What is Computer Graphics?

- Graphical hardware
- Mathematics of 3-D geometry and image creation
- Physics of light sources and scattering
- Psychology of light and color perception
- Methods for 3-D modeling
- Algorithms for converting 3-D models into color images

Applications of Computer Graphics

- **Entertainment: Games, Commercials, Movies, Virtual Reality**
- **Training: Simulated environments, Virtual Reality**
- **Scientific Visualization and Data Analysis**

  Molecular graphics.

  Geographic information systems (maps, topographic maps).

  Turbulence, temperature, stress, etc.
- **Computer-Aided Design**
- **User interfaces**

Hardware Devices

- **Bit-mapped raster graphics**

  Cathode-ray tubes

  LCD flat-panel displays

  Laser printers

- **Line drawing displays**

  Vector CRTs

  Pen-based plotters

We will focus almost entirely on raster devices, because of their flexibility and wide-spread use.

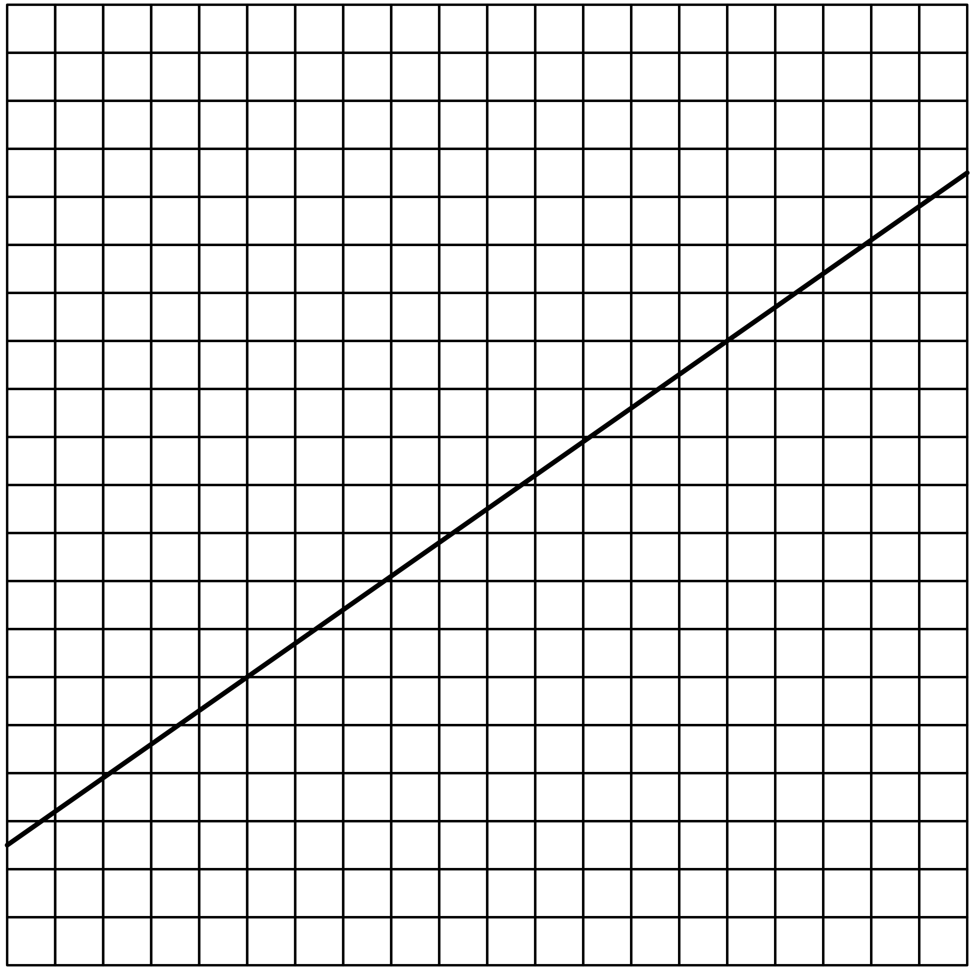
Raster Devices

The visible region is composed of a 2-D array of small dots or squares called "pixels" (picture elements).

Each pixel can be individually turned on and off.

- **Monochrome Displays** (single color)
  Each pixel can either be on or off.
  Inexpensive computer monitors.
  Laser printers.
- **Greyscale Displays**
  Each pixel can have one of n possible brightness values ("intensities").
- **Color Displays**
  Each pixel can have one of k possible colors.
  Each pixel is actually composed of a cluster of (usually 3) single-color pixels (RGB or CYMK) that fool the eye.

Drawing a Line: How hard can it be?

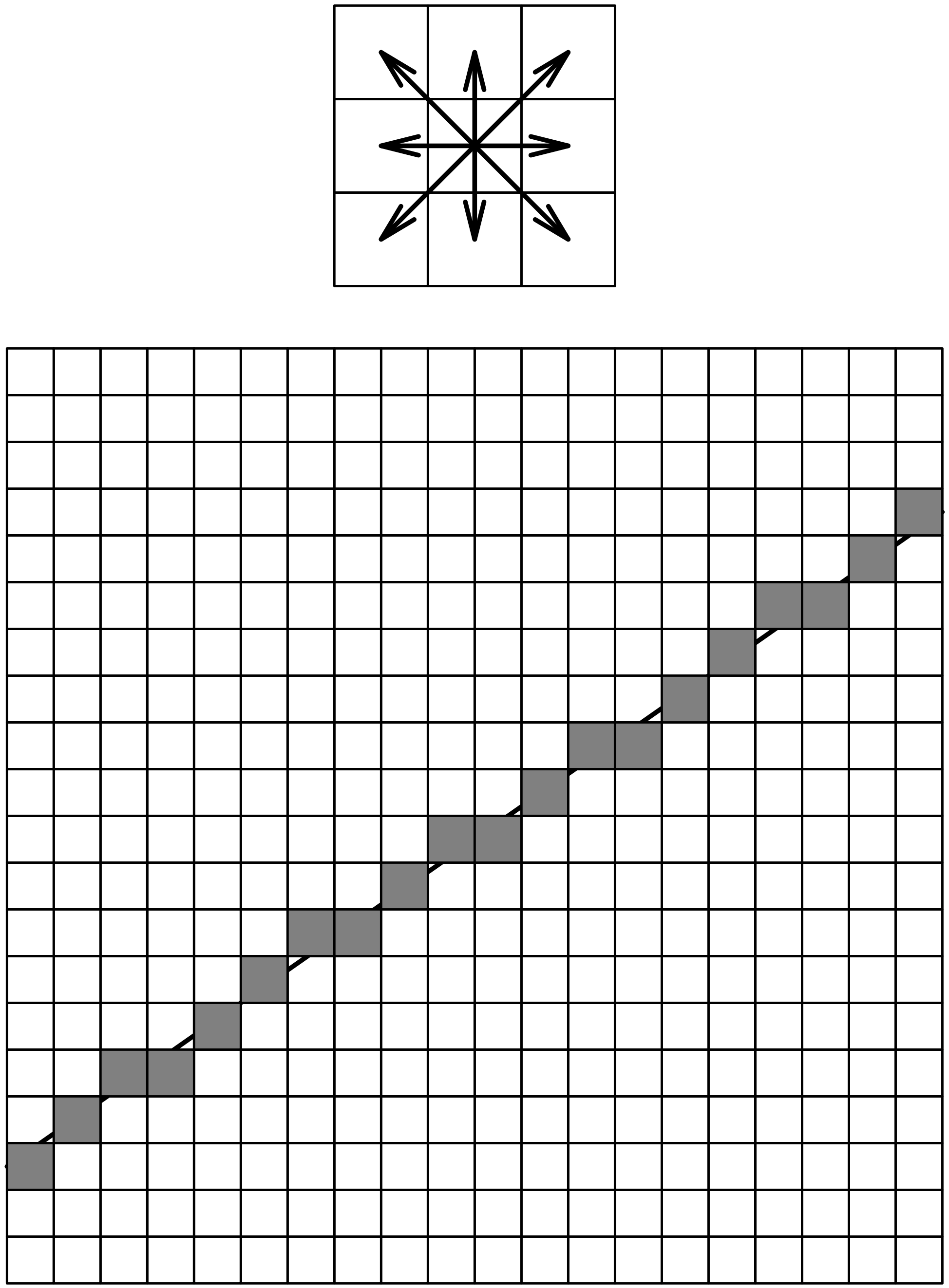


Which pixels should we turn on?

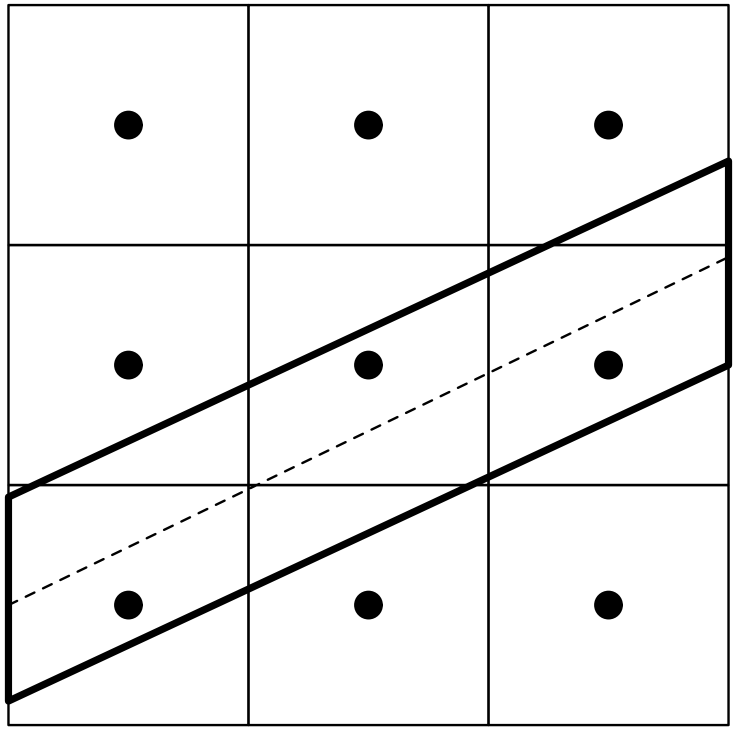
Specifying the Desired Output

- **The shortest sequence of 8-connected pixels that most closely approximates the line.**
- **The set of pixels whose centers are covered by a parallelogram of width 1 centered on the line.**

Eight-connectedness



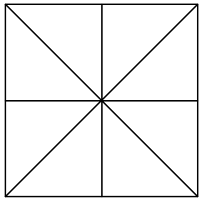
Parallelogram Coverage



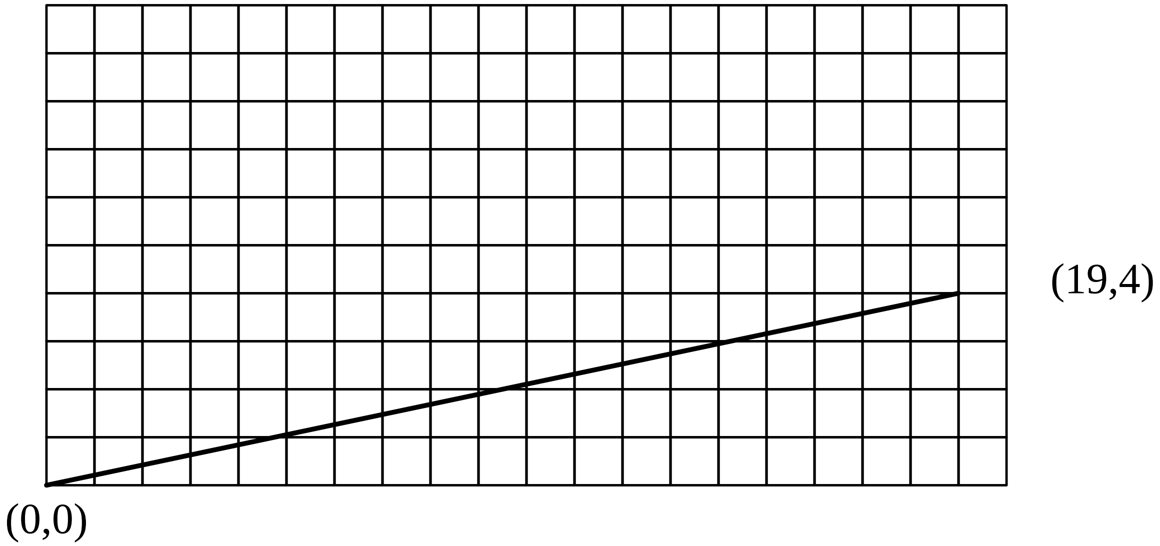
This is the definition used for a width 1 line in the X window system.

Bresenham's Algorithm

Consider the first octant. Assume the line begins at (0,0).



Consider a grid connecting the *centers* of each pixel.



Bresenham's Algorithm (2)

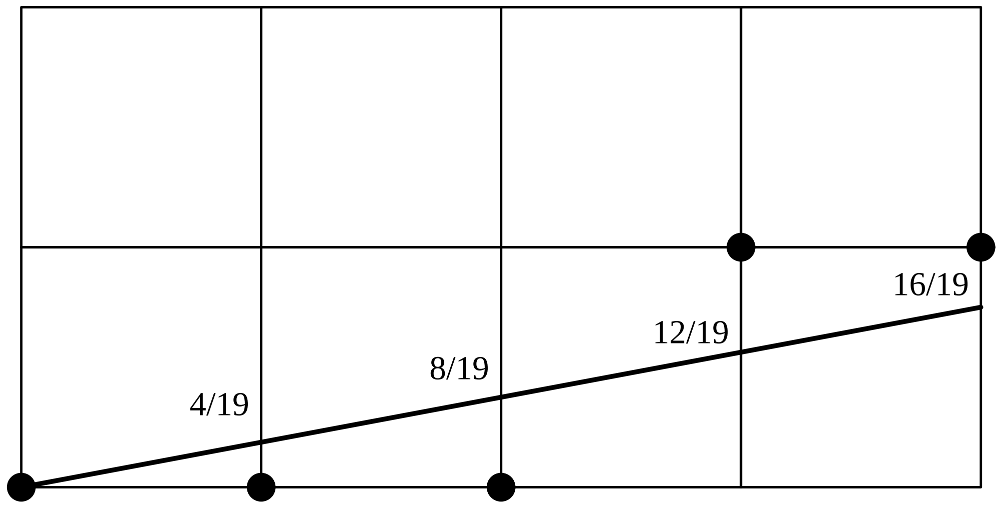


Figure 2.26

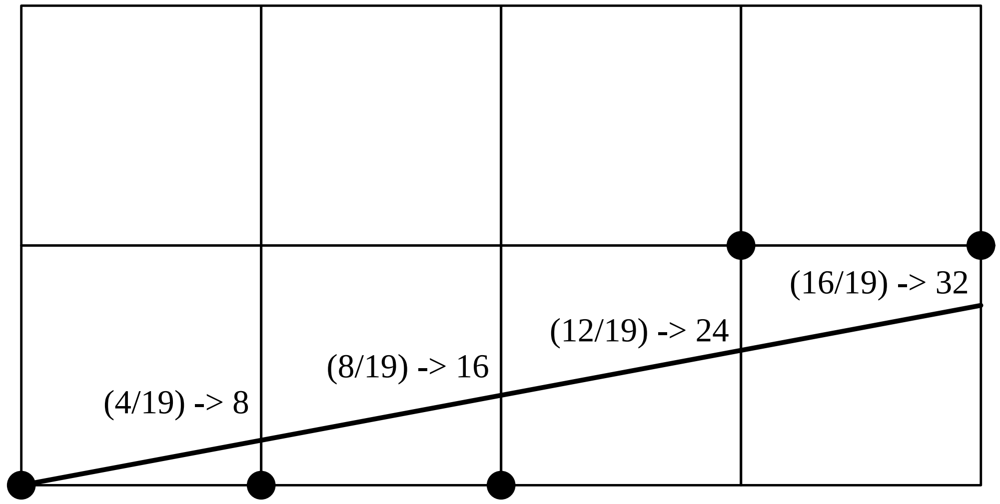
At each step, we move right one pixel and up by $\frac{4}{19}$. When our y value crosses the vertical midpoint between two pixel centers, we move up one row and compute a new "midpoint" target.

Bresenham's Algorithm (3)

```
void DrawTo(int EndX, int EndY)
{
  double midpoint = 0.5, elevation = 0.0;
  double step = float(EndY) / float(EndX);
  int x = 0, y=0;
  while (x <= EndX) {
    setpix(x,y);
    x++;
    elevation += step;
    if (elevation > midpoint) {
        y++;
        midpoint += 1.0;
        }
    }
}
```

Bresenham with Integers

We can eliminate all of the floating point computations by multiplying everything by twice the change in `x` (= `2*EndX`).

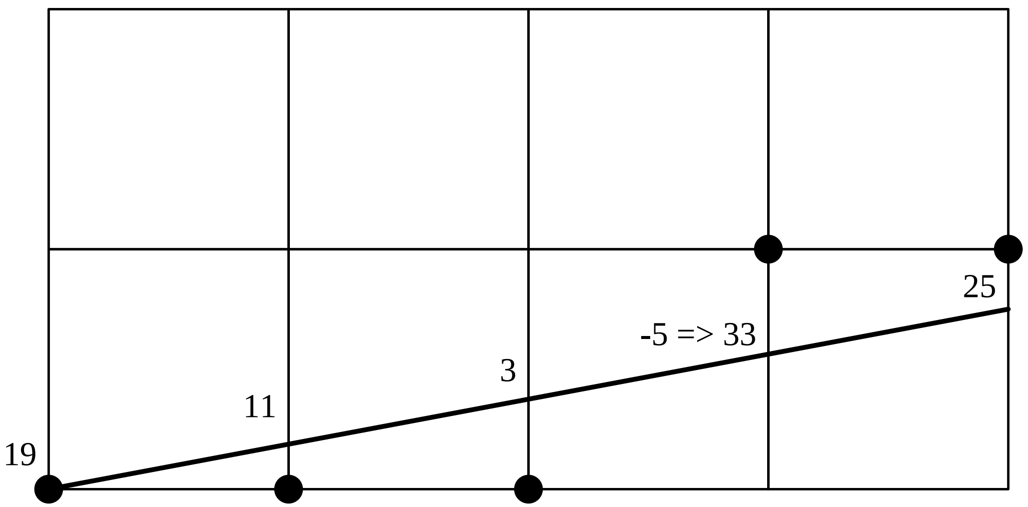


Bresenham with Integers

```
void DrawTo(int EndX, int EndY)
{
  int midpoint = EndX, elevation = 0;
  int step = (EndY << 1);
  int x = 0, y=0;
  while (x <= EndX) {
    setpix(x,y);
    x++;
    elevation += step;
    if (elevation > midpoint) {
        y++;
        midpoint += (EndX << 1);
        }
    }
}
```

Bresenham with Integers

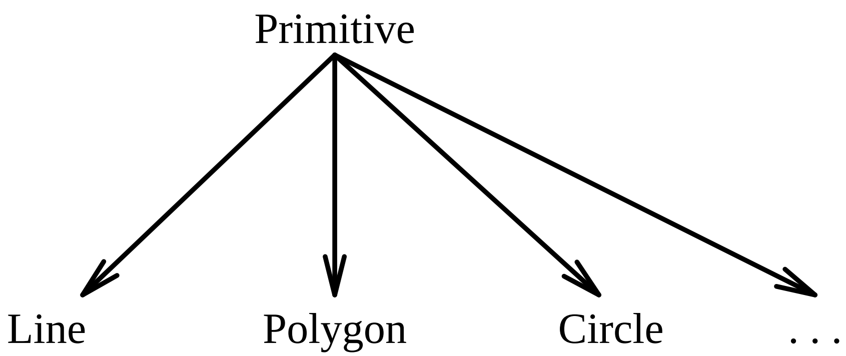
We can eliminate the `midpoint` variable, by adjusting `elevation` to start at the midpoint and *decrease* to zero.



Final Bresenham Algorithm

```
void DrawTo(int EndX, int EndY)
{
  int x = 0, y=0;
  int dx2 = (EndX << 1), dy2 = (EndY << 1);
  int elevation = EndX;
  while (x <= EndX) {
    setpix(x,y);
    x++;
    elevation -= dy2;
    if (elevation < 0) {
        y++;
        elevation += dx2;
        }
    }
}
```

Object-Oriented Design of the Graphics Library

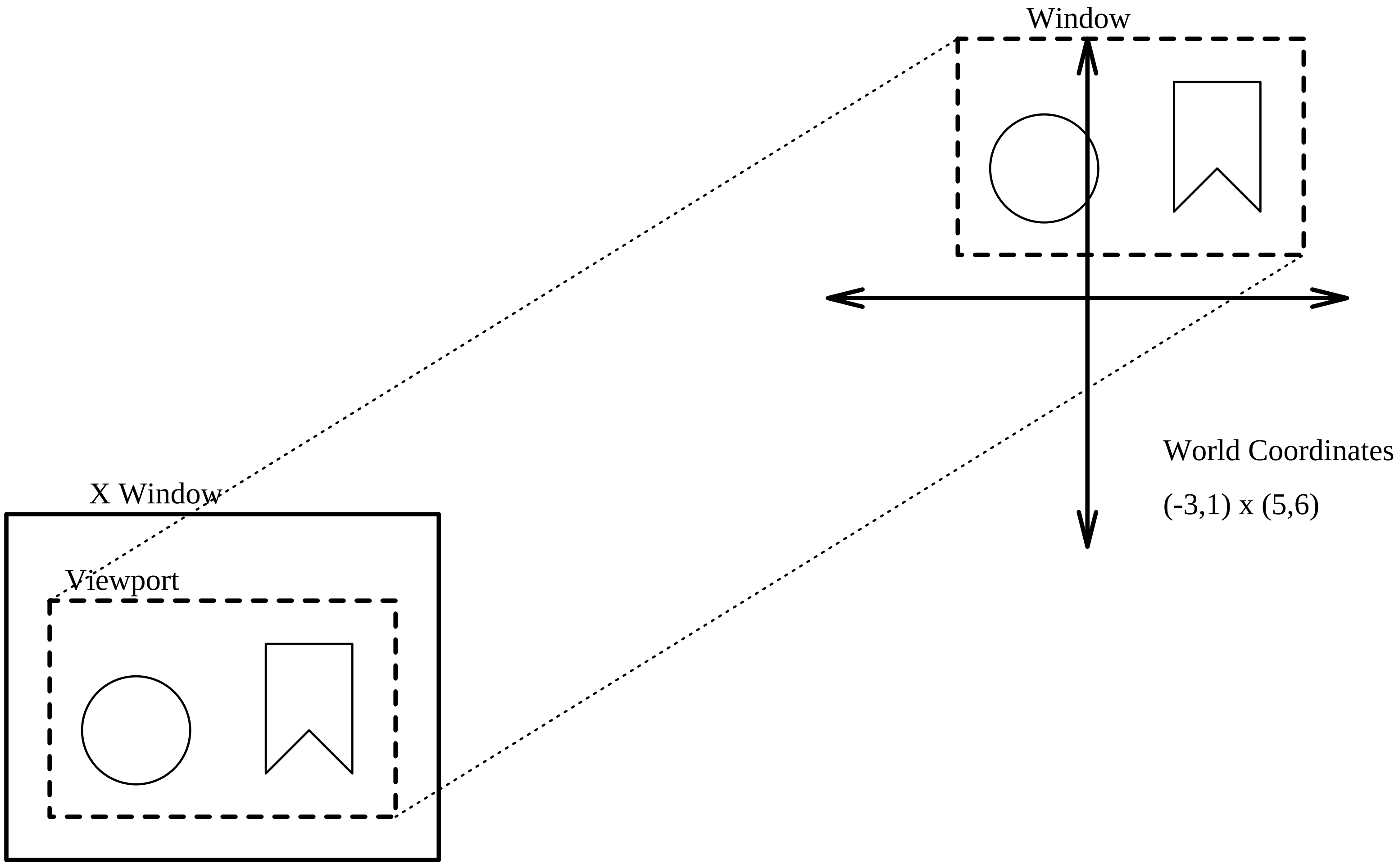


Each primitive object is responsible for drawing itself, filling itself, rotating itself, and translating itself.

Example: The line_t class.

```
class line_t : public prim_t  {                          // 2D line
public:
   line_t() ;
   line_t(double,double,double,double) ;
   line_t(const point_t&,const point_t&) ;
   line_t(mouse_t&) ;                          // construct with mouse
   line_t(const line_t&) ;                     // X(X&) constructor
   prim_t* copy() const ;
   void draw(const scene_t*) const ;
   prim_t& rot(double) ;
   prim_t& translate(const point_t&) ;
   point_t giva() const { return a; }
   point_t givb() const { return b; }
private:
   point_t a , b ;                             // line end points
   point_t cen ;                                     // centroid
} ;
```

Objects Describing the Image-Forming Process



X Window: Device Coordinates   1024 x 768

Viewport: Normalized Device Coordinates: (0.1,0.1) x (0.9,0.7)

Objects Describing the Image-Forming Process

`window_t` Describes window in world coordinates.

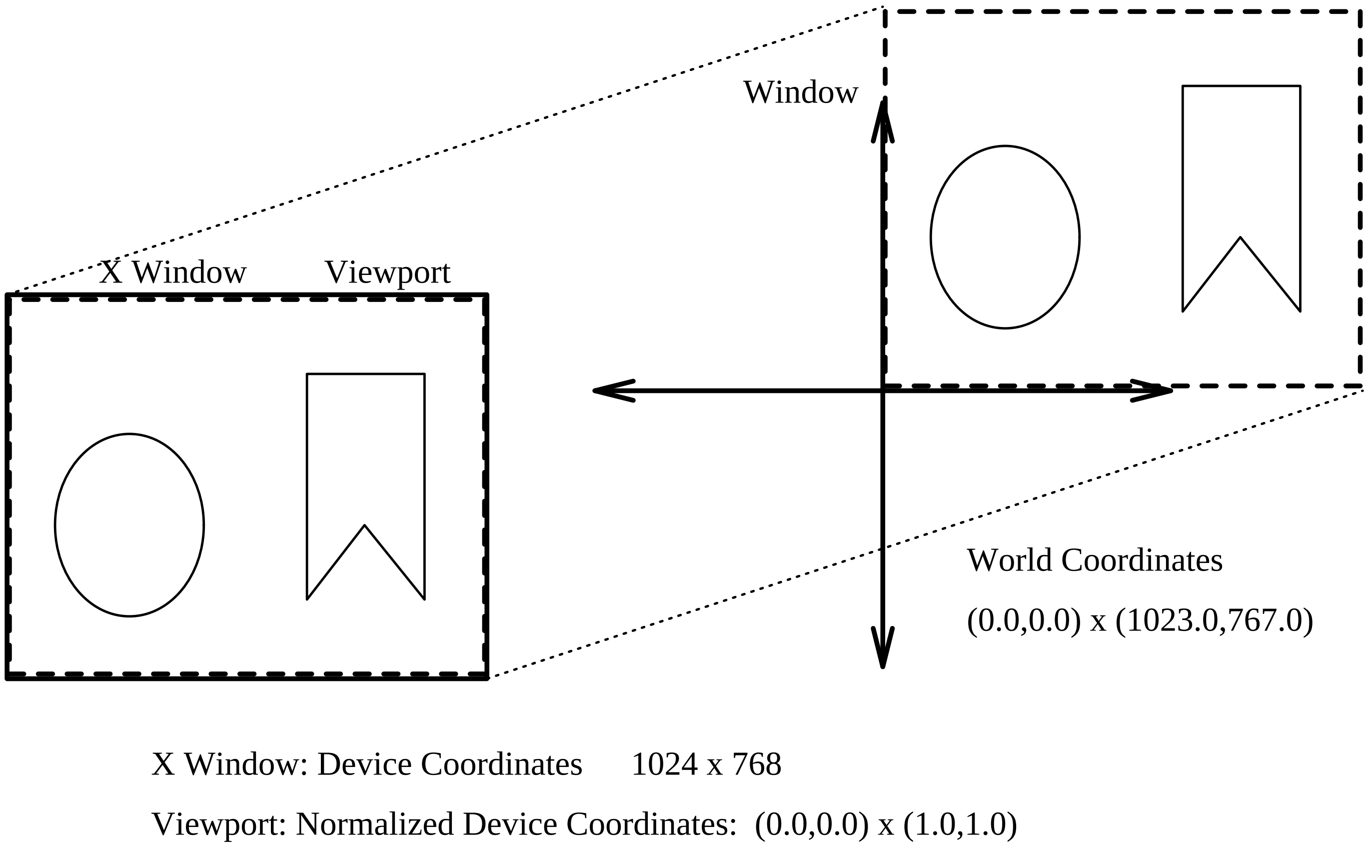
`viewport_t` Describes viewport in normalized device coordinates (i.e., fractional position within the X window).

`win_view_t` An object that merges information from the window and the viewport.

`scene_t` An object that contains a reference to the `win_view_t` and a list of primitives.

The `application_t` class

Handles creating and destroying the X window, viewport, and window.



A Simple Program to Draw a Line

```
#include <stdio.h>
#include <stdlib.h>
#include "../Shape2d/line.h"
#include "../Shape2d/application.h"
#include "../Shape2d/mouse.h"
#include "../Shape2d/scene.h"
void main()
{
    application_t ap;                  // create X window, window, viewport
    scene_t s(ap.wv);                  // create scene
    mouse_t mouse(ap.wv) ;             // create mouse
    line_t l(mouse);                   // ask user to define line with mouse
    s += l;                            // add line to scene
    s.draw();                          // draw the scene
    mouse.wait_button_click();         // let user look at scene
}
```

Convex Polygons: The `convex_t` primitive.

Convex polygons are the most important primitive in graphics.

A polygon is defined by a list of vertexes. The vertexes are just points supplemented with a pointer to the next vertex.

```
class point_t                          // point in 2D space
{                                      // many lines deleted
protected:
   double x , y ;
} ;

class vertex_t : virtual public point_t
{                                      // many lines deleted
protected:
   vertex_t* next ;
} ;
```

The convex_t class

```
class convex_t : public prim_t   // convex polygon is a container class
{
protected:                        // many lines deleted
   vertex_t* v ;                  // vertex list
   int vernum ;                   // vertex count
   point_t cen ;                  // 2-D centroid
} ;
```

This class is really misnamed. Nothing enforces convexity, but some functions *assume* convexity.

Operations on Polygons: Drawing

Drawing a polygon is easy. The underlying hardware supports two operations: `move(x,y)` and `draw(x,y)`. These move a "drawing pen" and cause it to draw a line.

Filling a Polygon

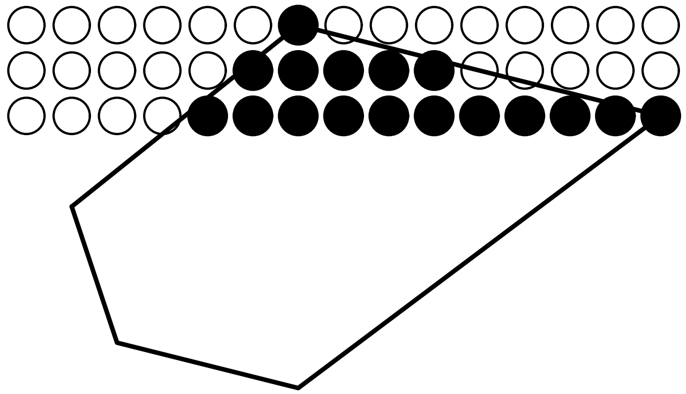
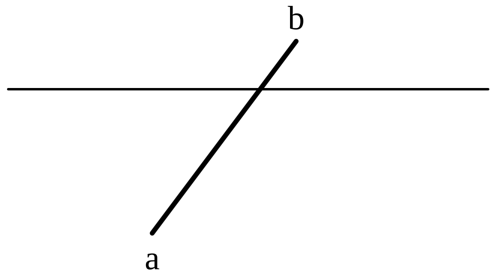


Figure 2.29

We fill by drawing short horizontal lines ("scan lines") that begin and end at the bounding line segments of the polygon. To determine whether an edge intersects the scan line, we can test whether $\min(a.y, b.y) \leq y \leq \max(a.y, b.y)$.



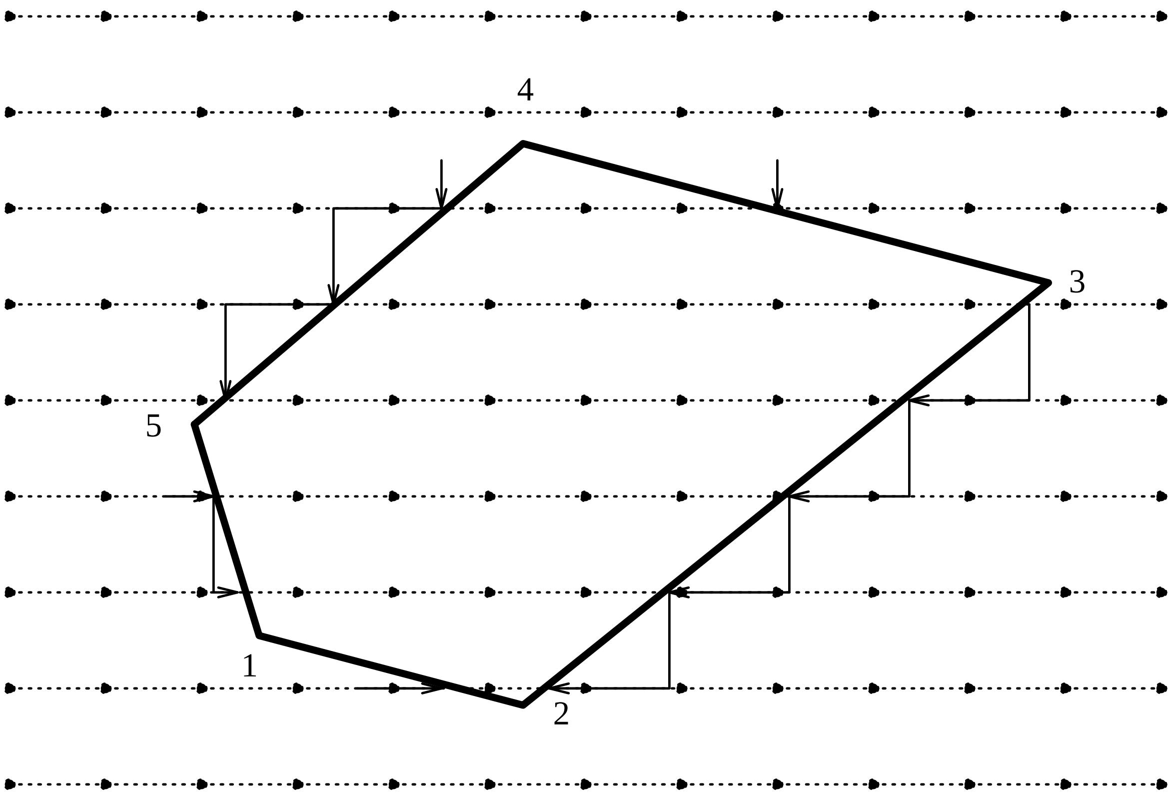
Representing a Polygon as an Edge List

For purposes of filling, it is helpful to convert the list of vertexes into a list of edges. We store the following information:

```
class edge_t            // polygon edge
{                       // member functions omitted
protected:
  double  ymax  ,       // larger of the edge's y-values
          ymin  ,       // smaller of the edge's y-values
          dx    ,       // negative inverse slope of the edge
          x     ;       // intersection x-value of edge and scanline
  edge_t* next  ;       // point to next edge, ordered by increasing ymax
} ;
```

$dx = -\dfrac{a.x - b.x}{a.y - b.y}$ is the amount that x will change when y decreases by 1.

Basic Algorithm



Edge	ymax	ymin	x	dx
4–5	6.6	3.8	4.5	−1.2
4–3	6.6	5.2	8.0	+3.8
3–2	5.2	0.8	10.6	−1.2
5–1	3.8	1.6	2.1	+0.4
1–2	1.6	0.8	4.6	+3.0

Executing the Basic Algorithm

Current y value	Current Edges	Current x range
6.0	4–5 and 4–3	4.5 to 8.0
5.0	4–5 and 3–2	3.3 to 10.6
4.0	4–5 and 3–2	2.1 to 9.4
3.0	5–1 and 3–2	2.1 to 8.2
2.0	3–2 and 5–1	2.5 to 7.0
1.0	3–2 and 1–2	4.6 to 5.8
0.0	none	none

Filling a Non-Convex Polygon

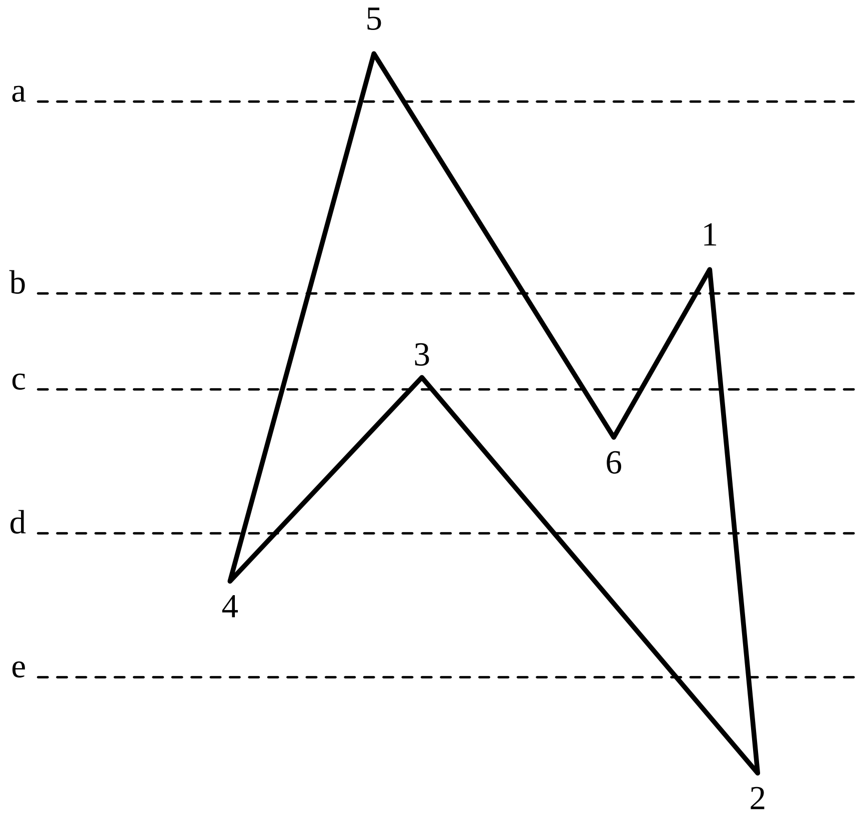


Figure 2.30

We can tell whether we are inside or outside by counting the number of times the "scan line" intersects an edge. An odd number of intersections means we are currently inside the polygon.

Managing the Edge List is Much More Complex

We must not only check for the end of the current edges, we must also check for the start of new edges.

Edge	`ymax`	`ymin`
5–6	16.0	8.0
5–4	16.0	5.0
1–6	11.5	8.0
1–2	11.5	1.0
3–4	9.0	5.0
3–2	9.0	1.0

We maintain a pointer to the *next* edge beyond the edges we are currently handling, and we check for y to capture that edge. After each update of x, we delete any edges that can no longer intersect the scan line and add any new edges that do intersect. Then we sort the current intersecting edges by their x values.

Summary for Polygons

- Polygons are represented by a list of vertexes.
- Polygons are drawn by connecting the vertexes with lines.
- Polygons are filled by converting to an edge list representation, sorting by the maximum y value of each edge, and then computing horizontal segments contained inside the polygon and bounded by two edges.

Circles

There are two different approaches to handling circles:

- Use center (a `point_t`) and radius (a `double`).
- Convert to a polygon with many vertexes. (Pokorny uses 90 vertexes.)

If we choose the center+radius approach, we must handle the problem of drawing and filling the circle. The polygon approach can just use the existing methods for polygons.

Bresenham's Algorithm for Drawing Circles

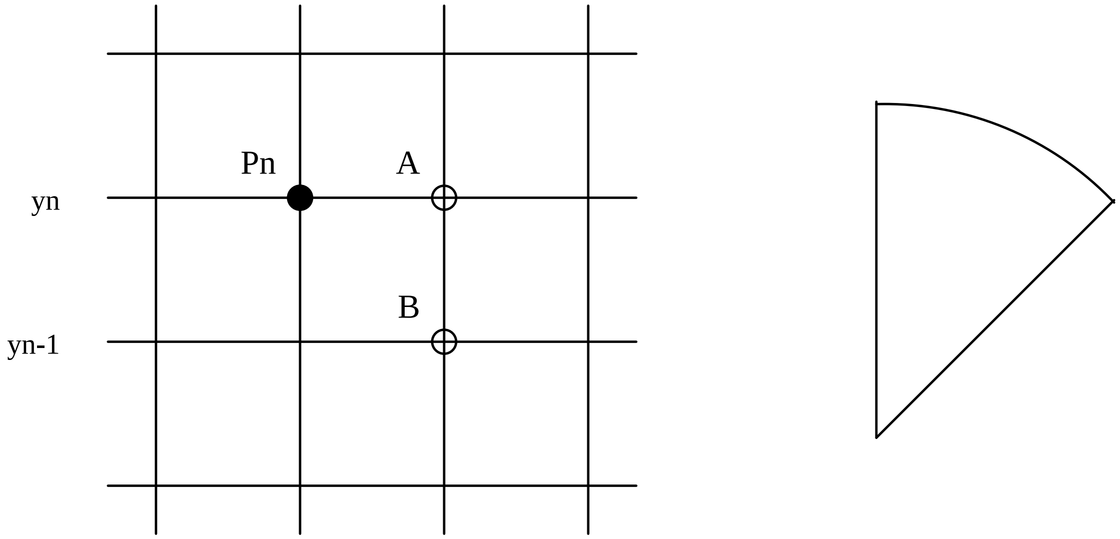


Figure 3.3

We work in the second octant only. After choosing pixel P_n at location (x_n, y_n), we must now choose either point A at (x_{n+1}, y_n) or point B at (x_{n+1}, y_{n-1}). We should choose the point that is closest to the true location of the circle.

Bresenham's Circle Algorithm (2)

Five cases can arise:

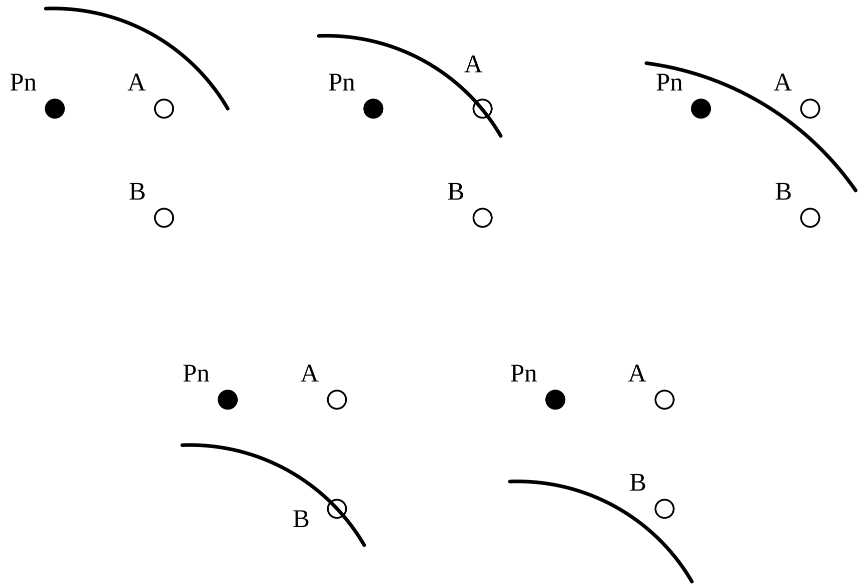


Figure 3.4

Define $f(x, y) = x^2 + y^2 - r^2$.

all points (x, y) for which $f(x, y) = 0$ lie on the circle.

all points (x, y) for which $f(x, y) > 0$ lie outside the circle.

all points (x, y) for which $f(x, y) < 0$ lie inside the circle.

Bresenham's Circle Algorithm (3)

Uses the sign of $f(A) + f(B)$ to decide which point to choose:

Case	$f(A)$	$f(B)$	$f(A) + f(B)$	Point Chosen
a	$-$	$-$	$-$	A
b	0	$-$	$-$	A
c	$+$	$-$	$-, 0$	A
			$+$	B
d	$+$	0	$+$	B
e	$+$	$+$	$+$	B

Bresenham's Circle Algorithm (4)

```
void draw()
{
 int x = 0;
 int y = radius;
 while ( x <= y ) {
    setpix_octs(x,y);
    x++;
    s = 2*x*x + 2*y*y - 2*y + 1 - 2*r*r;
    if (s > 0) y--;
    }
}
```

setpix_octs

```
void setpix_octs(int x, int y)
{
  // cen is the center of the circle
  setpix( x+cen.x, y+cen.y);   // four quadrants
  setpix( x+cen.x,-y+cen.y);
  setpix(-x+cen.x, y+cen.y);
  setpix(-x+cen.x,-y+cen.y);
  setpix( y+cen.x, x+cen.y);   // four quadrants, x and y interchanged
  setpix( y+cen.x,-x+cen.y);
  setpix(-y+cen.x, x+cen.y);
  setpix(-y+cen.x,-x+cen.y);
}
```

Bresenham's Circle Algorithm (5)

Computing the sign of $f(A) + f(B)$ incrementally using the previous value: $S_{n+1} - S_n$

$$S_n = f(A) + f(B) = (x+1)^2 + y^2 - r^2 + (x+1)^2 + (y-1)^2 - r^2$$

Case 1: $S_n \leq 0$ so we choose y again.

$$\begin{aligned}
S_{n+1} &= (x+2)^2 + y^2 - r^2 + (x+2)^2 + (y-1)^2 - r^2 \\
S_{n+1} - S_n &= 2(x+2)^2 - 2(x+1)^2 \\
&= 2(x^2 + 4x + 4 - x^2 - 2x - 1) \\
&= 2(2x+3) = 4x + 6
\end{aligned}$$

Case 2: $S_n > 0$ so we choose y_{n-1}.

$$\begin{aligned}
S_{n+1} &= (x+2)^2 + (y-1)^2 - r^2 + (x+2)^2 + (y-2)^2 - r^2 \\
S_{n+1} - S_n &= 2(x+2)^2 - 2(x+1)^2 + (y-2)^2 - y^2 \\
&= 4x + 6 + y^2 - 4y + 4 - y^2 \\
&= 4(x-y) + 10
\end{aligned}$$

So we can easily update S_n after each choice.

Final Bresenham's Circle Algorithm

```
void draw()
{
 int x = 0;
  int y = radius;
  int s = 3 - 2*radius;
  while ( x <= y ) {
     setpix_octs(x,y);
     s += (s <= 0) ? (4*x++ + 6) : 4*(x++ - y--) + 10;
     }
}
```