3. Raster Graphics

3.1 Display

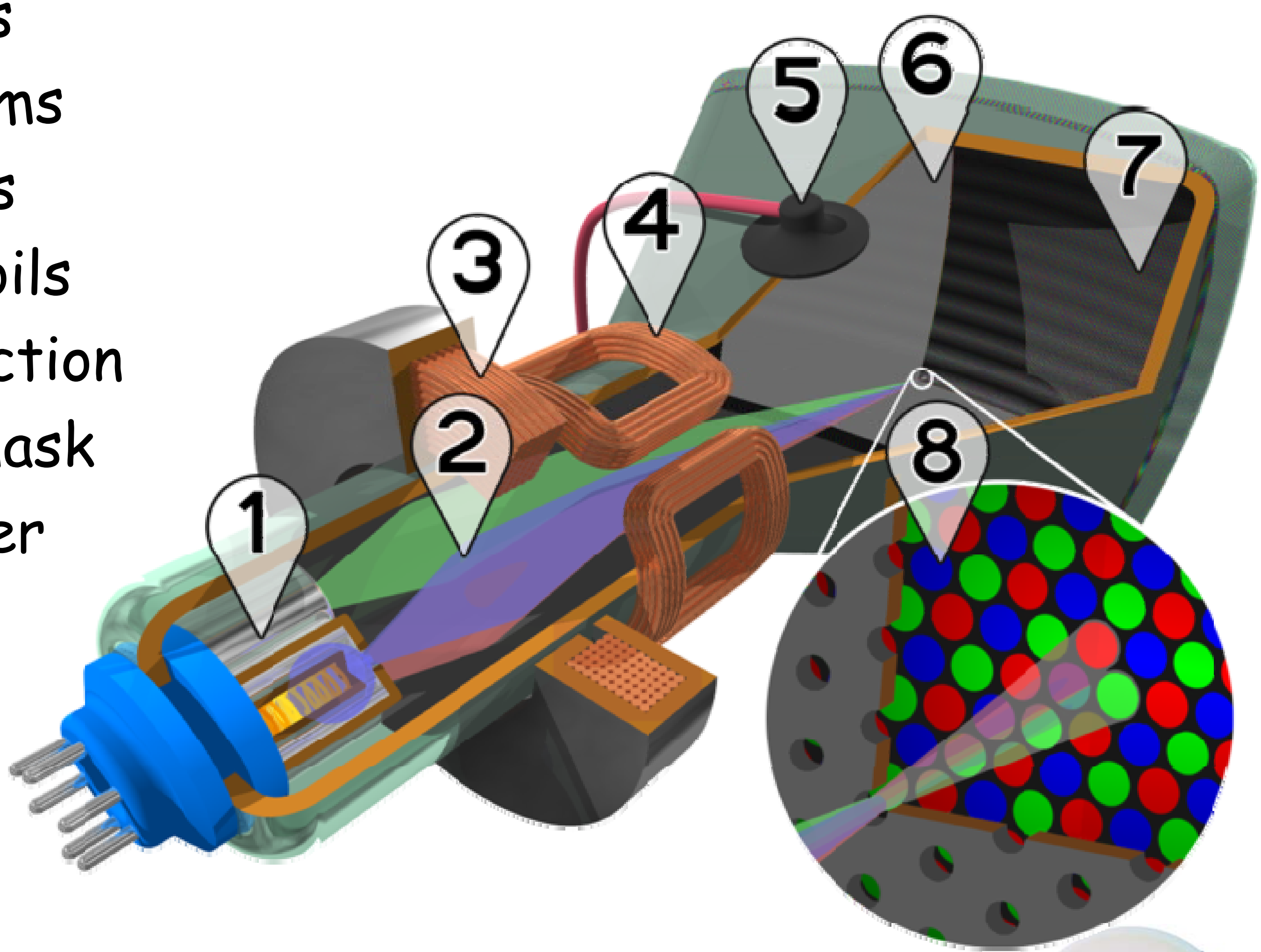
Displays

The most commonly used displays are

- Cathode Ray Tube (CRT)
- Liquid Crystal Display (LCD)
  - Thin-film transistor (TFT)

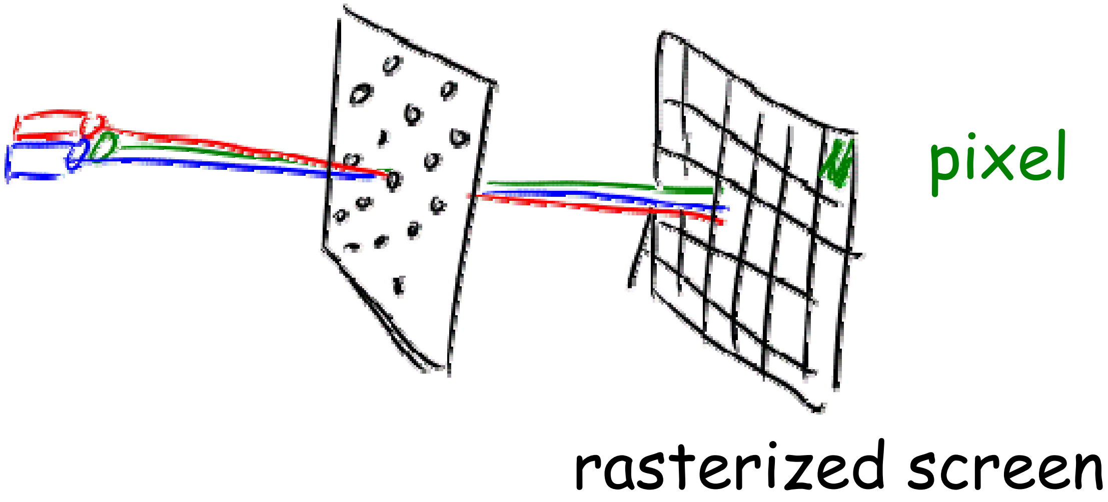
CRT

1. Electron guns
2. Electron beams
3. Focusing coils
4. Deflection coils
5. Anode connection
6. Separating mask
7. Phosphor layer
8. Close-up look



Masking

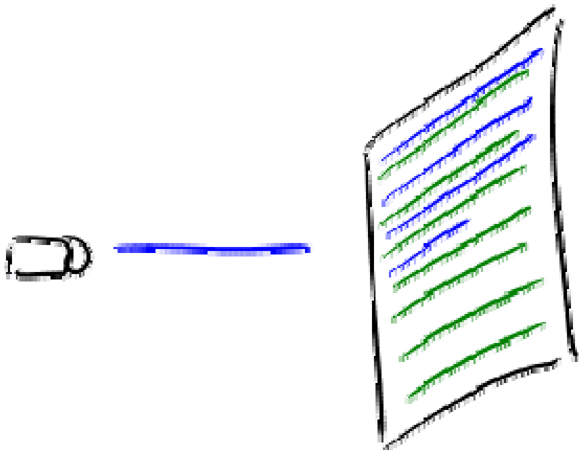
- Separating red, blue, and green light beams:



- Resolution = amount of pixels

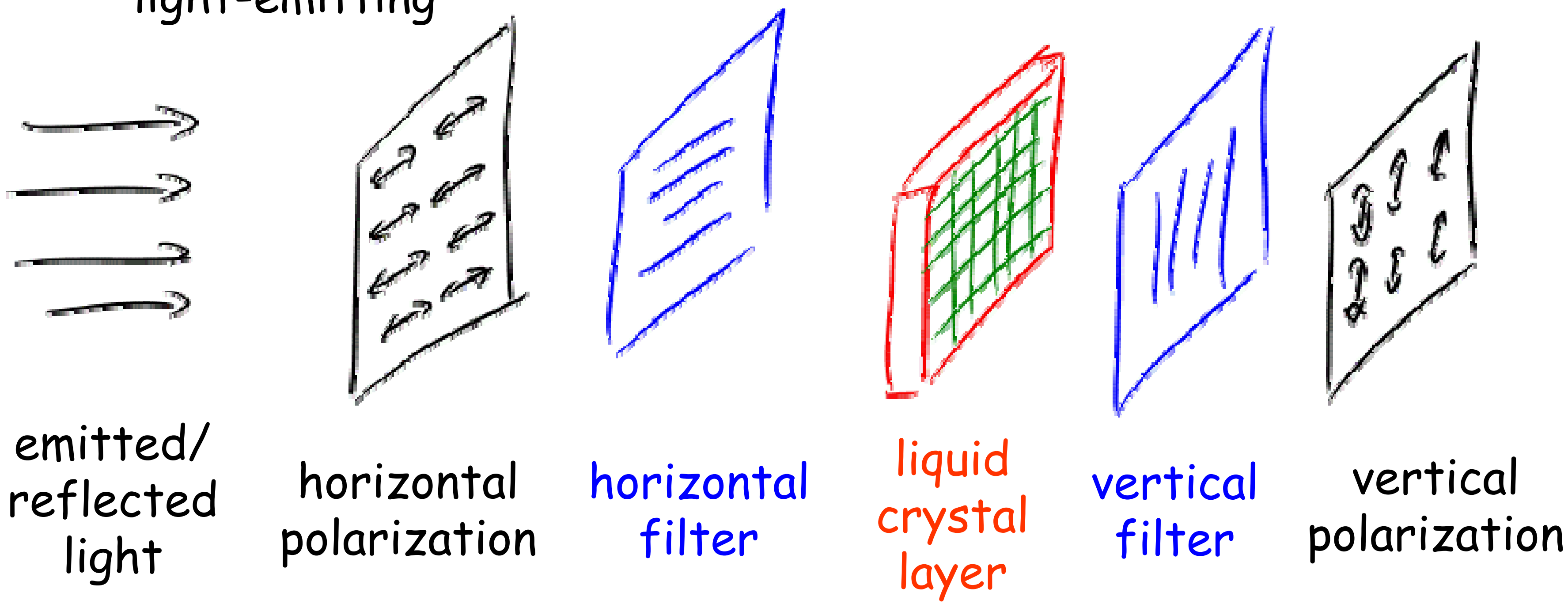
Interlacing

- Interlaced row drawing



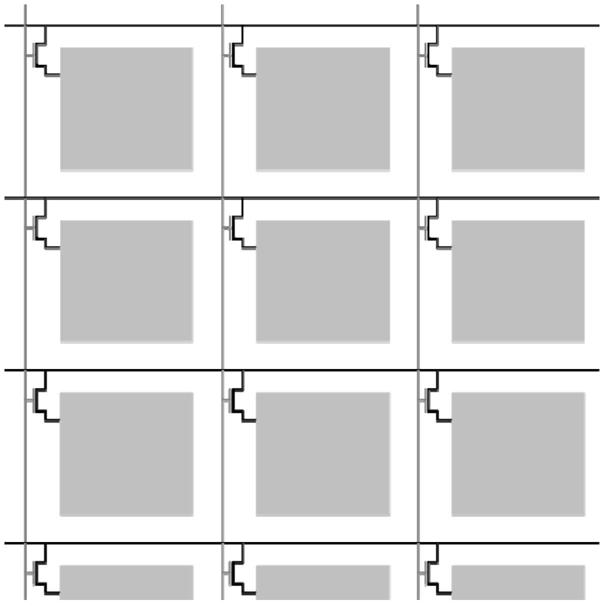
TFT-LCD

- Two versions:
  - mirroring light
  - light-emitting

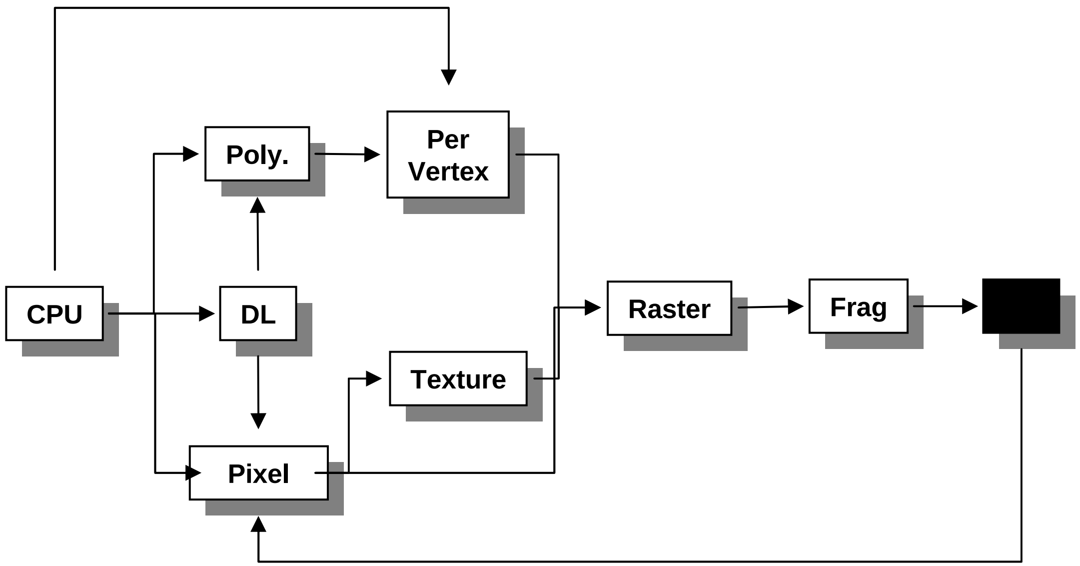


TFT-LCD

- The liquid crystal layer is deployed by a 2D array of thin film transistors.
- One has three transistors per pixel.
- Polarization is changed by 90°, unless voltage is applied.
- The higher the voltage, the darker the pixel.

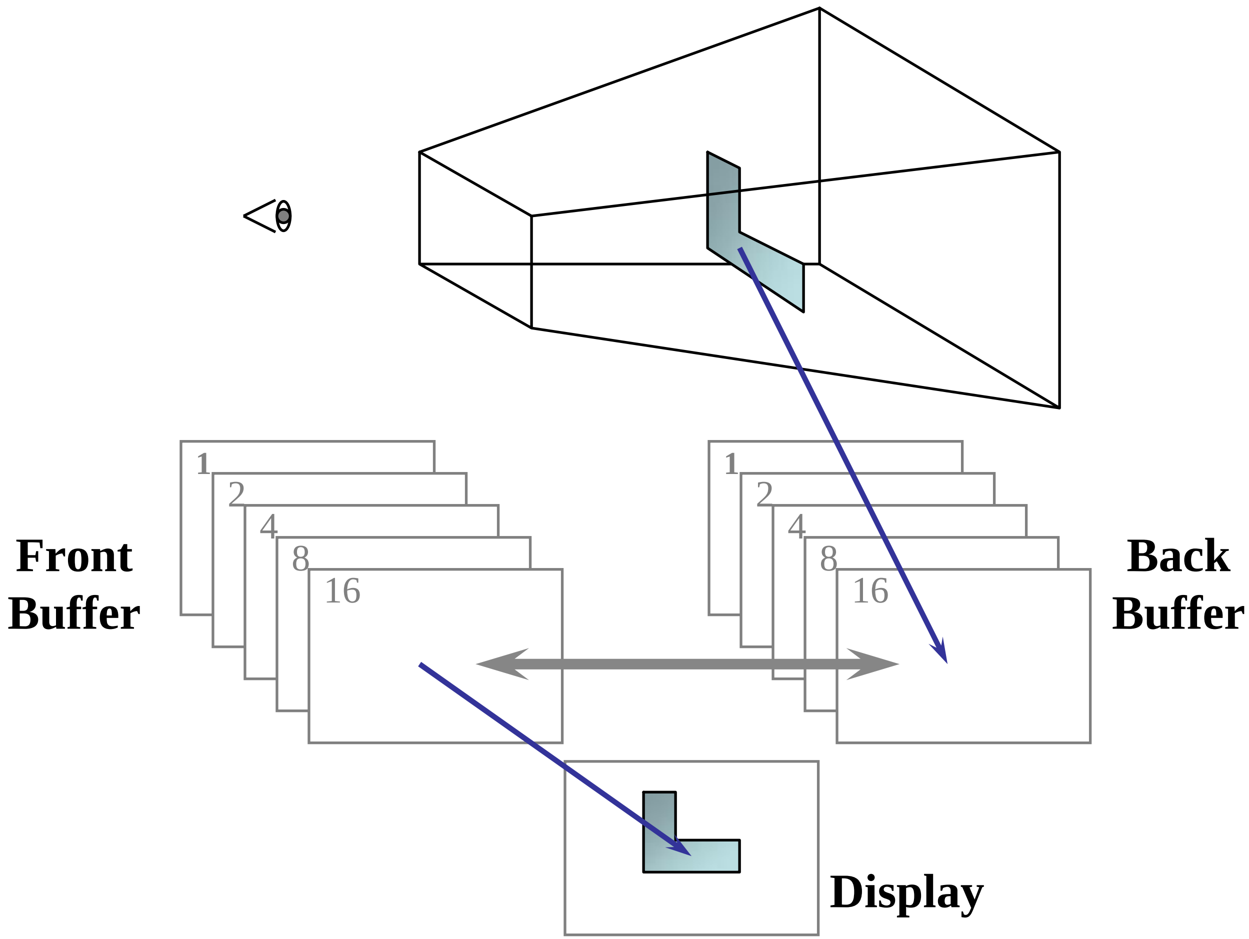


Graphics pipeline: Rasterization



- During rasterization, the projected 2D scene is discretized into a raster image.
- The raster image consists of pixels (called fragments in this context).
- The raster image is composed in the frame buffer.
- The input of the frame buffer is rendered on the screen.

Double buffering



Double buffering in OpenGL

1. Request a double buffered color buffer
   glutInitDisplayMode*(GLUT_RGB* | *GLUT_DOUBLE);*

2. Clear color buffer
   glClear*(GL_COLOR_BUFFER_BIT);*

3. Render scene
4. Request swap of front and back buffers
   glutSwapBuffers();

5. Repeat steps 2 - 4 for animation

Double buffering in OpenGL

```
• void drawScene( void )
• {
     GLfloat vertices[] = { … };
     GLfloat colors[] = { … };
     glClear( GL_COLOR_BUFFER_BIT);
     glBegin( GL_TRIANGLE_STRIP );
  /* calls to glColor*() and glVertex*() */
     glEnd();
     glutSwapBuffers();
• }
```

3.2 Scan conversion

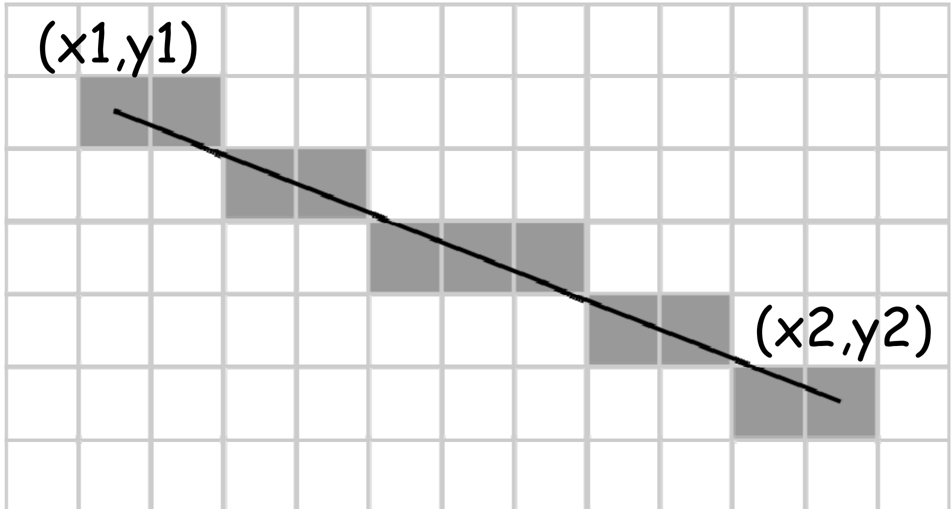
Scan conversion

Definition:

- Scan conversion is the process of mapping the screen-space projection of a 3D scene to the pixel raster of a screen.

Digital differential analyzer (DDA)

- Scan conversion of edges.
- Input: endpoints (x1,y1) and (x2,y2) of an edge as 2D Cartesian coordinates in the screen space coordinate system.
- Output: discretized edge, i.e., framebuffer with those pixels filled that are traversed by the edge.



DDA algorithm

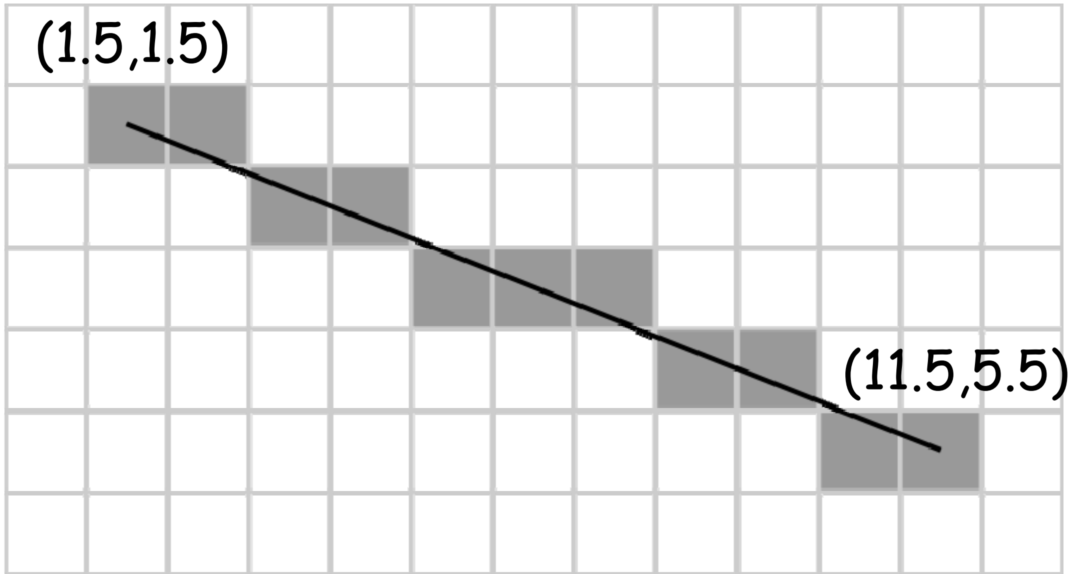
```
DDA (x1, y1, x2, y2)
{
   length = max (|x2-x1|, |y2-y1|);
   dx = (x2-x1)/length;
   dy = (y2-y1)/length;
   for (i=0; i<length; i++)
   {
        plot (⌊x1+i*dx⌋, ⌊y1+i*dy⌋);
   }
}
```

DDA algorithm

length = 10

dx = 1

dy = 2/5



(1.5,1.5), (2.5,1.9), (3.5,2.3), (4.5,2.7), (5.5,3.1), (6.5,3.5), (7.5,3.9), (8.5,4.3), (9.5,4.7), (10.5,5.1), (11.5,5.5)

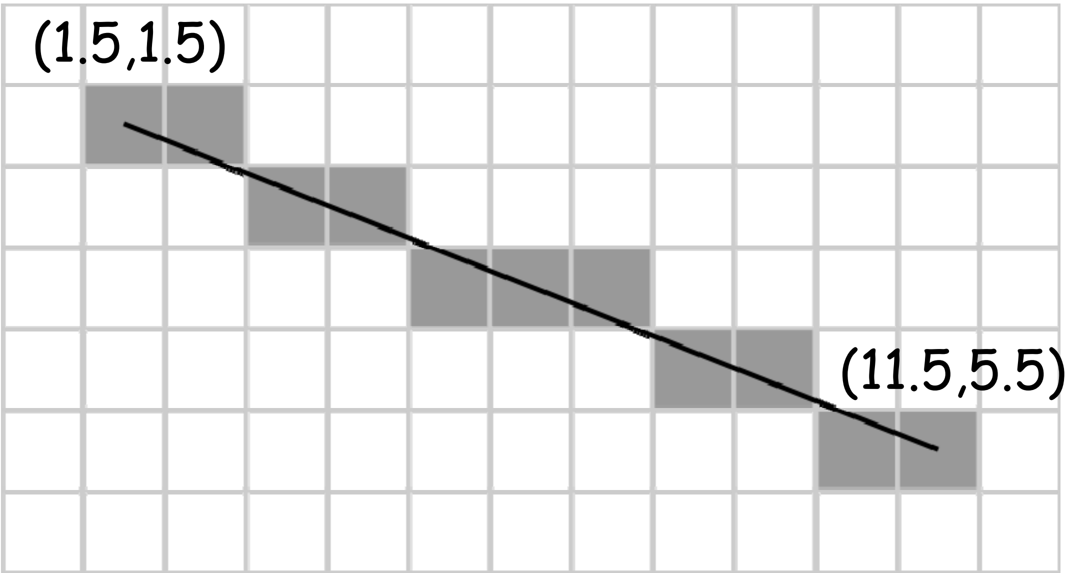
Bresenham algorithm

- Alternative approach to DDA
- Input/output: as before.
- Idea: local increments based on error term.

Bresenham algorithm

```
Bresenham (x1, y1, x2, y2)
{
    dx = x2-x1;
    dy = y2-y1;
    if (dx > dy > 0) // distinguish octants
    {
        (x,y) = (x1,y1); // start point
        error = dy/dx; // initialize with slope
        for (i=1; i <= dx; i++) // loop over x-direction
        {
                plot (⌊x⌋, ⌊y⌋);
                if (error >= 0.5) // if error has accumulated
                {
                        y++; // step in y-direction
                        e--;
                }
                x++; // step in x-direction
                e += dy/dx; // update error
        }}
    else  ... // treat other octants
```

Bresenham algorithm



dx = 10
dy = 4

(x,y)	error
(1.5,1.5)	0.4
(2.5,1.5)	0.8
(3.5,2.5)	0.2
(4.5,2.5)	0.6
(5.5,3.5)	0
(6.5,3.5)	0.4
(7.5,3.5)	0.8
(8.5,4.5)	0.2
(9.5,4.5)	0.6
(10.5,5.5)	0
(11.5,5.5)	

Bresenham algorithm

- The code can be restructured such that it only uses integer operations.
  - idea:
    - multiply all fractional numbers with dx
    - $(dy/dx > 0.5)$ equivalent to $(0.5\,dx - dy < 0)$
  - faster
  - more accurate

- The octants can be handled more efficiently by swapping x1, x2, y1, and y2 respectively.

Bresenham algorithm using integers

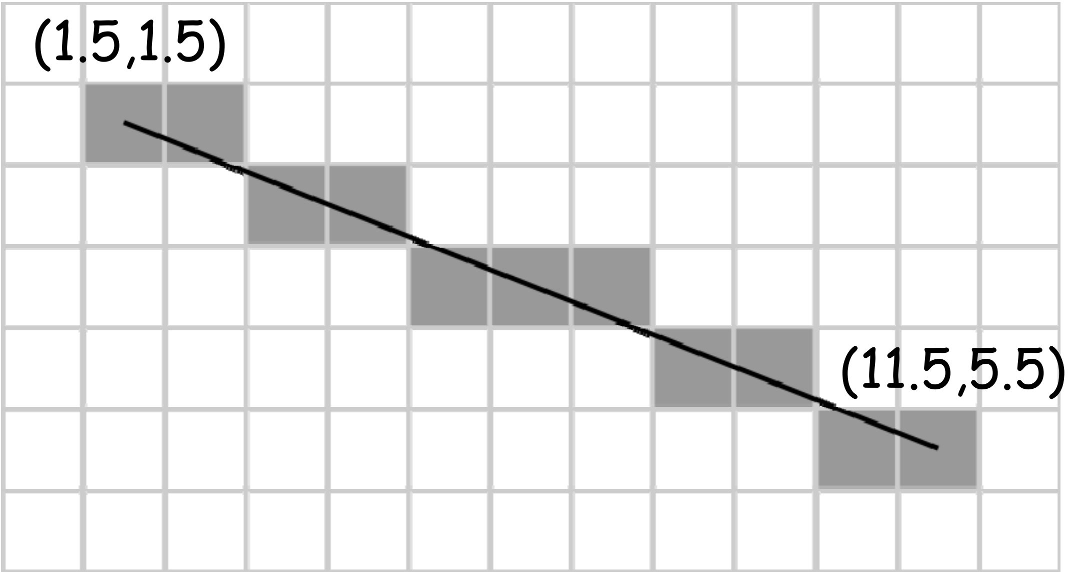
```
Bresenham_integer (x1, y1, x2, y2)
{
   if (|y2-y1| > |x2-x1|)
   {
         swap(x1, y1);
         swap(x2, y2);
   }
   if (x0>x1)
   {
         swap(x1, x2);
         swap(y1, y2);
   }

   ...
```

Bresenham algorithm using integers

```
int dx = x2 – x1;
int dy = |y2 – y1|;
(int,int) (x,y) = (x1,y1); // start point
int error = dx / 2; // initialization
int ystep = (y1 < y2)?1:-1; // step in + or – y-direction
for (i=1; i <= dx; i++)
{
      if (|y2-y1| > |x2-x1|) plot(y,x); else plot(x,y);
      error -= dy; // subtract dy
      x++;
      if (error < 0) // new error test
      {
              y += ystep;
              error += dx; // add dx
      }
}}
```

Bresenham algorithm

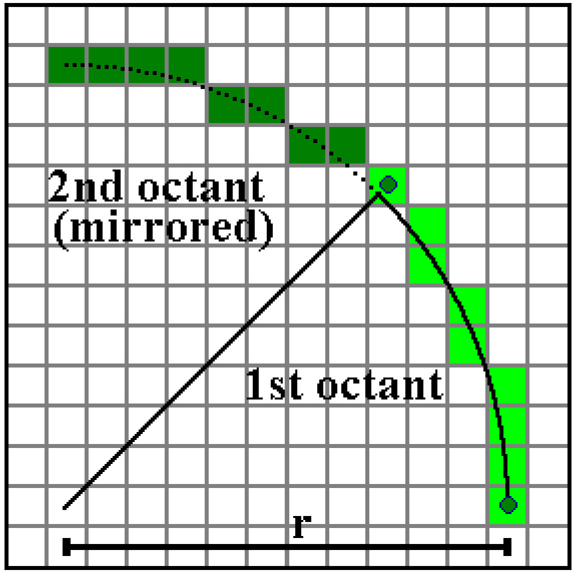


dx = 10
dy = 4

(x,y)	error
	5
• (1.5,1.5)	1
• (2.5,1.5)	7
• (3.5,2.5)	3
• (4.5,2.5)	9
• (5.5,3.5)	5
• (6.5,3.5)	1
• (7.5,3.5)	7
• (8.5,4.5)	3
• (9.5,4.5)	9
• (10.5,5.5)	5
• (11.5,5.5)	

Generalization

- The incremental structure of the algorithm allows for scan conversion of any curve.
- What needs to be adapted is the adjustment of the error term.



Bresenham algorithm for circles

```
BresenhamCircle(int x0, int y0, int radius)
{
    int f = 1 - radius;
    int ddF_x = 1;
    int ddF_y = -2 * radius;
    int x = 0;
    int y = radius;

    plot(x0 + x, y0 + y);
    plot(x0 + x, y0 - y);
    plot(x0 + y, y0 + x);
    plot(x0 - y, y0 + x);

    …
```

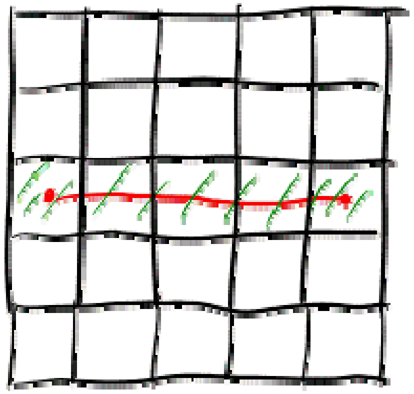
Bresenham algorithm for circles

```
...
while(x < y)
{
        if(f >= 0)
        {
                y--;
                ddF_y += 2; // adjusting slope
                f += ddF_y; // update error
        }
        x++;
        ddF_x += 2; // adjusting slope
        f += ddF_x; // update error

        plot(x0 + x, y0 + y); plot(x0 - x, y0 + y);
        plot(x0 + x, y0 - y); plot(x0 - x, y0 - y);
        plot(x0 + y, y0 + x); plot(x0 - y, y0 + x);
        plot(x0 + y, y0 - x); plot(x0 - y, y0 - x);
}
}
```

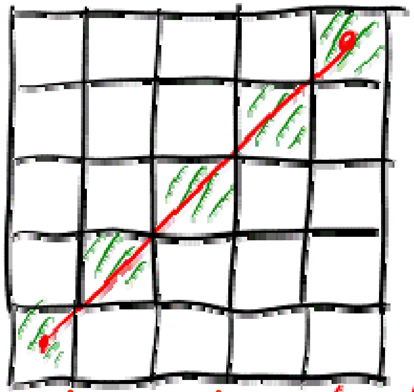
Problems

- Line thickness depends on orientation:



length = 4

#pixels = 5



length = $4 \cdot \sqrt{2}$

#pixels = 5

- Aliasing (see later)

3.3 Polygon filling

Polygon filling

- Polygon filling treats the drawing of polygonal faces after being projected to the screen space.
- The input is a (set of) polygon(s).
- The output is a raster image stored in the framebuffer that represents a discrete version of the projected polygons.

Seed fill algorithm

- The seed fill (also: flood fill) algorithm starts with a seed point inside the polygon and keeps on filling in all directions until the boundary of the polygon is hit.

1. Scan convert all edges of the polygon.
2. Choose a seed point inside the polygon and fill the respective pixel.
3. Recursively fill all pixels of the 4-point neighborhood, if they are not already filled.

   Stop recursion at already filled pixels.

Seed fill algorithm

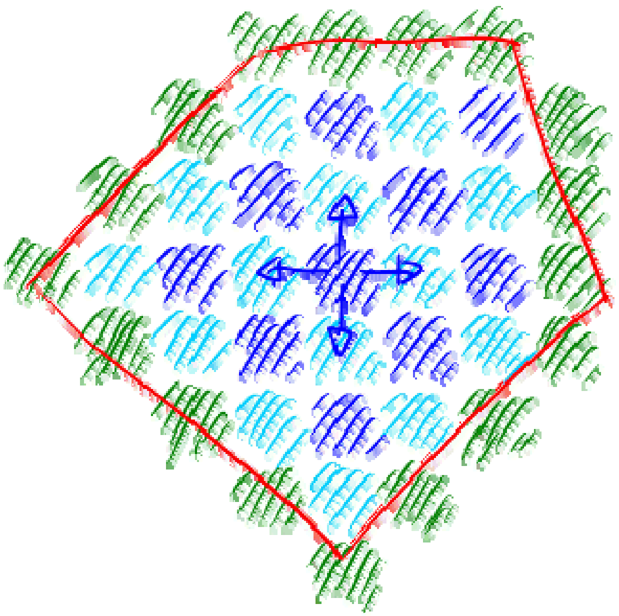
polygon

scan conversion

seeding

neighbors

seed fill iterations



Seed fill algorithm

- How to pick appropriate seed point?
- It must lie inside the polygon

Strategy:

1. Use heuristic (e.g., barycenter of polygon) to pick any point.
2. Check inside-property using a ray-intersection test.

Ray-intersection test

- Let **x** be the chosen seed point.
- Shoot a ray to infinity (typically along a coordinate axis)
- Compute intersection of ray with all edges of the polygon
- Iff number of intersections is odd, x lies inside the polygon