# 计算机图形学

## 光栅化

陶钧
taoj23@mail.sysu.edu.cn
中山大学 数据科学与计算机学院
国家超级计算广州中心

# 课程概要

- 光栅化简介
- 线段光栅化
- 多边形光栅化
- 抗锯齿

# 光栅化简介

- 如何绘制一张图像？

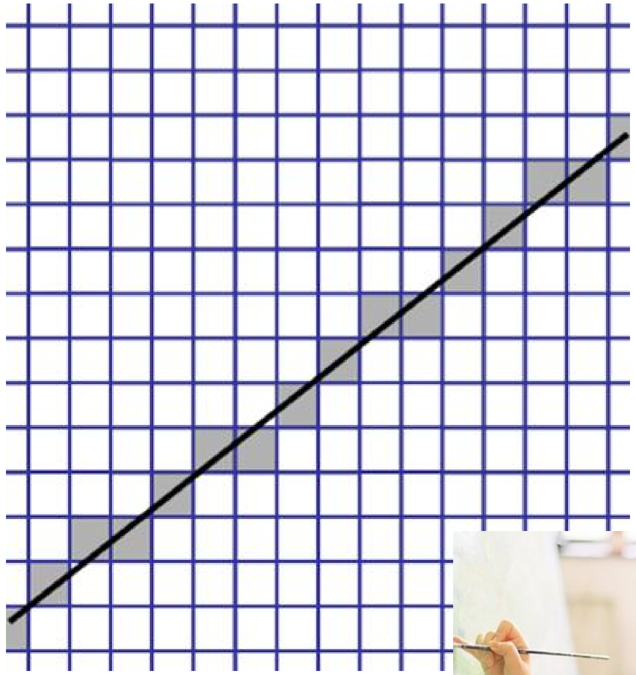
绘画

摄影

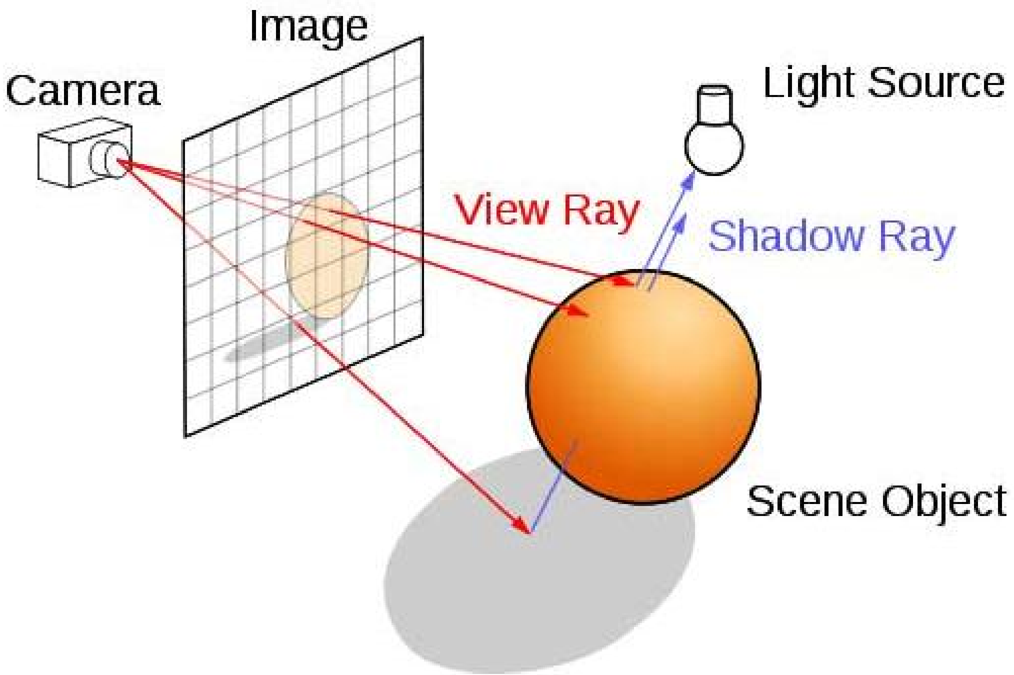
# 光栅化简介

- 如何绘制一张图像？
  - 翻译成计算机图形学的语言

摄影：光线追踪

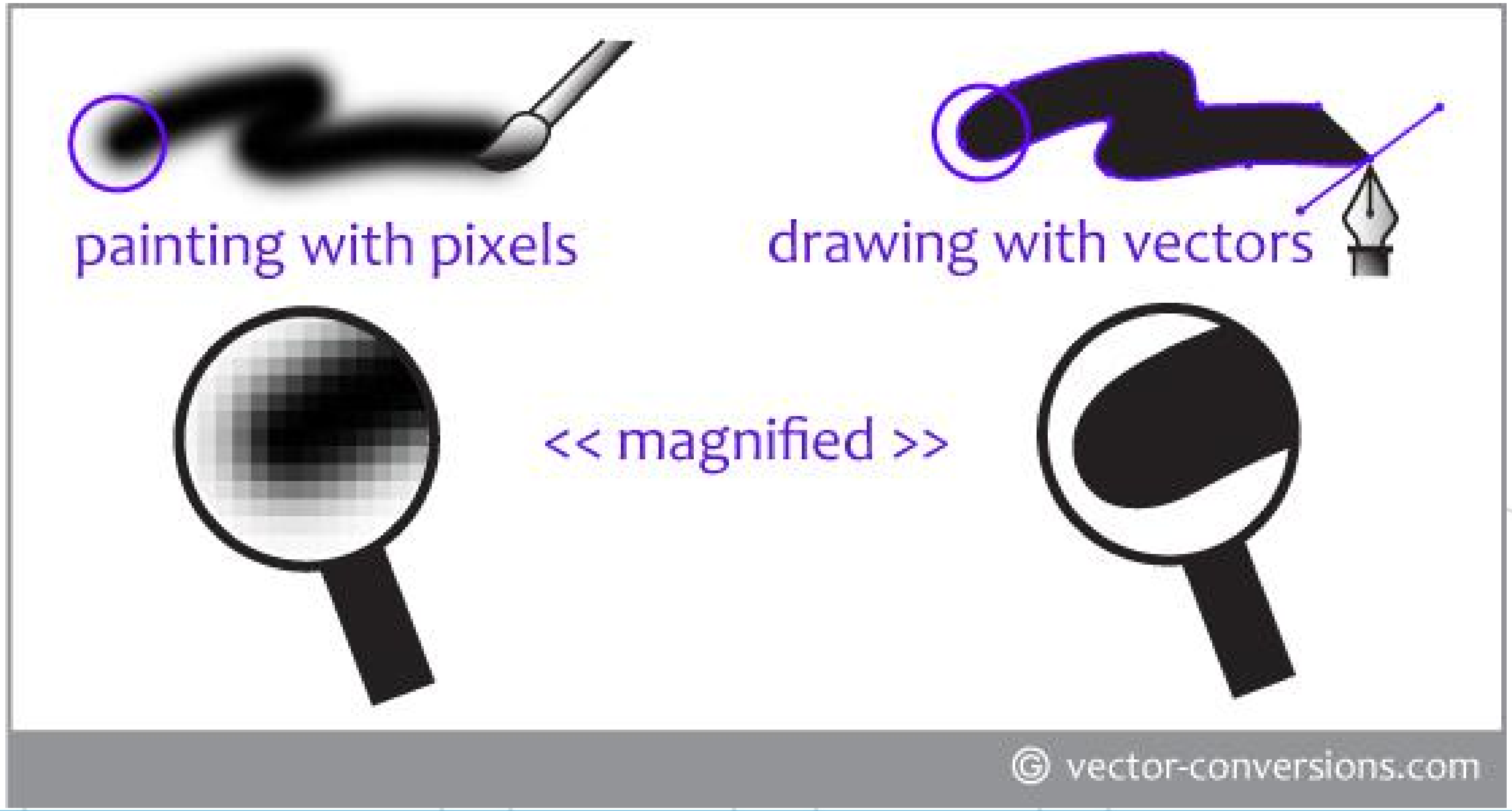


绘画：光栅化



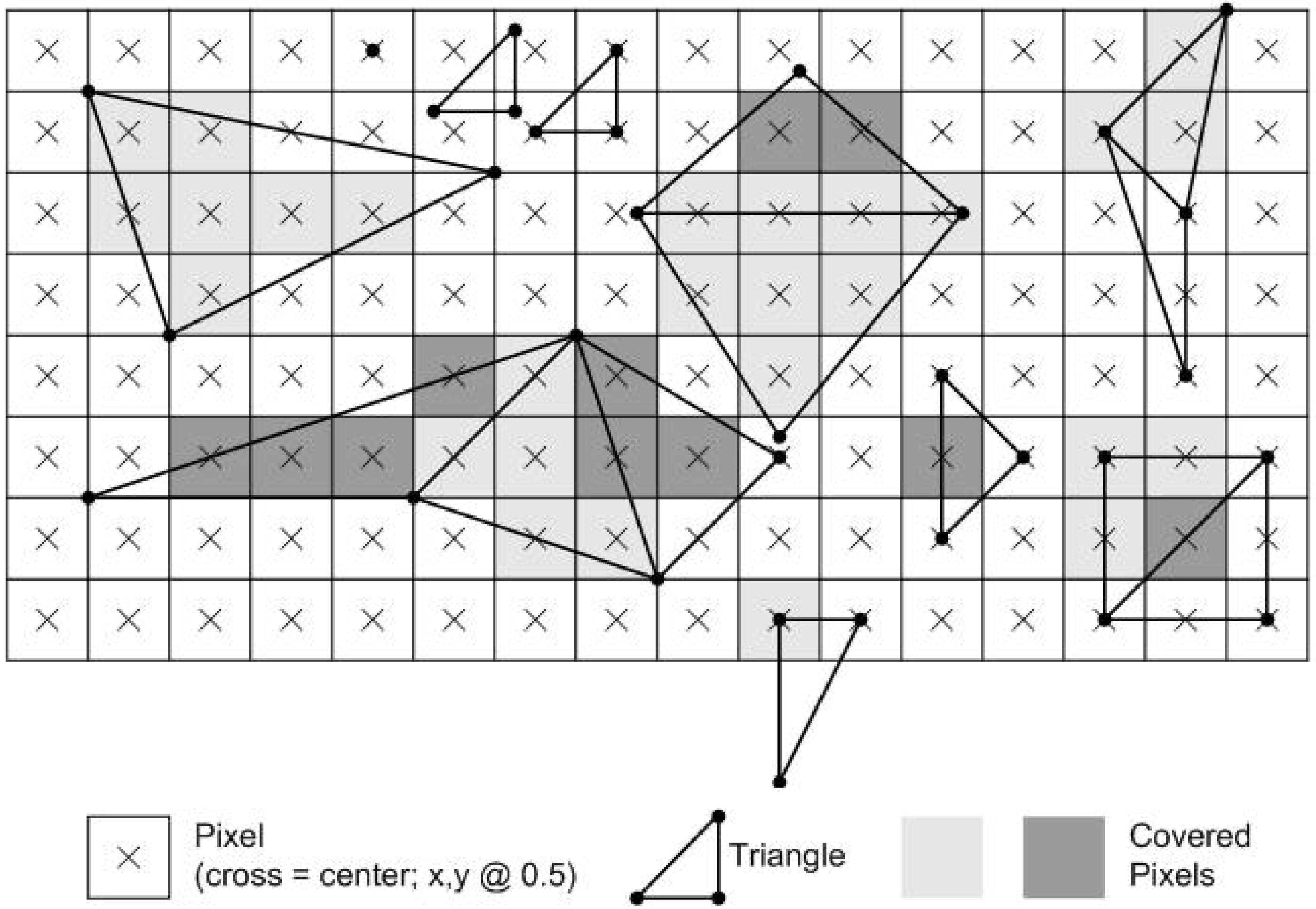

# 光栅化简介

- Raster graphics vs vector graphics
  - Raster graphics对像素进行操作
  - Vector graphics对几何元素进行操作
    - 但最终仍需转换成像素！



# 光栅化简介

- 计算机图形绘制过程
  - 投影：将顶点从空间映射到平面
  - 光栅化：将几何图元从离散化到像素
    - 决定每个图元对应哪些像素
    - 决定每个像素的颜色
    - 光栅化由一系列高效的scan conversion算法在硬件上实现

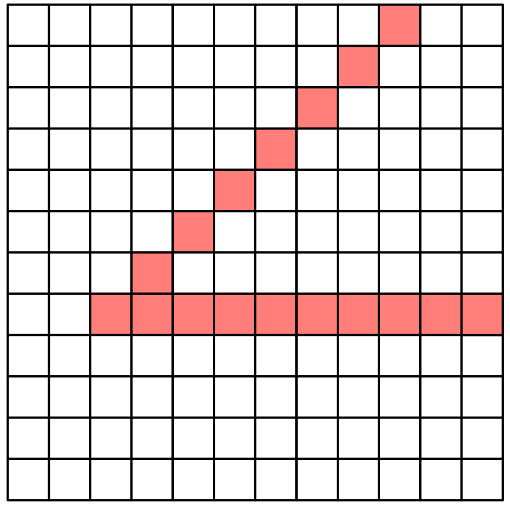


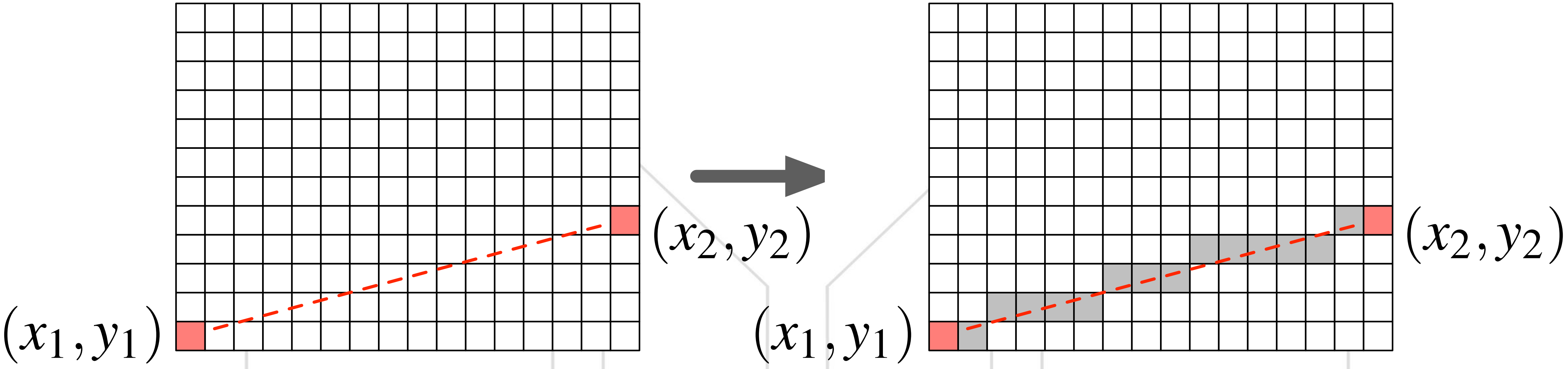
# 课程概要

- 光栅化简介
- 线段光栅化
- 多边形光栅化
- 抗锯齿

# 线段光栅化

- 绘制目标：使用像素绘制两点$(x_1, y_1)$与$(x_2, y_2)$之间的线段
  - 选中像素点尽可能靠近理想线段
  - 选中像素序列看起来尽可能“直”
  - 沿线亮度一致（亮度会随斜率改变！）
  - 起始点应当在选中像素序列中
  - 绘制应尽可能快



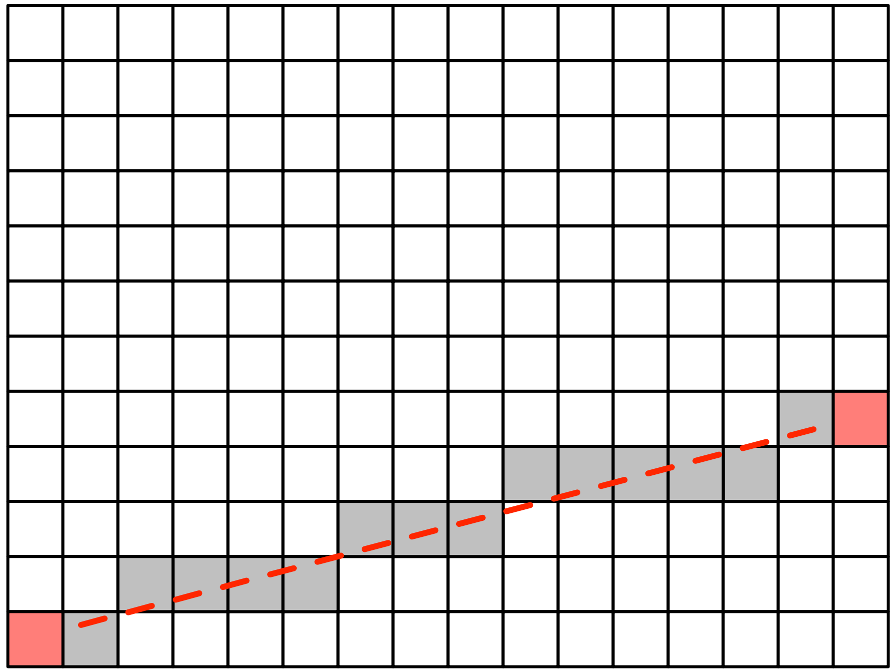


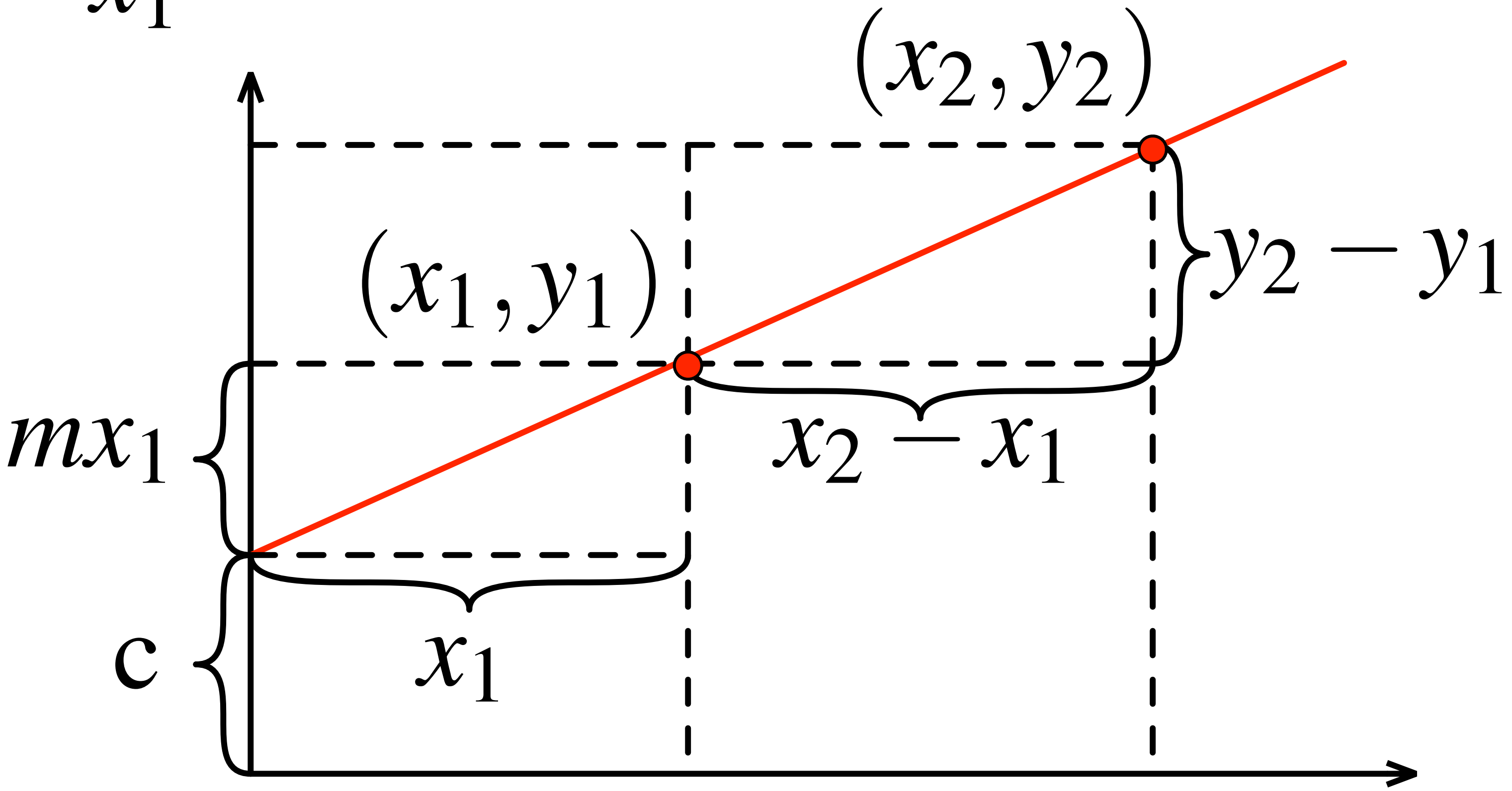
# 线段光栅化

- 思路一：将直线表示为解析式

$$y = mx + c$$

$$= \frac{y_2 - y_1}{x_2 - x_1} x + \left(y_1 - \frac{y_2 - y_1}{x_2 - x_1} \cdot x_1\right)$$



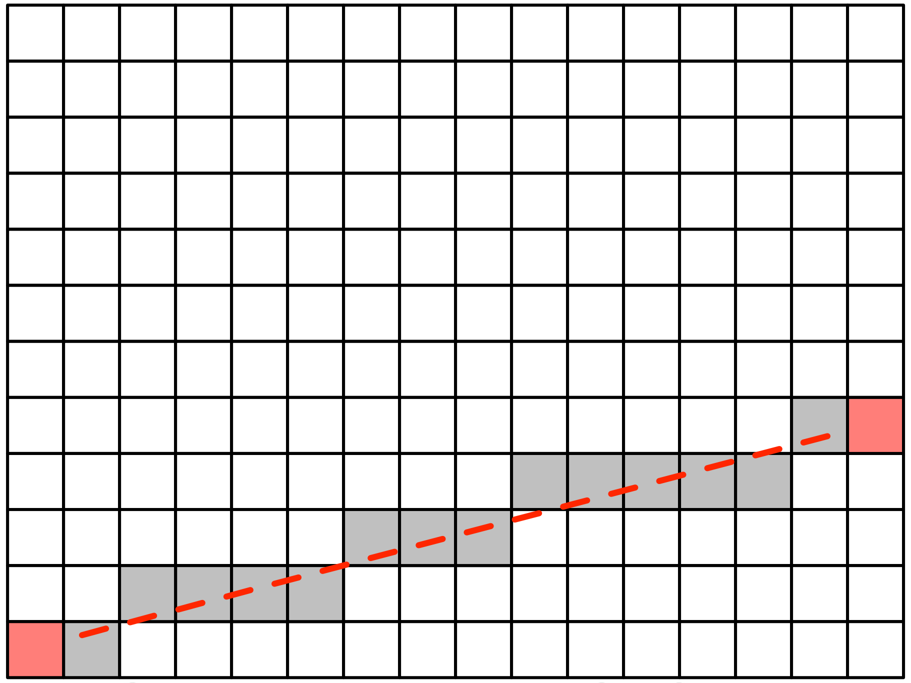


## 思路一：将直线表示为解析式

– 编程实现：Digital Differential Analyzer (DDA)

$$y = mx + c$$

$$= \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1$$



1. 直接代入解析式：

$$\text{for } x = x_1 \cdots x_2$$

$$y = mx + c$$

2. 自增：

$$x = x_1, y = y_1$$

$$\Delta x = 1, \Delta y = m$$

需要取整！

# 线段光栅化

## 思路一：将直线表示为解析式

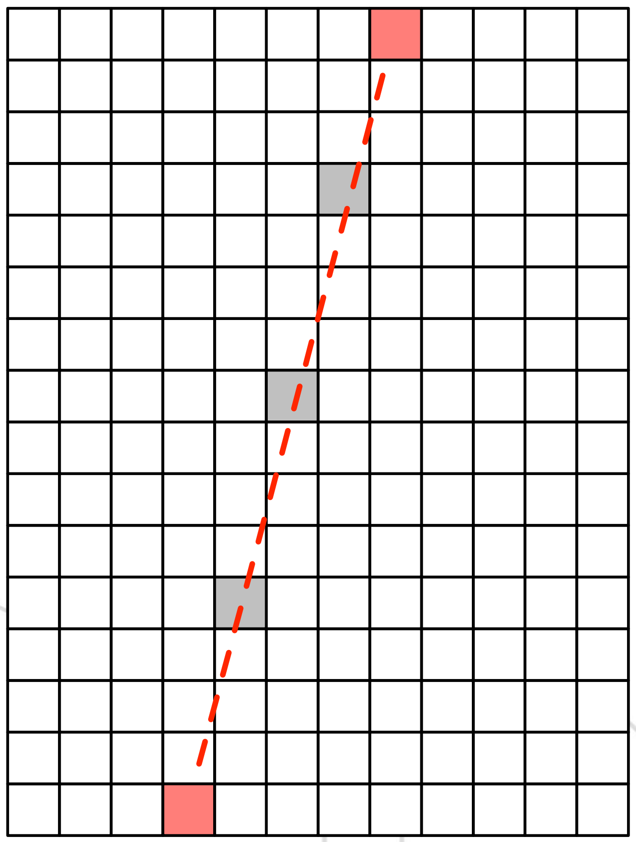
– 编程实现：Digital Differential Analyzer (DDA)

- 固定$\Delta x = 1$存在问题:

$$\Delta x < \Delta y$$

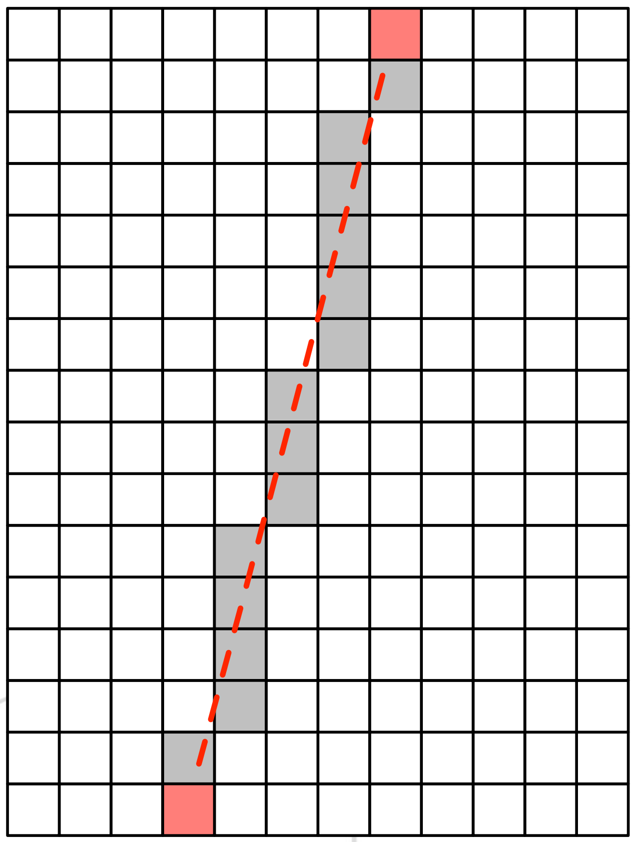
$$\Delta x = 1$$

$$\Delta y = 15/4$$



$$\Delta x = 4/15$$

$$\Delta y = 1$$



## 思路一：将直线表示为解析式

– 编程实现：Digital Differential Analyzer (DDA)

• 以下代码是否还有改进空间？

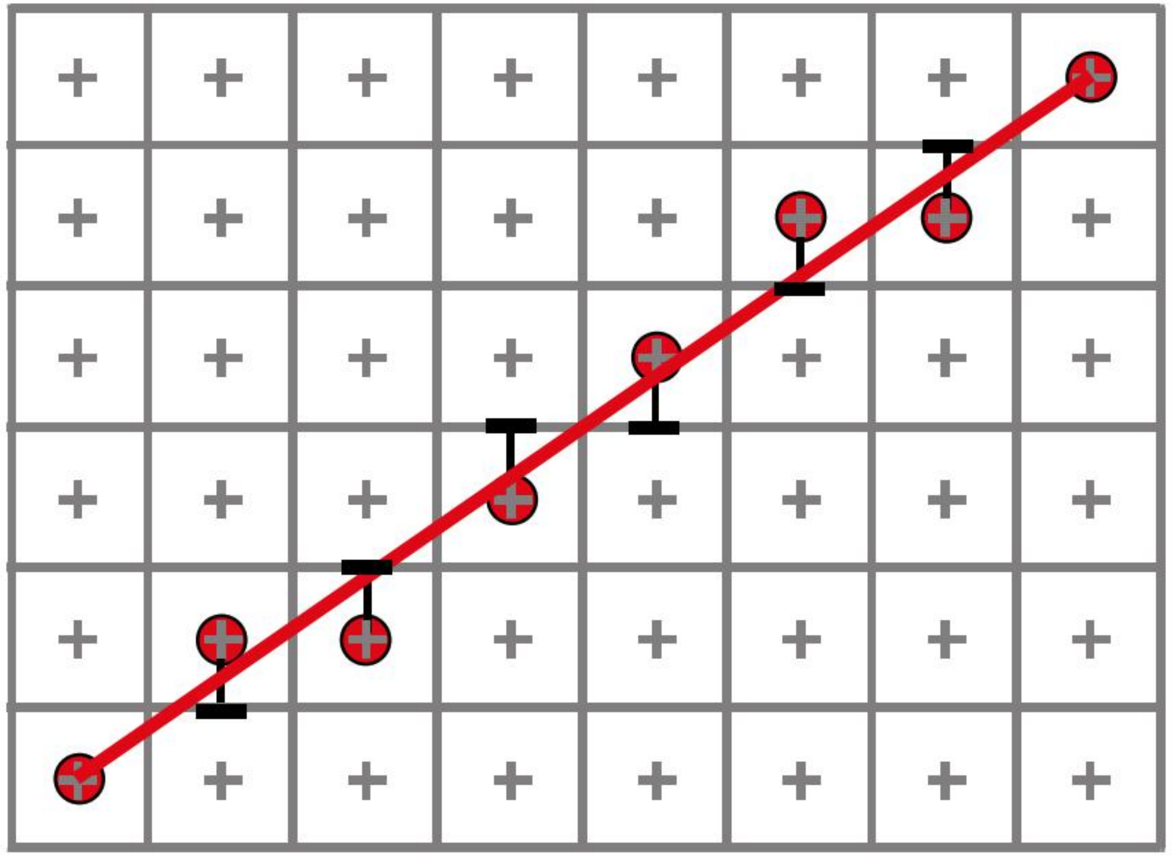
```
void line_DDA(int x1, int y1, int x2, int y2){
    int dx = x2-x1, dy = y2-y1, steps;
    float delta_x, delta_y, x=x1, y=y1;

    step = (abs(dx)>abs(dy))?(abs(dx)):(abs(dy));
    delta_x = dx/(float)steps;
    delta_y = dy/(float)steps;

    set_pixel(round(x), round(y));
    for(int i=0; i<steps; ++i){
        x += delta_x;
        y += delta_y;
        set_pixel(round(x), round(y));
    }
}
```

## 思路二：寻找最靠近直线的像素

- 假设$\Delta x > \Delta y > 0 \ (|m| < 1)$
- 需要绘制的每个像素距离直线的垂直距离都不超过0.5
  - 与四舍五入取整的DDA结果完全一致

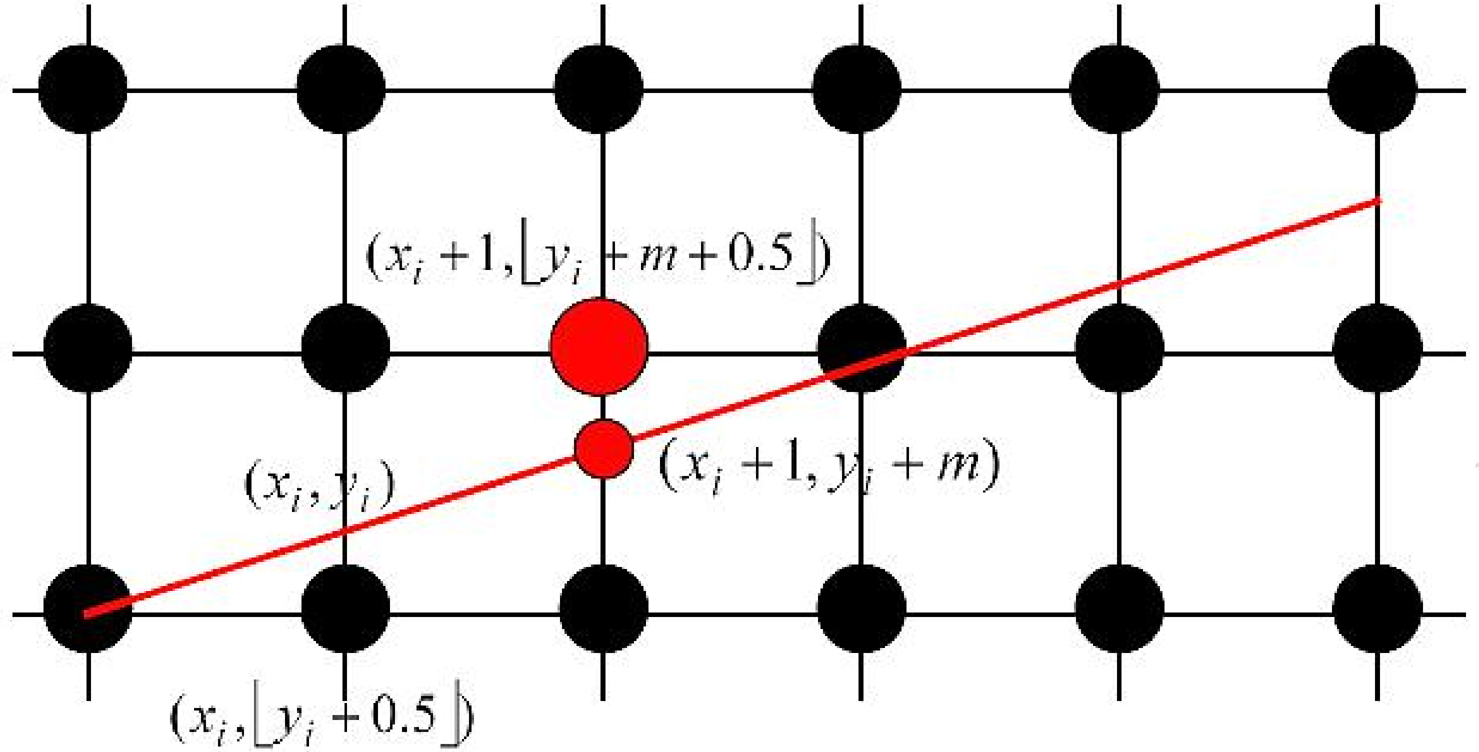


# 线段光栅化



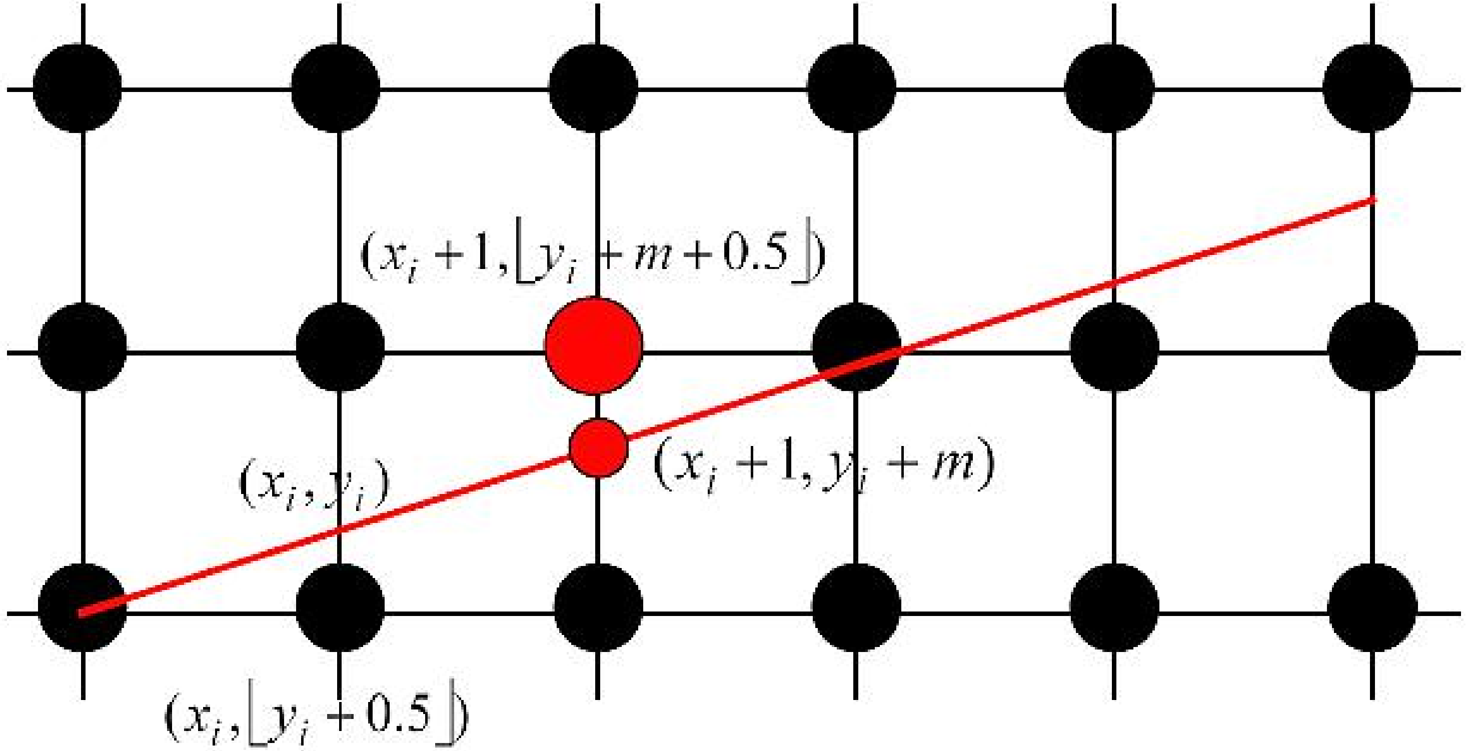
## 思路二：寻找最靠近直线的像素

- Bresenham’s algorithm（布兰森汉姆算法）
  - 1967年， 由IBM工程师Jack Elton Bresenham发明
  - 在DDA中，我们使用$y_i$计算$y_{i+1}$但 $y_i$所对应的像素信息并没有保留
  - Bresenham’s algorithm充分利用了这一信息，以推测下一个像素的位置



# 线段光栅化

- Bresenham’ s algorithm
  - 假设$\Delta x > \Delta y > 0\ (|m| < 1)$
  - 与DDA类似，Bresenham算法也$x = x_1$从开始，此后x每次自增1
  - 假设第$i$步绘制的像素为$(\overline{x_i}, \overline{y_i})$
  - 下一步需要绘制的像素为$(\overline{x_i}+1, \overline{y_i})$与$(\overline{x_i}+1, \overline{y_i}+1)$中的一个

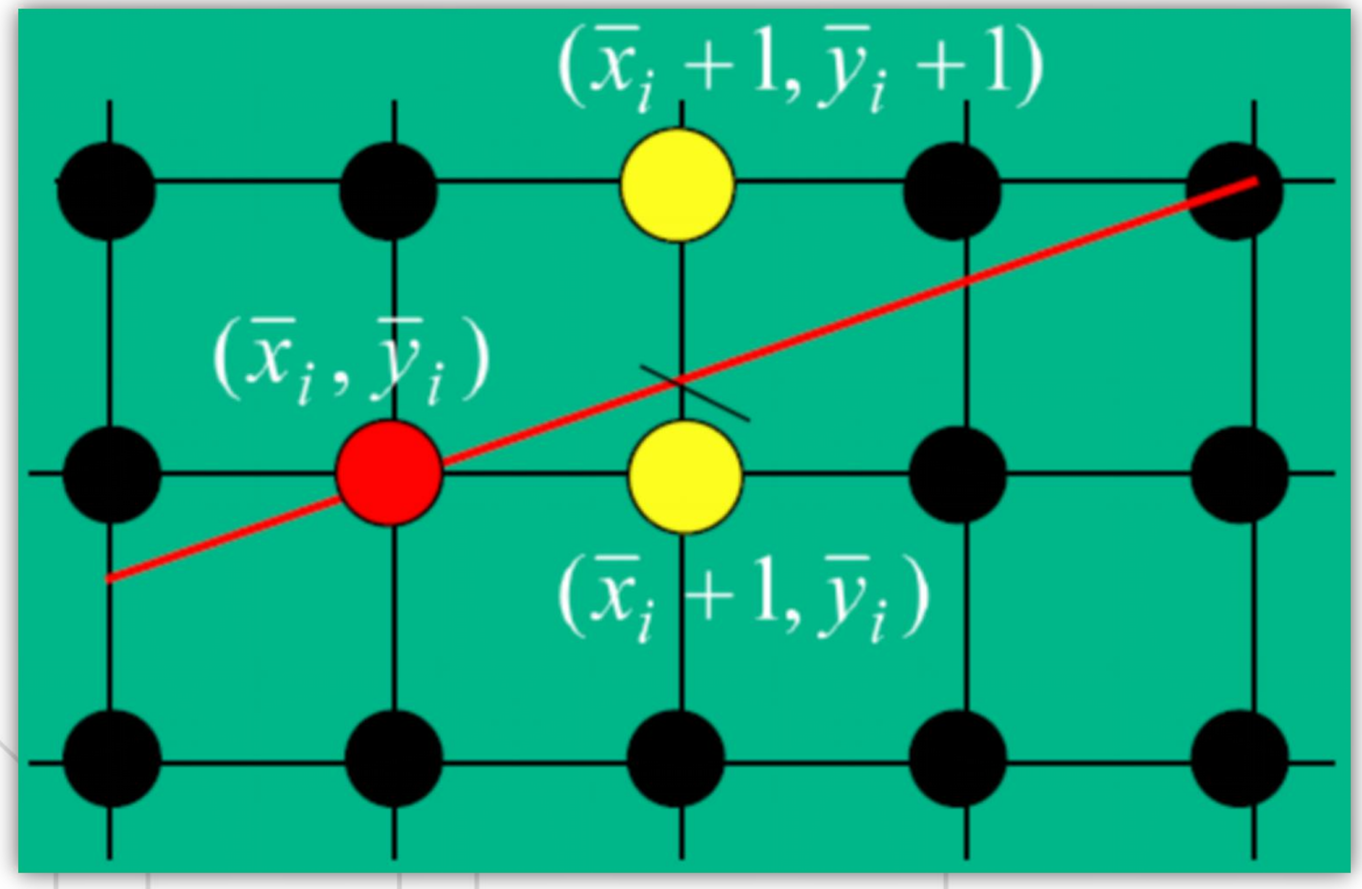


# 线段光栅化

- Bresenham’s algorithm
  - 如何在$(\overline{x_i}+1,\overline{y_i})$与$(\overline{x_i}+1,\overline{y_i}+1)$中进行选择？
    - 哪一个离直线更近！

$$x_{i+1} = x_i + 1$$

$$\begin{aligned} y_{i+1} &= m x_{i+1} + c \\ &= m(x_i + 1) + c \end{aligned}$$

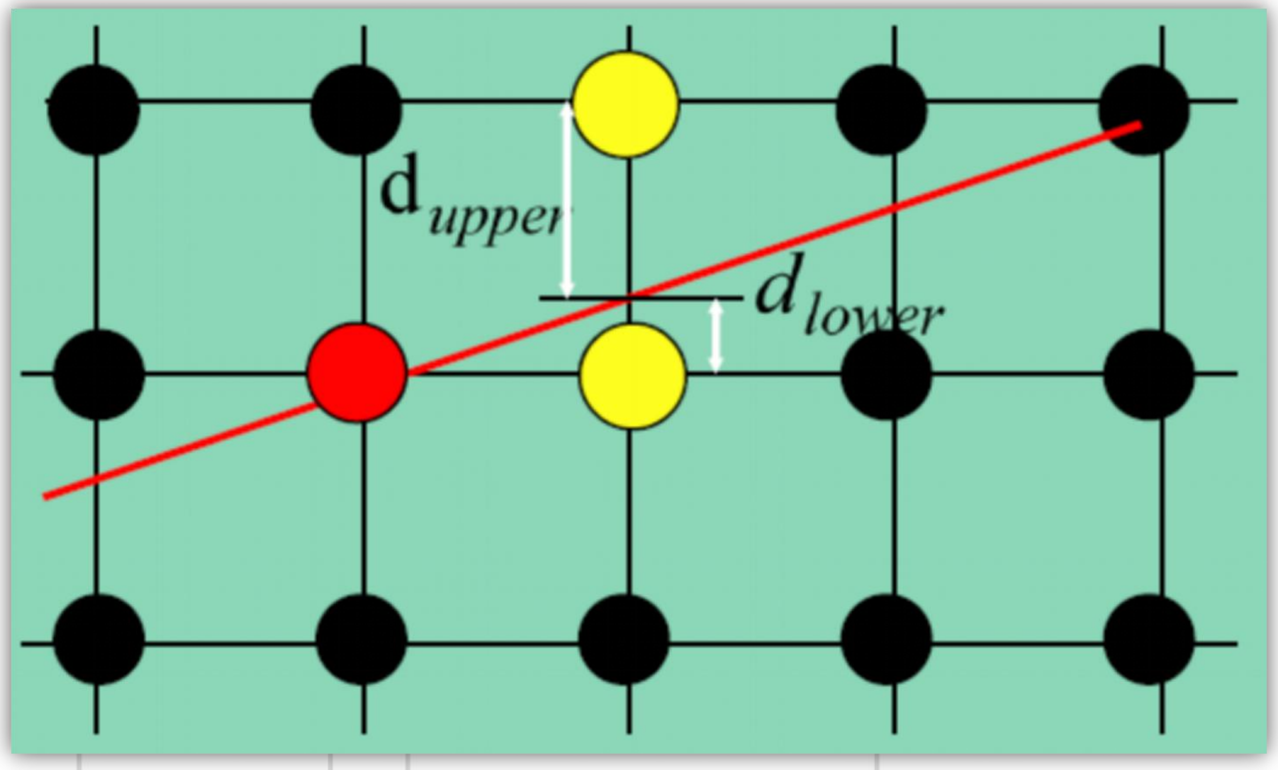


# 线段光栅化

- Bresenham’s algorithm
  - 如何在$(\bar{x}_i+1, \bar{y}_i)$与$(\bar{x}_i+1, \bar{y}_i+1)$中进行选择？
    - 比较$(\bar{x}_i+1, \bar{y}_i)$与$(\bar{x}_i+1, \bar{y}_i+1)$到直线上的点$(x_{i+1}, y_{i+1})$的距离
      - $d_{lower} > d_{upper}$则取上方的像素，$d_{lower} < d_{upper}$则取下方的像素，$d_{lower} = d_{upper}$则取任意像素皆可

$$
\begin{aligned}
d_{upper} &= \bar{y}_i + 1 - y_{i+1} \\
&= \bar{y}_i + 1 - (mx_{i+1} + c)
\end{aligned}
$$

$$
\begin{aligned}
d_{lower} &= y_{i+1} - \bar{y}_i \\
&= (mx_{i+1} - c) - \bar{y}_i
\end{aligned}
$$



# 线段光栅化

- Bresenham’ s algorithm
  - 如何在$(\overline{x}_i+1,\overline{y}_i)$与$(\overline{x}_i+1,\overline{y}_i+1)$中进行选择?
    - 只需判断$d_{lower}-d_{upper}$的符号

$$
\begin{aligned}
d_{lower}-d_{upper} &= (y_{i+1}-\bar{y}_i)-(\bar{y}_i+1-y_{i+1}) \\
&= 2y_{i+1}-2\bar{y}_i-1 \\
&= 2(m(x_i+1)+c)-2\bar{y}_i-1 \\
&= 2m(x_i+1)-2\bar{y}_i+2c-1
\end{aligned}
$$

    - 目前为止，运算得到简化了吗?

- Bresenham's algorithm
  - 如何在$(\bar{x}_i+1, \bar{y}_i)$与$(\bar{x}_i+1, \bar{y}_i+1)$中进行选择？
    - 只需判断$d_{lower}-d_{upper}$的符号
    - 将浮点运算转变为整型运算

$$d_{lower}-d_{upper}=2\boxed{m}(x_i+1)-2\bar{y}_i+2\boxed{c}-1$$

    - 注意到两个浮点型数字表示为分数是有相同的分母$\Delta x$

$$\begin{aligned}p_i&=\Delta x\cdot(d_{lower}-d_{upper})\\&=2\Delta y\cdot(x_i+1)-2\Delta x\cdot\bar{y}_i+(2c-1)\Delta x\\&=2\Delta y\cdot x_i-2\Delta x\cdot\bar{y}_i+(2c-1)\Delta x+2\Delta y\end{aligned}$$

# 线段光栅化

- Bresenham’ s algorithm
  - 如何在$(\overline{x}_i+1,\overline{y}_i)$与$(\overline{x}_i+1,\overline{y}_i+1)$中进行选择？
    - 如何计算$p_i$?
    - 初始值

$$
\begin{aligned}
p_0 &= 2\Delta y \cdot x_0 - 2\Delta x \cdot \bar{y_0} + (2c-1)\Delta x + 2\Delta y \\
&= 2\Delta y \cdot x_0 - 2\Delta x \cdot (\Delta y \cdot x_0 + c \cdot \Delta x) + (2c-1)\Delta x + 2\Delta y \\
&= 2\Delta y - \Delta x
\end{aligned}
$$

# 线段光栅化

- Bresenham’s algorithm
  - 如何在$(\overline{x_i}+1, \overline{y_i})$与$(\overline{x_i}+1, \overline{y_i}+1)$中进行选择？
    - 如何计算$p_i$?
    - 迭代更新

$$
\begin{aligned}
p_{i+1}-p_i &= (2\Delta y\cdot x_{i+1}-2\Delta x\cdot \bar{y}_{i+1}+\tilde{c})-(2\Delta y\cdot x_i-2\Delta x\cdot \bar{y}_i+\tilde{c}) \\
&= 2\Delta y-2\Delta x\cdot(\bar{y}_{i+1}-\bar{y}_i)
\end{aligned}
$$

- 当$p_i \le 0$时，可知$d_{lower} \le d_{upper}$，取下方像素，因此$\overline{y}_{i+1}-\overline{y}_i=0$
  - $p_{i+1}=p_i+2\Delta y$
- 当$p_i > 0$时，可知$d_{lower} > d_{upper}$，取上方像素，因此$\overline{y}_{i+1}-\overline{y}_i=1$
  - $p_{i+1}=p_i+2\Delta y-2\Delta x$
- 至此，所有运算均可使用整数完成

- Bresenham's algorithm
  - 伪代码
    - draw $(x_0, y_0)$
    - Calculate $\Delta x, \Delta y, 2\Delta y, 2\Delta y - 2\Delta x$, $p_0 = 2\Delta y - \Delta x$
    - If $p_i \le 0$ draw $(x_{i+1}, \bar{y}_{i+1}) = (x_i + 1, \bar{y}_i)$

      and compute $p_{i+1} = p_i + 2\Delta y$
    - If $p_i > 0$ draw $(x_{i+1}, \bar{y}_{i+1}) = (x_i + 1, \bar{y}_i + 1)$

      and compute $p_{i+1} = p_i + 2\Delta y - 2\Delta x$
    - Repeat the last two steps

# 线段光栅化



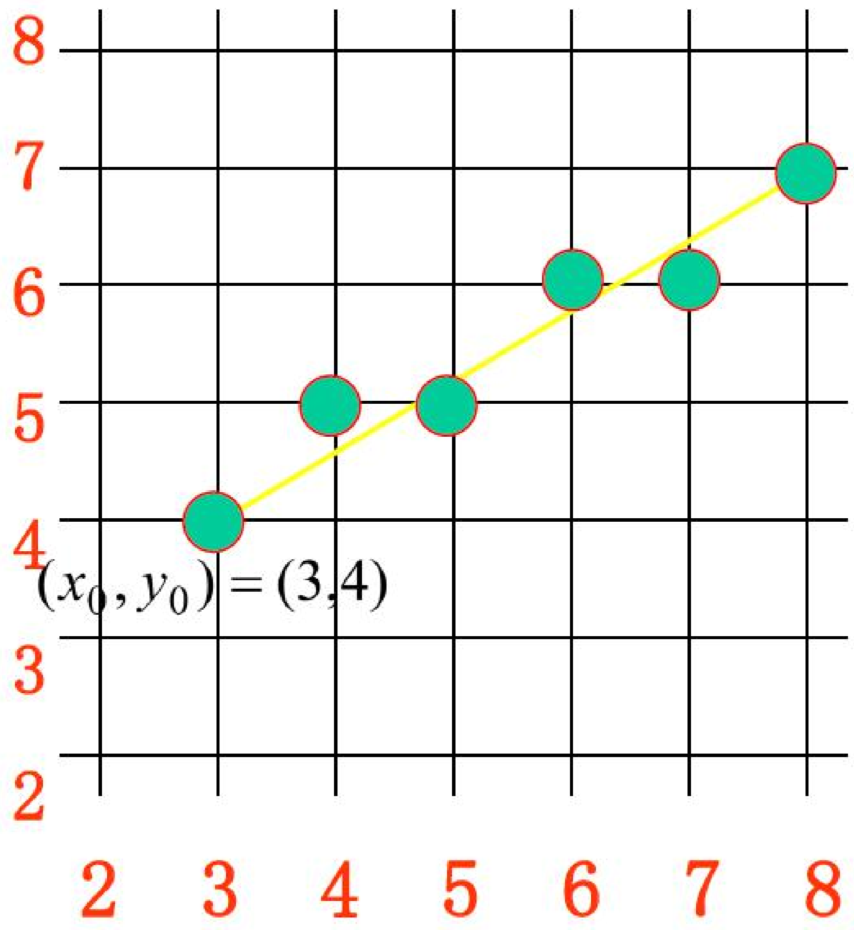
- Bresenham’s algorithm
  - 举例，绘制线段(3,4)→(8,7)
    - $\Delta x = 5,\ \Delta y = 3$
    - $2\Delta y = 6$，$2\Delta y - 2\Delta x = -4$

- draw $(x_0, y_0)$
- Calculate $\Delta x, \Delta y, 2\Delta y, 2\Delta y - 2\Delta x,\ p_0 = 2\Delta y - \Delta x$
- If $p_i \le 0$ draw $(x_{i+1}, \bar{y}_{i+1}) = (x_i + 1, \bar{y}_i)$

  and compute $p_{i+1} = p_i + 2\Delta y$
- If $p_i > 0$ draw $(x_{i+1}, \bar{y}_{i+1}) = (x_i + 1, \bar{y}_i + 1)$

  and compute $p_{i+1} = p_i + 2\Delta y - 2\Delta x$
- Repeat the last two steps

| k | $p_k$ | $(x_{k+1}, y_{k+1})$ |
|---|---|---|
| 0 | 1 | (4,5) |
| 1 | -3 | (5,5) |
| 2 | 3 | (6,6) |
| 3 | -1 | (7,6) |
| 4 | 5 | (8,7) |

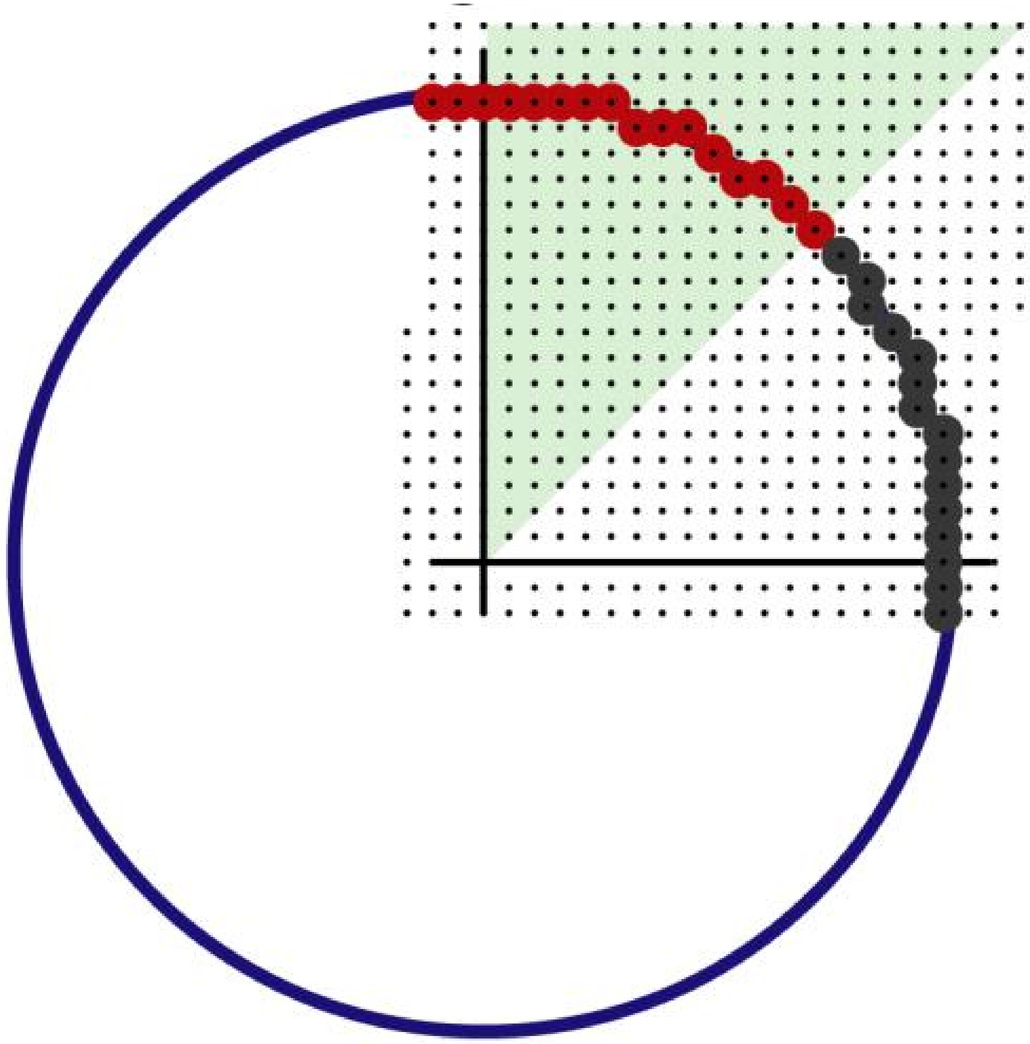


# 线段光栅化



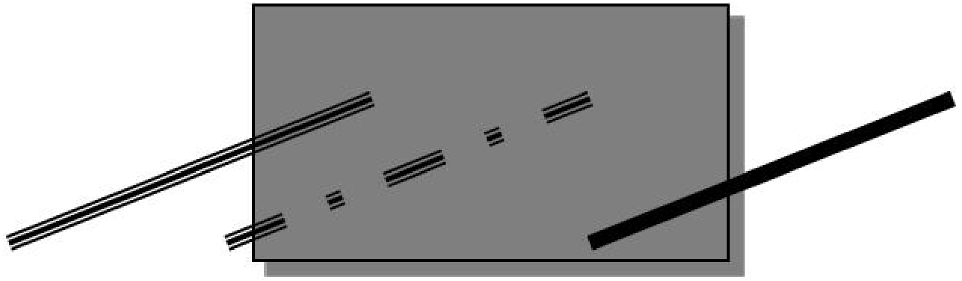
## 为什么要学习Bresenham算法？

- OpenGL里已经有画线函数了
- 理解硬件的工作过程
- 学习思路
  - 将Bresenham算法视作一个范例
  - 理解核心思想并应用于未来遇到的问题中
  - Ray marching与direct volume rendering
  - Bresenham画圆算法
    - 类比Bresenham线段算法
    - 只需要对八分之一段弧进行计算
    - 斜率从0增加到1
    - https://www.geeksforgeeks.org/bresenhams-circle-drawing-algorithm/



# 线段光栅化

- 在OpenGL中画出完美的直线并非容易的事
  - https://mattdesl.svbtle.com/drawing-lines-is-hard
  - 抗锯齿
  - 在不同角度下保持线段的宽度不变
  - 从p到q及从q到p绘制的直线应该是一致的
  - 线的绘制风格
    - 宽度、虚线、端点、转折点

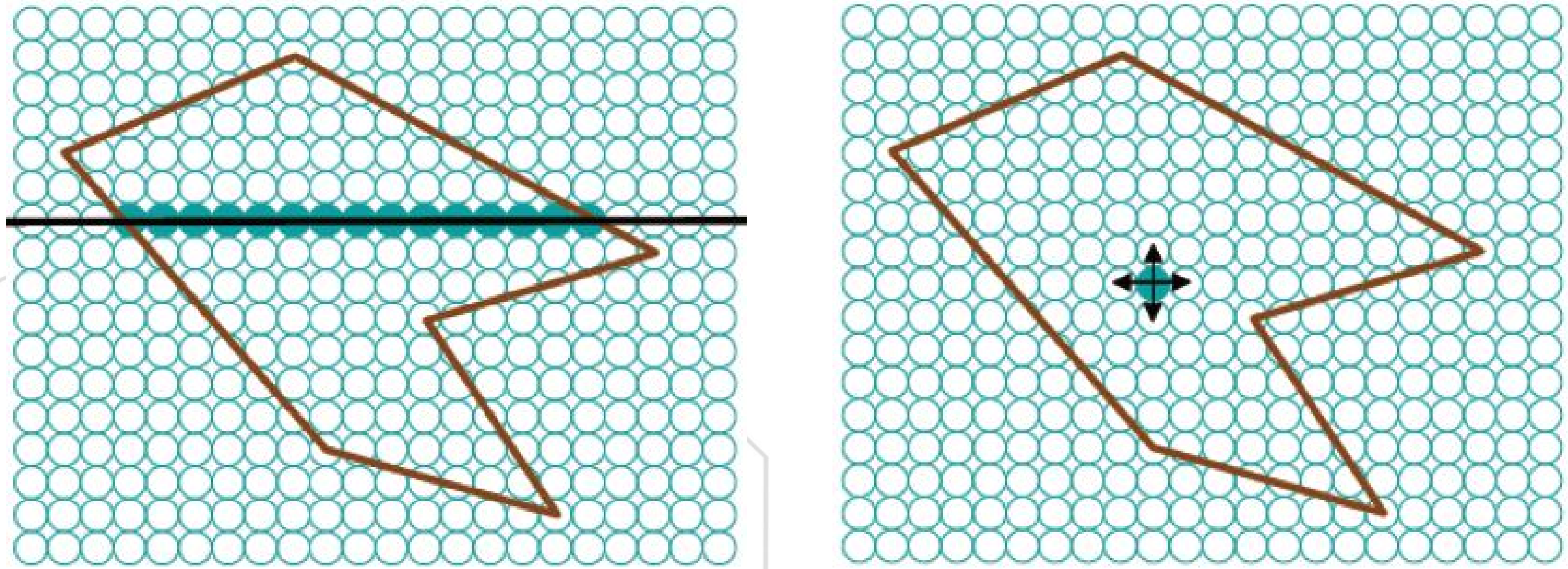


# 课程概要

- 光栅化简介
- 线段光栅化
- 多边形光栅化
- 抗锯齿

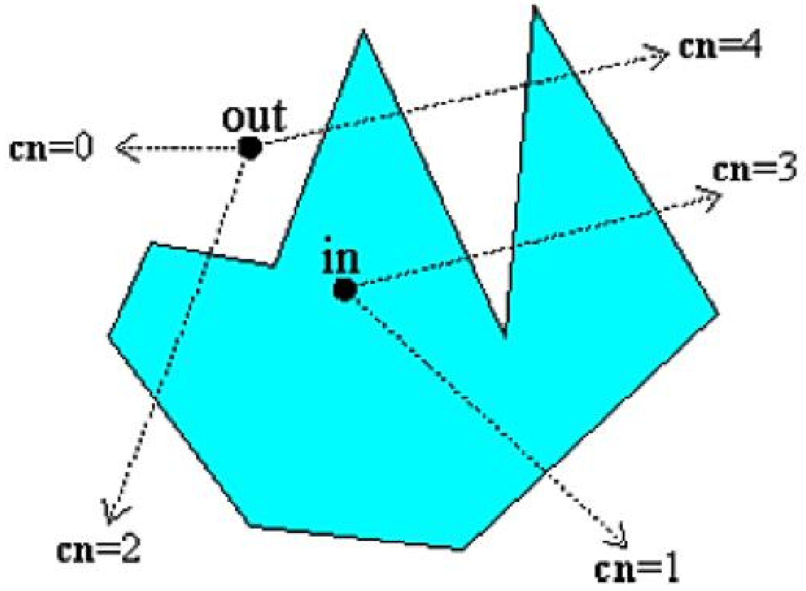
# 多边形光栅化

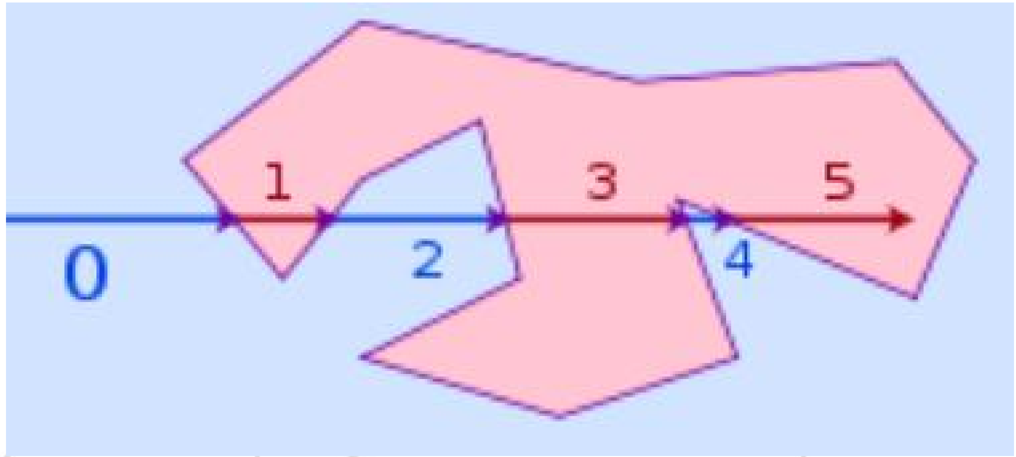
- 多边形光栅化：决定哪些像素与多边形相关
  - 两种常见思路
  - Scan-conversion：沿scan line逐行扫描，设置边界内的像素
  - Fill：选择多边形内的一个像素作为起点向外扩张，直至充满整个多边形（不易并行）
    - 也可以对屏幕中每个点进行测试，看是否在多边形内（可并行完成）



# 多边形光栅化

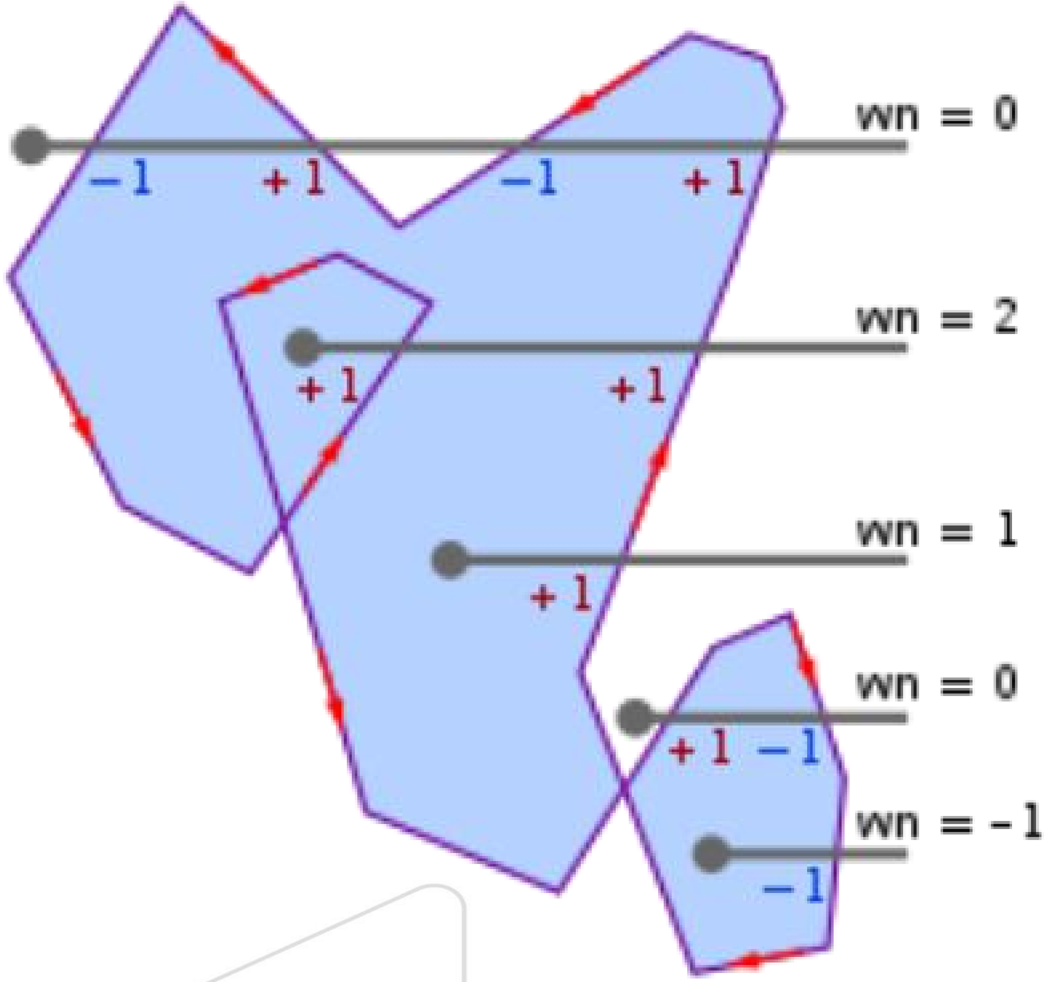
- 如何判断一个点在多边形内？
  - Even-odd test （ray casting algorithm）
    - 计算与多边形外任意一点的连线与多边形边界相交次数
    - 奇数次则该点在多边形内，否则在多边形外
    - 需要求射线与线段的交点（容易引起精度问题）



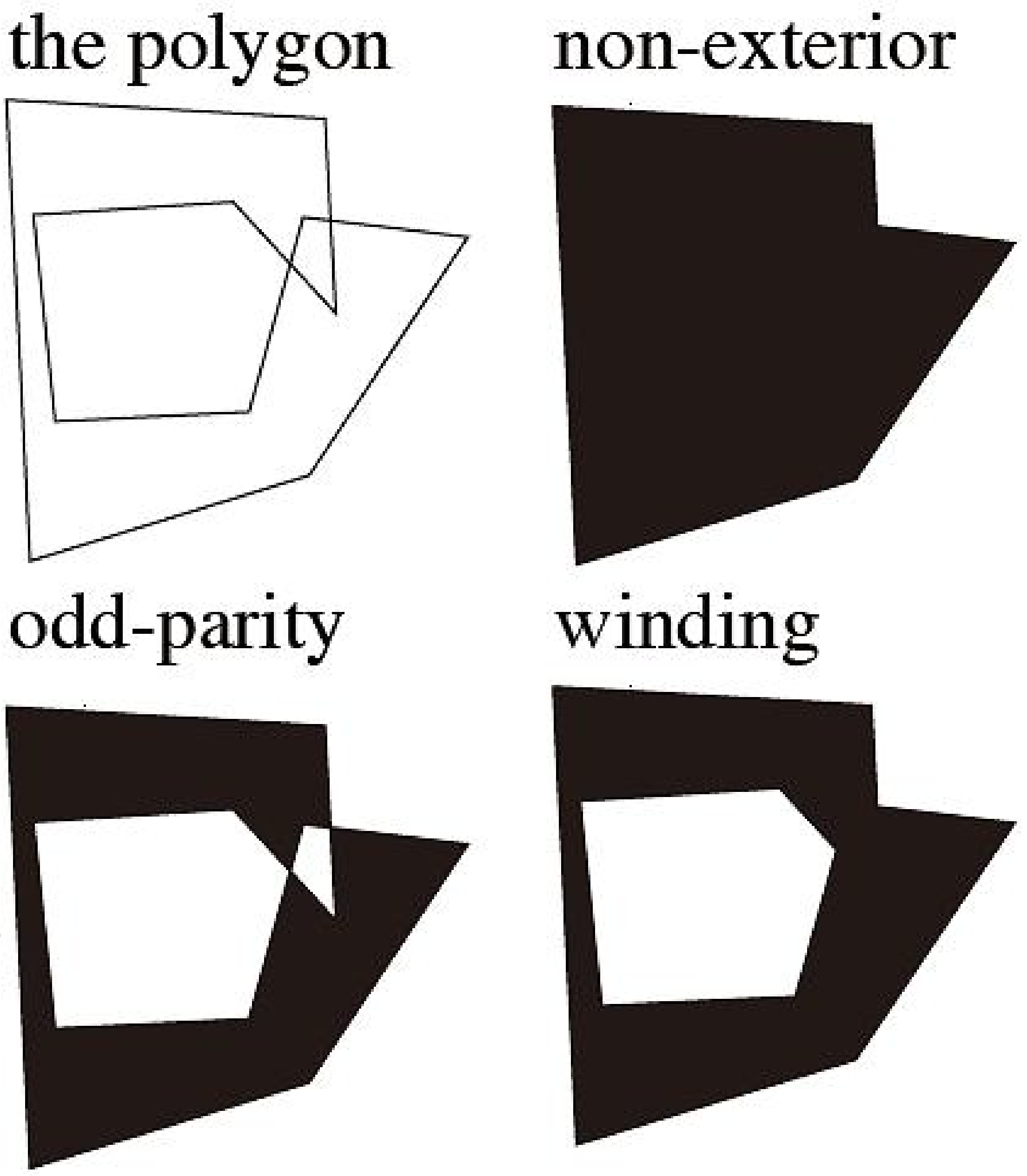
# 多边形光栅化

- 如何判断一个点在多边形内？
  - Winding number test
    - 根据多边形是绕该点所转角度判断
    - 若该角度非0，则该点在多边形内
      - 计算角度：需使用反三角函数，因此可能较慢
      - 检查所跨过的象限：等效于计算角度，较快
      - Dan Sunday’s algorithm：从测试点出发引出的水平线与多边形交点上判断累计角度
    - 除判断是否为0外，也可以判断累计转角为$2\pi$的奇数/偶数倍



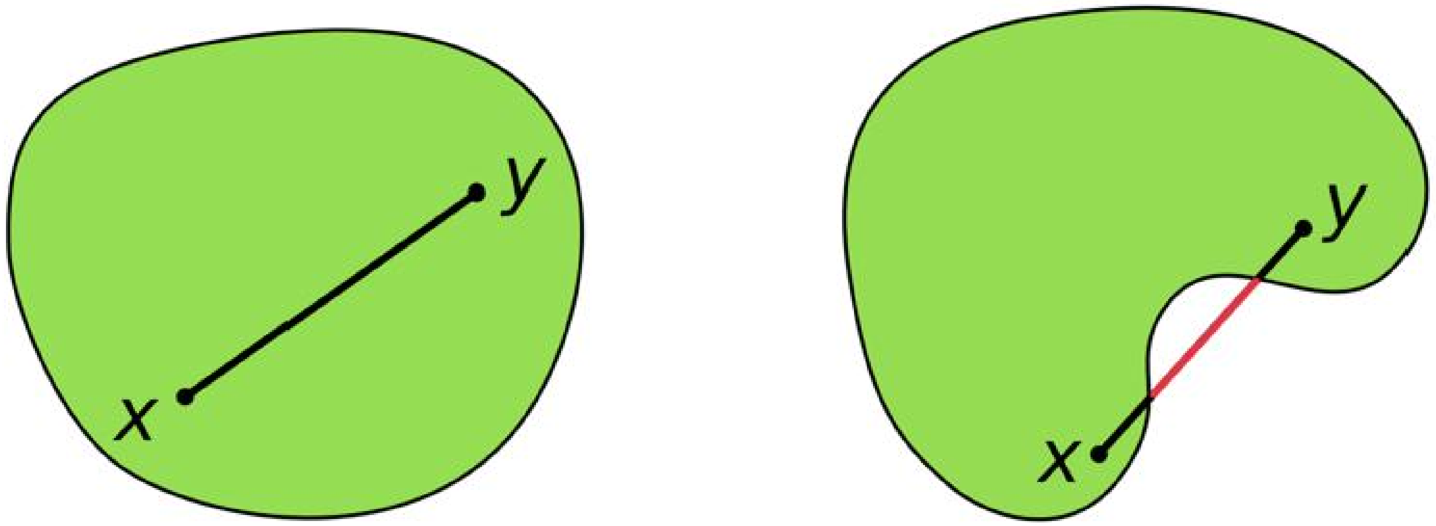
# 多边形光栅化

- 如何判断一个点在多边形内？
  - 不同判定规则可能产生不同的结果
    - 针对非“简单”多边形而言
    - 哪一种是正确的可能需要由实际应用决定
    - Odd-parity rule：奇数次相交则在内
    - Non-exterior rule：该点与无穷远点间连线与多边形相交则在多边形内
    - Winding rule：绕该点角度不为0则在多边形内



## 三角形光栅化

– 任意简单多边形都可拆解为多个三角形
  - Convex或non-convex




A set C in S is said to be **convex** if, for all x and y in C and all $t$ in the interval [0,1], the point

$$(1-t)x + ty$$

is in C.

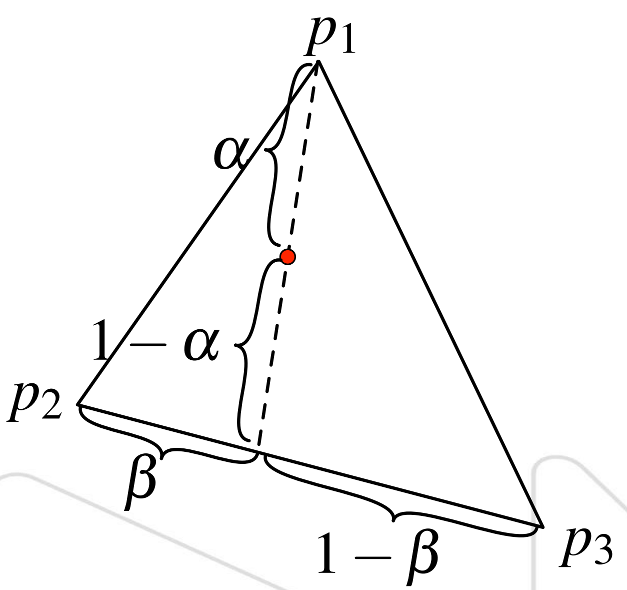
# 多边形光栅化

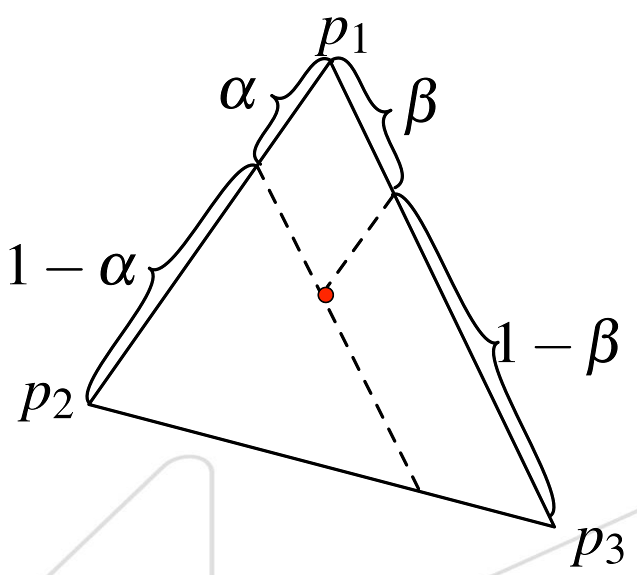
## 三角形光栅化

– 如何计算三角形相关的所有像素？
  - Fill: edge-equation; Scan conversion: edge-walking

– 如何计算每个像素的颜色？
  - 常使用基于barycentric coordinate的插值方法

$$a \cdot p_1 + b \cdot p_2 + c \cdot p_3, \text{ where } a + b + c = 1$$



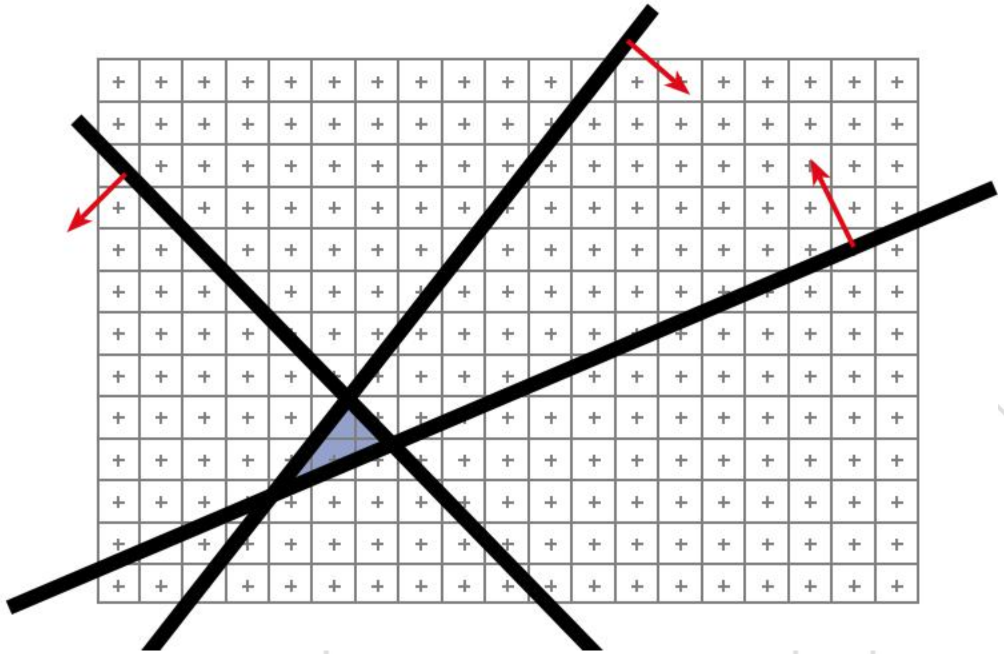

$$(1-\alpha)p_1 + \alpha((1-\beta)p_2 + \beta p_3)$$
$$=(1-\alpha)p_1 + \alpha(1-\beta)p_2 + \alpha\beta p_3$$

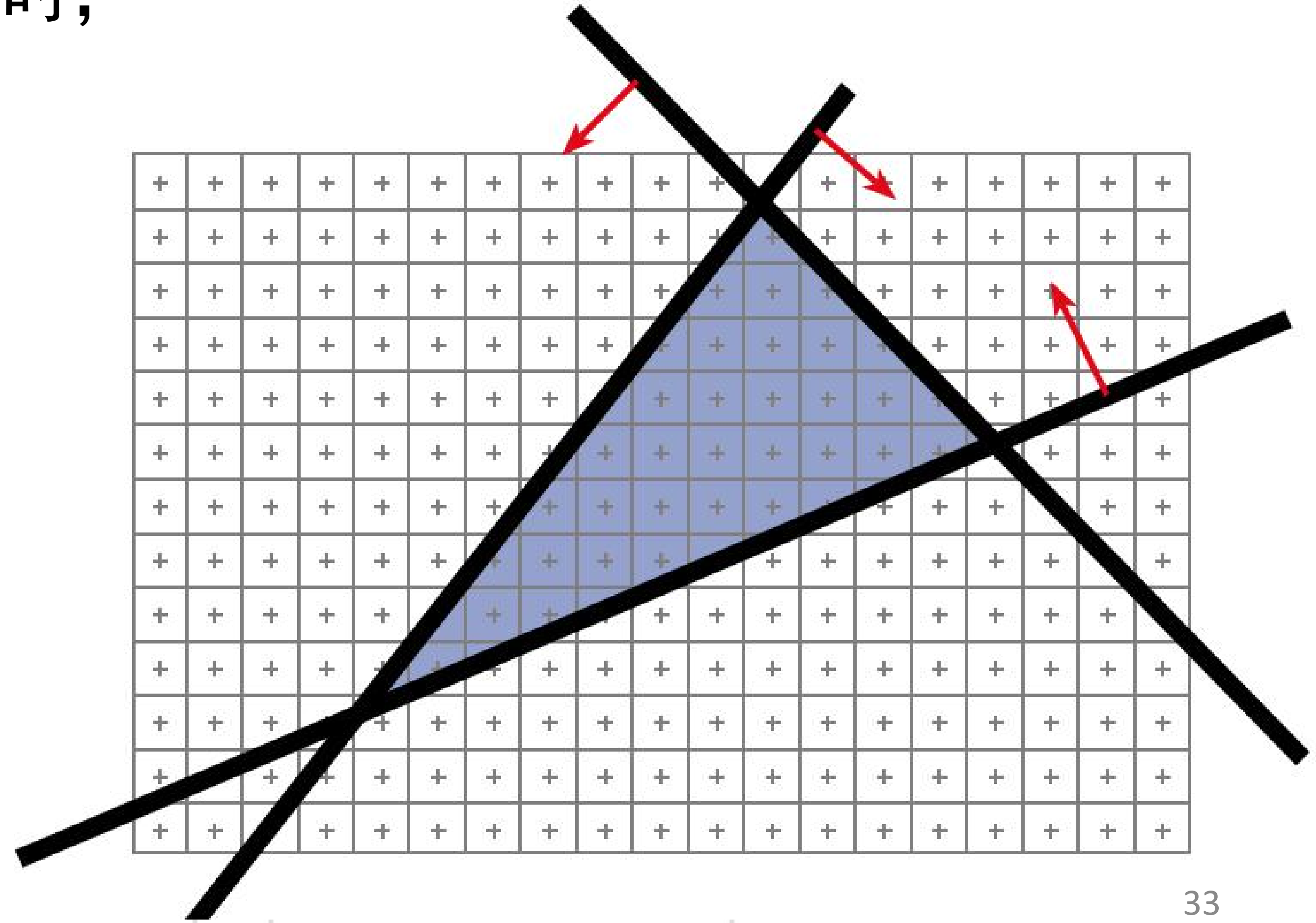
$$p_1 + \alpha(p_2 - p_1) + \beta(p_3 - p_1)$$
$$=(1-\alpha-\beta)p_1 + \alpha p_2 + \beta p_3$$

# 多边形光栅化

## 三角形光栅化：edge-equation

- 使用三角形三边的表达式
- 判断屏幕上某个点是否处于三角形中：对每条边，判断该点是否与边相对的顶点处于边的同侧
  - 存在问题：当三角形相对屏幕而言很小时，需要进行大量不必要的判断

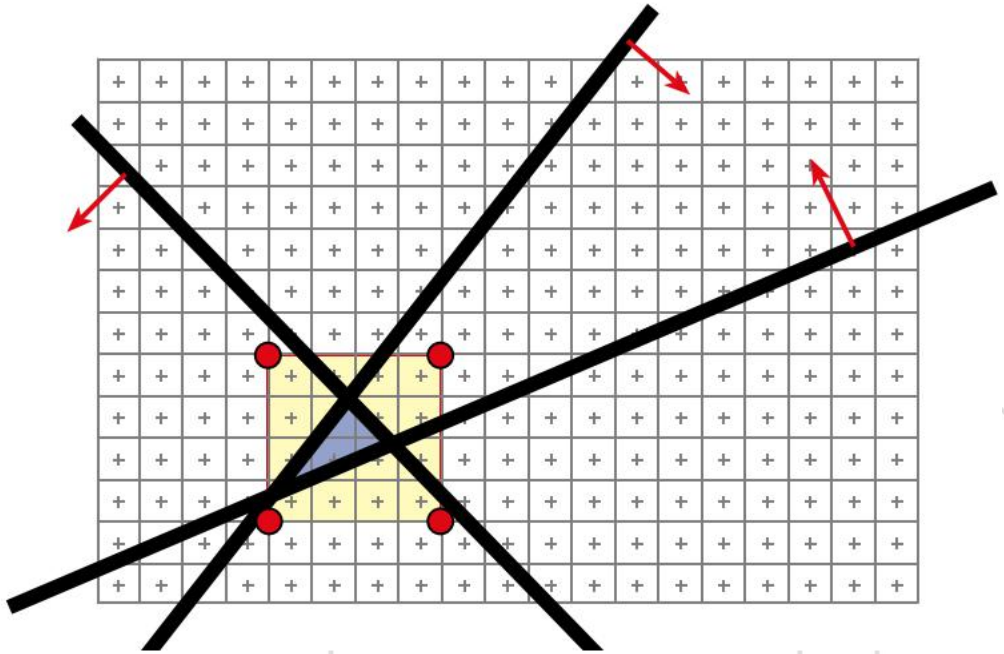


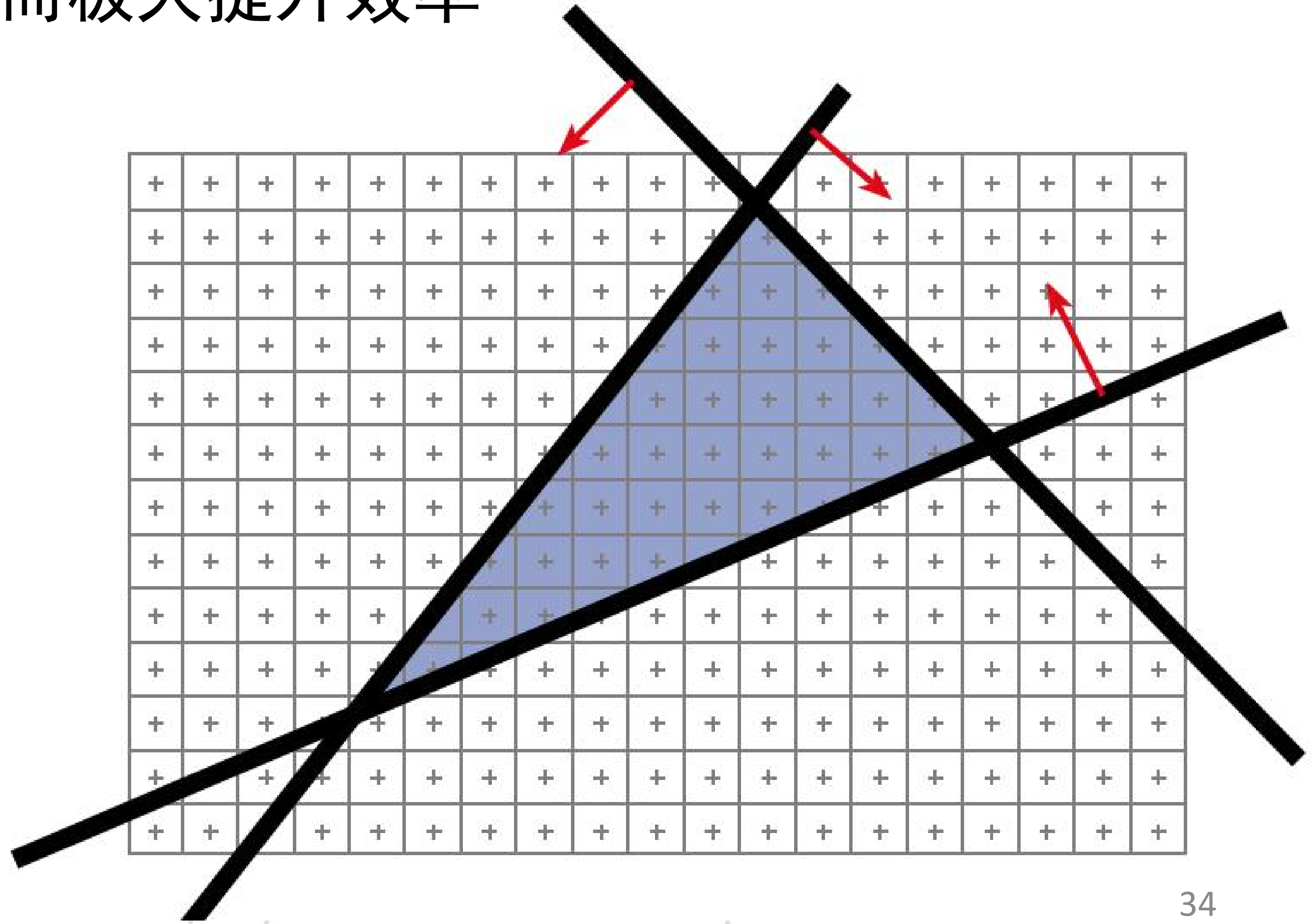


# 多边形光栅化

## 三角形光栅化：edge-equation

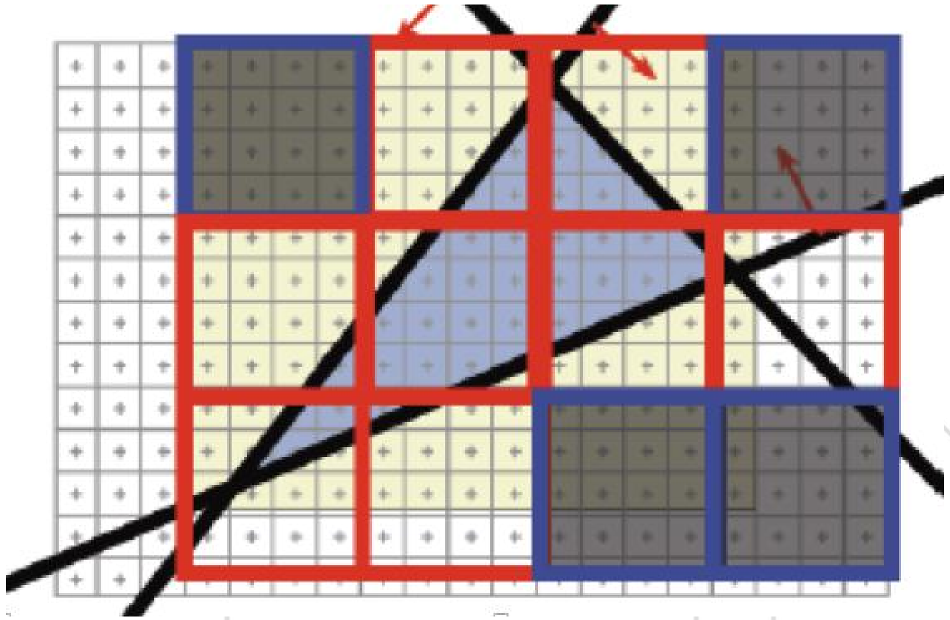
- 使用三角形三边的表达式
- 判断屏幕上某个点是否处于三角形中：对每条边，判断该点是否与边相对的顶点处于边的同侧
  - 可使用bounding box限制判断范围，从而极大提升效率

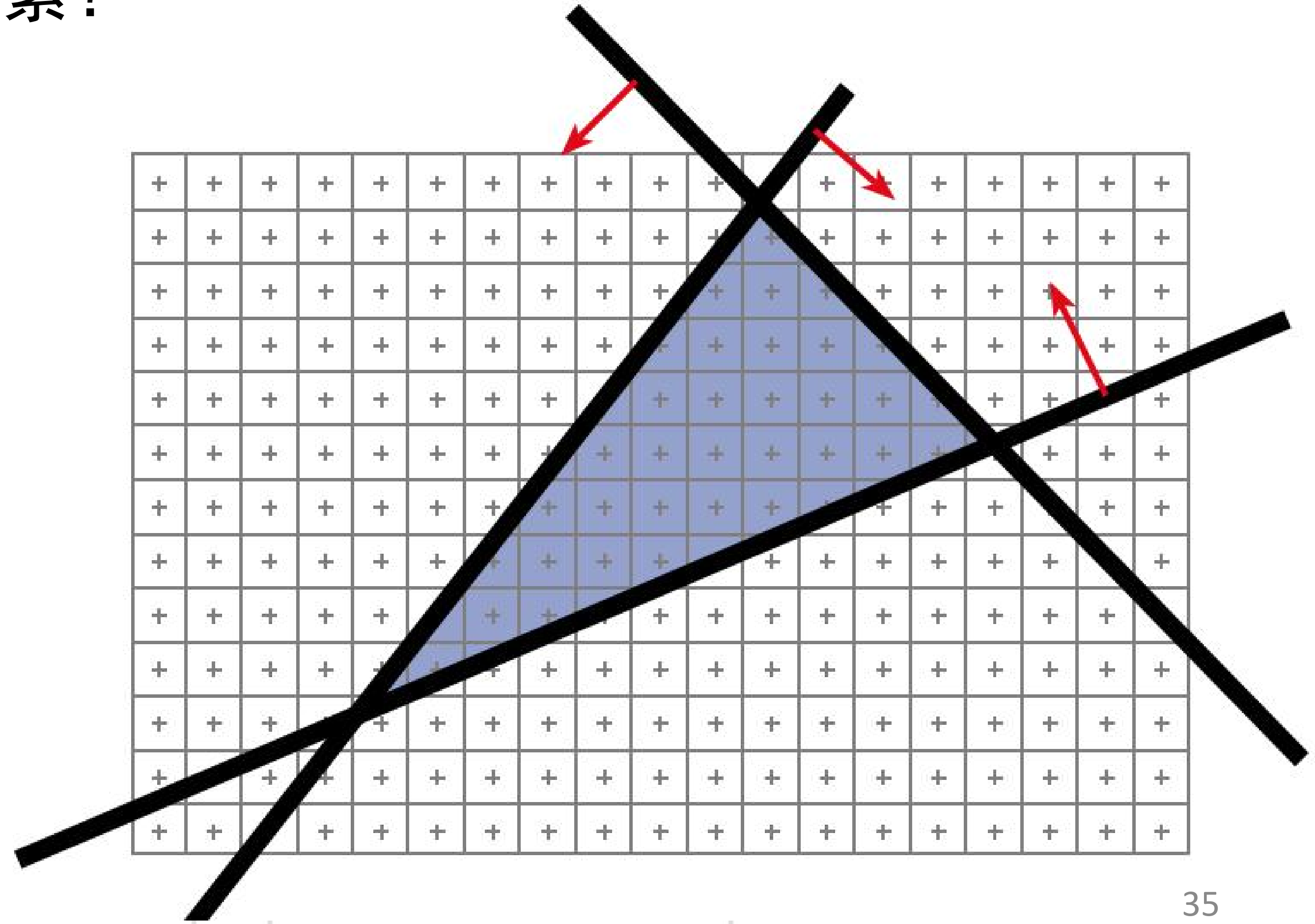




# 多边形光栅化

- 三角形光栅化：edge-equation
  - 使用三角形三边的表达式
  - 判断屏幕上某个点是否处于三角形中：对每条边，判断该点是否与边相对的顶点处于边的同侧
    - 如何使用层次bounding box排除一片像素？

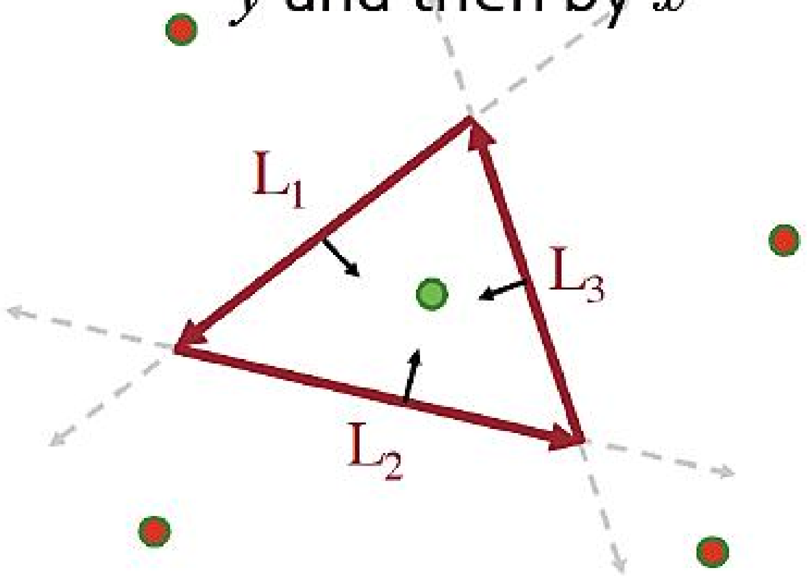




## 三角形光栅化：edge-equation

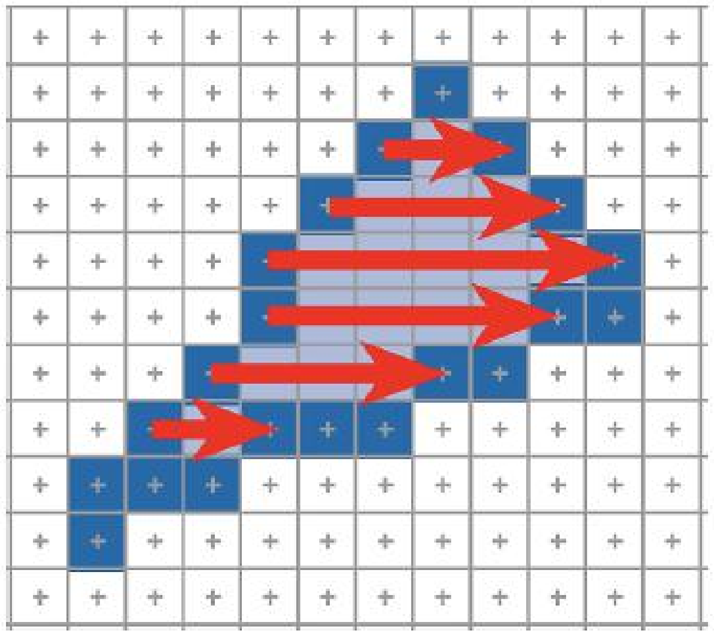
```
void edge_equations(vertices T[3])
{
  bbox b = bound(T);
  foreach pixel(x, y) in b {
    inside = true;
    foreach edge line L_i of Tri {
      if (L_i.A*x+L_i.B*y+L_i.C < 0) {
        inside = false;
      }
    }
    if (inside) {
      set_pixel(x, y);
    }
  }
}
```

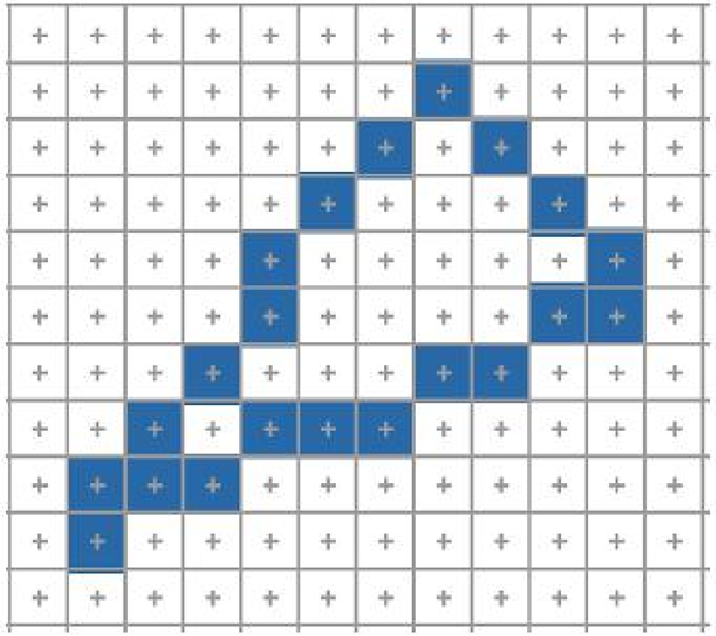
can be rewritten to update the $L$'s incrementally by $y$ and then by $x$



# 多边形光栅化

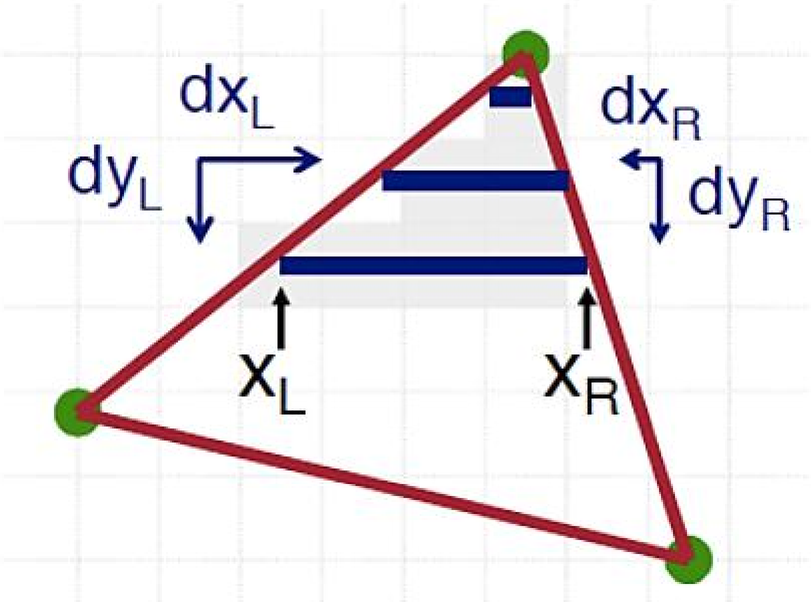
- 三角形光栅化：edge-walking
  - 自顶向下逐行扫描
  - 沿着边的坡度更新边上像素的坐标
    - 可使用Brehensem算法
  - 对每一个scan-line从左向右设置像素
  - 当遇到底部顶点（或边）时停止





## 三角形光栅化：edge-walking

```
void edge_walking(vertices T[3])
{
  for each edge pair of T {
    initialize xL, xR;
    compute dxL/dyL and dxR/dyR;
    for scanline at y {
      for (int x = xL; x <= xR; x++) {
        set_pixel(x, y);
      }
    }
    xL += dxL/dyL;
    xR += dxR/dyR;
  }
}
```

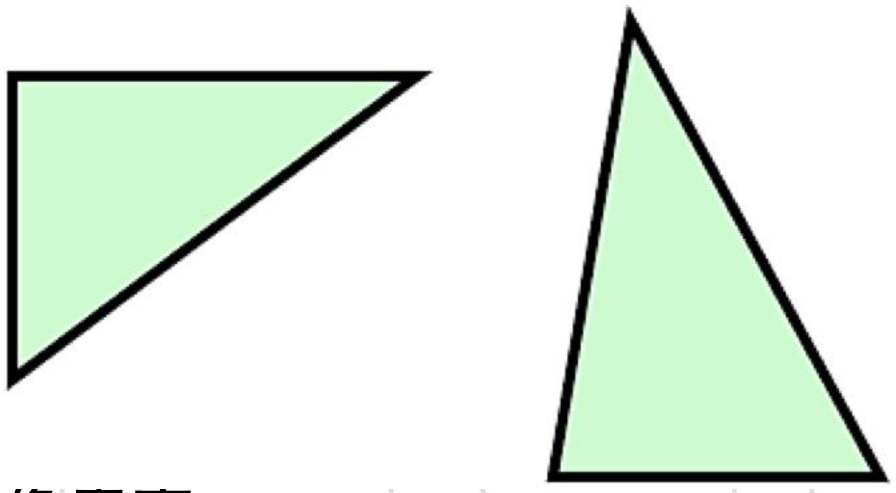



Funkhouser09

# 多边形光栅化

## 三角形光栅化：edge-walking

- 优点：简单，每个像素上所需计算少
- 缺点
  - 串行化
    - 需要按顺序一个像素到下一个像素→难以并行
    - 但行与行之间可以并行
  - 特殊情况需要另行处理
    - 平行边：$\Delta y = 0$



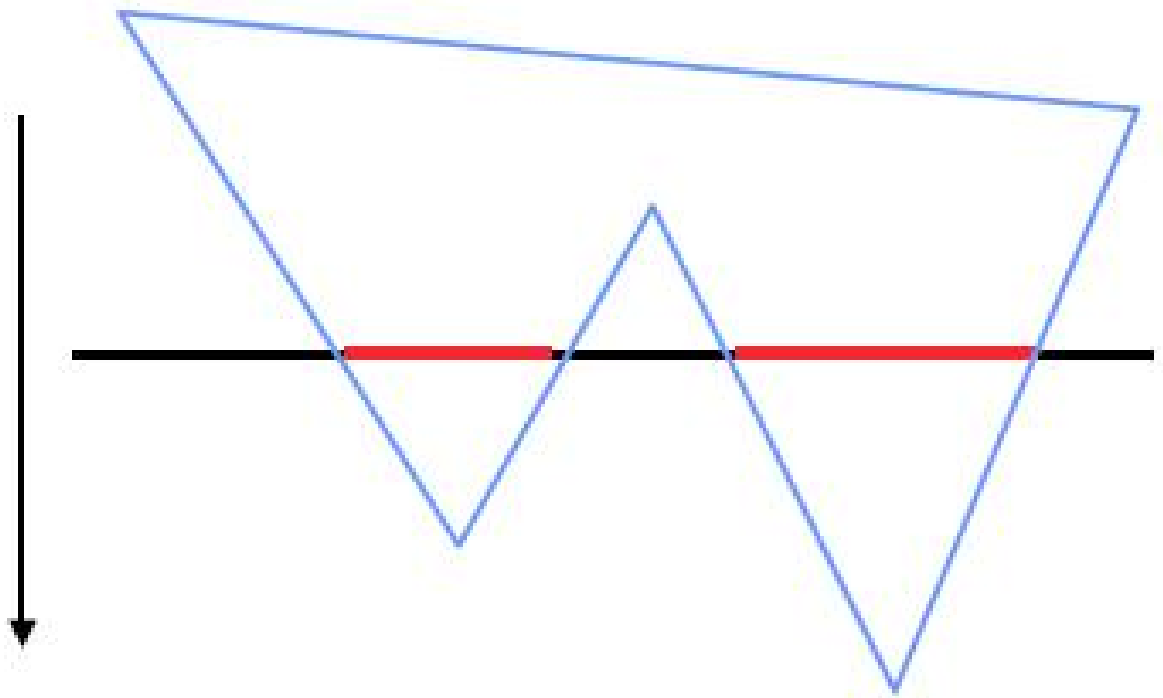
    - 三角形过窄：不到1像素高

# 多边形光栅化



## 多个三角形或不规则多边形的扫描

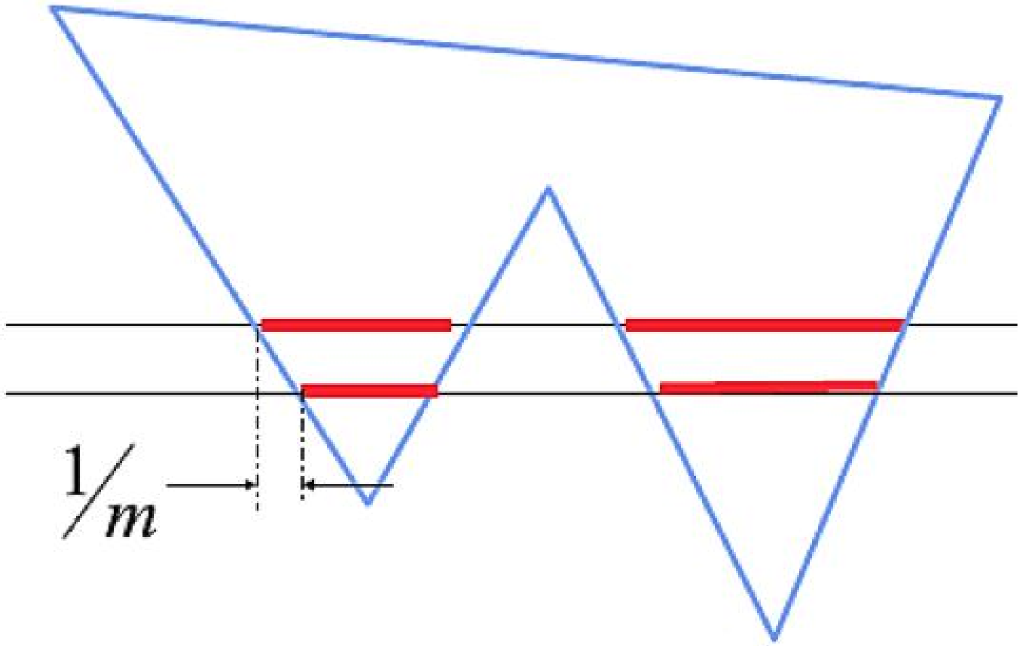
- 充分利用多边形的连贯性
  - 空间连贯性：除边界上的像素外，相邻像素具有同样特征
  - scan-line连贯性：相邻scan-line具有相同特性
- 使用边界线的交点与scan-lines判断像素是否在区域中
- 算法：与三角形edge-walking类似
  - 自顶向下逐行扫描
  - 求scan-line与所有edge之间的交点
  - 将交点按x坐标进行排序
    - 最为耗时
  - 设置相邻的每对交点之间的像素



# 多边形光栅化

- 多个三角形或不规则多边形的扫描
  - 算法：与三角形edge-walking类似
    - 如何快速求scan-lines与edge的交点及对其进行排序？
      - 利用scan-line连贯性

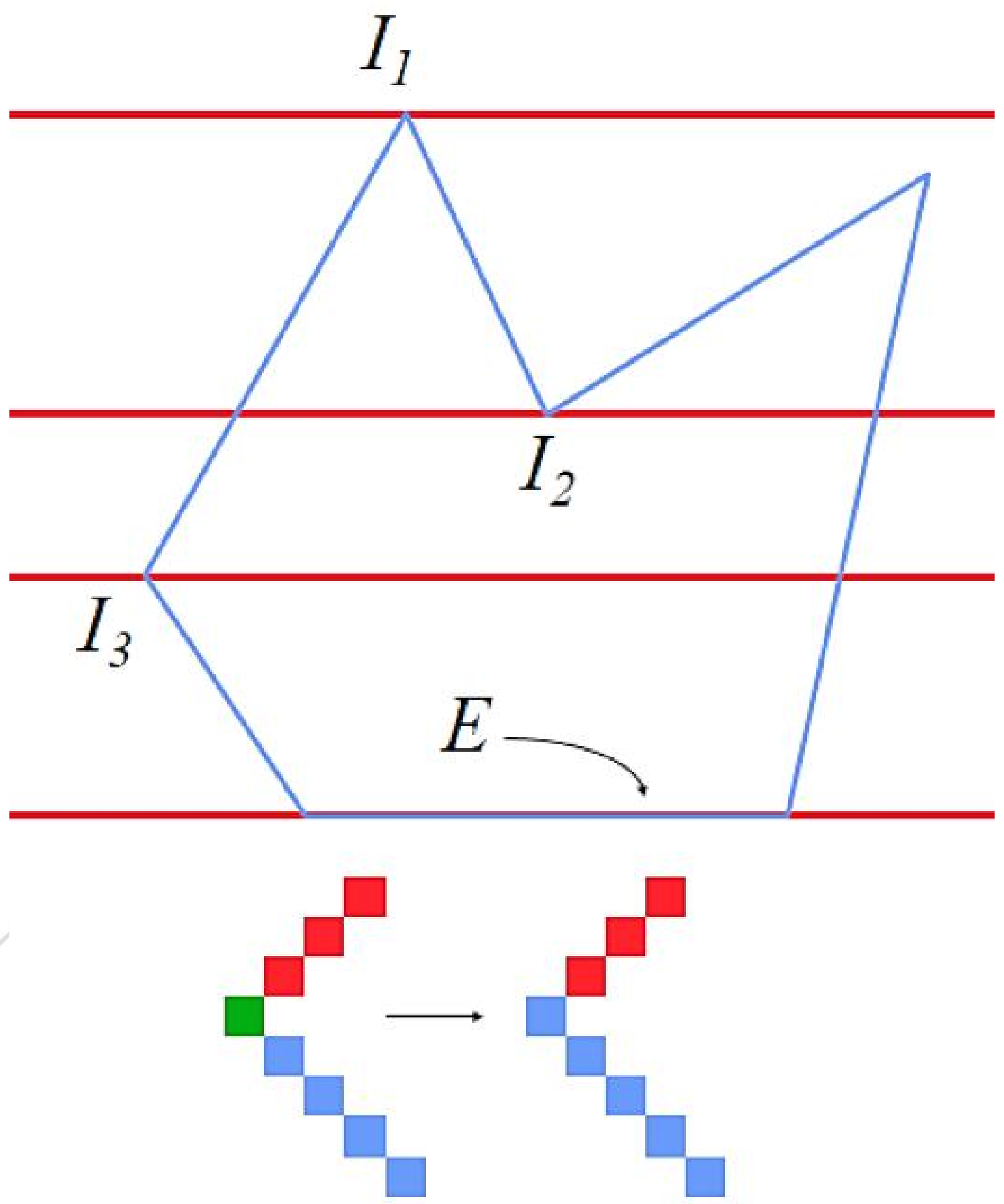
$$x_{i+1} = x_i + 1/m$$




      - 可使用维护active edge列表以提高效率
        - » 对行记录行内的所有顶点，active edge只在顶点处发生变化

# 多边形光栅化

- 多个三角形或不规则多边形的扫描
  - 需要考虑整体拓扑结构
  - $I_1$及$I_2$应被视为两个交点！
  - $I_3$仍然为一个交点
    - 但坡度需要发生变化
  - 水平边不需要考虑

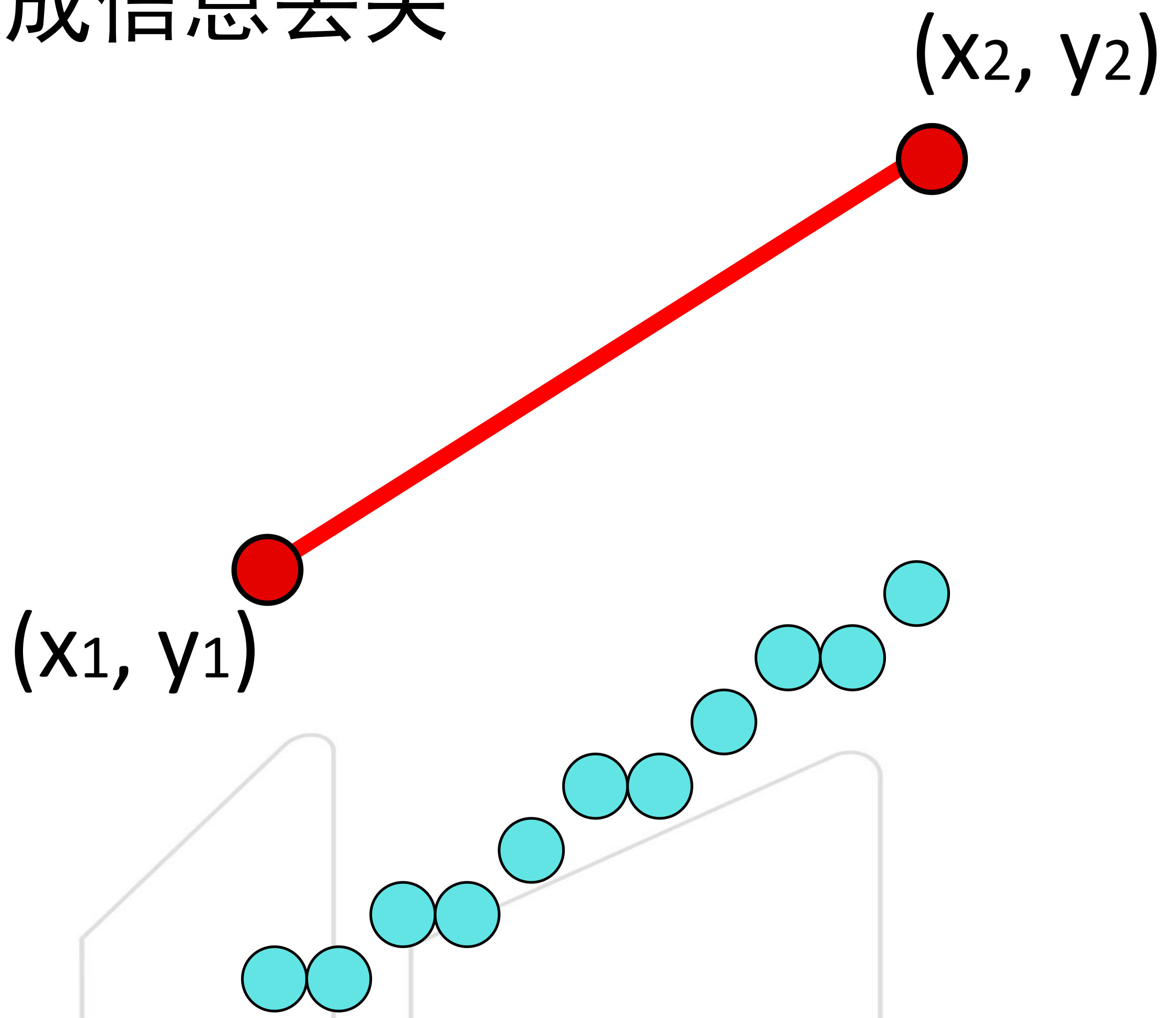


# 课程概要

- 光栅化简介
- 线段光栅化
- 多边形光栅化
- 抗锯齿

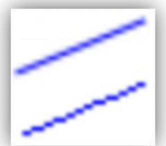
# 抗锯齿

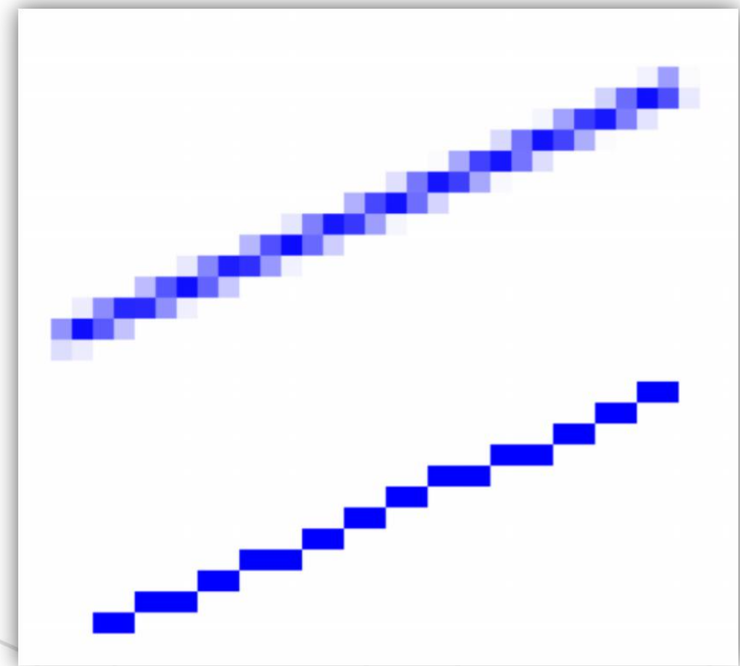
- 对几何物体的离散化会产生锯齿（aliasing）
  - 使用有限个数据对连续信号进行采样
  - 采样过程中受限于采样率，造成信息丢失
- 锯齿包括
  - 锯齿状的边
  - 渲染细节不正确
  - 微小物体可能丢失
- 平滑的边并不容易

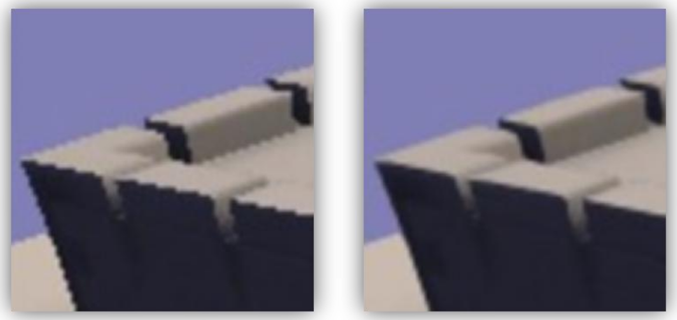


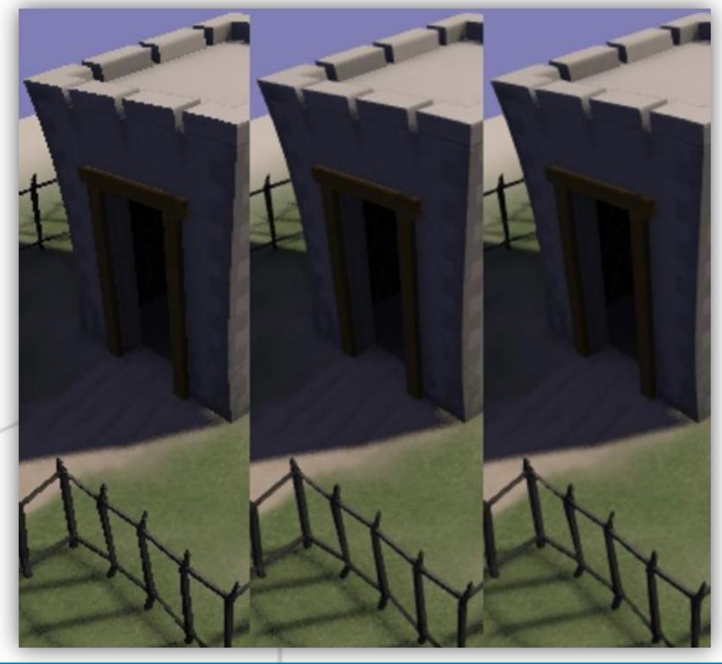
# 抗锯齿

- 对几何物体的离散化会产生锯齿（aliasing）
  – 使用有限个数据对连续信号进行采样
  – 采样过程中受限于采样率，造成信息丢失
- 锯齿包括
  – 锯齿状的边
  – 渲染细节不正确
  – 微小物体可能丢失
- 平滑的边并不容易





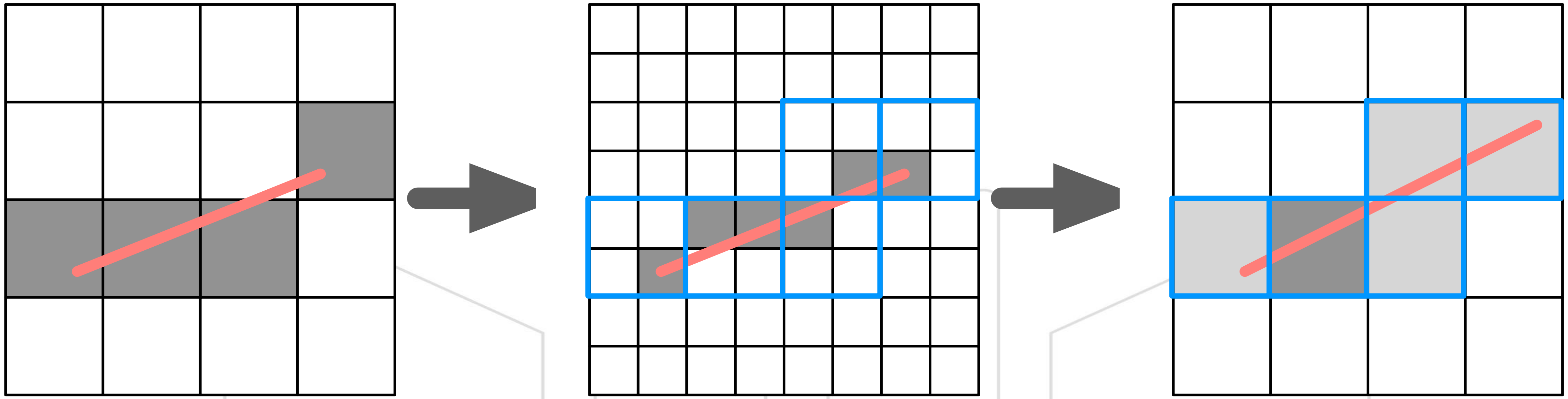




# 抗锯齿

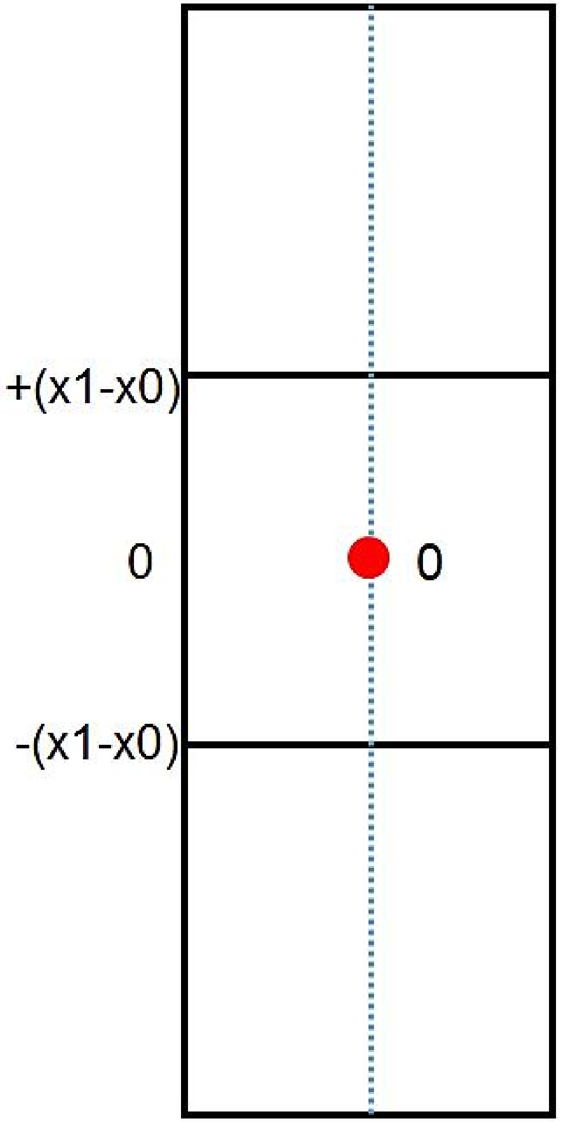


- 抗锯齿方法：super-sampling
  - 在高于原图的分辨率下进行重新采样
    - 2x2，4x4，或8x8等
  - 将线段端点置于相应位置
  - 使用线段光栅化方法在超采样图像中计算对应像素点
  - 将2x2（或4x4，8x8）的像素块合并成一个像素



# 抗锯齿

- 抗锯齿方法：ratio method
  - 根据每个像素到线段上的采样点的实际距离决定颜色比例



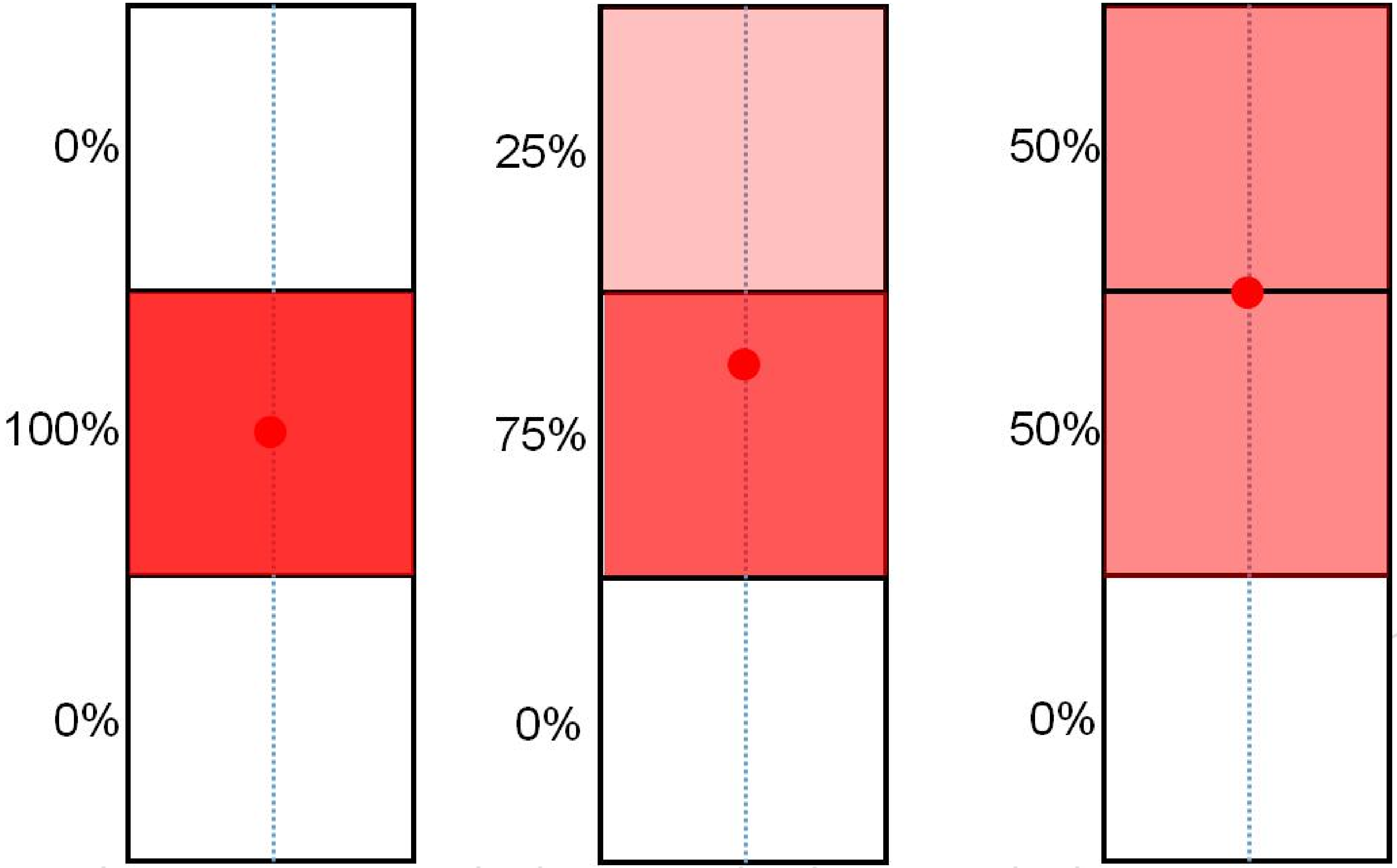

$$\left(.5 * MAX\left(\frac{error}{x_1 - x_0}, 0\right)\right) RGB$$

$$\left(1.0 - .5 * abs\left(\frac{error}{x_1 - x_0}\right)\right) RGB$$

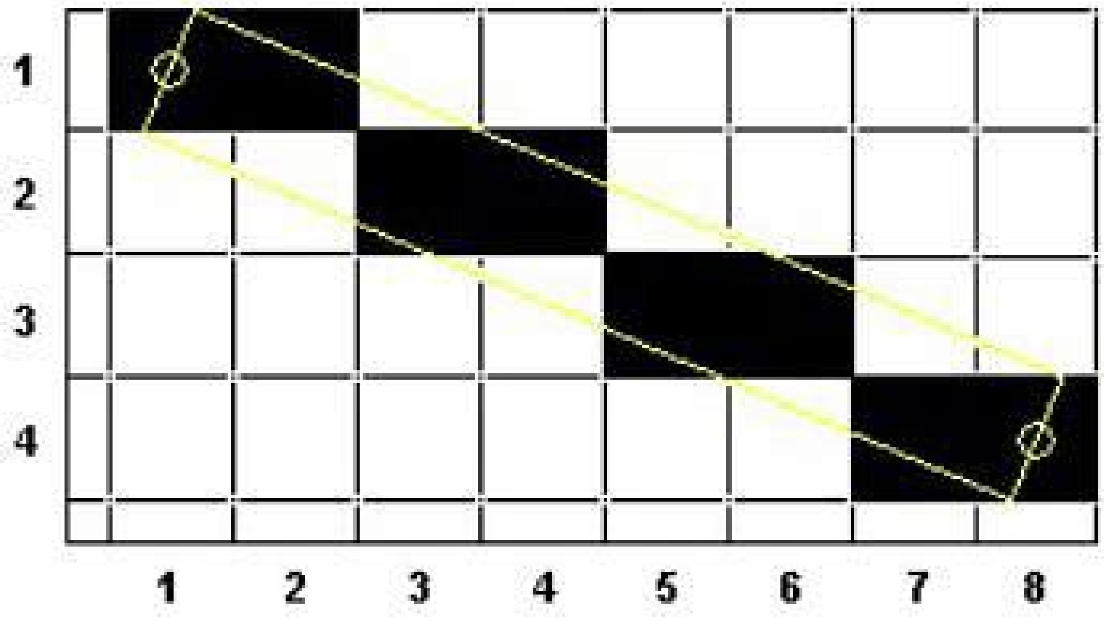
$$\left(.5 * MAX\left(\frac{-error}{x_1 - x_0}, 0\right)\right) RGB$$

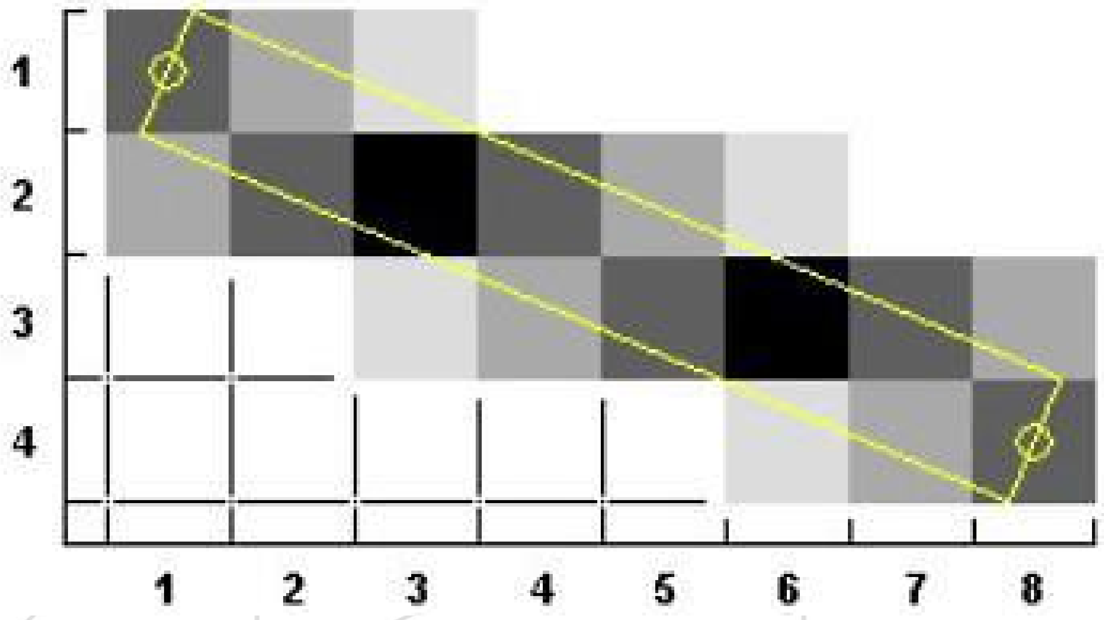
# 抗锯齿

- 抗锯齿方法：ratio method
  - 根据每个像素到线段上的采样点的实际距离决定颜色比例



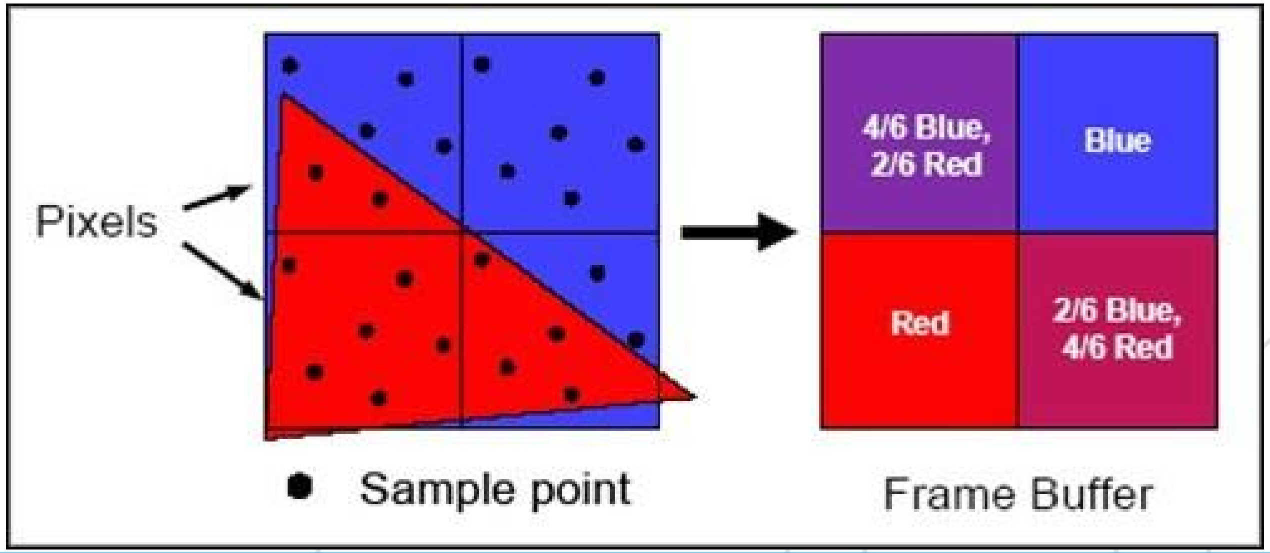
- 抗锯齿方法：area sampling
  - 扫描一个几何图元所覆盖的像素
    - 如，使用长方体替换无宽度的线
    - 像素颜色可依据离边界的距离进行调整（weighted area sampling）



# 抗锯齿

- 显卡上常见的抗锯齿设置
  - 多重采样抗锯齿：multi-sampling anti-aliasing (MSAA)
    - 是一种特殊的超级采样抗锯齿（SSAA）
    - MSAA首先来自于OpenGL，具体是MSAA只对Z缓存（Z-Buffer）和模板缓存(Stencil Buffer)中的数据进行超级采样抗锯齿的处理
    - 可以简单理解为只对多边形的边缘进行抗锯齿处理。这样的话，相比SSAA对画面中所有数据进行处理，MSAA对资源的消耗需求大大减弱，不过在画质上可能稍有不如SSAA



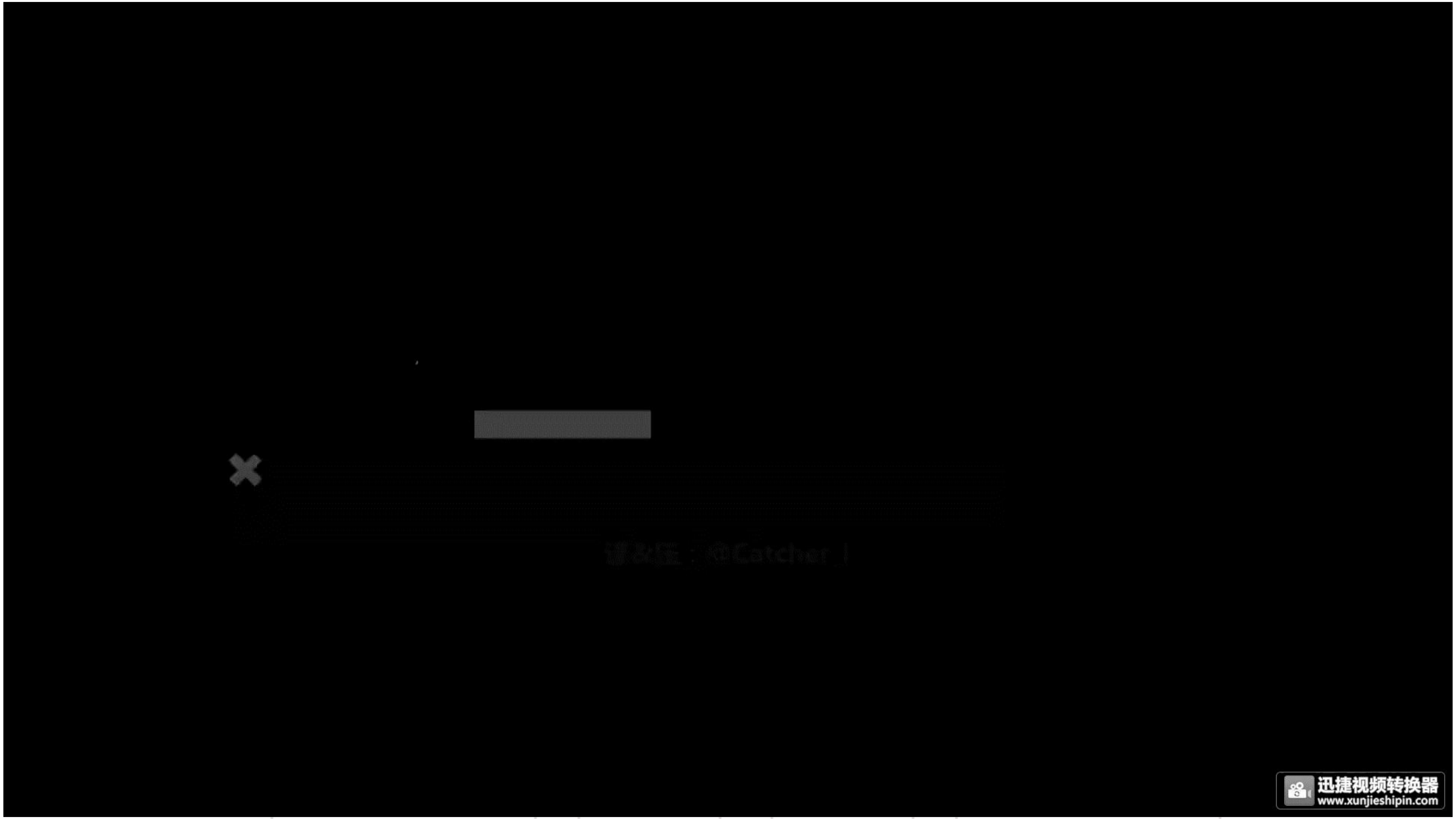

# 抗锯齿

- 显卡上常见的抗锯齿设置
  - 自适应抗锯齿：adaptive anti-aliasing (AAA)
    - MSAA有一个重要缺陷就是不能处理透明纹理，因此在一些栅栏、树叶、铁丝网等细长的物体上就不能起作用了
    - 为了解决这种问题，ATI在X1000系列加入了自适应（Adaptive）抗锯齿
    - 自适应抗锯齿可专门针对透明纹理选择性地进行多级或是超级采样，这样就比完全采用SSAA拥有更低的性能损失

# 抗锯齿

- 显卡上常见的抗锯齿设置
  - 快速近似抗锯齿：fast approximate anti-aliasing (FXAA)
    - 传统MSAA（多重采样抗锯齿）效果的一种高性能近似值
    - 只是单纯的后期处理着色器，不依赖于任何GPU计算API
    - FXAA技术对显卡没有特殊要求，完全兼容NVIDIA、AMD的不同显卡（MLAA仅支持A卡）和DX9、DX10、DX11
    - 完全使用图像处理的方式进行模糊


# 抗锯齿



# Questions?