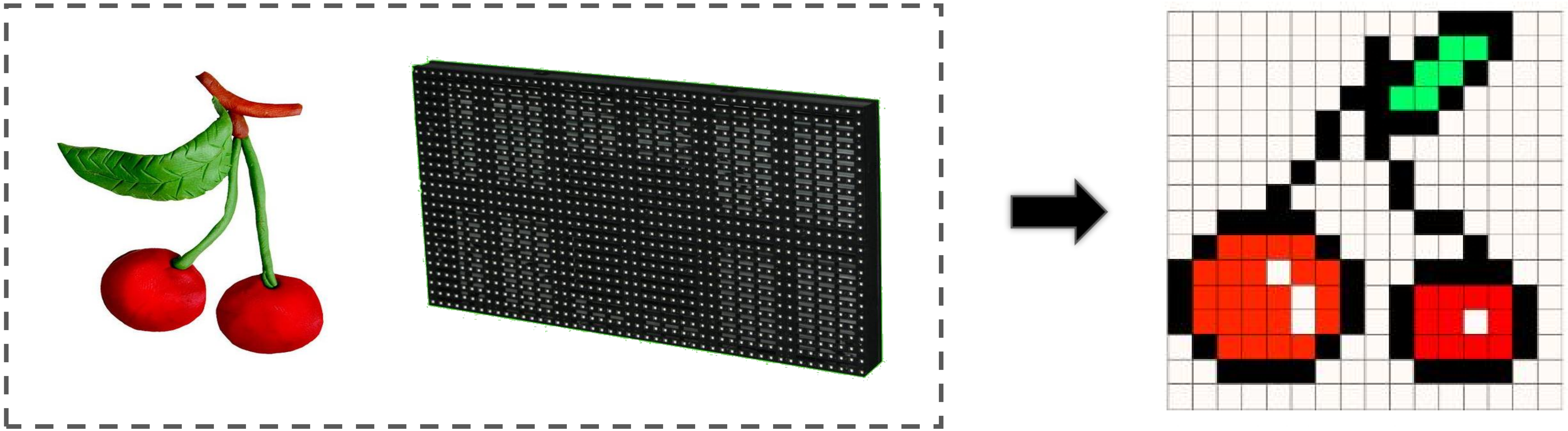
第四章 光栅化算法

什么是光栅化算法？

光栅化的目的是将图形绘制在光栅显示器屏幕上。

图形表示：用代数曲面或者三维网格表示**（“连续表示”）**

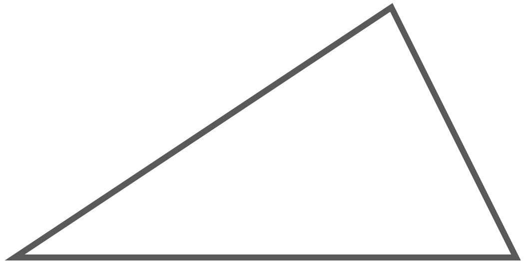
光栅显示器：光栅扫描式图形显示器的简称，是画点设备，可看作是一个点阵单元发生器，并可控制每个点阵单元的亮度**（“离散表示”）**



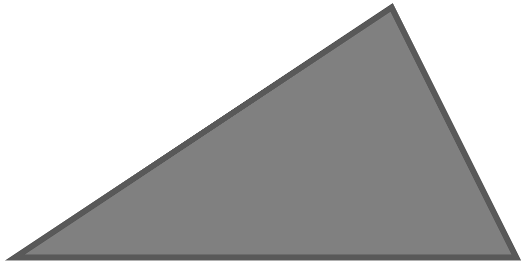
什么是光栅化算法?

- **Rasterization** (scan conversion)
  - Determine which pixels that are inside primitive specified by a set of vertices
  - Produces a set of fragments
  - Fragments have a location (pixel location) and other attributes such color, depth and texture coordinates that are determined by interpolating values at vertices
- Pixel colors determined later using color, texture, and other vertex properties.

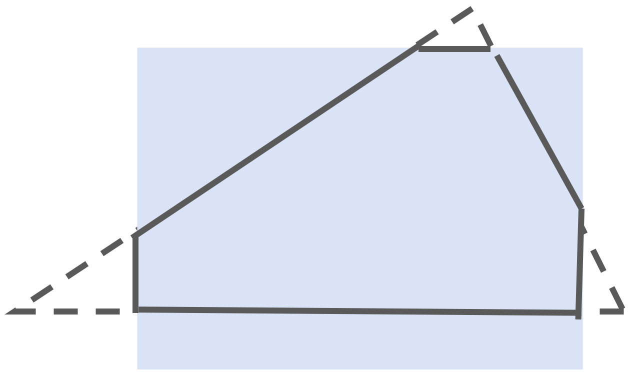
本章内容



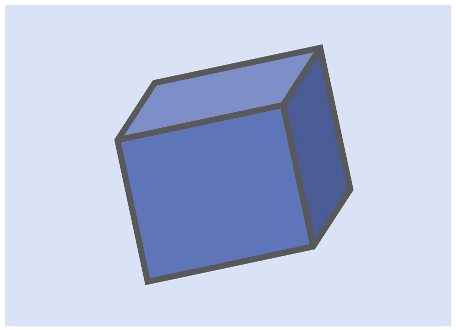
4.1 图形生成算法



4.2 区域填充算法



4.3 剪裁算法

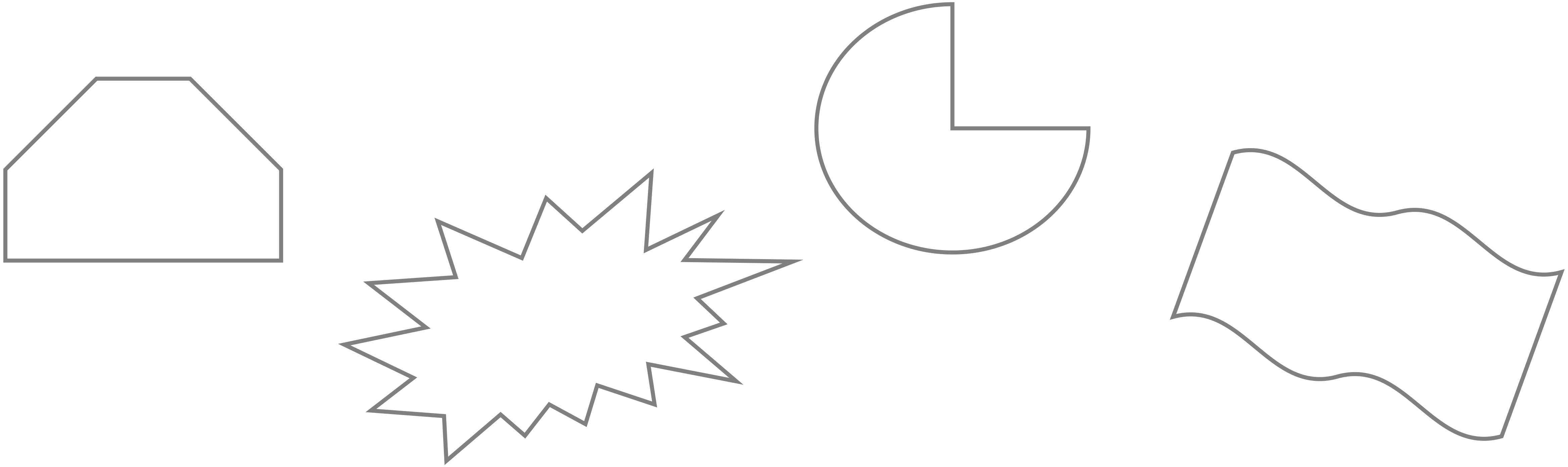


4.4 消隐算法

4.1 图形生成算法

如何在屏幕上绘制一个二维图形?

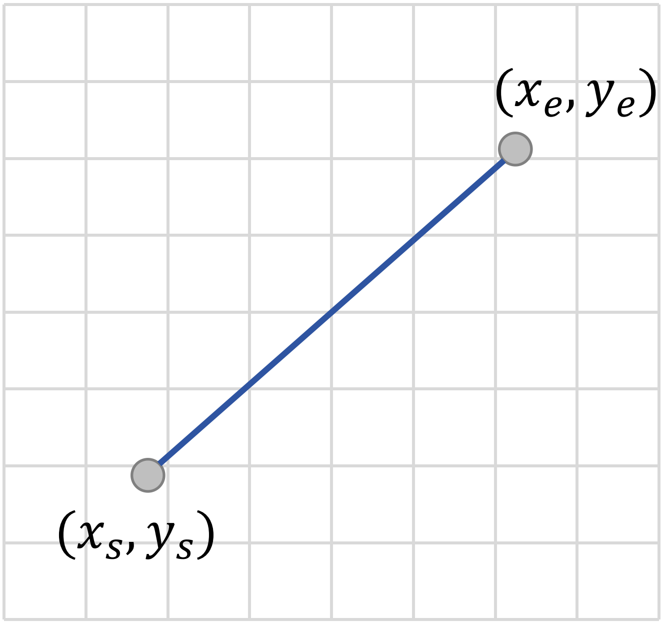
- Lines and curves are basic 2D primitives
- The triangle’s 3D edges project to line segments in the image (thanks to planar perspective)



线段的表示

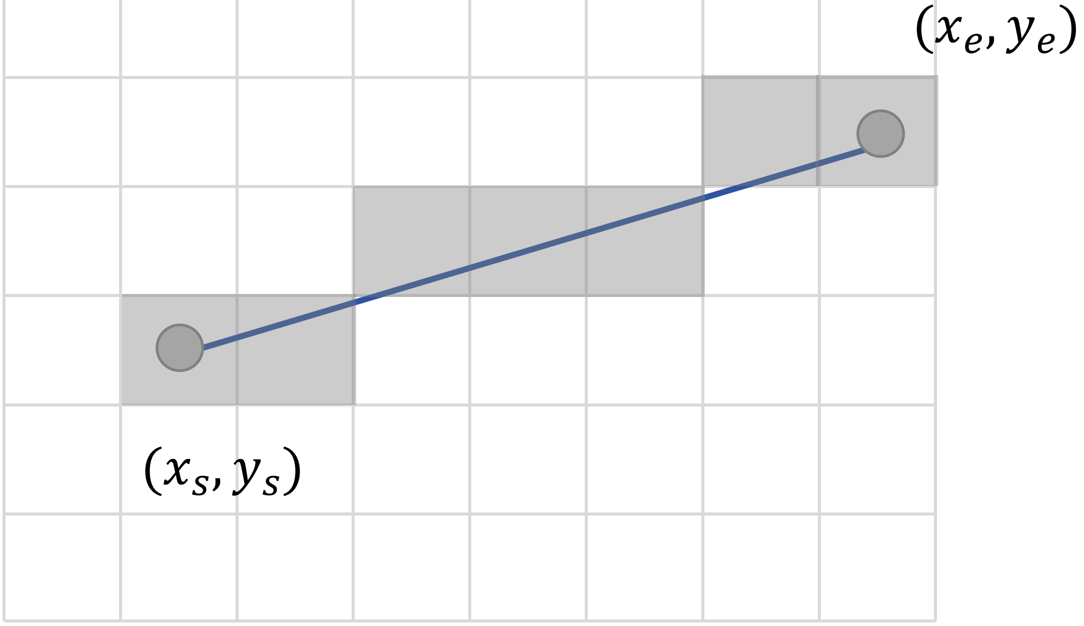
- **Line Segment**: start point (x_s, y_s) and end point (x_e, y_e)
- **Parametric Representation：**

$$\begin{cases} x = x_s + \Delta x \cdot t \\ y = y_s + \Delta y \cdot t \end{cases} \qquad 0 \leq t \leq 1$$



1. DDA Algorithm

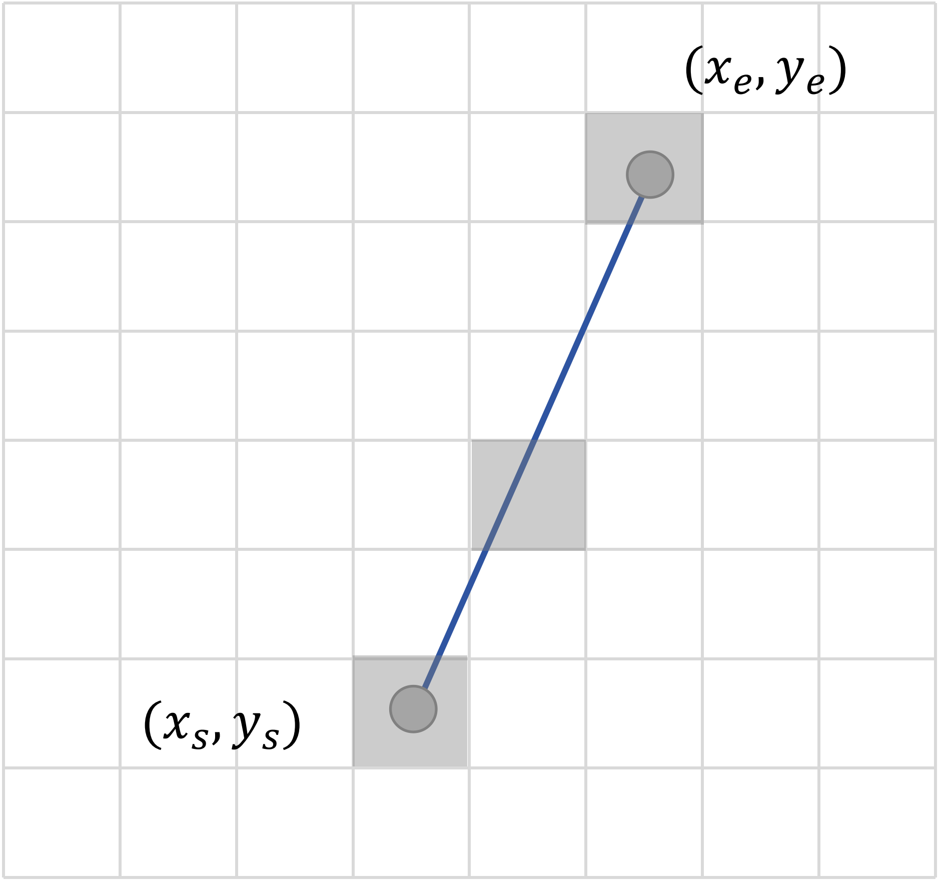
- Move to next pixel with Δt
- $\begin{cases} x = x_s + \Delta x \cdot \Delta t \\ y = y_s + \Delta y \cdot \Delta t \end{cases}$



沿x方向每次增加1

1. DDA Algorithm

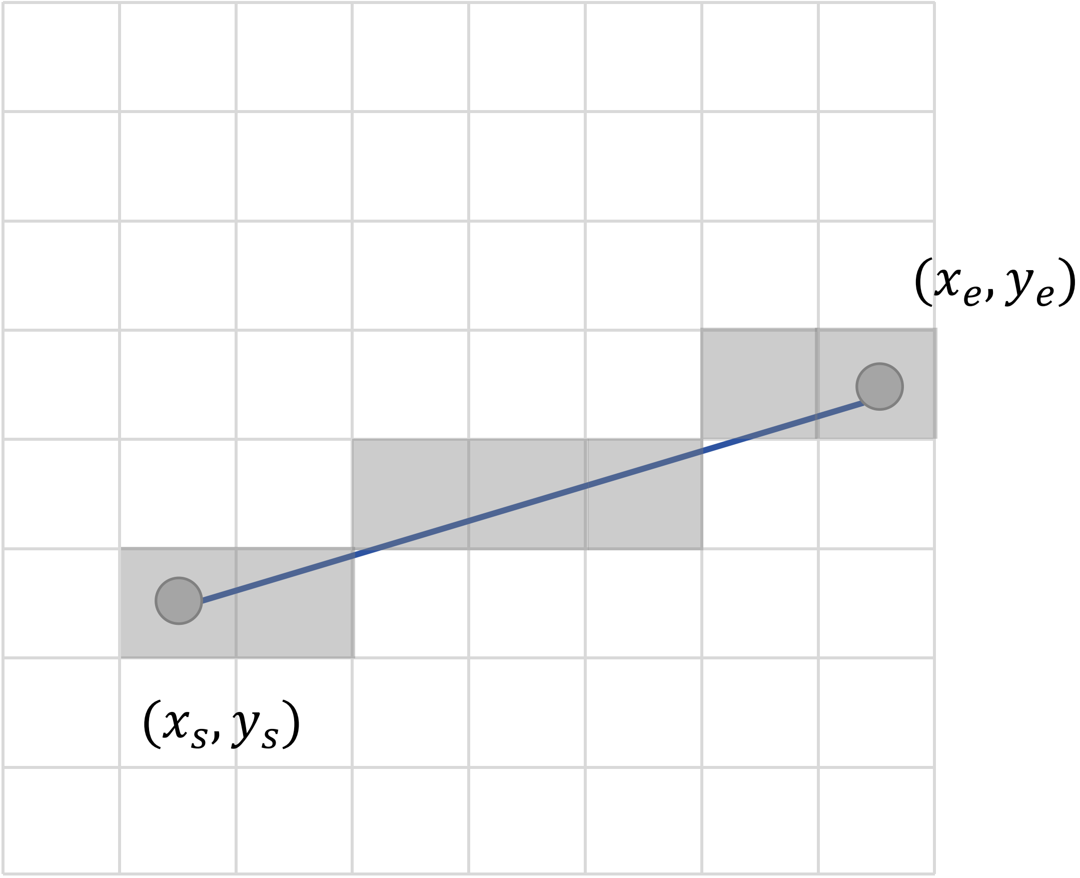
- What if $\Delta y \geq \Delta x \geq 0$
- Discontinuity problem!



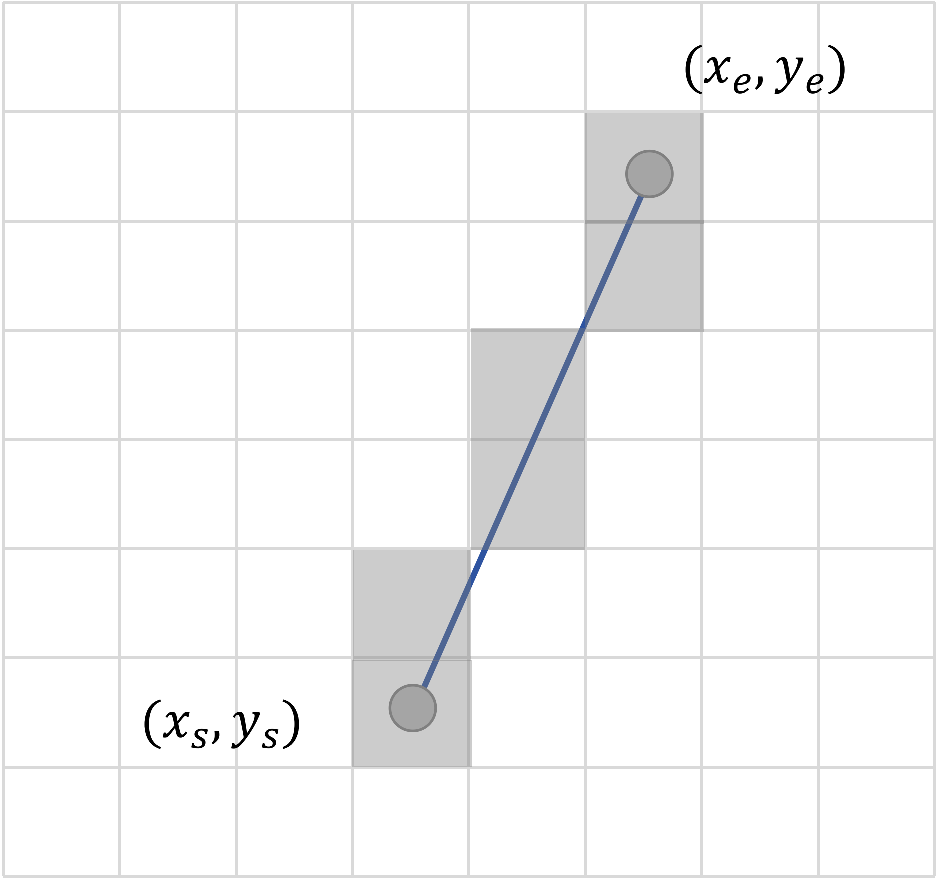
沿x方向每次增加1

1. DDA Algorithm

- Move along x or y axis



沿x方向每次增加1



沿y方向每次增加1

1. DDA Algorithm

DDA算法

- 利用Δt控制步长来计算线段上的点

$$\begin{cases} \boldsymbol{x = x_s + \Delta x \cdot \Delta t} \\ \boldsymbol{y = y_s + \Delta y \cdot \Delta t} \end{cases}$$

- 若直线斜率小于1($\Delta x \geq \Delta y \geq 0$)，$x$方向每次增加1，$\Delta t = 1/\Delta x$
- 若直线斜率大于1($\Delta y \geq \Delta x \geq 0$)，$y$方向每次增加1，$\Delta t = 1/\Delta y$
- 所以

$$\Delta t = min\{1/\Delta x, 1/\Delta y\}$$

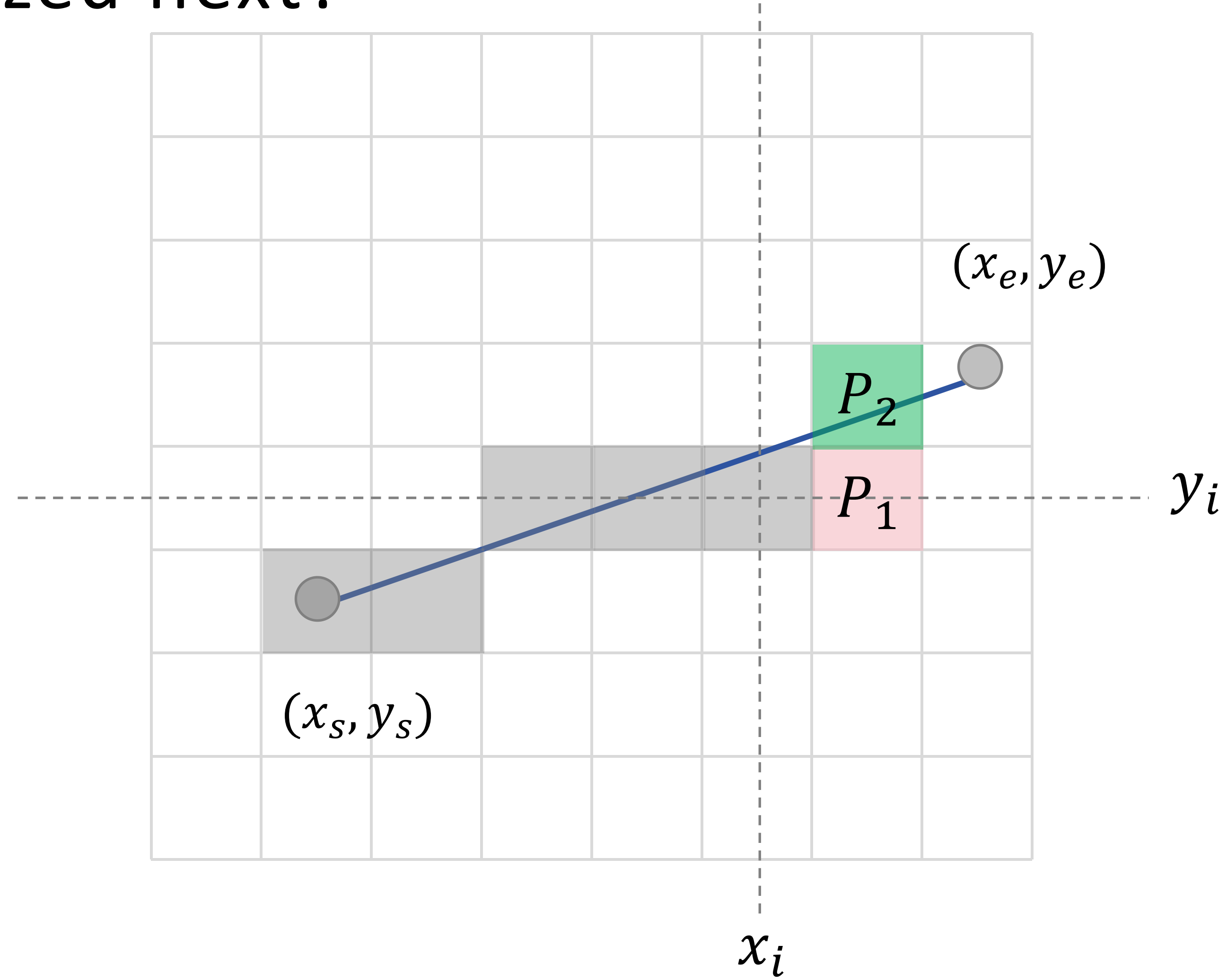
1. DDA Algorithm

- High computation cost!
  - Requires division operation,
  e.g. $\Delta t = 1/\Delta x$
  - **Two sum and round every time**

```
float dx=x1-x0;
float dy=y1-y1;

int steps=abs(y1-y0);
// 这里是取绘制点数最多的值
if(fabs(dx)>fabs(dy))
 steps=abs(x1-x0);

// 先初始化两个坐标为起点
float x=x0;
float y=y0;
// 然后定义两个x和y的增量变量
// 分别用两点的x、y的差除与需要绘制的点数来获得每绘制结束一个点后需要前进多少
float xinc=dx/steps;
float yinc=dy/steps;

// 绘制起点
drawPoint(round(x),round(y));
for(int i=0;i<steps;++i)
{
 // 注意这里需要放到本次循环开始
 // 因为第一个点已经绘制了
 x+=xinc;
 y+=yinc;

 drawPoint(round(x),round(y));
}
```

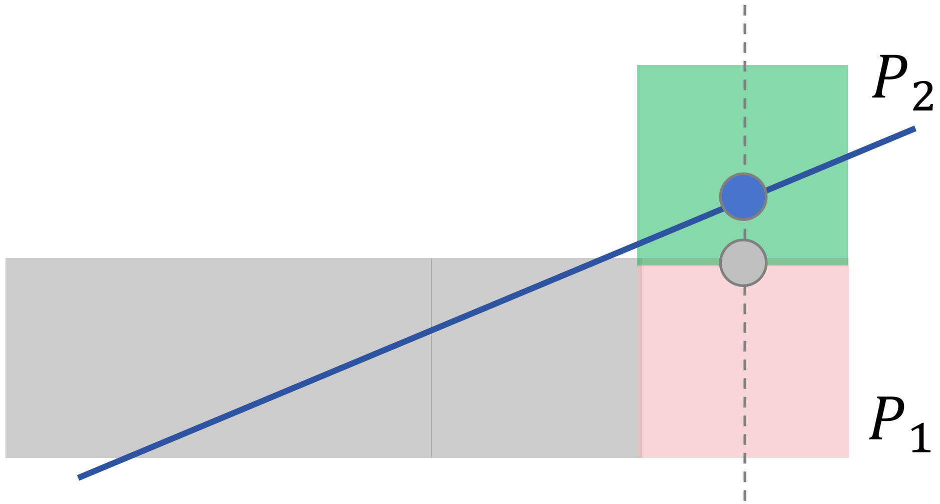
2. Midpoint Line Algorithm

- Which pixel to be rasterized next?
  - P_1 or P_2
- Which is more close?



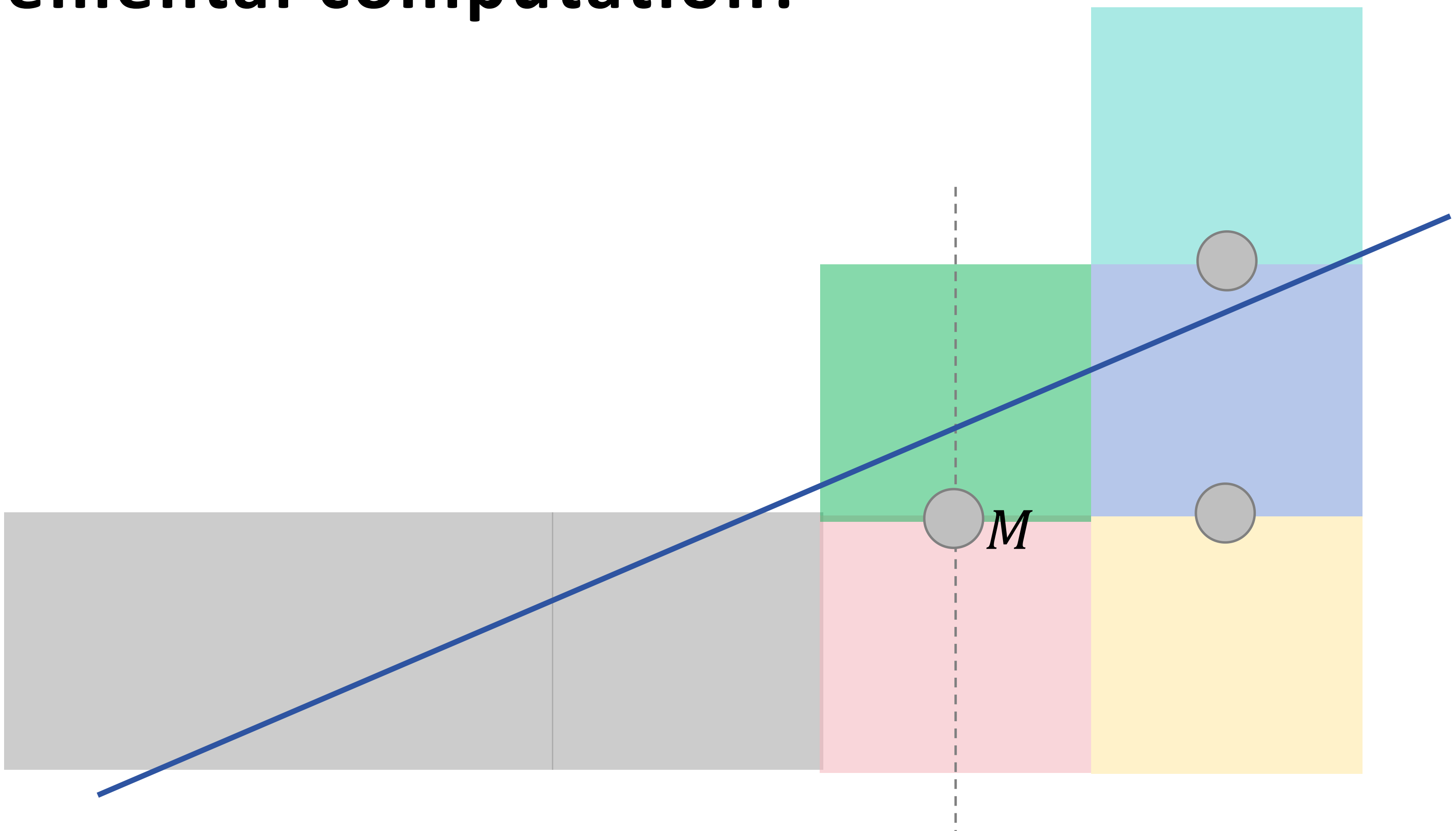
2. Midpoint Line Algorithm

- Which is more close?
  - Compare the midpoint of P_1P_2 and the interaction with target line segment
  - Use line representation: $\mathrm{ax} + \mathrm{by} + \mathrm{z} = 0$
    - $d = F(M) = a(x_p + 1) + b(y_p + 0.5) + c$
  - **Compute d every time!**



3. Bresenham Algorithm

- Shall we use sum to obtain the next midpoint info?
- **Incremental computation!**



3. Bresenham Algorithm

Do the math...

- $d = F(M) = F(x_p + 1, \ y_p + 0.5)$
  - $d < 0$, select top-right pixel
    - Next: $d_1 = F(x_p + 2, y_p + 1.5) = d + a + b$
  - $d \geq 0$, select right pixel
    - Next: $d_2 = F(x_p + 2, y_p + 0.5) = d + a$
  - $a = y_0 - y_1, b = x_1 - x_0$

3. Bresenham Algorithm

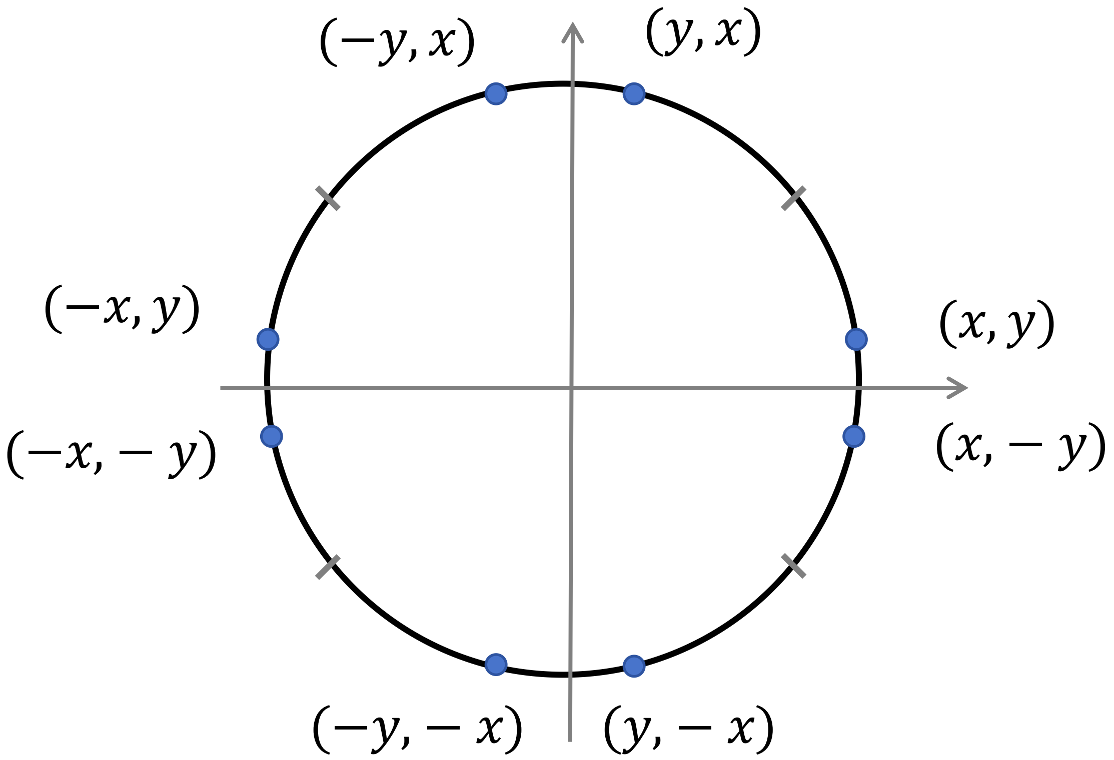
- It's actually the midpoint algorithm with incremental computation
- High efficiency!

```
/* mid PointLine */
void Midpoint Line (int x0,int y0,int x1, int y1,int color)
     {   int a, b, d1, d2, d, x, y;
    a=y0-y1, b=x1-x0, d=2*a+b;
    d1=2*a, d2=2* (a+b);
    x=x0, y=y0;
    drawpixel(x, y, color);
    while (x<x1)
    { if (d<0)       {x++, y++, d+=d2; }
     else       {x++, d+=d1;}
     drawpixel (x, y, color);
     } /* while */
 }
```

如何绘制一段圆弧?

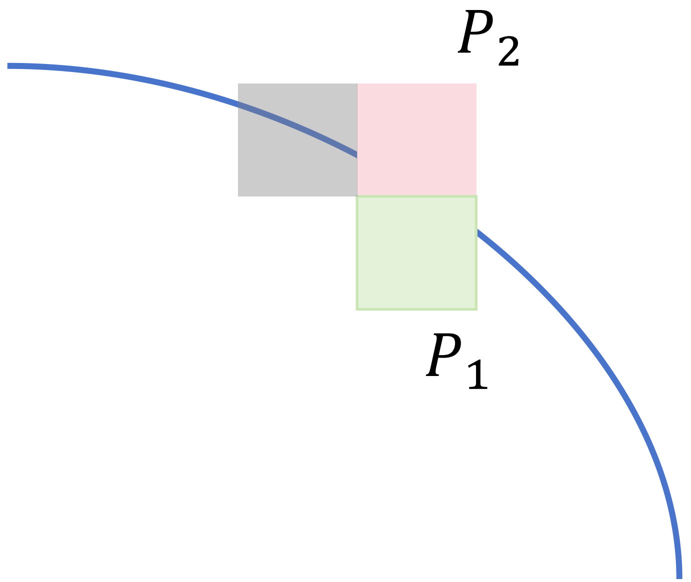
圆具有对称性

- 只需要计算1/8圆弧的坐标位置，并同时显示其他7个对称点



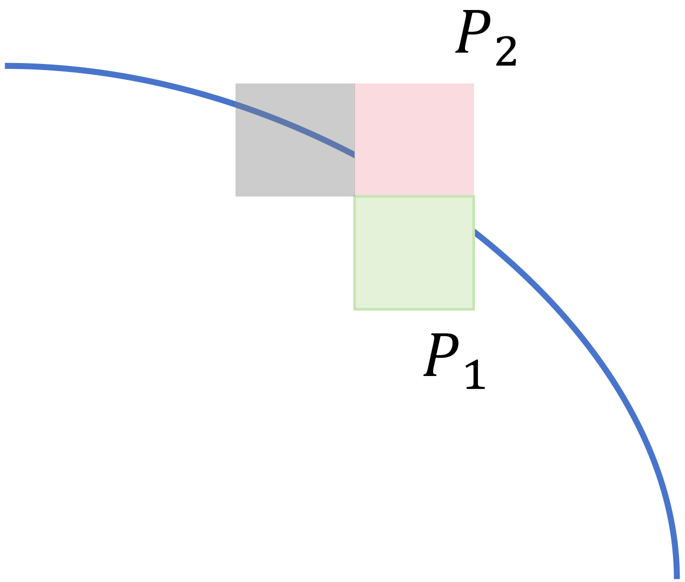
Bresenham Algorithm for Circle

- Again, we are now at P
- Next we need to select from P_1 or P_2
  - Which one is close to the circle?



Bresenham Algorithm for Circle

- Use circle representation: $D(P) = (x^2 + y^2) - R^2 = 0$
- We have $D(L_i) < 0,\ \ D(H_i) > 0$
- Compute $d_i = D(L_i) + D(H_i)$
  - $d_i < 0$, select P_2
  - $d_i > 0$, select P_1



Bresenham Algorithm for Circle

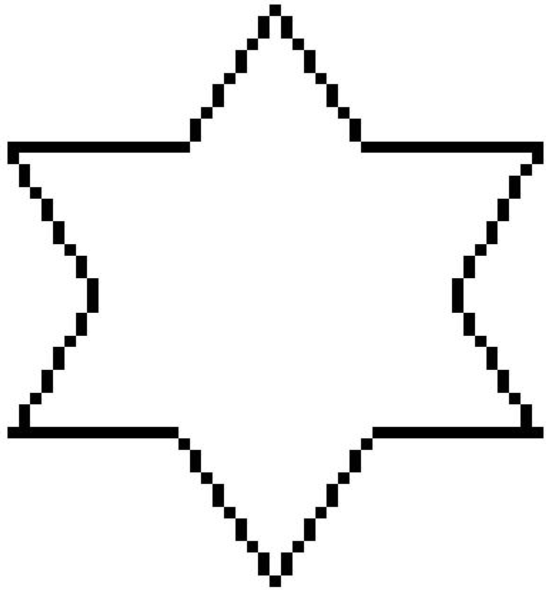
- Incremental computation:
  - $d_i < 0$, select P_2
    - Next: $d_{i+1} = d_i + 4x_i + 6$
  - $d \geq 0$, select P_1
    - Next: $d_{i+1} = d_i + 4(x_i - y_i) + 10$

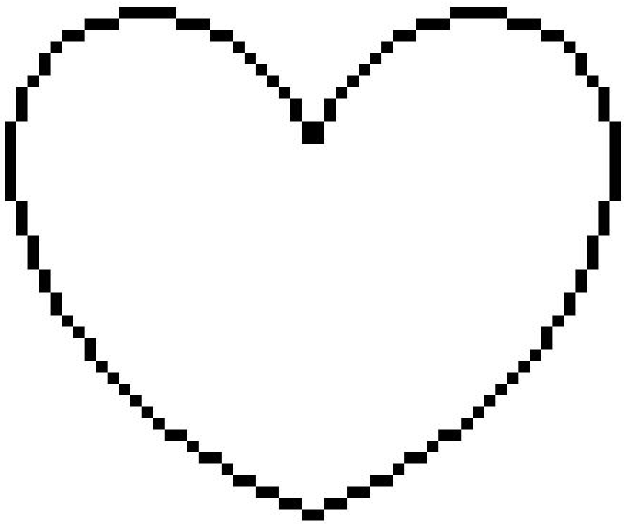
```
void bresenham_arc(int R){
    int x,y,d;
    x=0; y=R; d=3-2*R;
    while(x<y){
        gl_Point(x,y);
        if(d<0) d=d+4*x+6;
        else{
            d=d+4*(x-y)+10;
            y=y-1;
        }
        x=x+1;
    }
    if(x==y) gl_Point(x,y);
}

```

Now, Wireframes Done!

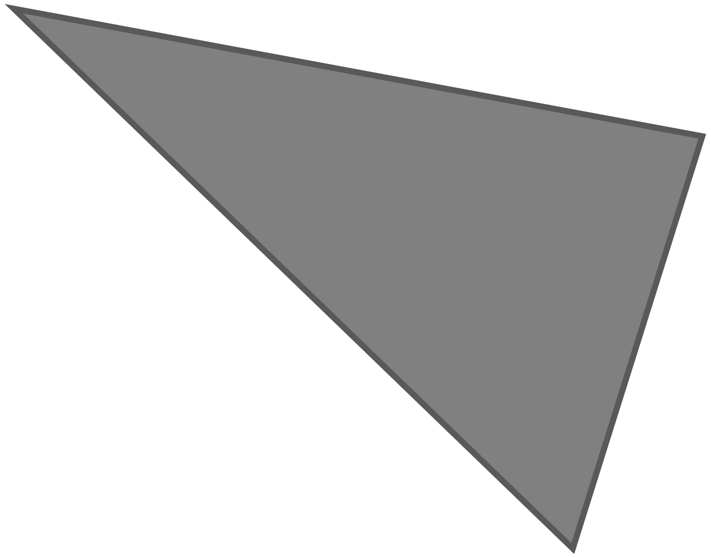
Now we can draw random wireframe shapes!

- Every shape can be approximated by lines and arcs from circles
- But how to fill the shapes with colors?



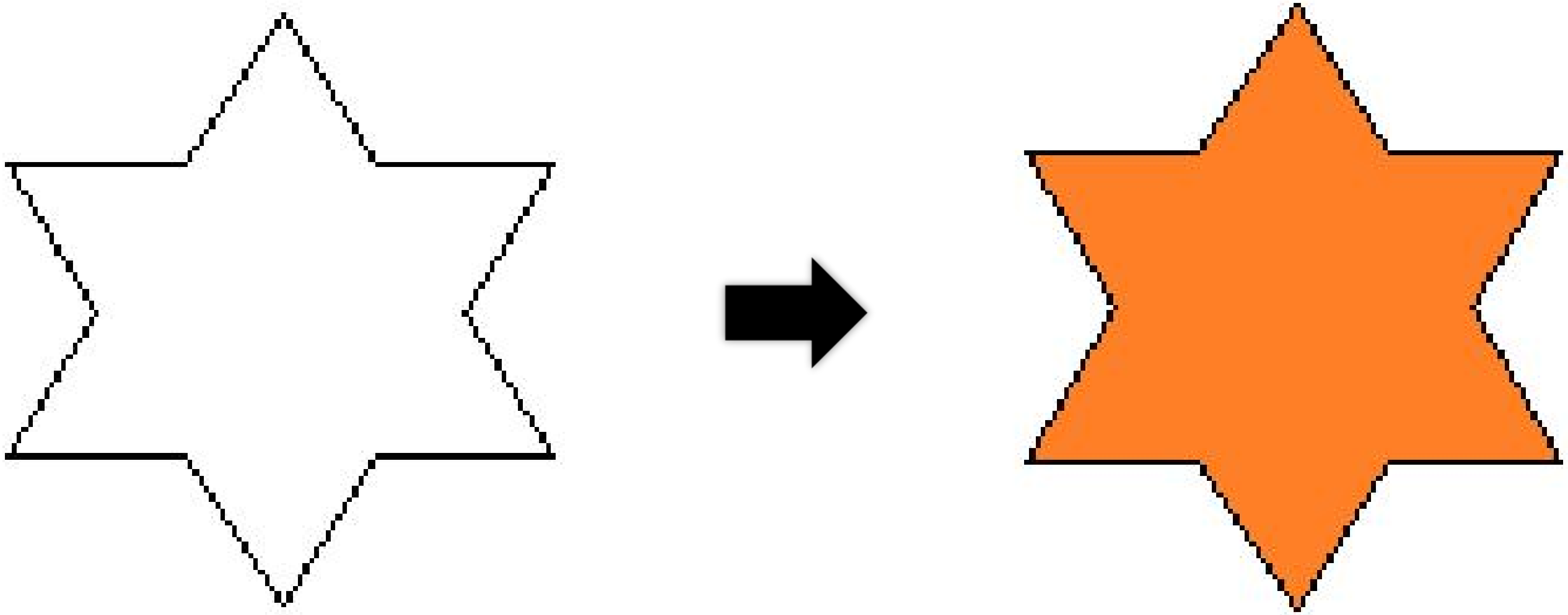


区域填充算法



如何填充二维区域?

- **区域填充**：即给出一个区域的边界，要求对边界范围内的所有像素单元赋予指定的颜色代码。区域填充中最常见的是多边形填色。

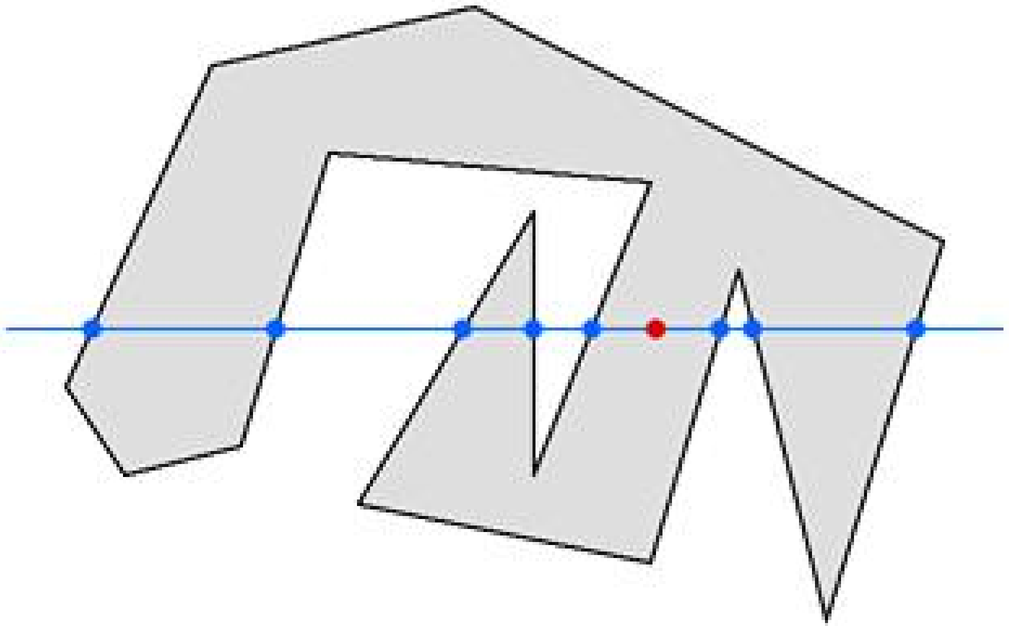


Brute Force Rasterizer

- Exhaustively test all the pixels
- Is this pixel inside or outside the shape?
  - Odd-even Rule
  - Non-zero Winding Number Rule
  - Triangle Testing

1. Odd-even Rule (奇偶规则)

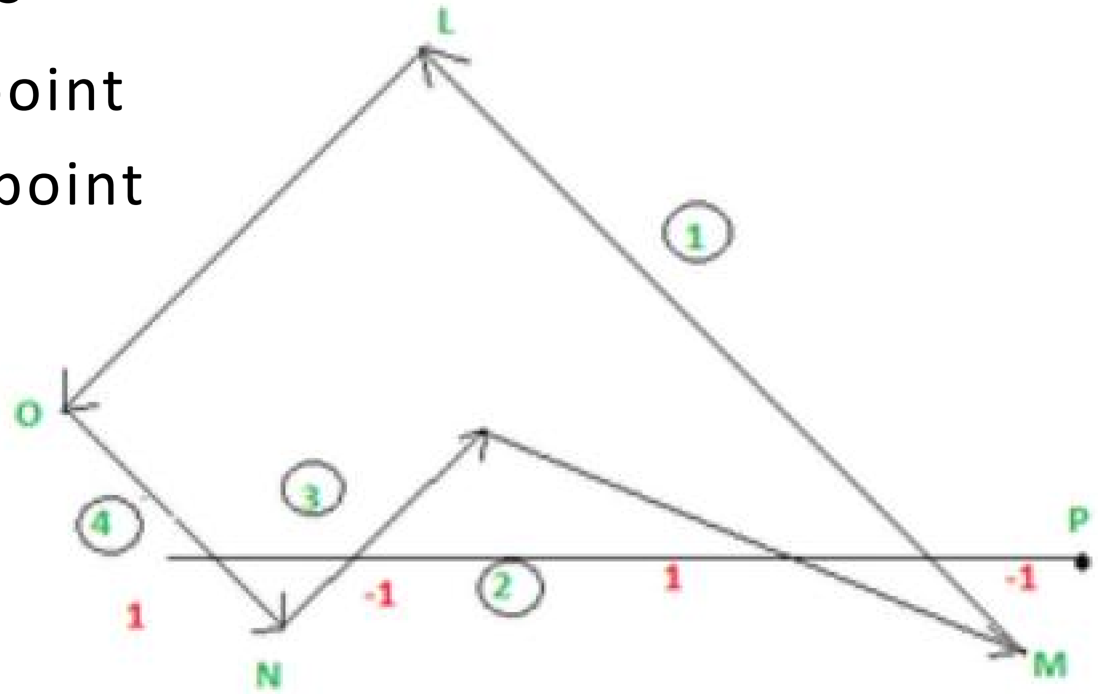
- Connecting point P and a know point outside the shape
- Line segment intersects with the shape boundary
- Count the number of intersections (**Crossing Number**)
  - Odd number, P is interior point
  - Even number, P is outside point



2. Non-zero Winding Number Rule

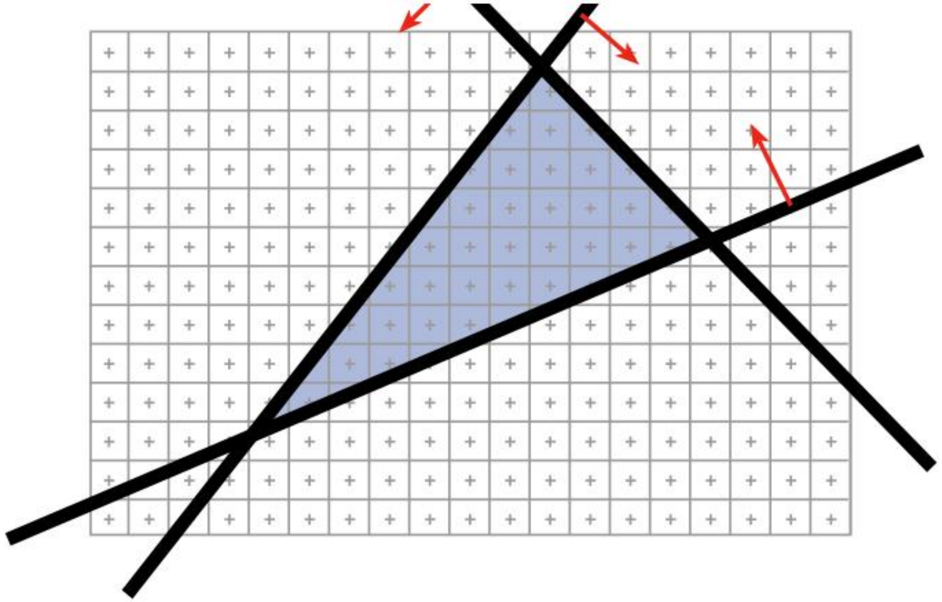
Winding number (环绕数)

- Find all the sides which crossed this line segment
- Two directions with different signs
    - Winding number = 0, P is outside point
    - Winding number ≠ 0, P is interior point



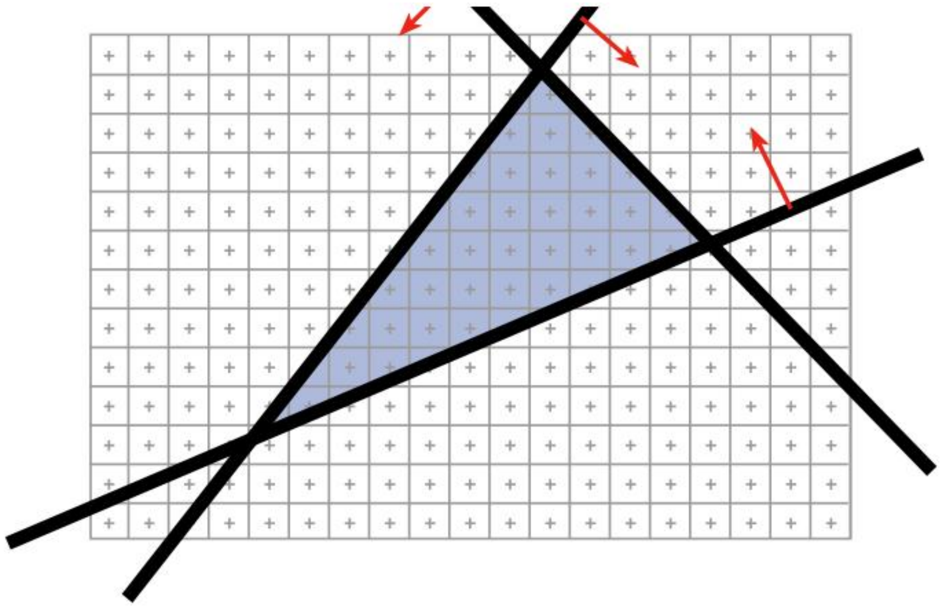
3. Triangle Testing

- The triangle's 3D edges project to line segments in the image (thanks to planar perspective)
- The interior of the triangle is the set of points that is inside all three halfspaces defined by these lines



3. Triangle Testing

- The triangle's 3D edges project to line segments in the image (thanks to planar perspective)
- The interior of the triangle is the set of points that is inside all three halfspaces defined by these lines

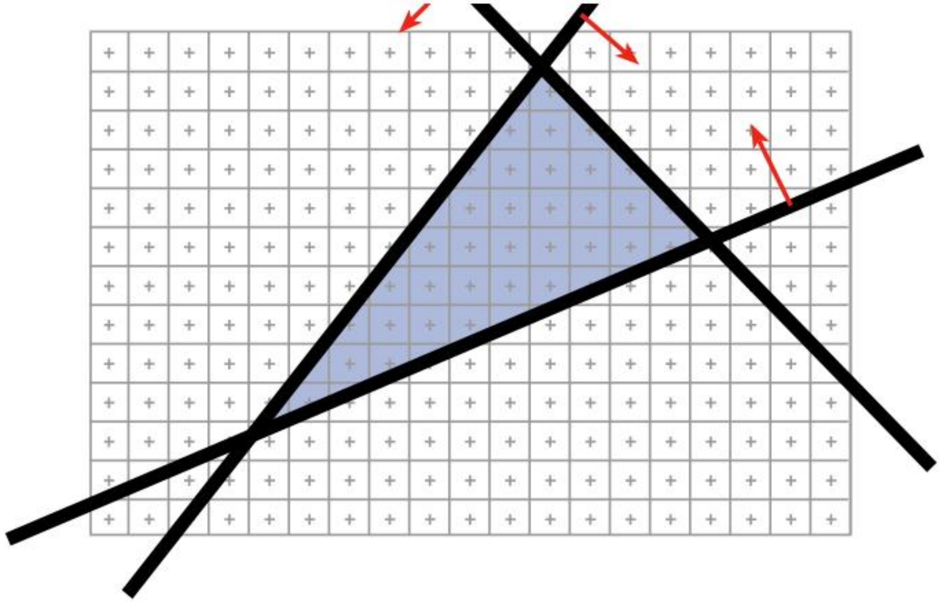


$$E_i(x, y) = a_i x + b_i y + c_i$$

$$(x, y) \text{ within triangle} \Leftrightarrow E_i(x, y) \geq 0, \quad \forall i = 1, 2, 3$$

3. Triangle Testing

- Compute E1, E2 , E3 coefficients from projected vertices
  - Called “triangle setup”, yields ai, bi, ci for i=1,2,3
- For each pixel (x, y)
  - Evaluate edge functions at pixel center
  - If all non-negative, pixel is in!

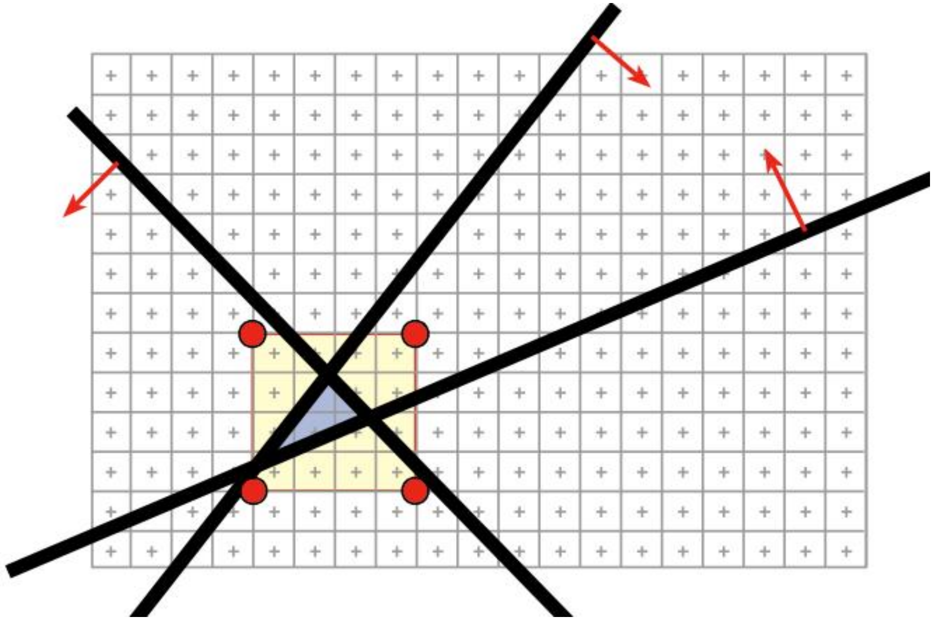


If the triangle is small, lots of useless computation if we really test all pixels

Too much wasted computation!

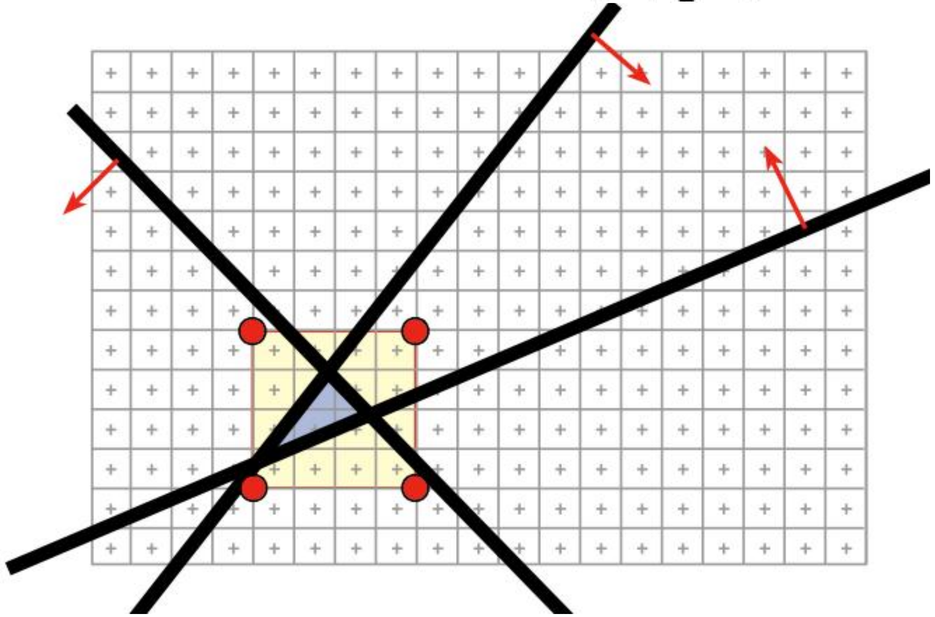
Improve the Rasterizer

- Improvement: Scan over only the pixels that overlap the screen bounding box of the triangle
- How do we get such a bounding box?
  - Xmin, Xmax, Ymin, Ymax of the projected triangle vertices



Improve the Rasterizer

```
For every triangle
   Compute projection for vertices, compute the Ei
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Evaluate edge functions Ei
      If all > 0
          Framebuffer[x,y ] = triangleColor
```

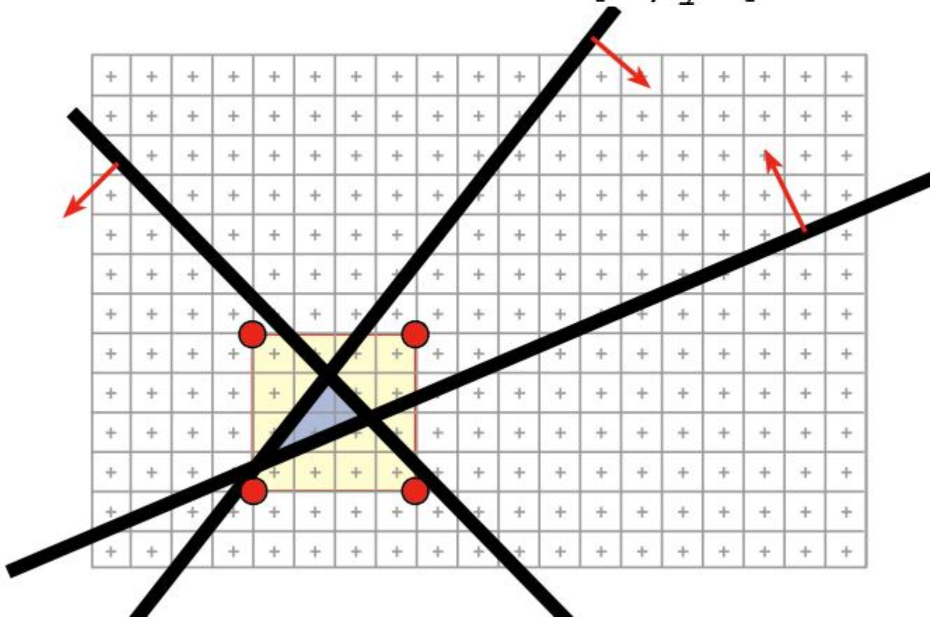


Bounding box clipping is easy, just clamp the coordinates to the screen rectangle

Can We Do Better?

```
For every triangle
    Compute projection for vertices, compute the Ei
    Compute bbox, clip bbox to screen limits
    For all pixels in bbox
        Evaluate edge functions Ei
        If all > 0
            Framebuffer[x,y ] = triangleColor
```

$$a_i x + b_i y + c_i$$



These are linear functions of the pixel coordinates (x,y), i.e., they only change by a constant amount when we step from x to x+1 (resp. y to y+1)

Incremental Edge Functions

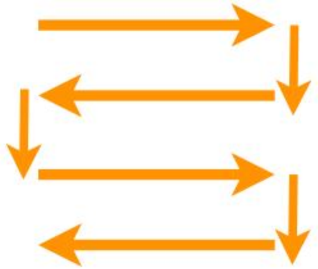
```
For every triangle
    ComputeProjection
    Compute bbox, clip bbox to screen limits
    For all scanlines y in bbox
        Evaluate all Ei's at (x0,y): Ei = aix0 + biy + ci
        For all pixels x in bbox
            If all Ei>0
                Framebuffer[x,y ] = triangleColor
            Increment line equations: Ei += ai
```

- We save ~ two multiplications and two additions per pixel when the triangle is large

Incremental Edge Functions

```
For every triangle
    ComputeProjection
    Compute bbox, clip bbox to screen limits
    For all scanlines y in bbox
        Evaluate all E_i's at (x0,y): E_i = a_i x0 + b_i y + c_i
        For all pixels x in bbox
            If all E_i>0
                Framebuffer[x,y ] = triangleColor
            Increment line equations: E_i += a_i
```

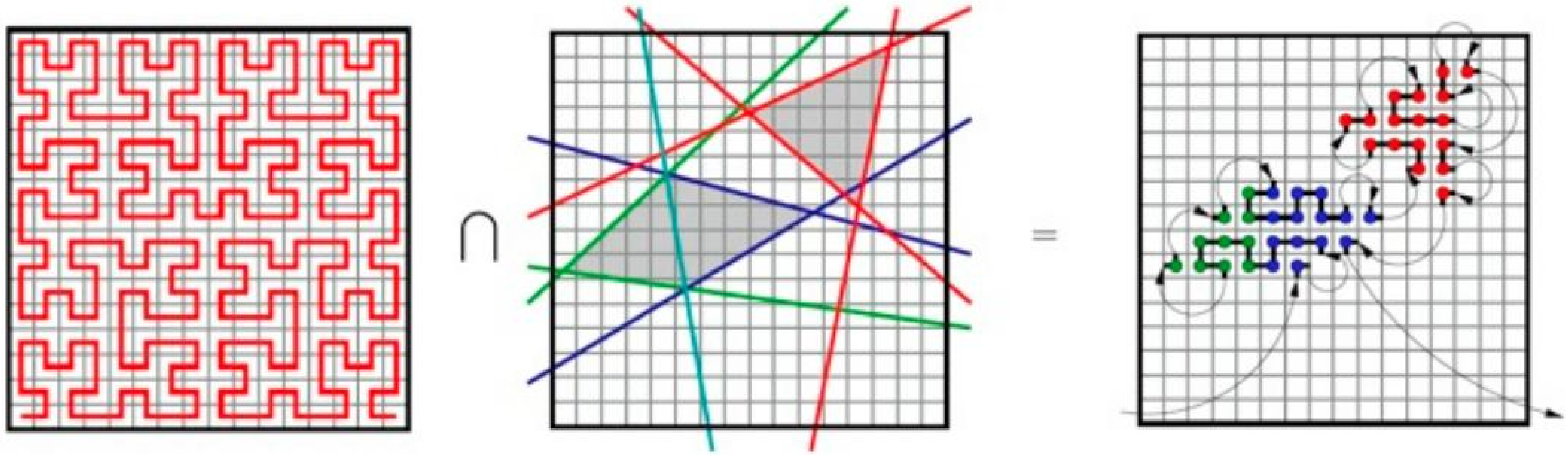
- We save ~ two multiplications and two additions per pixel when the triangle is large



Can also zig-zag to avoid reinitialization per scanline, just initialize once at x0, y0

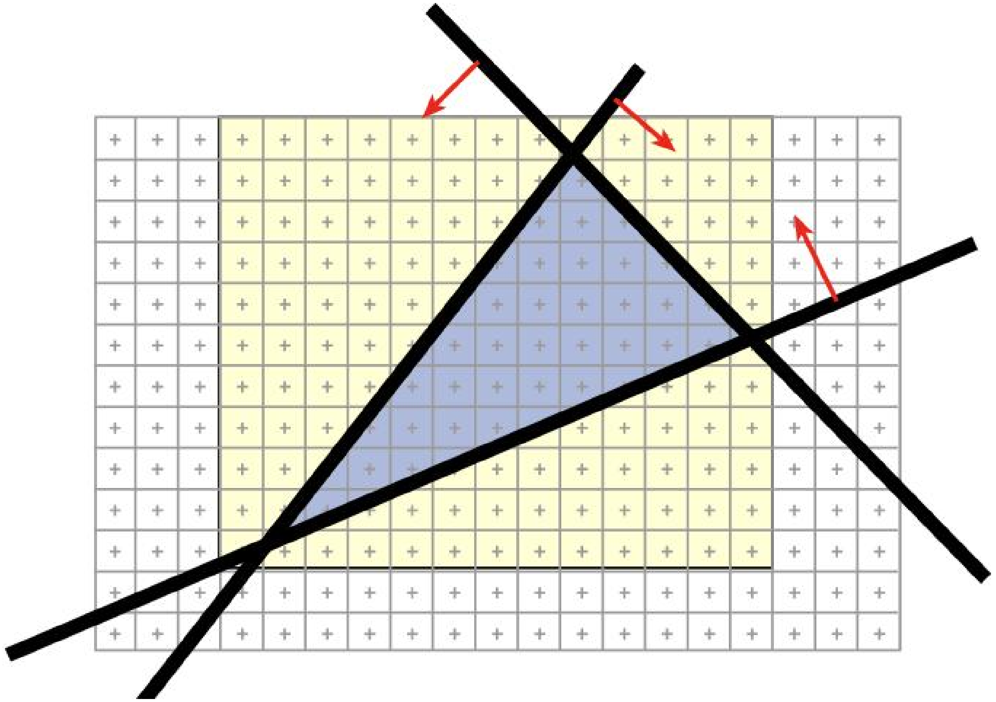
Incremental with Space Filling Curves

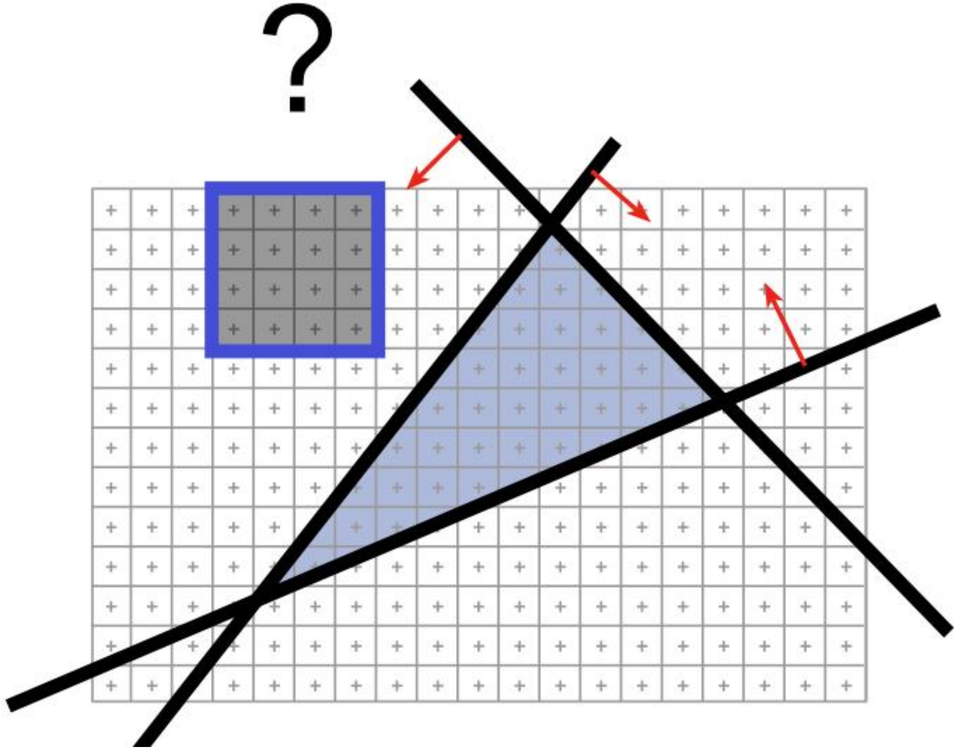
- Hilbert curve rasterizer by McCool, Wales and Moule.



Can We Do Even Better?

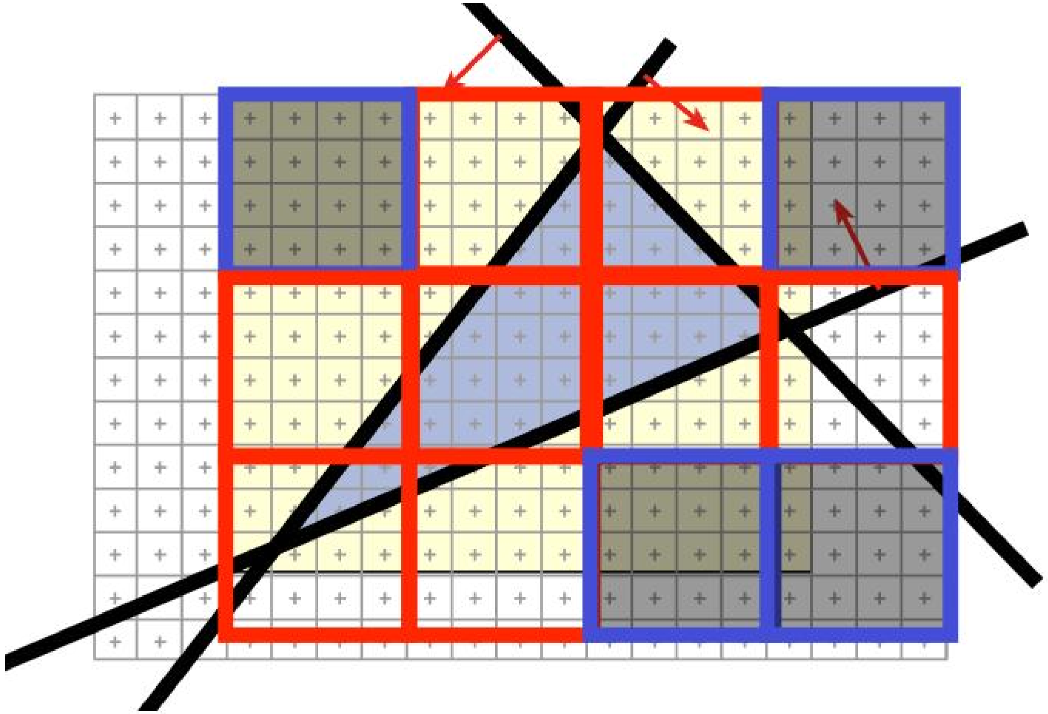
- We compute the line equation for many useless pixels
- What could we do?





Hierarchical Rasterization

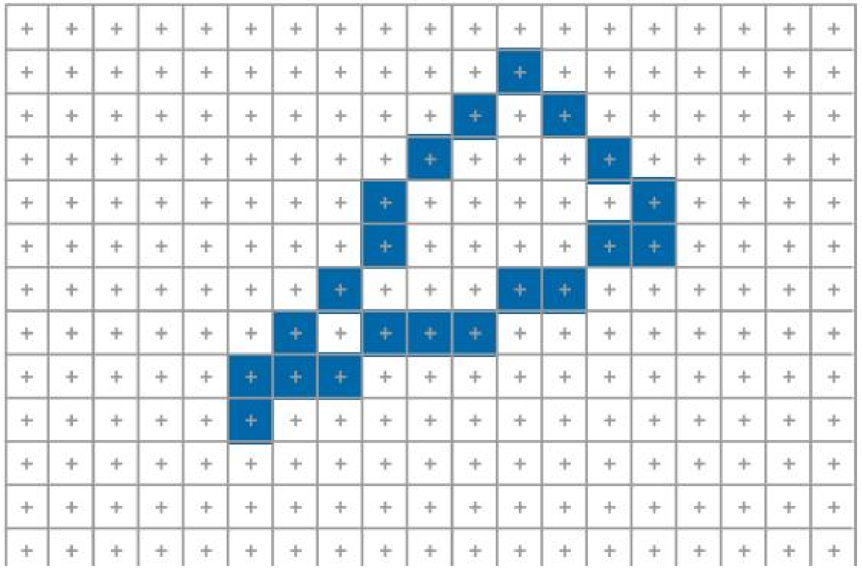
- Conservatively test blocks of pixels before going to per-pixel level (can skip large blocks at once)
- Usually two levels

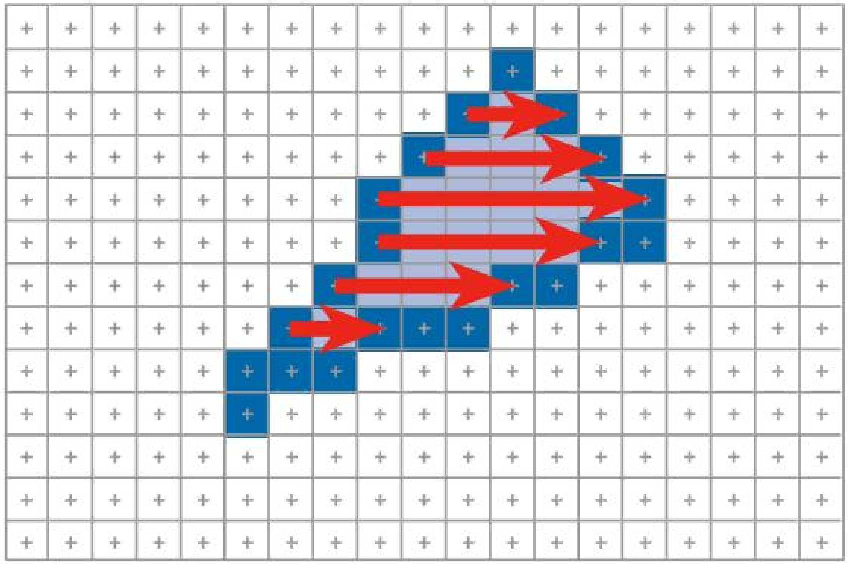


Conservative tests of axis-aligned blocks vs. edge functions are not very hard, thanks to linearity. See Akenine-Möller and Aila, Journal of Graphics Tools 10 (3), 2005.

Other Rasterization

- Compute the boundary pixels using line rasterization
- Fill the spans





More annoying to implement than edge functions

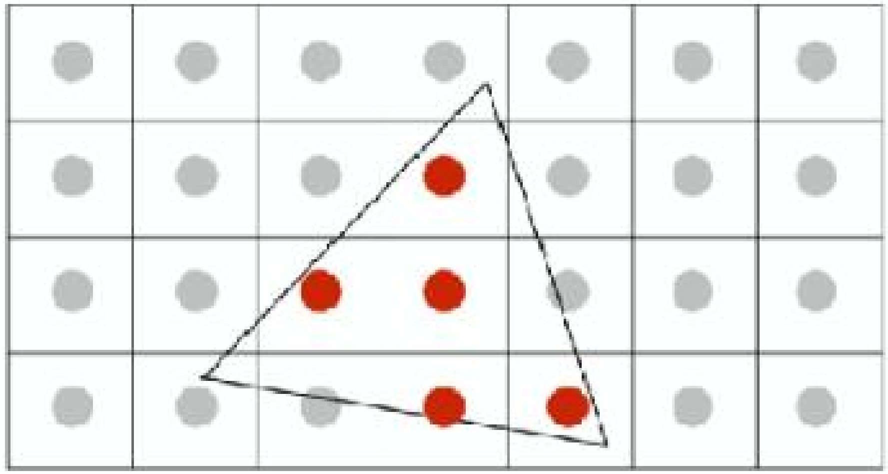
Not faster unless triangles are huge

Anti-aliasing

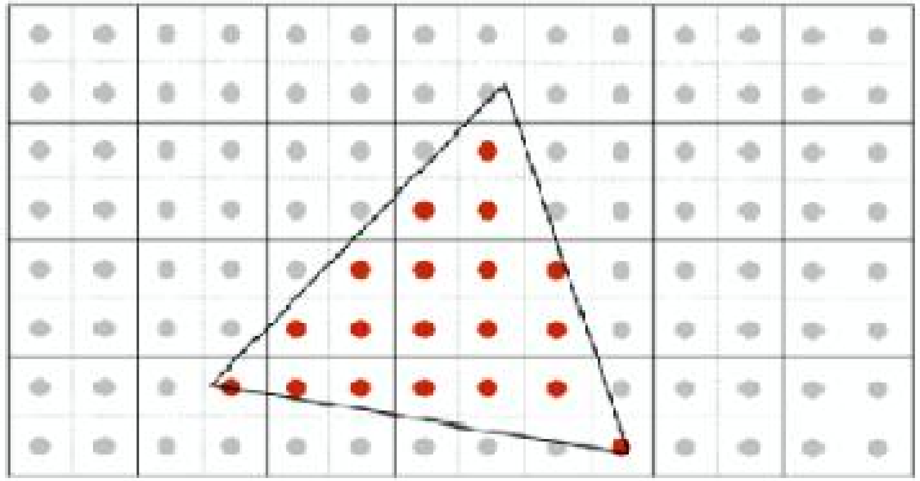
- Jagged edges or “jaggies” in a rasterized image
- The cause of anti-aliasing is Undersampling

Super-sampling

- Sampling at a higher resolution and display the image at a lower resolution



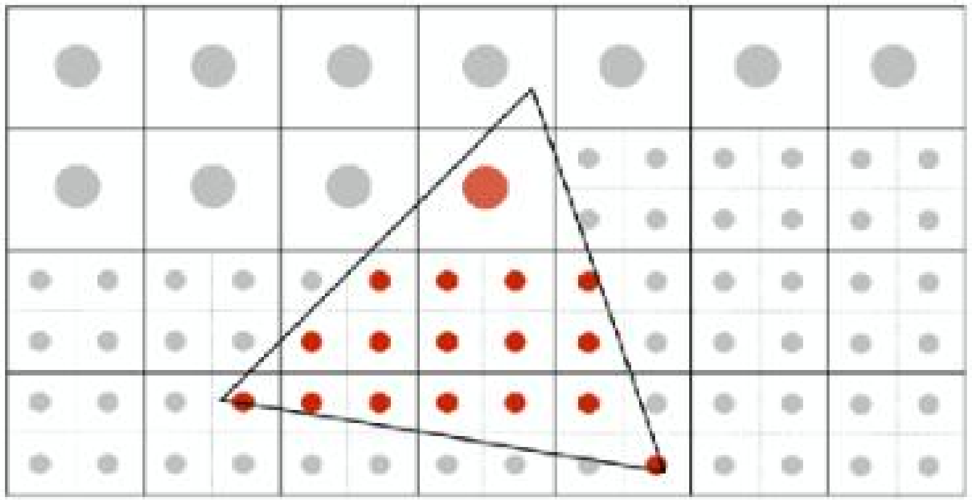
One sample per pixel



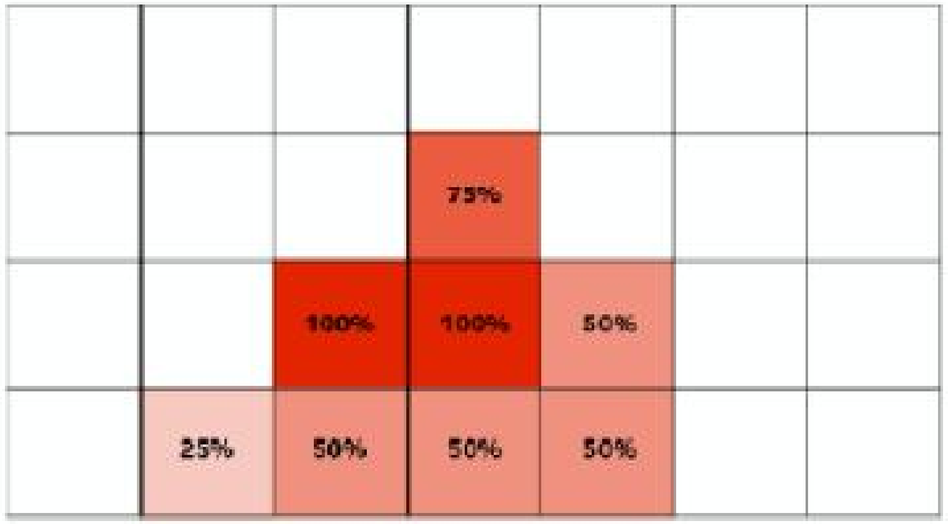
2x2 supersampling

Super-sampling

- Sampling at a higher resolution and display the image at a lower resolution
- The pixel intensity is the average of intensities of subpixels.

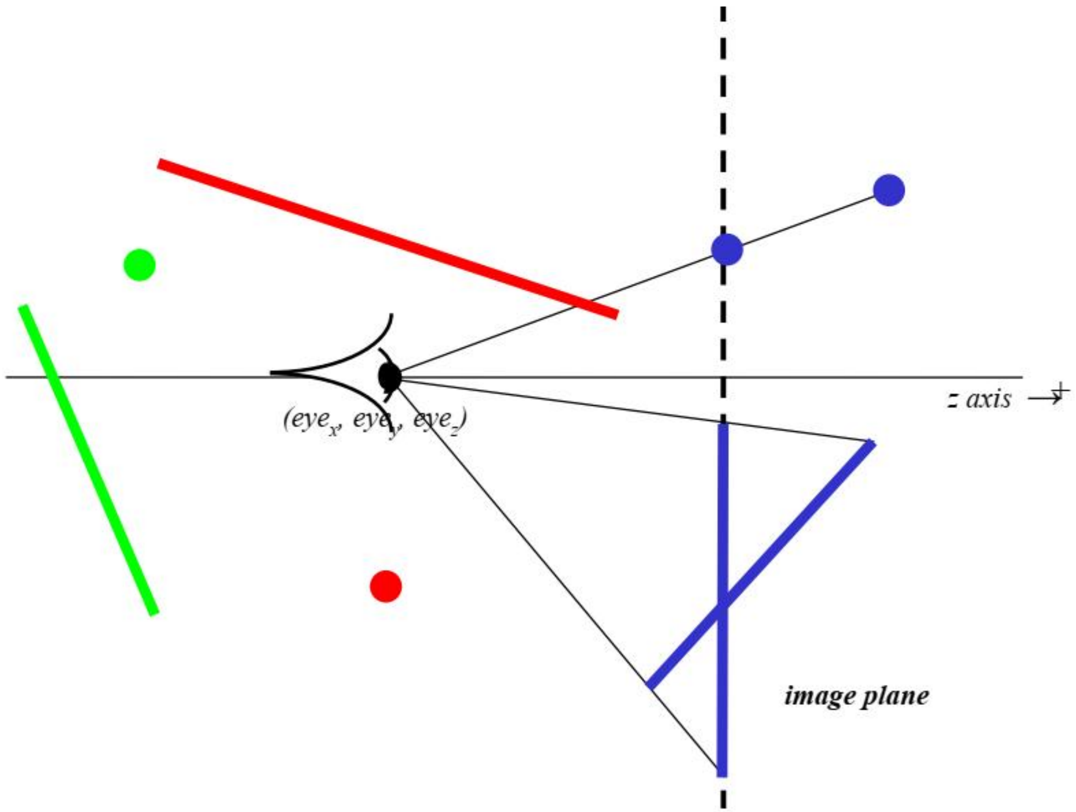


Averaging down



Now We Have Colored Shapes!

- Now we are able to rasterize simple shapes with colors
- What if 3D to 2D?



Some of them are invisible in particular viewpoint !

剪裁算法