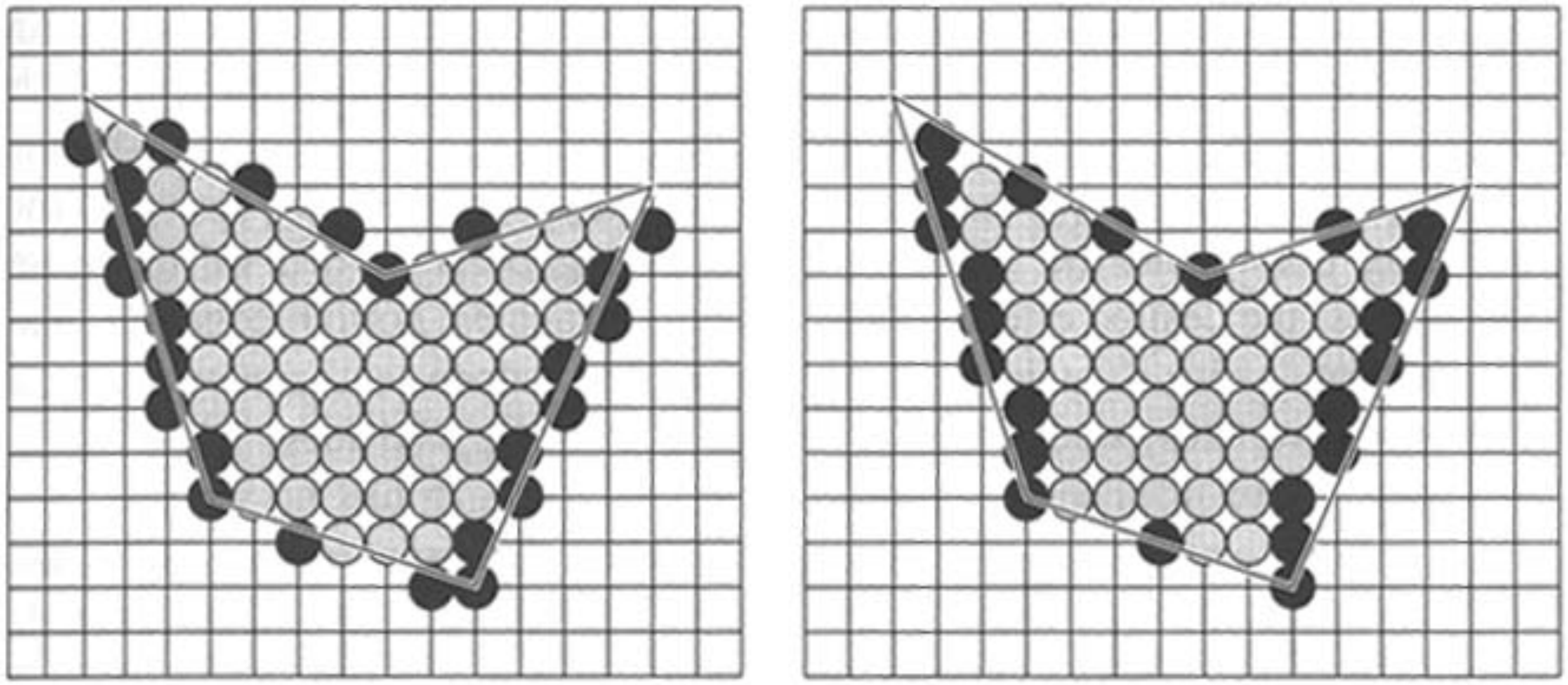
- Problem: what if two rectangles are next to each other ? border pixels drawn twice ?

Filling Rectangles

- Problem: what if two rectangles are next to each other ? border pixels drawn twice ?
- Solution: pixels up and right are not drawn
- There is no perfect solution

Filling Polygons

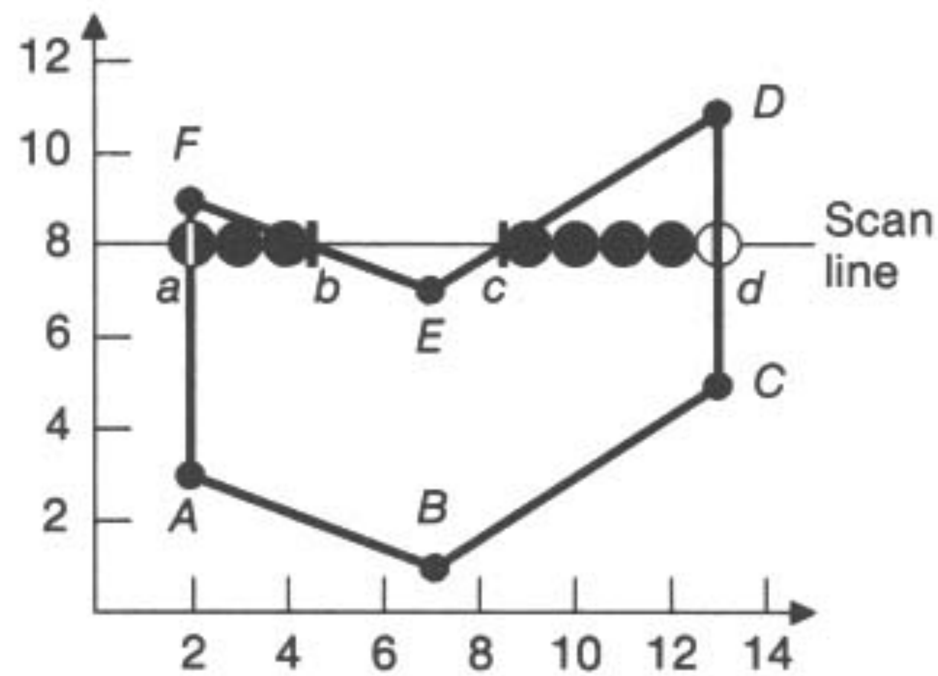
Black: span extrema
Grey: interior pixels



- What is the difference between these two solutions ?
- Which one is "better" ?

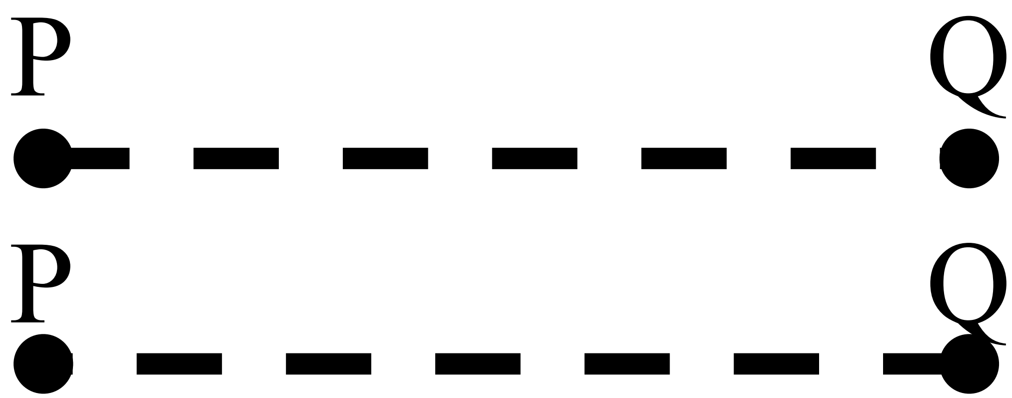
Filling Polygons

- We start with the pixel right of a, until the pixel left of b, and then again with the pixel right of c until the pixel left of d
- We count if we have an even or odd number of intersections



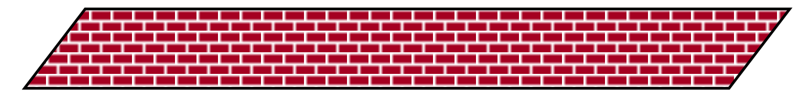
Patterned Lines

- Patterned line from *P* to *Q* is <u>not</u> same as patterned line from *Q* to *P*.

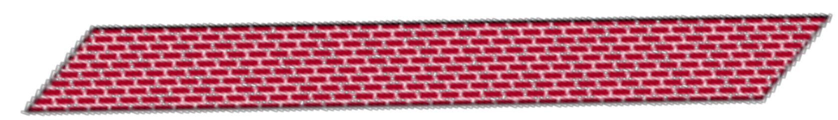


- Patterns can be ***cosmetic*** or ***geometric***
  - Cosmetic: Texture applied after transformations
  - Geometric: Pattern subject to transformations

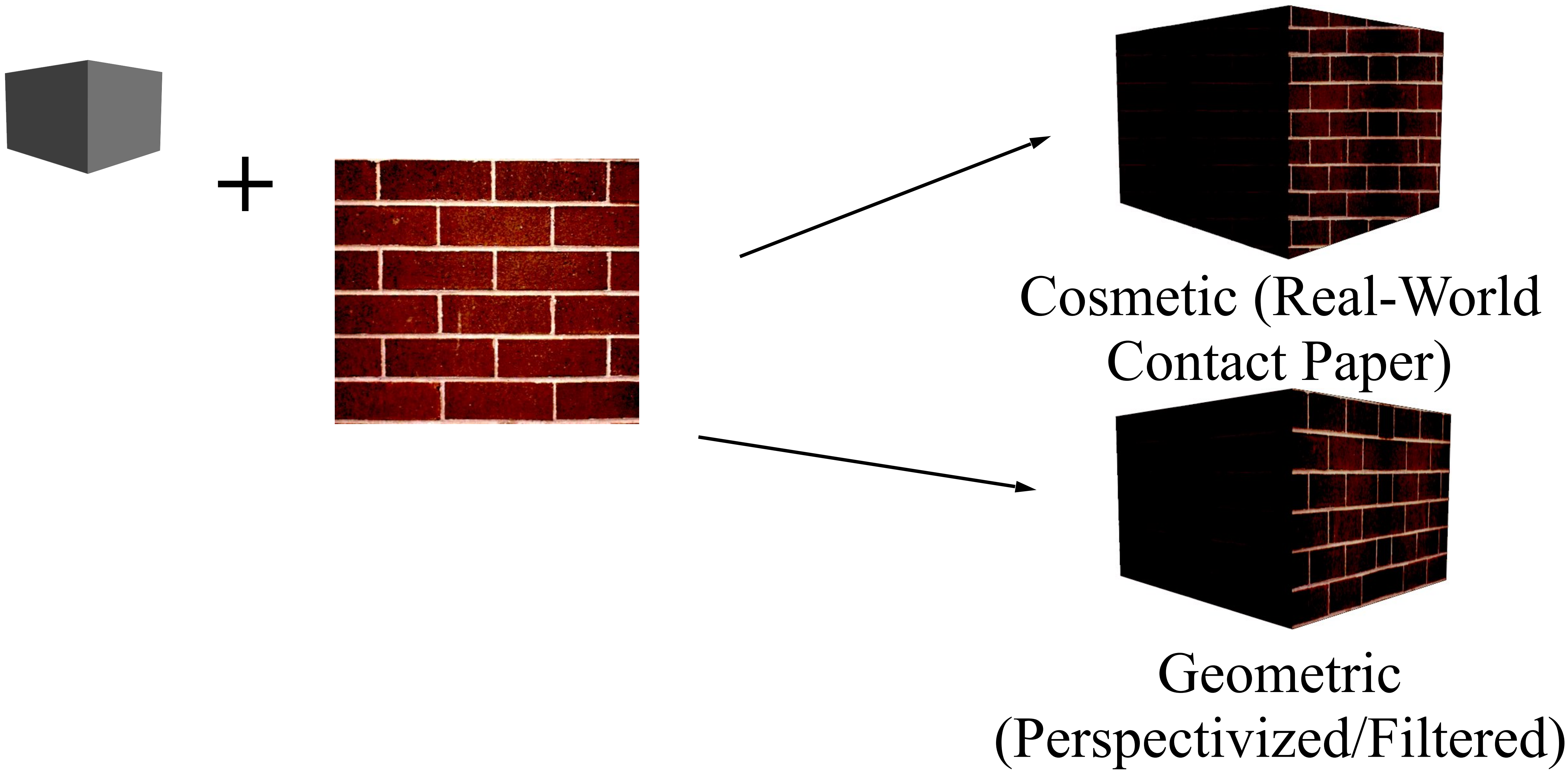
Cosmetic patterned line



Geometric patterned line



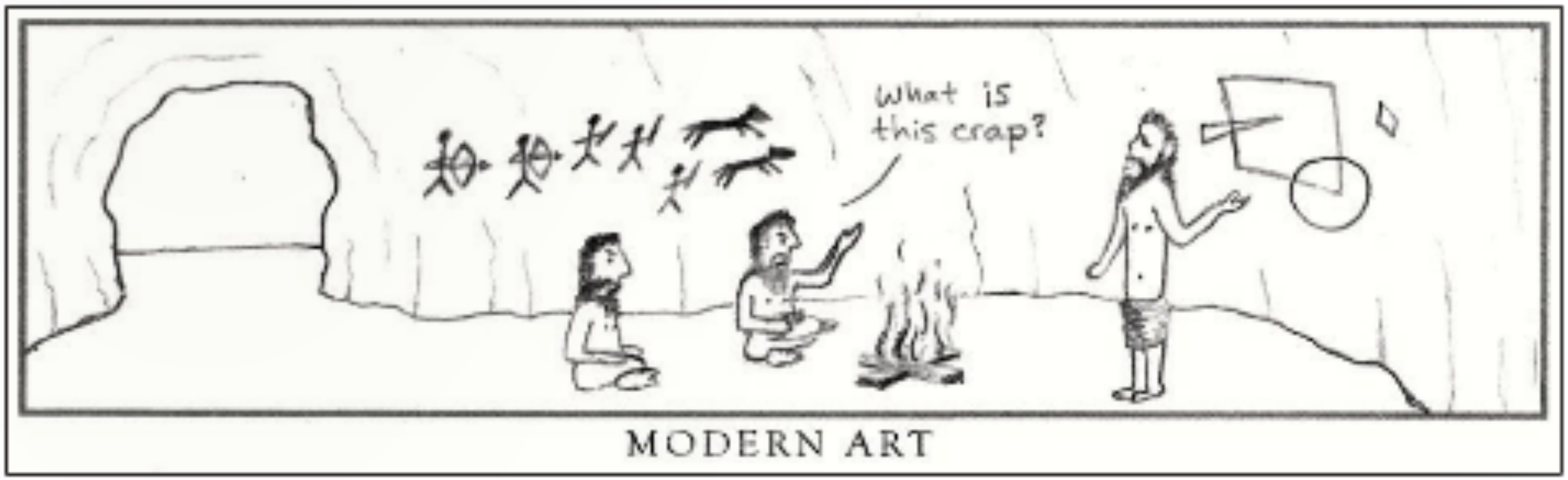
Geometric vs. Cosmetic



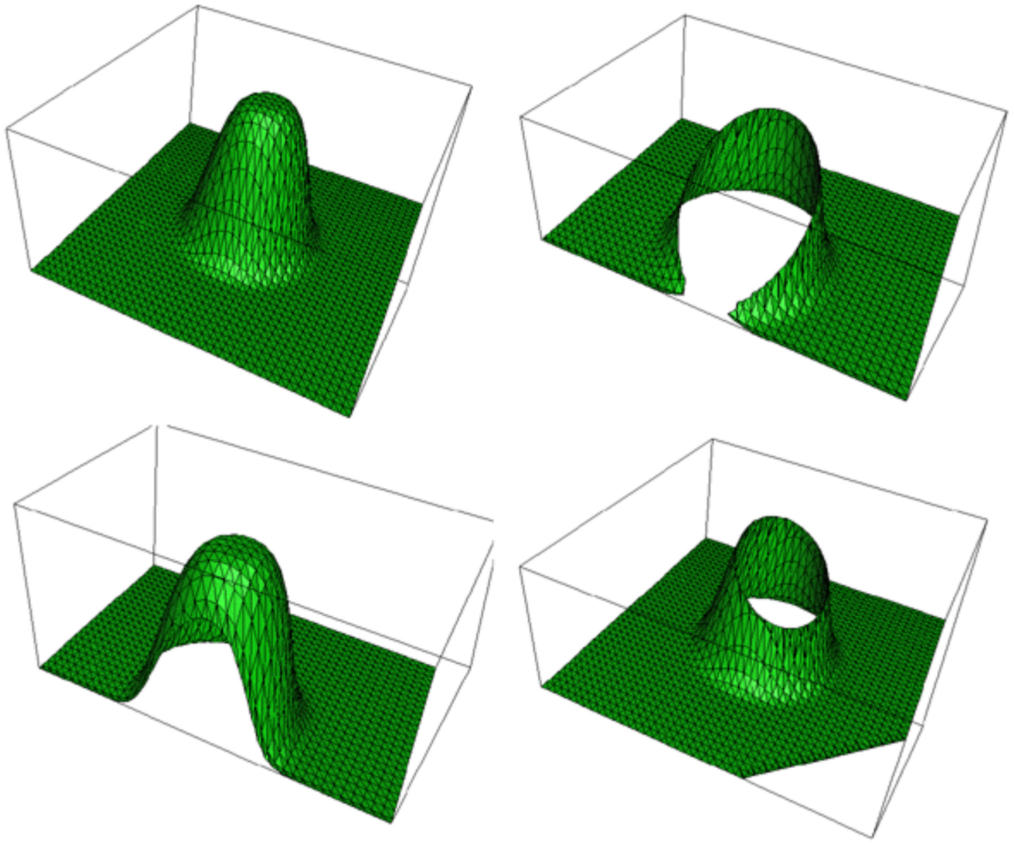
Scan Converting Arbitrary Solids

- Rapid Prototyping is becoming cheaper and more prevalent
  - 3D printers use various methods to print rasterized slices of whole solid objects
    - Extrude layer after layer of material from a nozzle (Makerbot)
    - Use UV light to cure material from a liquid bath (Formlabs)
    - Use heat to cure material from powdered materials (3D Systems)
    - And more! http://3dprintingindustry.com/3d-printing-basics-free-beginners-guide/processes/

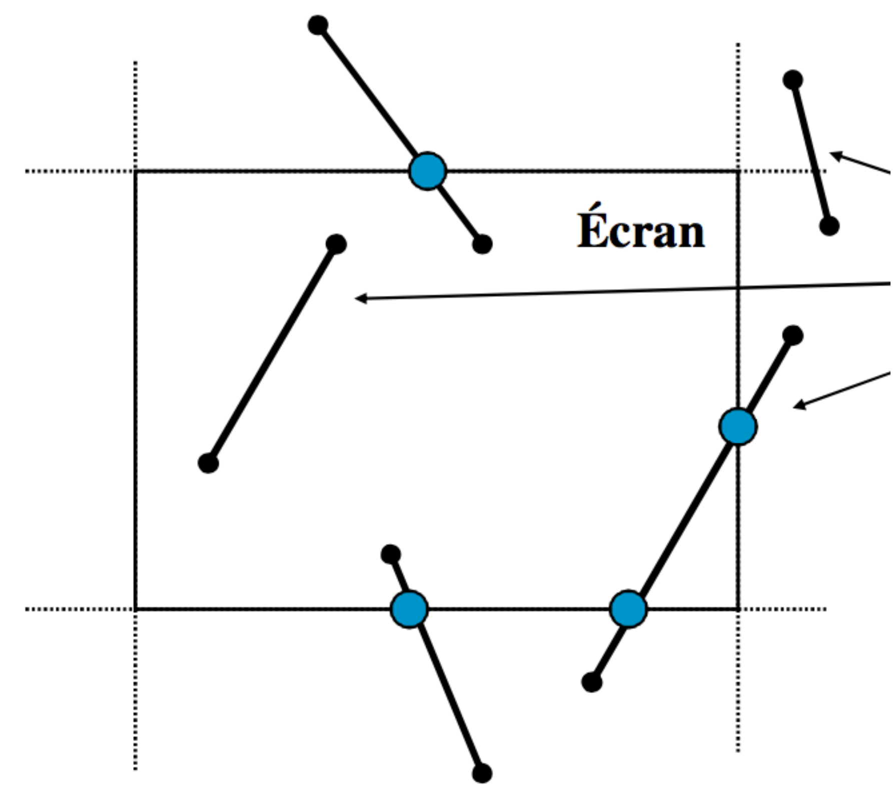
- Prosthetics - http://www.youtube.com/watch?v=6dI-dNE2yQ0
- ISS 3D Printer - https://www.youtube.com/watch?v=vgZymJC4a-g



Clipping



Clipping

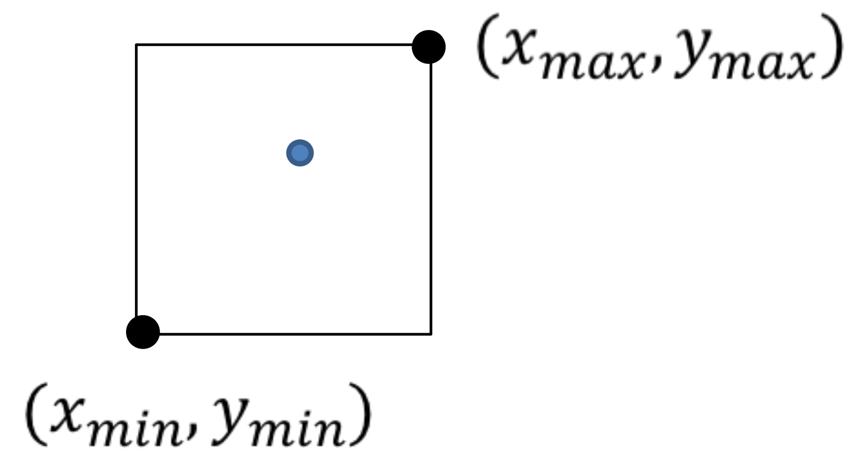


- Clipping a rectangle gives a rectangle
- Clipping a convex polygon gives a convex polygon
- Clipping a concave polygon can lead to several concave polygons
- Clipping a circle can create up to 4 arcs

Line Clipping
in 2D

- Clipping endpoints
  - If $x_{min} \leq x \leq x_{max}$ and $y_{min} \leq y \leq y_{max}$

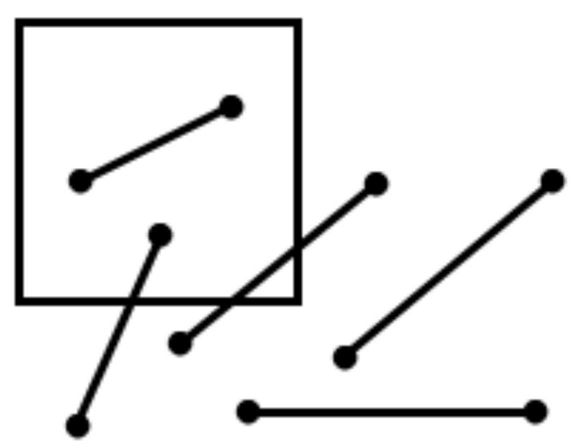
  the point is inside the clip rectangle



Line Clipping
in 2D

Point inside if
$x_{min} \leq x \leq x_{max}$ and $y_{min} \leq y \leq y_{max}$

- Clipping endpoints
- Endpoint analysis for segments
  - if both endpoints inside, do “trivial acceptance”
  - if one endpoint inside, one outside, must clip
  - if both endpoints out, we don’t know



Line Clipping

in 2D

Point inside if

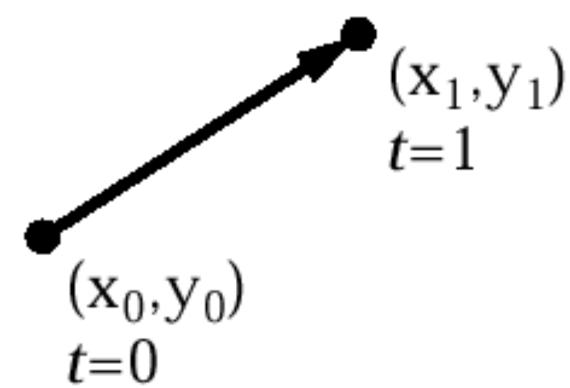
$x_{min} \leq x \leq x_{max}$ and $y_{min} \leq y \leq y_{max}$

- Clipping endpoints
- Endpoint analysis for segments
  - if both endpoints inside, do “trivial acceptance”
  - if one endpoint inside, one outside, must clip
  - if both endpoints out, we don’t know
- Brute force clip: solve simultaneous equations using line and four clip edges
  - Slope-intercept formula $y = mx + b$ handles infinite lines only
  - Need to use parametric line equation

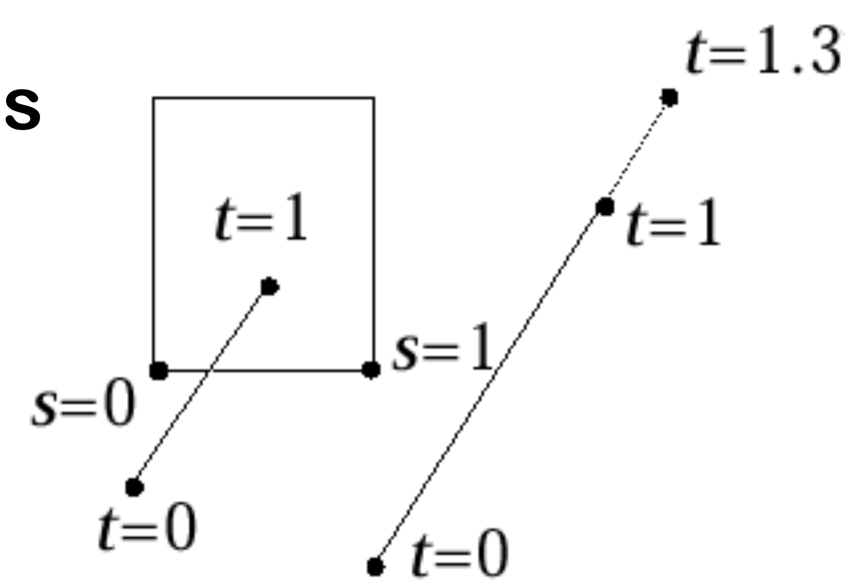
Line Clipping
in 2D

- Parametric form for line segment

$$X = x_0 + t(x_1 - x_0)$$
$$Y = y_0 + t(y_1 - y_0) \qquad 0 \le t \le 1$$
$$P(t) = P_0 + t(P_1 - P_0) = (1 - t)P_0 + t(P_1)$$



- Line is in clip rectangle if parametric variables t_{line} and s_{edge} both in [0,1] at intersection point between line and edge of clip rectangle
  - Slow, must intersect lines with all edges



Line Clipping in 2D

Cohen-Sutherland Algorithm

- Divide plane into 9 regions
- Compute 4-bits outcode for each vertex
- Each sign bit comes from the comparison between the vertex and the 4 edges
  - First bit: above top edge

    $$y_{max} - y$$

  - Second bit: below bottom edge

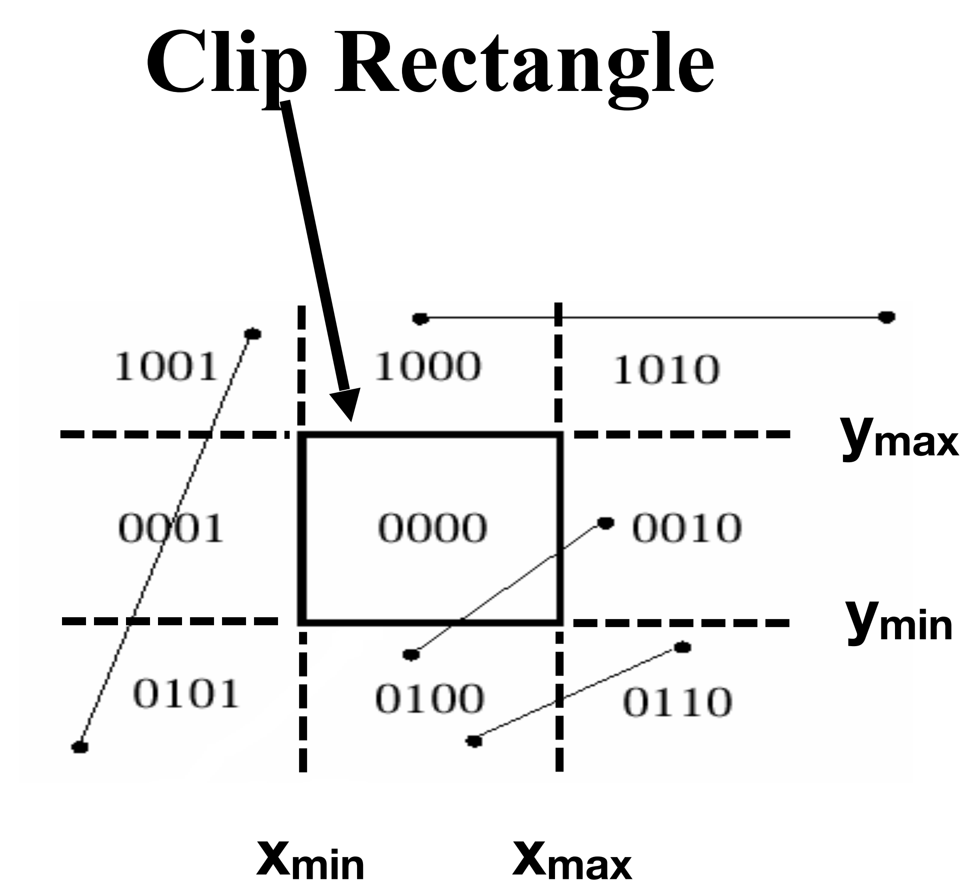
    $$y - y_{min}$$

  - Third bit: to the right of right edge

    $$x_{max} - x$$

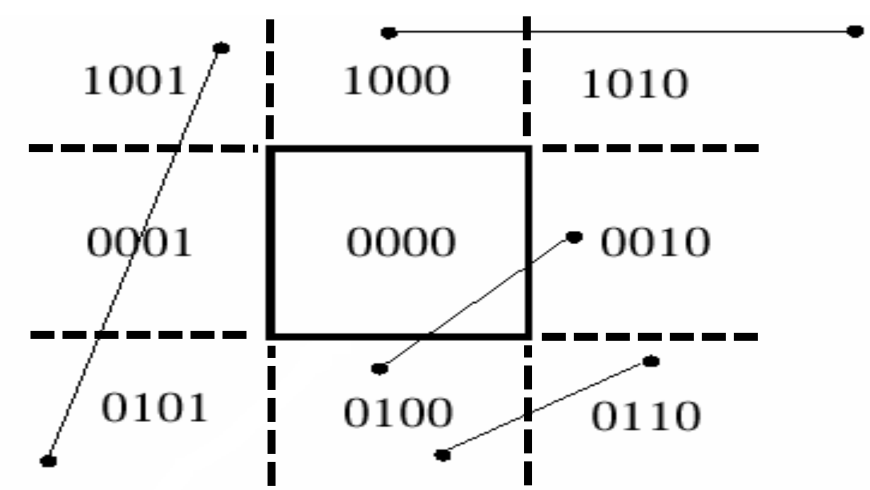
  - Fourth bit: to the left of left edge

    $$x - x_{min}$$



Line Clipping in 2D

Cohen-Sutherland Algorithm



- Vertex lies inside only if all bits are 0
  - Otherwise exceeds edge
- With codes for both vertices

  (denoted as OC_0 and OC_1)
  - Lines with $OC_0 = 0$ (i.e. 0000) and $OC_1 = 0$ can be trivially accepted

    (both end points inside)
  - Lines lying entirely in a half plane outside and edge can be trivially rejected $OC_0 \wedge OC_1 \neq 0$

    (they share an “outside” bit)

Line Clipping in 3D
Cohen-Sutherland Algorithm

- Very similar to 2D
- Divide volume into 27 regions
- 6-bit outcode records results of 6 bounds tests
  - **First bit**: behind back plane: Z - Zmin
  - **Second bit**: in front of front plane: Zmax -Z
  - **Third bit**: above top plane: Ymax - Y
  - **Fourth bit**: below bottom plane: Y - Ymin
  - **Fifth bit**: to the right of right plane: Xmax -X
  - **Sixth bit**: to the left of left plane: X - Xmin
- Again, lines with $OC_0 = 0$ and $OC_1 = 0$ can be *trivially accepted*
- Lines lying entirely in a volume outside of a plane can be *trivially rejected*: OC_0 **AND** $OC_1 \neq 0$ (i.e., they share an "outside" bit)

Back plane

000000 (in front)

100000 (behind)

Front plane

010000 (in front)

000000 (behind)

Top plane

001000 (above)

000000 (below)

Bottom plane

000000 (above)

000100 (below)

Right plane

000000 (to left of)

000010 (to right of)

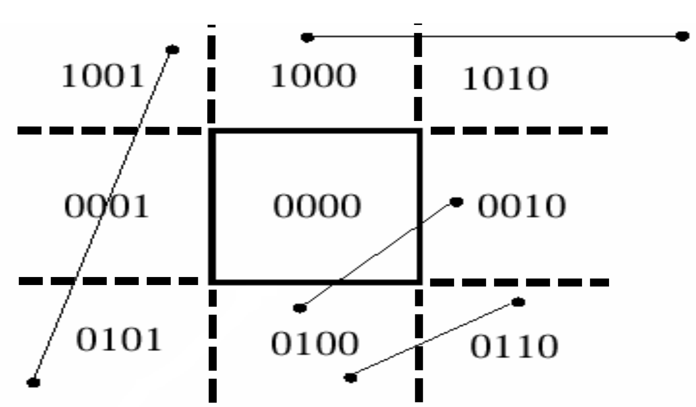
Left plane

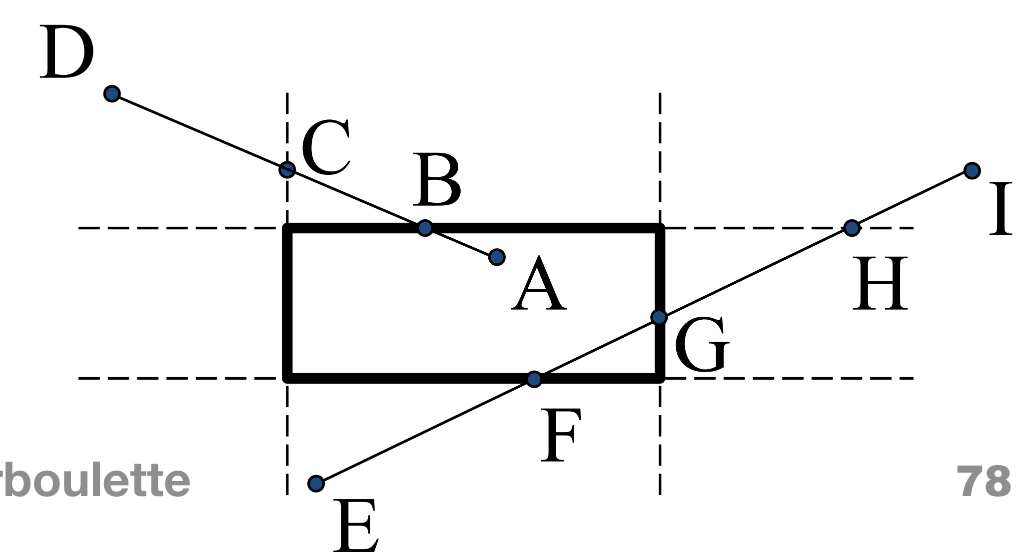
000001 (to left of)

000000 (to right of)

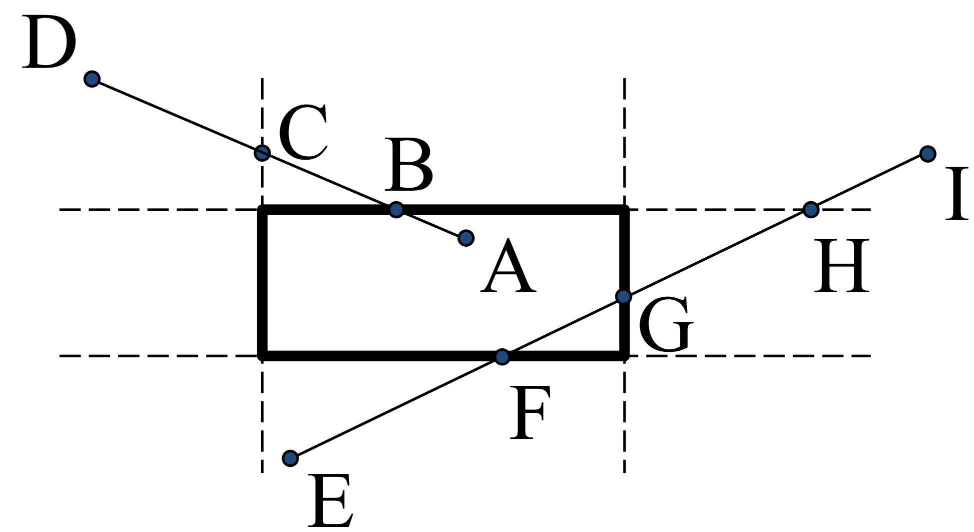
Line Clipping in 2D
Cohen-Sutherland Algorithm

- If we can neither trivially accept/reject, divide and conquer
- Subdivide line into two segments
  - Use outcodes to choose the edges that are crossed
    - For a given clip edge, if a line's two outcodes differ in the corresponding bit, the line has one vertex on each side of the edge, thus crosses
    - Compare first bit, then second bit…
  - Choose an outside point (D ; I or E)
  - Based on its outcode, gets the edge to clip (top – bottom – right – left)





Line Clipping in 2D
Cohen-Sutherland Algorithm



- Compute the intersection point
- Point-slope form of line equation

$$y - y_0 = m(x - x_0)$$

$$y = y_0 + m(x - x_0) \qquad x = x_0 + \frac{1}{m}(y - y_0)$$

- The clip edge fixes either x (RIGHT or LEFT) or y (TOP or BOTTOM)
- Can substitute into the line equation

```
if TOP then
  x = x0 + (x1 - x0) * (ymax - y0) / (y1 - y0);
  y = ymax;
```

- Iterate for the newly shortened line

- Might require multiple iterations and needless clipping (H)

Line Clipping in 2D

Cohen-Sutherland Algorithm

```
ComputeOutCode(x0, y0, outcode0); ComputeOutCode(x1, y1, outcode1);

Repeat
      Check for trivial reject or trivial accept;
      Pick a point (x0,y0) or (x1,y1) that is outside the clip rectangle;

      if TOP then
        x = x0 + (x1 - x0) * (ymax - y0) / (y1 - y0);
        y = ymax;
        else if BOTTOM then
          x = x0 + (x1 - x0) * (ymin - y0) / (y1 - y0);
          y = ymin;
          else if RIGHT then
            y = y0 + (y1 - y0) * (xmax - x0) / (x1 - x0);
            x = xmax;
            else if LEFT then
             y = y0 + (y1 - y0) * (xmin - x0) / (x1 - x0);
             x = xmin;

      if (x0,y0) was chosen
        x0 = x; y0 = y; ComputeOutCode(x0, y0, outcode0);
      else
        x1 = x; y1 = y; ComputeOutCode(x1, y1, outcode1);
Until done
```

Cohen-Sutherland Algorithm (3/3)

- Similar algorithm for using 3D outcodes to clip against canonical parallel view volume:

```
xmin = ymin = -1;  xmax = ymax = 1;
zmin = -1;  zmax = 0;

ComputeOutCode(x0, y0, z0, outcode0);
ComputeOutCode(x1, y1, z1, outcode1);
repeat
    check for trivial reject or trivial accept
    pick the point that is outside the clip rectangle
    if TOP then
        x = x0 + (x1 – x0) * (ymax – y0) / (y1 – y0);
        z = z0 + (z1 – z0) * (ymax – y0) / (y1 – y0);
        y = ymax;
    else if BOTTOM then
        x = x0 + (x1 – x0) * (ymin – y0) / (y1 – y0);
        z = z0 + (z1 – z0) * (ymin – y0) / (y1 – y0);
        y = ymin;
    else if RIGHT then
        y = y0 + (y1 – y0) * (xmax – x0) / (x1 – x0);
        z = z0 + (z1 – z0) * (xmax – x0) / (x1 – x0);
        x = xmax;
    else if LEFT then
        y = y0 + (y1 – y0) * (xmin – x0) / (x1 – x0);
        z = z0 + (z1 – z0) * (xmin – x0) / (x1 – x0);
        x = xmin;
    else if NEAR then
        x = x0 + (x1 – x0) * (zmax – z0) / (z1 – z0);
        y = y0 + (y1 – y0) * (zmax – z0) / (z1 – z0);
        z = zmax;
    else if FAR then
        x = x0 + (x1 – x0) * (zmin – z0) / (z1 – z0);
        y = y0 + (y1 – y0) * (zmin – z0) / (z1 – z0);
        z = zmin;

    if (x0, y0, z0 is the outer point) then
        x0 = x; y0 = y; z0 = z;
        ComputeOutCode(x0, y0, z0, outcode0)
    else
        x1 = x; y1 = y; z1 = z;
        ComputeOutCode(x1, y1, z1, outcode1)
until done
```

Line Clipping in 2D

Parametric Line-Clipping Algorithm

- Cohen-Sutherland Algorithm : for lines that cannot be trivially accepted of rejected, the (x,y) intersection has to be calculated
- Using an algorithm based on parametric line equations means calculating 4 parameters t (one for each clip edge) : 1D instead of 3D
- Liang-Barsky improvement: examine each t value to eventually reject line

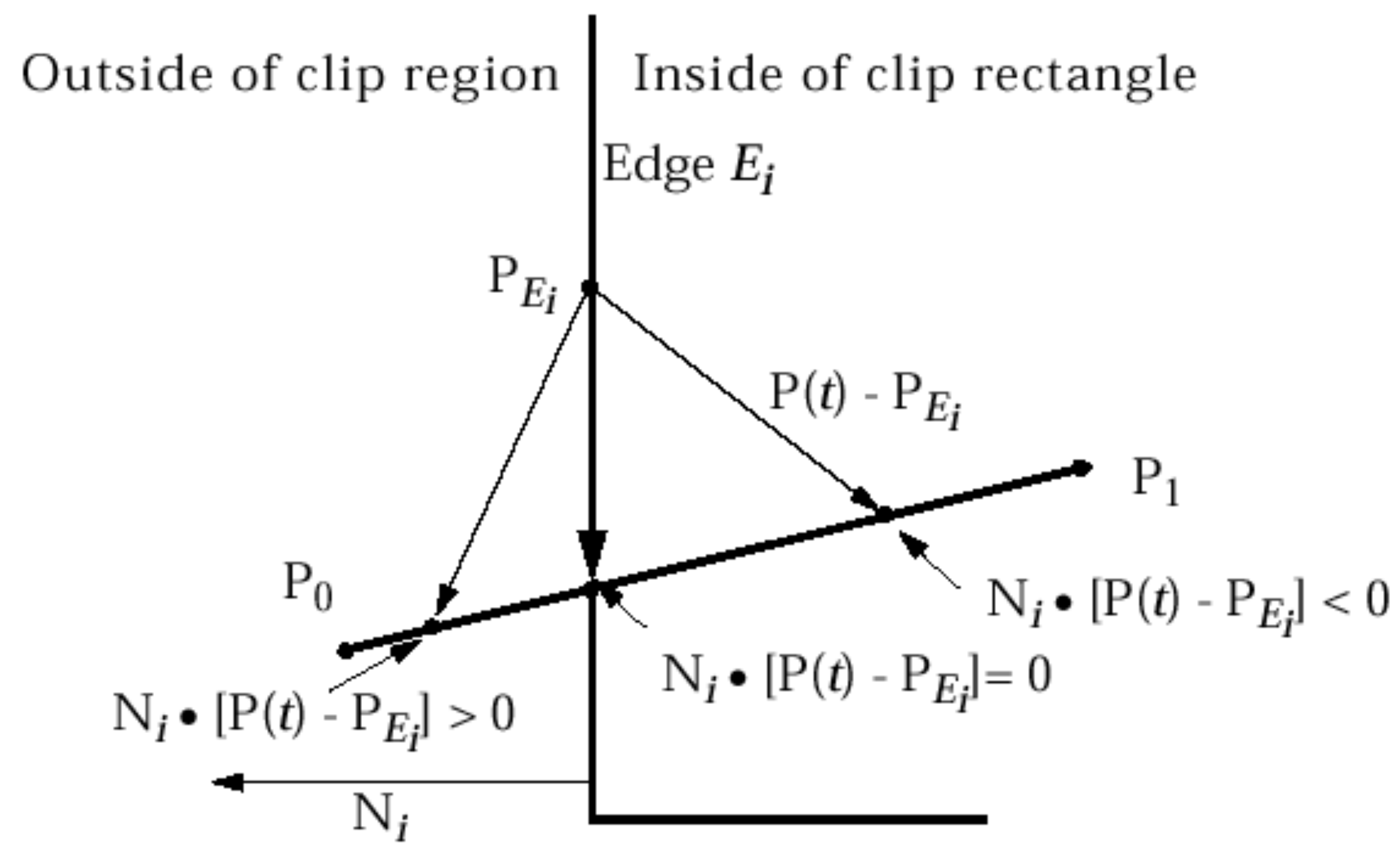
Parametric Line Clipping
Cyrus-Beck/Liang-Barsky Algorithm

- Uses parametric form of line equation

$$P(t) = P_0 + t \cdot (P_1 - P_0)$$

1. Computes t for each edge
2. Determines if the corresponding intersection is on the clipping rectangle

- Principle
  - $N_i \cdot [P(t) - P_{E_i}] > 0$
    point in outside half plane
  - $N_i \cdot [P(t) - P_{E_i}] = 0$
    intersection point
  - $N_i \cdot [P(t) - P_{E_i}] < 0$
    point in inside half plane



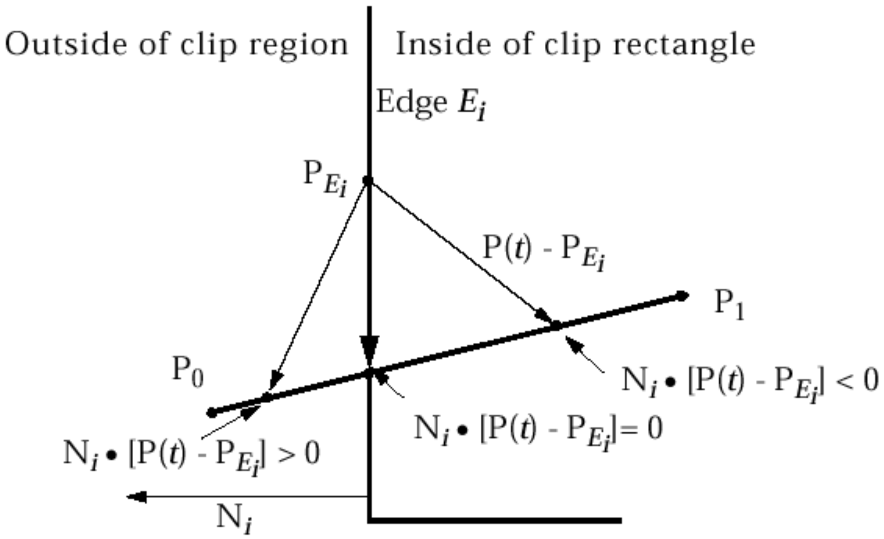
Parametric Line Clipping

Cyrus-Beck/Liang-Barsky Algorithm

1. Compute t at the intersection of $\overrightarrow{P_0P_1}$ with the edge E_i

- Pick any point P_{E_i} on edge E_i

$$N_i \cdot [P(t) - P_{E_i}] = 0$$



- Substitute for $P(t) = P_0 + t \cdot (P_1 - P_0)$

$$N_i \cdot [P_0 + t \cdot (P_1 - P_0) - P_{E_i}] = 0$$

- Group terms and distribute dot product

$$N_i \cdot [P_0 - P_{E_i}] + N_i \cdot t \cdot (P_1 - P_0) = 0$$

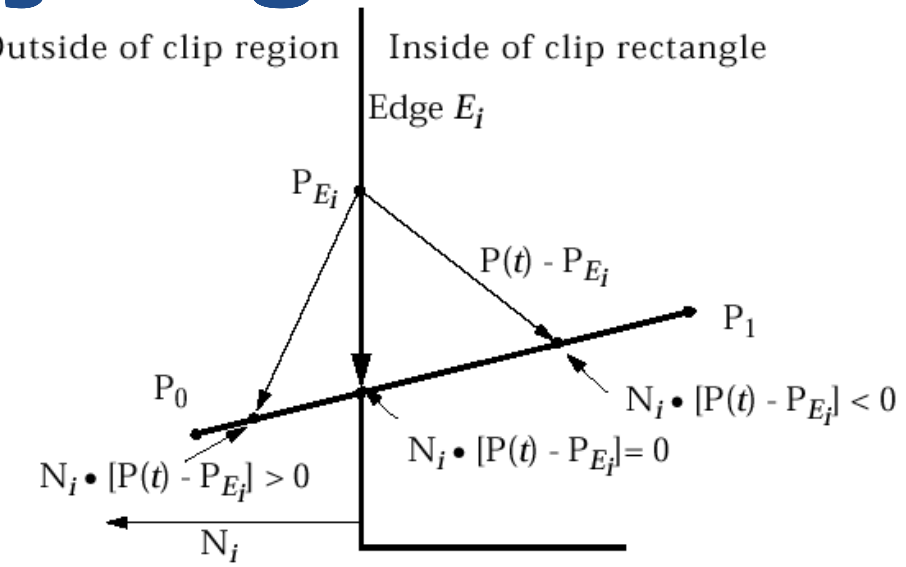
- Let D be the vector from P_0 to P_1 = ($P_1 - P_0$), and solve for t

$$t = \frac{N_i \cdot [P_0 - P_{E_i}]}{-N_i \cdot D}$$

Parametric Line Clipping

Cyrus-Beck/Liang-Barsky Algorithm

$$t = \frac{N_i \cdot [P_0 - P_{E_i}]}{-N_i \cdot D}$$

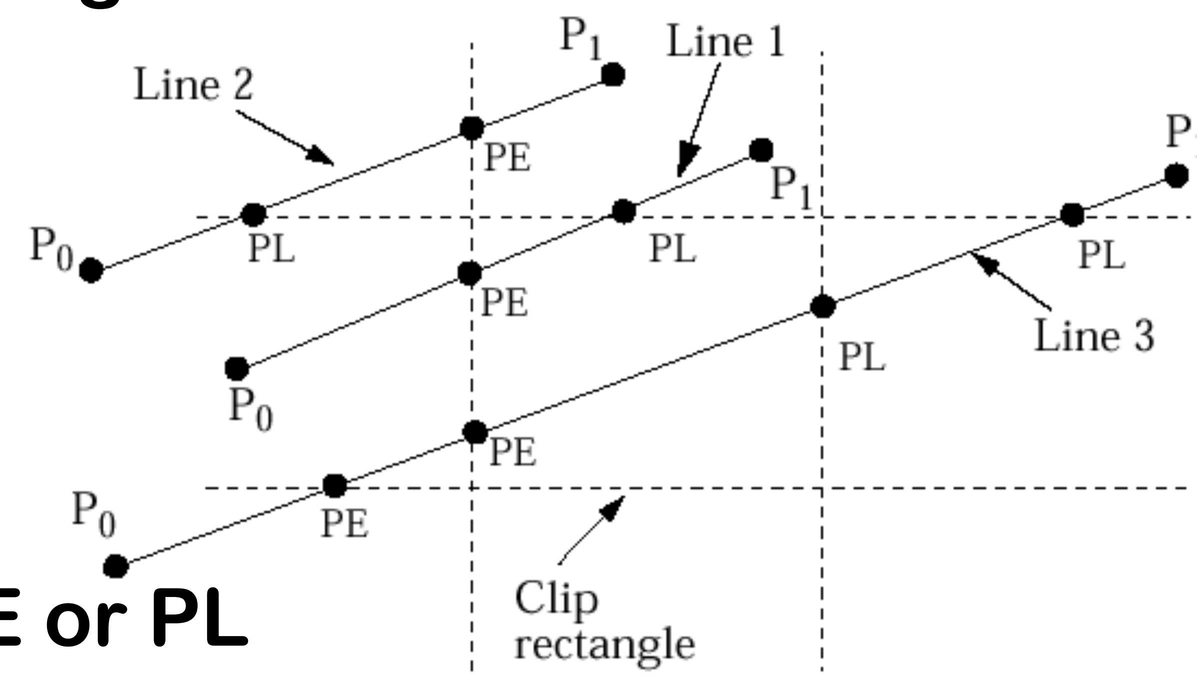


- This gives a valid value of t only if the denominator of the expression is non zero:
  - $N_i \neq 0$ (i.e. the normal should not be 0; this could occur only as a mistake)
  - $D \neq 0$ (i.e. $P_1 \neq P_0$)
  - $N_i \bullet D \neq 0$ (edge E_i and line D are not parallel; if they are, no intersection).
- The algorithm checks these conditions

Parametric Line Clipping

Cyrus-Beck/Liang-Barsky Algorithm

2. Determines if the corresponding intersection is on the clipping rectangle



- Eliminate t's outside [0,1]
- Find interior intersections
  - Line 2: doesn't work
- Need to tag intersections as PE or PL
  - If $N_i \cdot D < 0$, angle greater than 90, **P**otentially **E**ntering
  - If $N_i \cdot D > 0$, angle less than 90, **P**otentially **L**eaving
- Pick t_E for P_{PE} with max t and t_L for P_{PL} with min t
- If $t_L < t_E$, line is rejected

Cyrus-Beck/Liang-Barsky Line Clipping Algorithm

Pre-calculate N_i and select P_{E_i} for each edge;

for each line segment to be clipped
  if $P_1 = P_0$ **then** line is degenerate so clip as a point;
  else
    begin
      $t_E = 0$; $t_L = 1$;
      for each candidate intersection with a clip edge
        if $Ni \bullet D \neq 0$ **then** {Ignore edges parallel to line}
          begin
            calculate t; {of line and clip edge intersection}
            use sign of $N_i \bullet D$ to categorize as PE or PL;
            if PE **then** t_E = **max**(t_E,t);
            if PL **then** t_L = **min**(t_L,t);
          end
      if $t_E > t_L$ **then** return **nil**
      else return $P(t_E)$ and $P(t_L)$ as true clip intersections
  end

Parametric Line Clipping for Upright Clip Rectangle (1/2)

$$t = \frac{N_i \cdot [P_0 - P_{E_i}]}{-N_i \cdot D}$$

- $D = P_1 - P_0 = (x_1 - x_0, y_1 - y_0)$
- Leave P_{E_i} as an arbitrary point on clip edge: it's a free variable and drops out

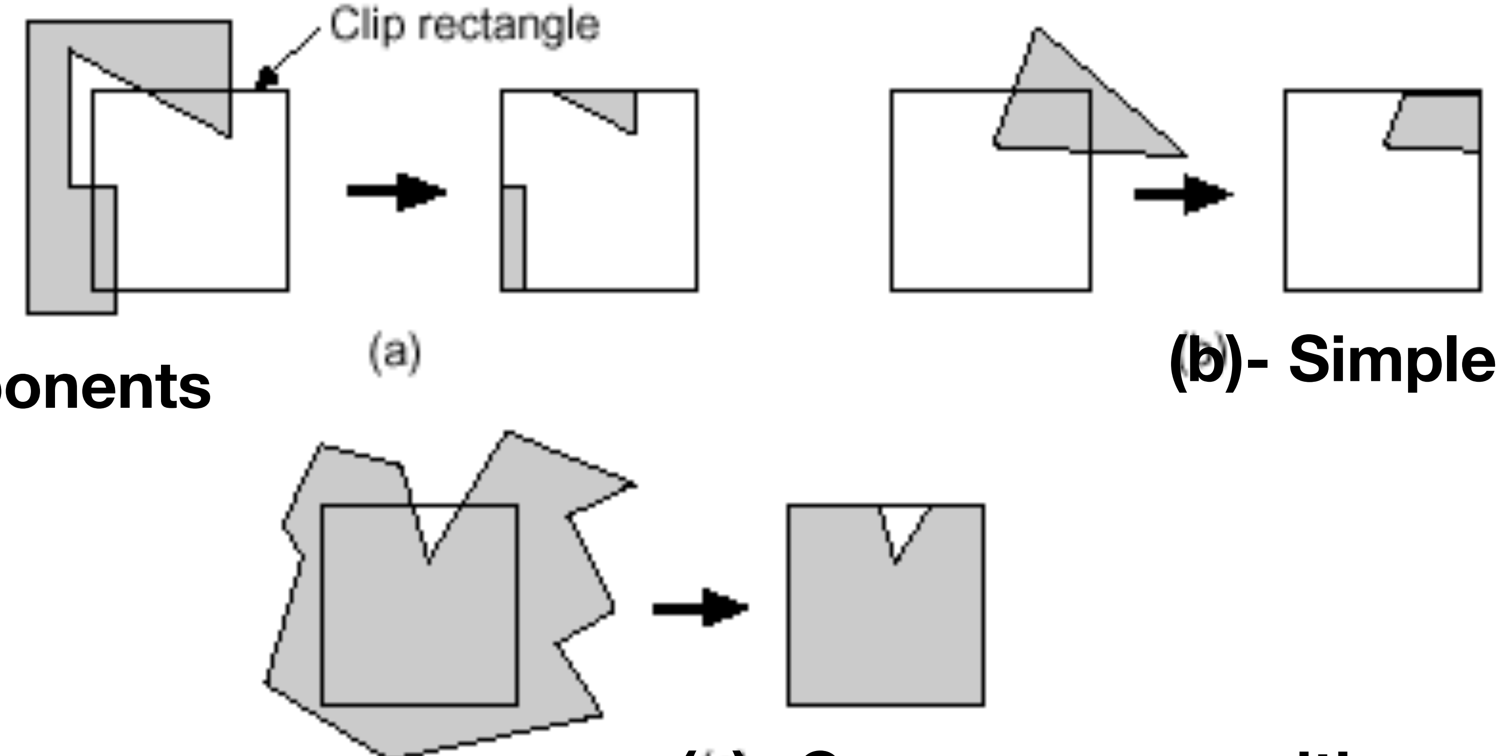
Calculations for Parametric Line Clipping Algorithm

Clip Edge$_i$	Normal N_i	P_{E_i}	P_0-P_{E_i}	$t = \frac{N_i \bullet (P_0 - P_{E_i})}{-N_i \bullet D}$
left: $x = x_{min}$	(-1,0)	(x_{min}, y)	$(x_0 - x_{min}, y_0 - y)$	$\frac{-(x_0 - x_{min})}{(x_1 - x_0)}$
right: $x = x_{max}$	(1,0)	(x_{max}, y)	$(x_0 - x_{max}, y_0 - y)$	$\frac{-(x_0 - x_{max})}{(x_1 - x_0)}$
bottom: $y = y_{min}$	(0,-1)	(x, y_{min})	$(x_0 - x, y_0 - y_{min})$	$\frac{-(y_0 - y_{min})}{(y_1 - y_0)}$
top: $y = y_{max}$	(0,1)	(x, y_{max})	$(x_0 - x, y_0 - y_{max})$	$\frac{-(y_0 - y_{max})}{(y_1 - y_0)}$

Parametric Line Clipping for Upright Clip Rectangle (2/2)

- Examine t:
  - Numerator is just the directed distance to an edge; sign corresponds to OC
  - Denominator is just the horizontal or vertical projection of the line, ***dx*** or ***dy***; sign determines PE or PL for a given edge
  - Ratio is constant of proportionality: "how far over" from P_0 to P_1 intersection is relative to ***dx*** or ***dy***

Polygon Clipping



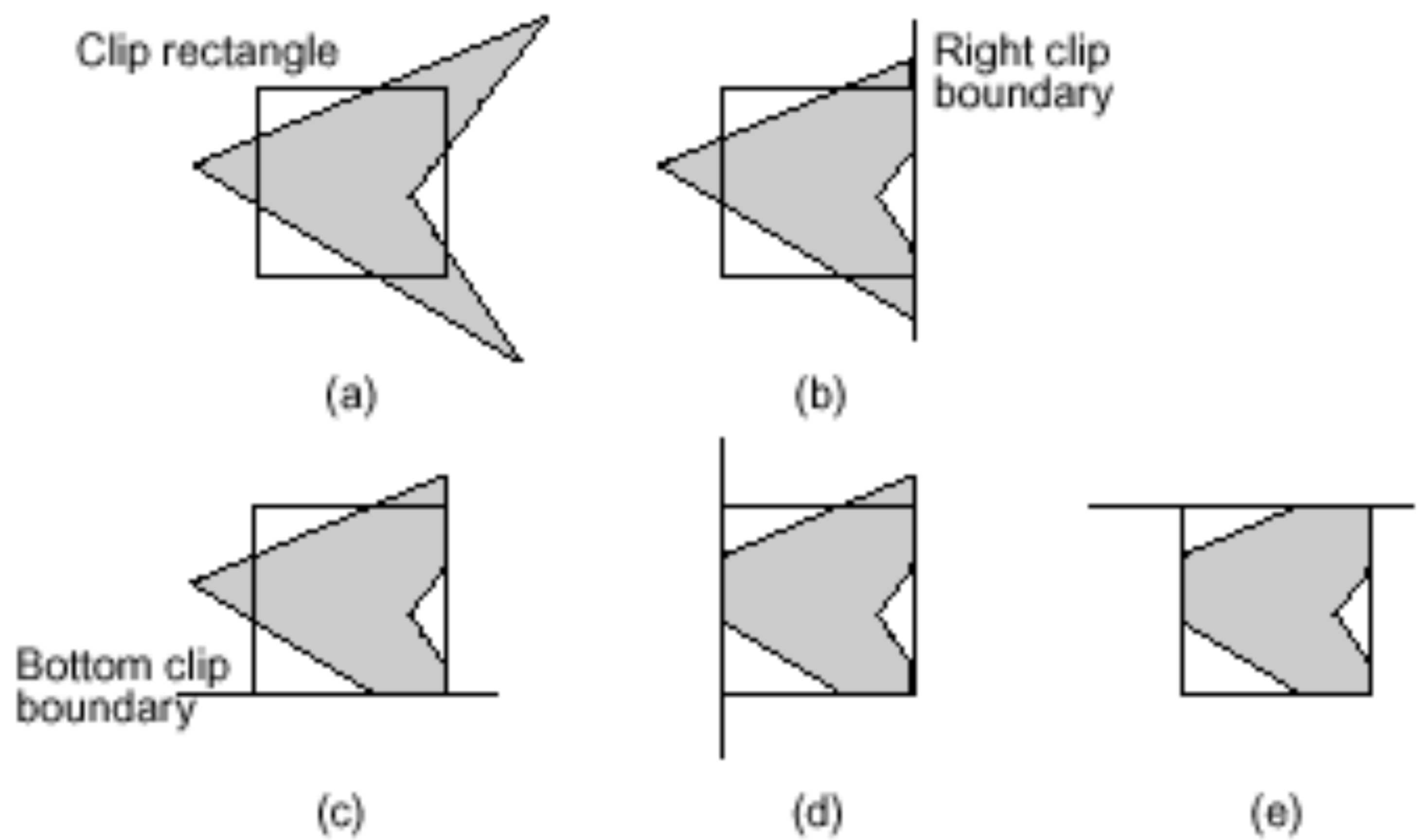
(a)- Multiple components

(b)- Simple convex case

(c)- Concave case with many exterior edges

- (b)- Clipping a convex polygon gives a convex polygon
- (a) & (c)- Clipping a concave polygon can lead to several concave polygons
- Works successively with each side of the clip rectangle

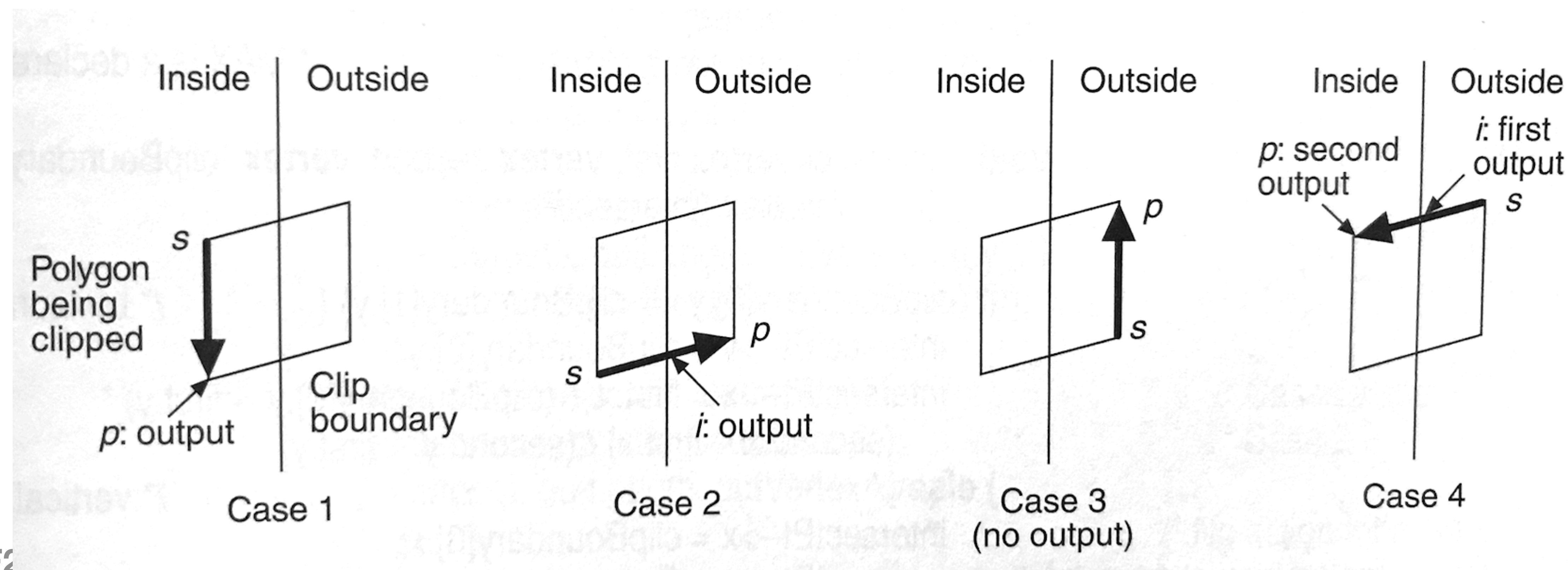
Polygon Clipping
Sutherland-Hodgman Algorithm



- Divide and Conquer strategy
  - Clip the polygon against a single infinite clip edge
  - Right, Bottom, Left, Top

Polygon Clipping
Sutherland-Hodgman Algorithm

- Algorithm is general
  - Can be used to clip against any convex polygon
  - Can be extended to 3D to clip against convex polyhedral volumes defined by planes
- Go through each polygon edge and create a list of output vertices

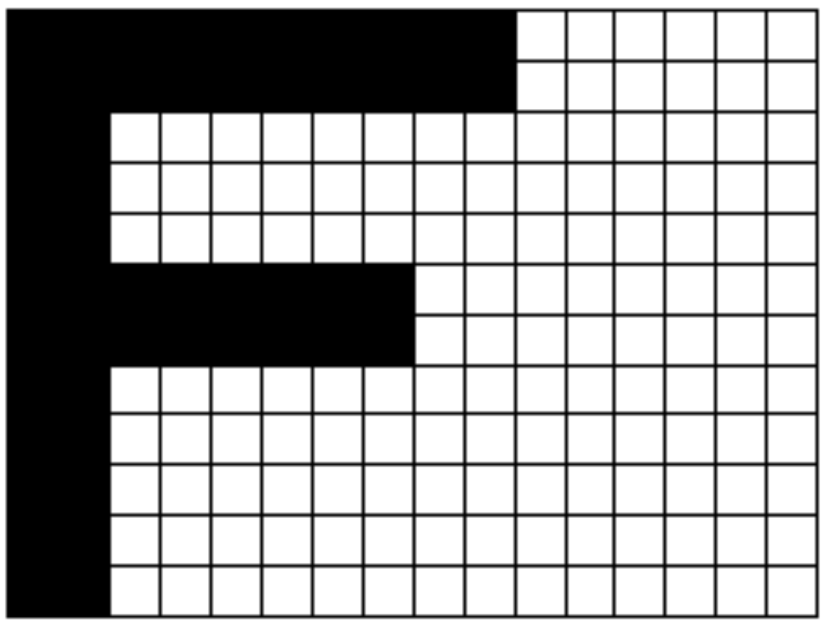


Generating Characters

Generating Characters

- Two techniques
  - Using bitmaps
  - Using polygons and curves

Bitmap Characters



```
0xff, 0xc0
0xff, 0xc0
0xc0, 0x00
0xc0, 0x00
0xc0, 0x00
0xff, 0x00
0xff, 0x00
0xc0, 0x00
0xc0, 0x00
0xc0, 0x00
0xc0, 0x00
0xc0, 0x00
```

1111 1111 1100 0000

1100 0000 0000 0000

- Each character needs to be specified and stored
- One bitmap per size (8)
- One bitmap per style (4): normal, italic, bold, italic bold
- 32 bitmaps per letter

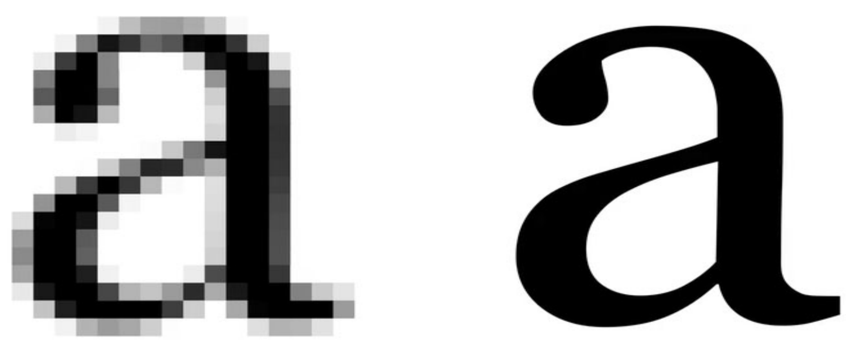
Bitmap Characters

- Created by
  - scanning printed characters
  - drawing with paint software
  - completely by hand, specifying which pixels are black

Abstract Characters

- Polygons and spline curves
- Material indépendant
- Can be scaled
- Italic can be obtained by tilting the polygon/curve
- Use of standard scan conversion for drawing, standard clipping algorithm
- Slower

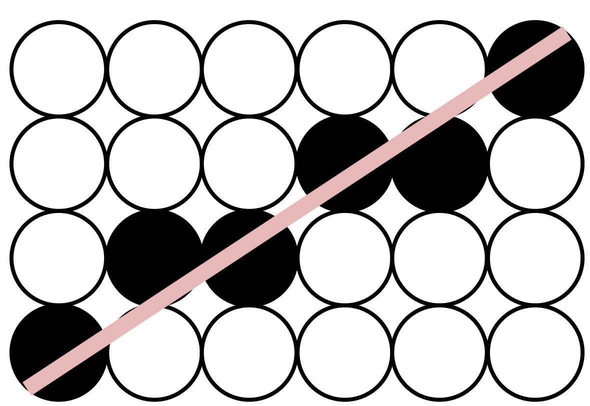
Antialiasing



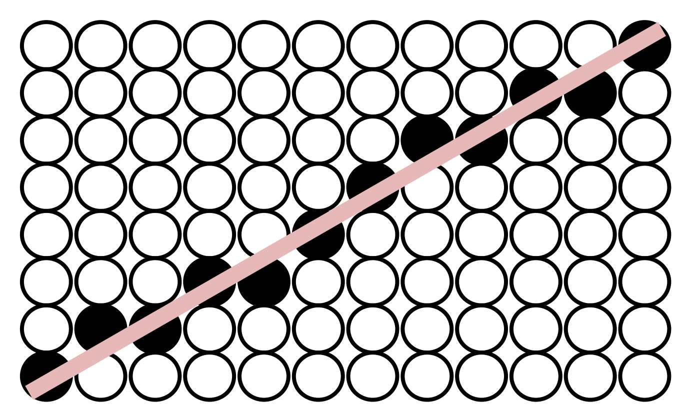
Point Sampling

Resolution

- Aliasing totally depends on the resolution
- Increasing resolution improves the rendering but is not a solution
- Costly (memory, bandwidth, scan conversion time) : doubling resolution costs 4 times more



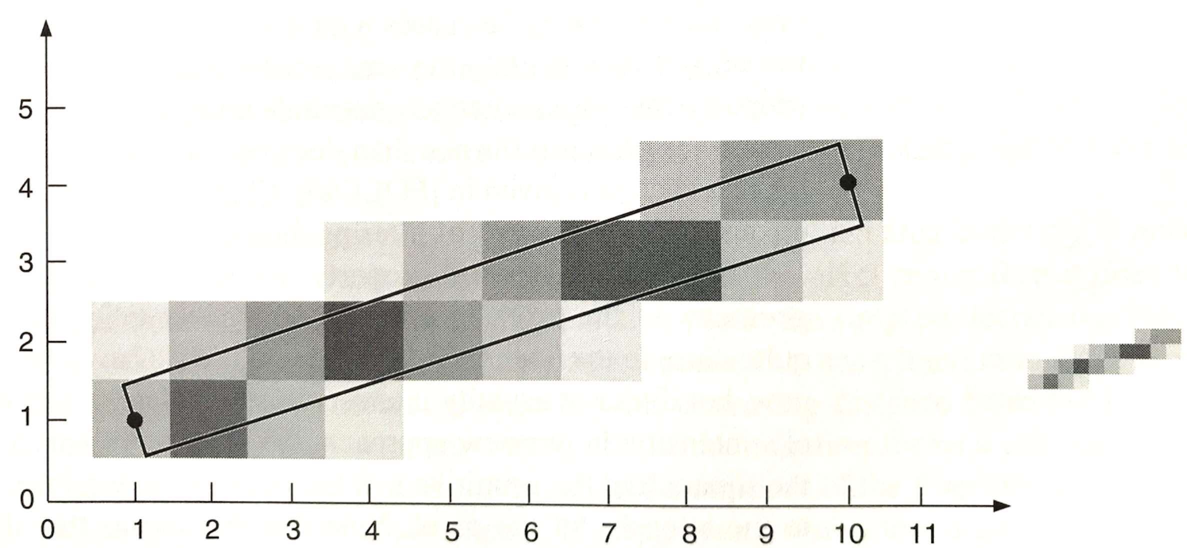
Line approximation using point sampling



Approximating same line at 2x the resolution

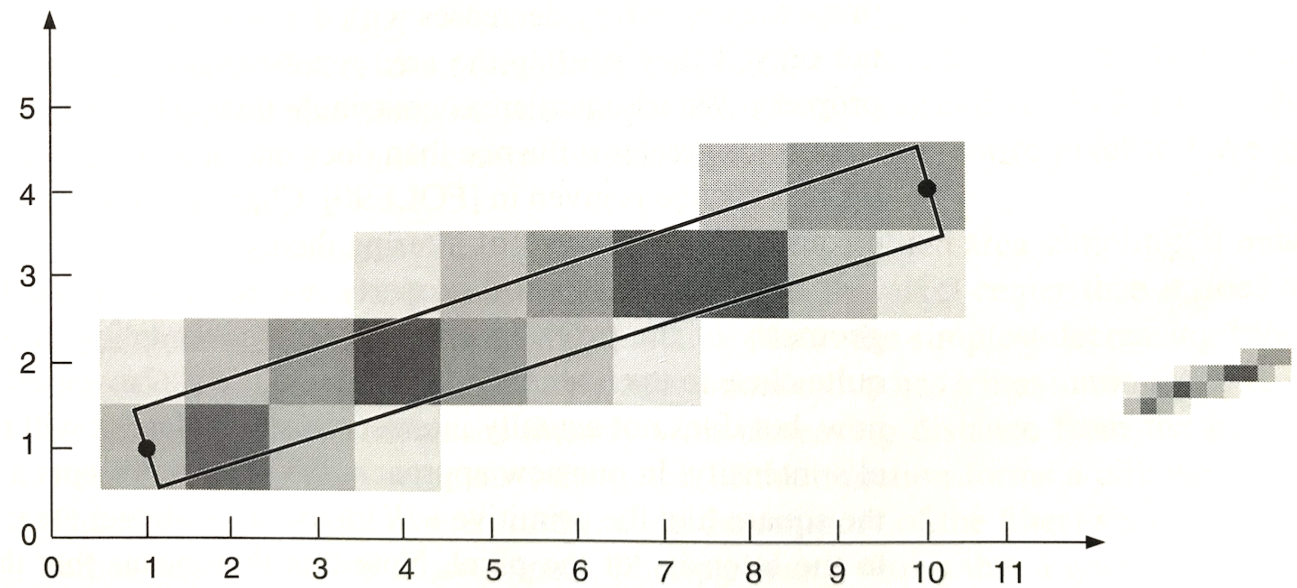
Area Sampling

- Line has a width (minimum 1 pixel for horizontal and vertical lines)
- Idea:
  - Represent the line as a unit width rectangle
  - Use multiple pixels overlapping the rectangles



Area Sampling

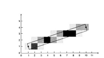
Unweighted



(2,1) : 70% black
(2,2) : 25% black
(2,3) : 0% black (100% white)

- Each pixel intensity is proportional to the area covered by the unit rectangle
  - Only pixels covered by primitive contribute
  - Distance of pixel to center of line doesn't matter
- Works only if screen has multiple bits per pixel

Area Sampling
Unweighted



- How do you compute the area covered ?
- Divide the pixel in smaller sub-pixels and count the number of sub-pixels covered

Area Sampling
Adding Filtering

- Problem 1: a small surface in a pixel corner gives the same intensity as a small surface near a pixel center
  - Solution: add a weighting function to make a surface near a pixel center contribute more to intensity
- Problem 2: a pixel near the line but not covered is totally white
  - Solution: make pixel bigger than reality for computation

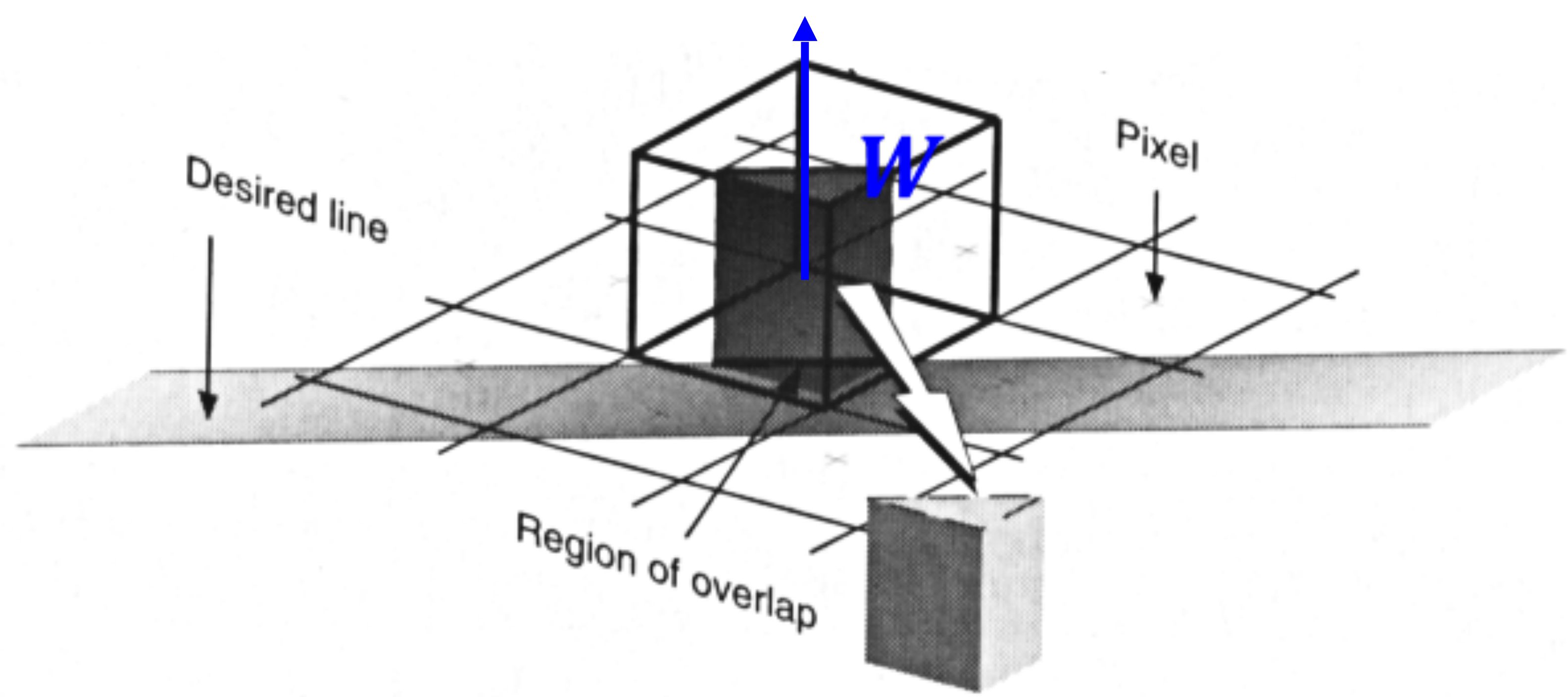
Area Sampling
Adding Filtering

- Let $W(x, y)$ be a weighing function
- $W(x, y)$ is applied to dA (area covered)
- Intensity of a small area = $W(x, y) \cdot dA$
- Pixel intensity = $\int_A W(x, y) \cdot dA$
- Weight $W(x, y)$ is function of the distance of dA to the center of the pixel:

  If distance increases, $W(x, y)$ decreases

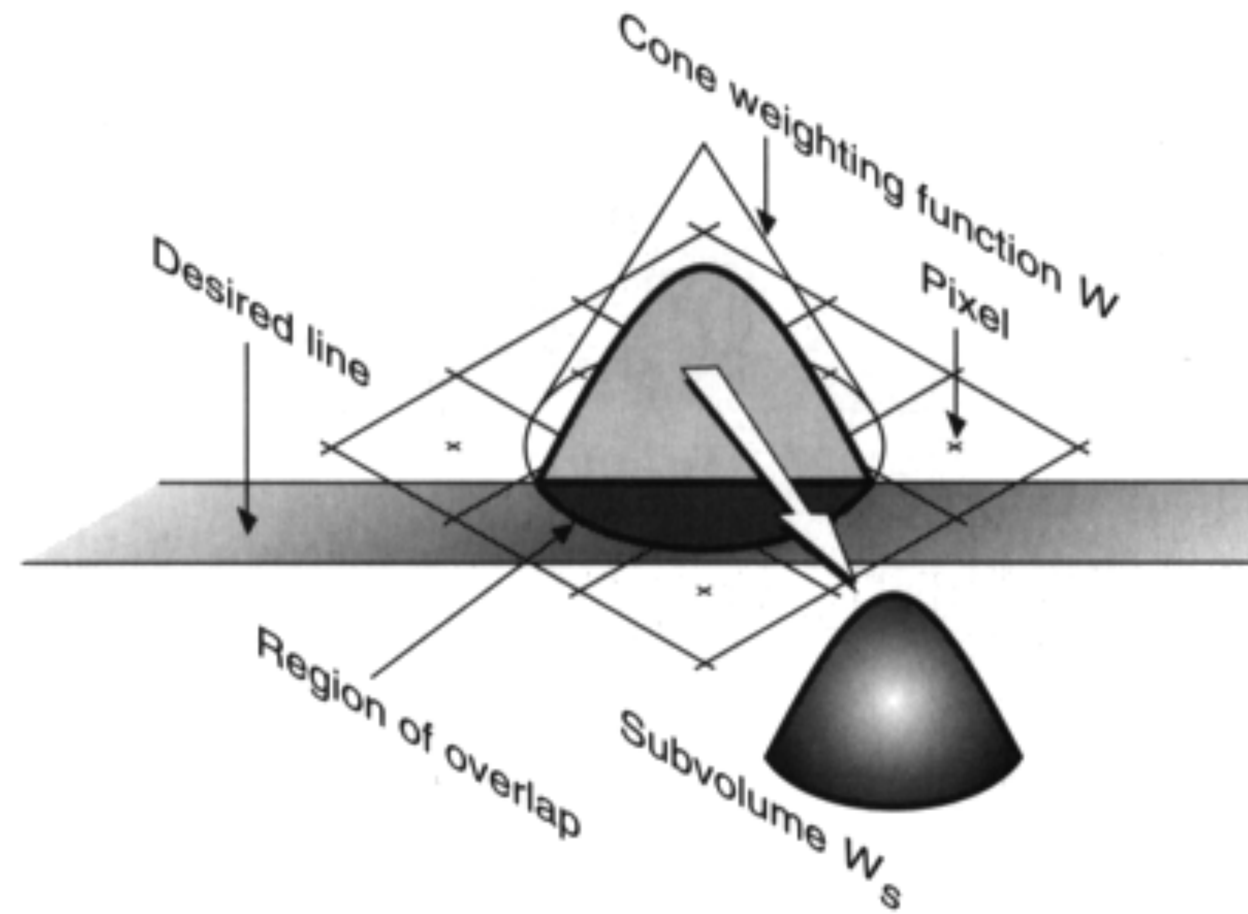
Area Sampling
“Box Filter” (no weight)

- Imagine function $W(x, y)$ be the height of the plane over the surface at (x,y)
- If no weighting, this area is a plane, so W is a cube



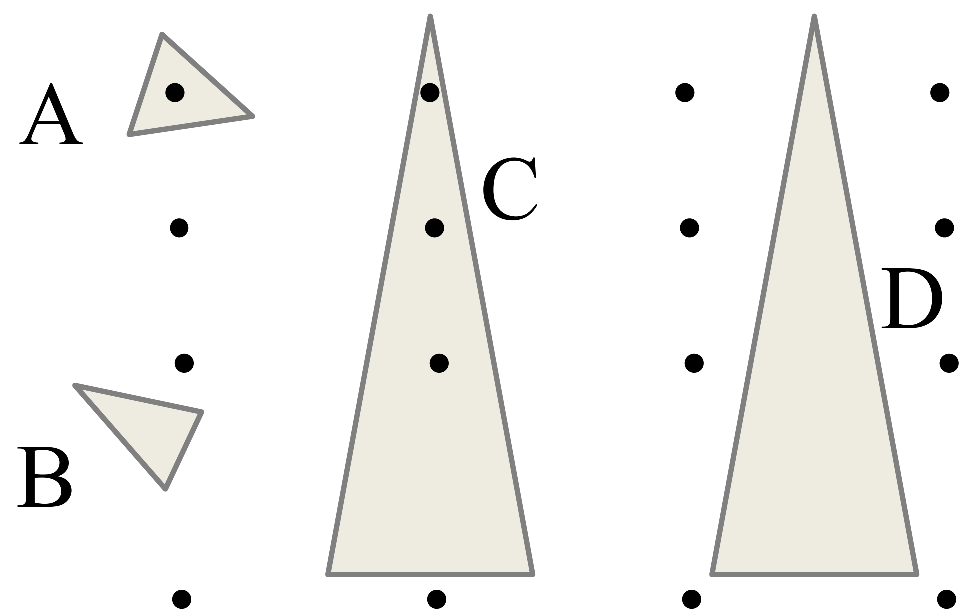
Area Sampling

“Cone Filter” (with weight)



- Now, let W be a cone…
- Radius = pixel’s width
- Linear falloff
- Circular symmetry
- Horizontal / Vertical lines are now more than 1 pixel wide

Another Look at Point Sampling – Even Box Filter is Better!

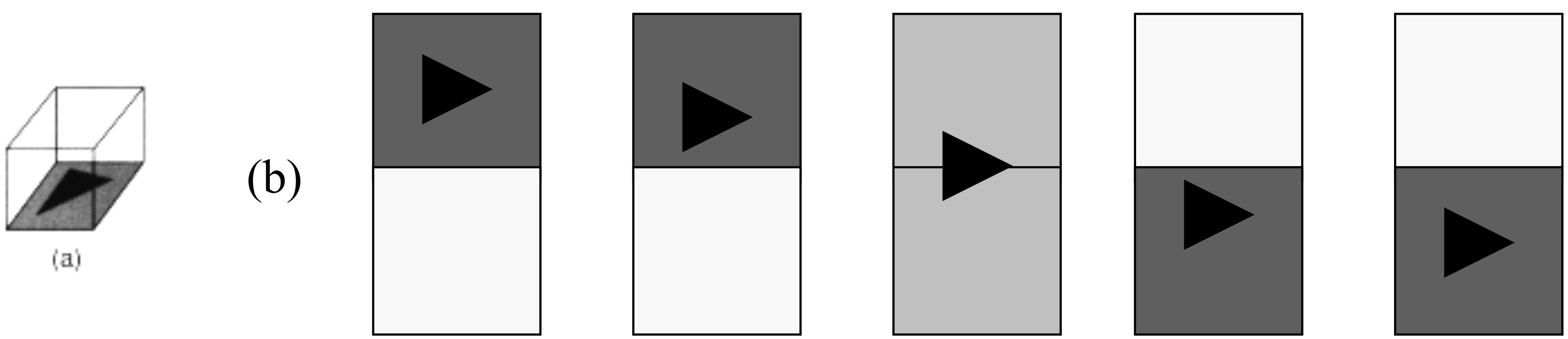


Point-sampling problems. Samples are shown as black dots. Object *A* and *C* are sampled, but corresponding objects *B* and *D* are not.

- This simplistic scan conversion algorithm only asks if a mathematical point is inside the primitive or not
  - Bad for sub-pixel detail which is very common in high-quality rendering where there may be many more micro-polygons than pixels!

Another Look at Unweighted Area Sampling : Box filter

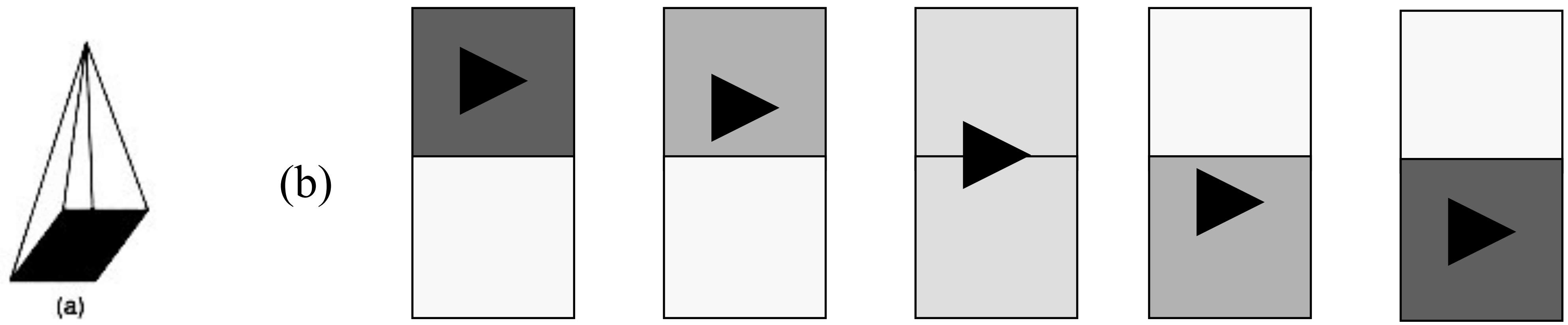
- Support: 1 pixel
- Sets intensity proportional to area of overlap
- Creates "winking" of adjacent pixels as a small triangle translates



Unweighted area sampling. (a) All sub-areas in the pixel are weighted equally. (b) Changes in computed intensities as an object moves between pixels.

Another Look at Weighted Area Sampling : Pyramid filter

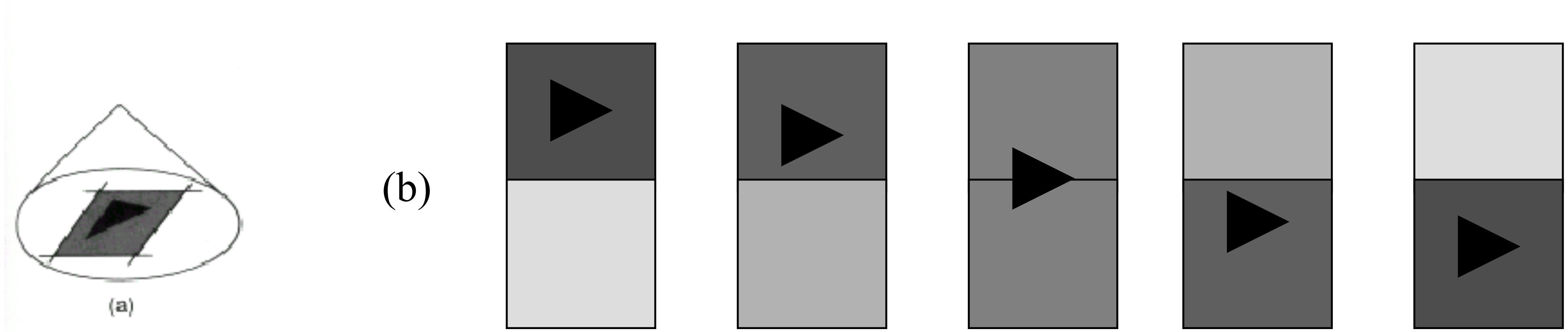
- Support: 1 pixel
- Approximates circular cone to emphasize area of overlap close to center of pixel



Weighted area sampling. (a) sub-areas in the pixel are weighted differently as a function of distance to the center of the pixel. (b) Changes in computed intensities as an object moves between pixels.

Another Look at Weighted Area Sampling : Cone filter

- Support: 2 pixels
- Greater smoothness in the changes of intensity

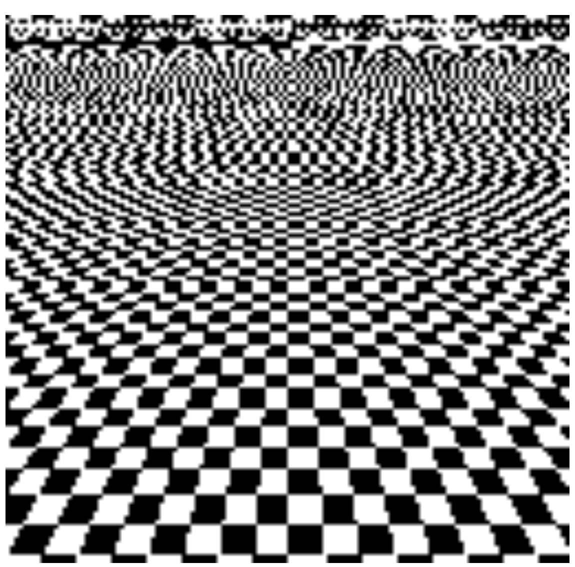


Weighted area sampling with overlap. (a) Typical weighting function. (b) Changes in computed intensities as an object moves between pixels.

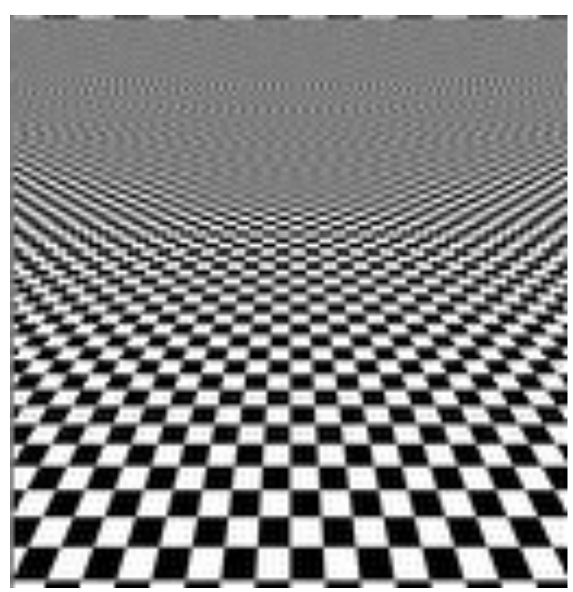
Pseudocode and Results

```
for each sample point p: //p need not be integer!
      place filter centered over p
      for each pixel q under filter:
            weight = filter value over q
            p.intensity += weight * q.intensity
```

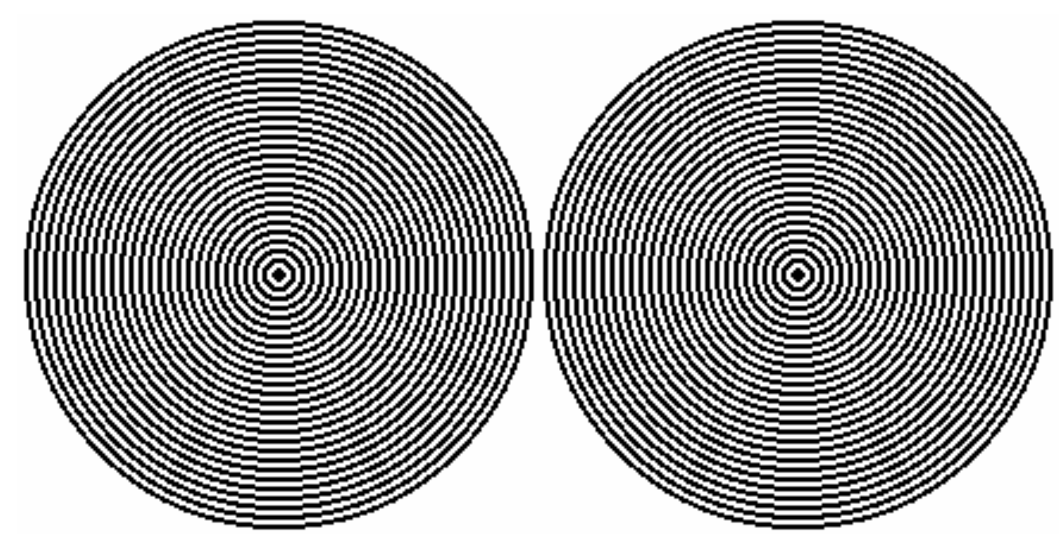
Related phenomenon: Moire patterns



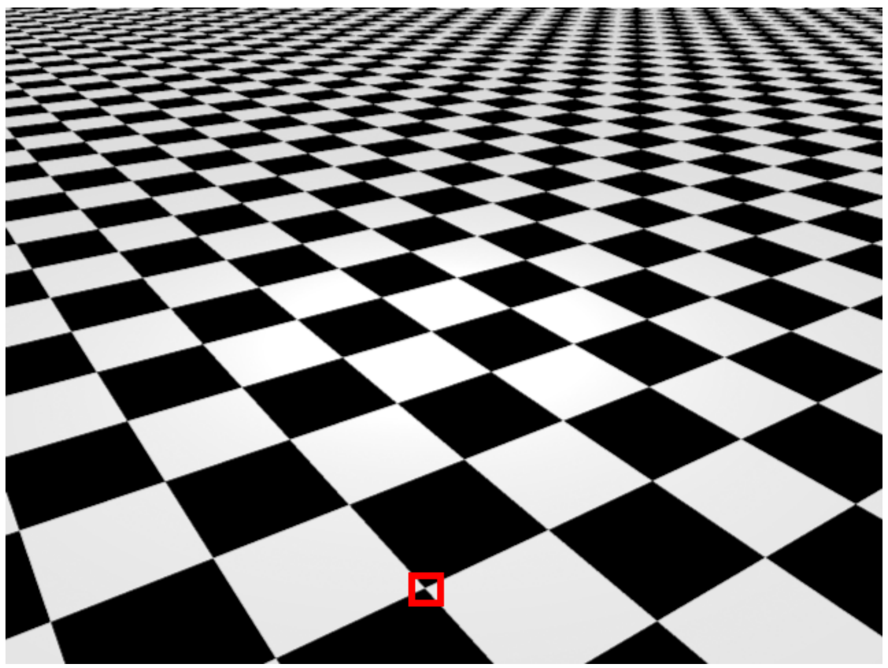
Aliased



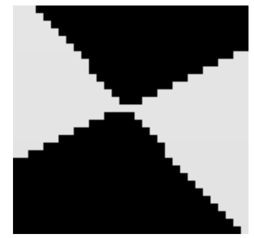
Anti-aliased



Anti-Aliasing Example



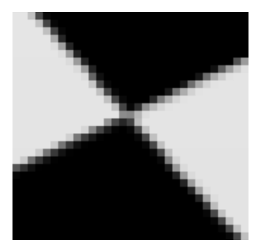
Checkerboard with Supersampling



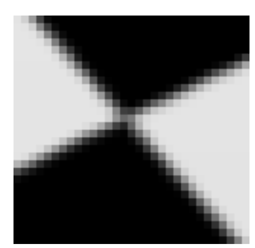
Close-up of original, aliased render

Antialiasing Techniques:

Blur filter – weighted average of neighboring pixels



Supersampling - sample multiple points within a given pixel and average the result



Supersampling and Blurring