光栅图形学

高林

中国科学院计算技术研究所

2023-03-23

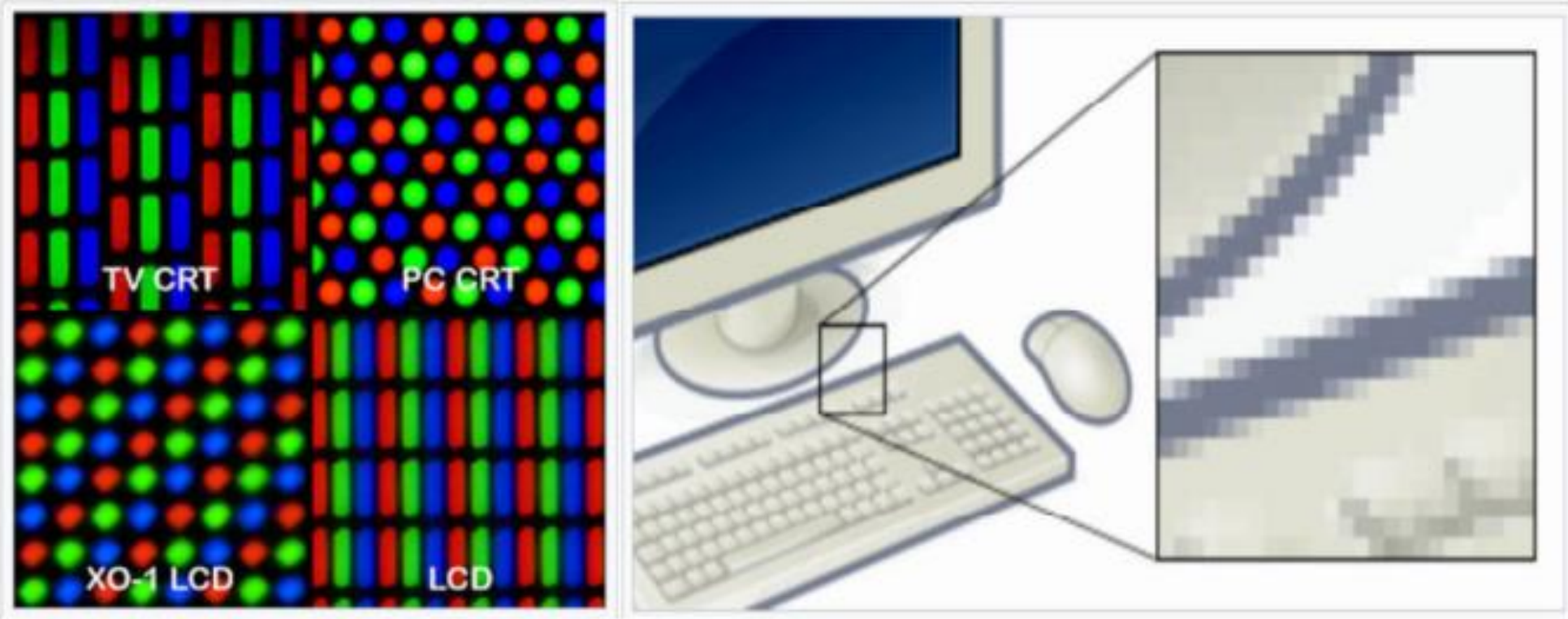
光栅图形显示技术

- 光栅显示器上的图像是由光栅(raster)形成的；
- 光栅是一组互相平行的水平扫描线（scanline），每行扫描线是由大小一致的显示单元组成的显示序列，每一显示单元称为一个像素(picture element, pixel)，可显示给定的颜色或者灰度；
- 光栅显示器将显示图元(primitive)如线、文字、填充颜色或图案区域等，以像素的形式存储到一个刷新缓冲器(refresh buffer)中。

光栅图形显示技术

- 像素（pixel）
  - 最小可编程物理单元
  - 具体的物理形式有不同形式
  - 单一物理单元/复合物理单元（三原色）
  - 四边形，六角形

光栅图形显示技术 （续）

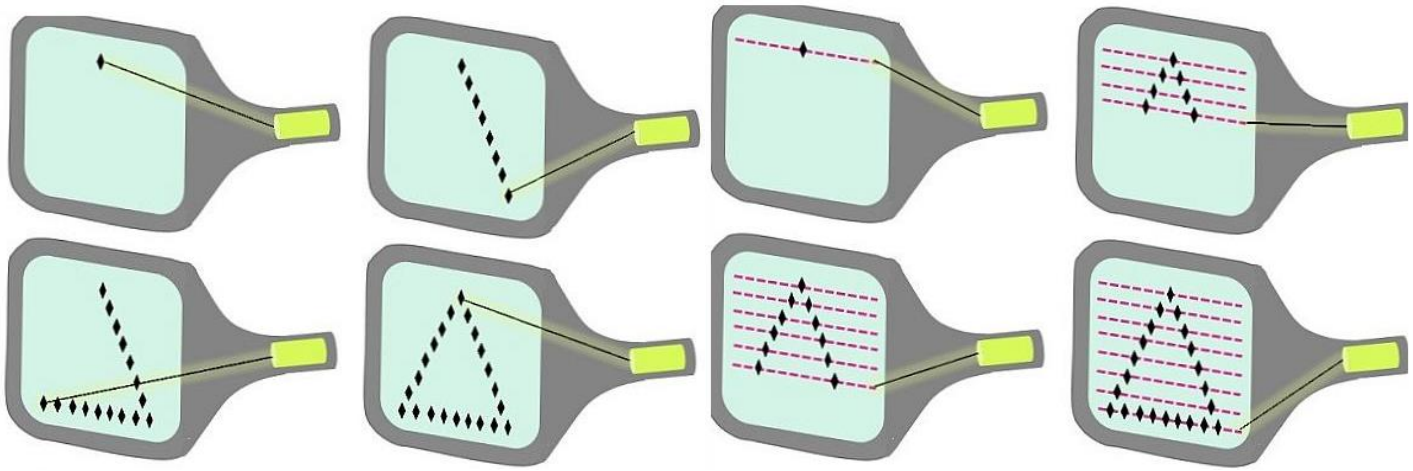


光栅图形显示技术 （续）

- 图元（primitive）
  - 描述图形元素的基本函数，可直接调用；
  - 点，直线……；
  - 但圆不是；
- 缓冲器(buffer)
  - refresh ~
  - Z~

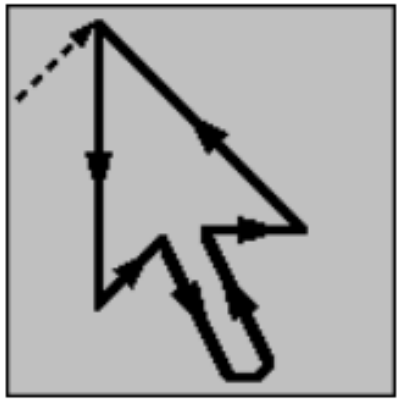
随机扫描与光栅扫描

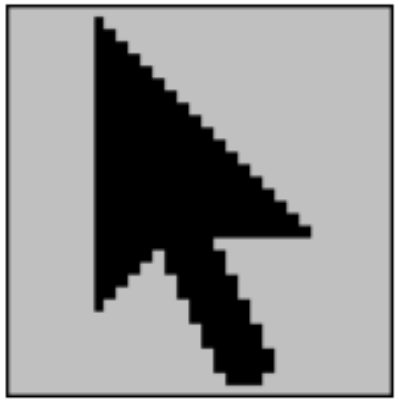
- 随机扫描技术
  - 按照显示命令的任意顺序，将电子束从一个端点偏转到另一个端点；
  - 单枪顺序扫描；
  - 屏幕上的点不可编程；
- 光栅图形技术
  - 像素可编程；
  - 可并行处理；
  - 可对显示区域填充颜色或图案；
  - 存储的图像易于操作；



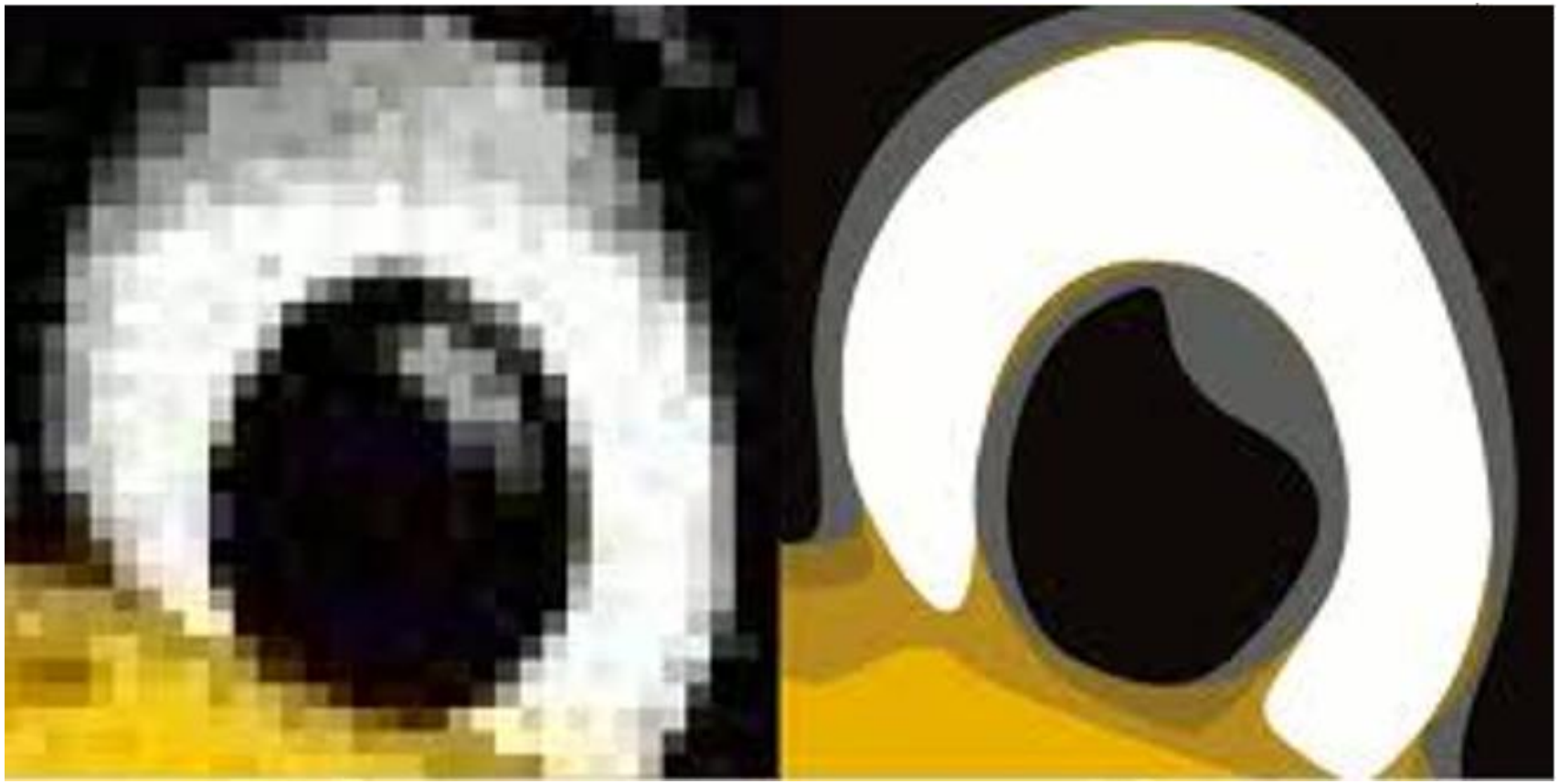
随机扫描技术　　光栅扫描技术

两种扫描方法对比





两种扫描方法对比

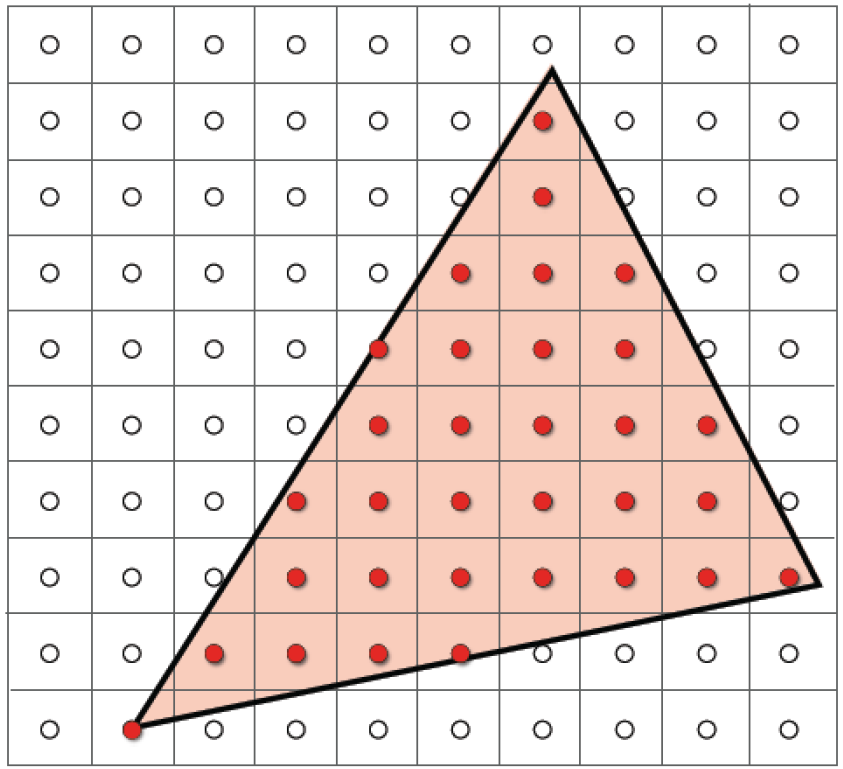


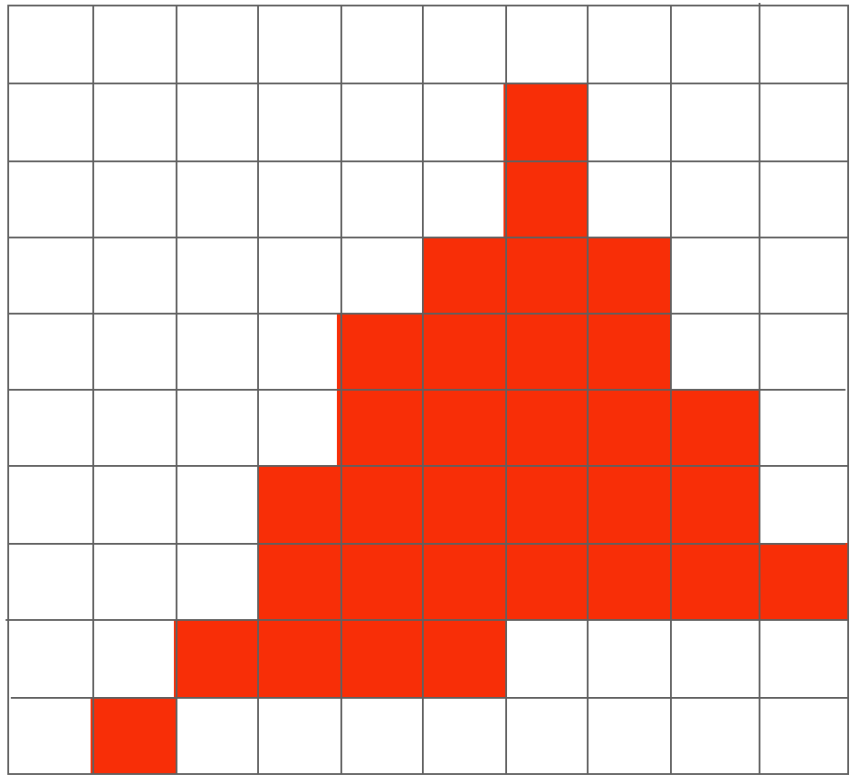
走样与反走样

- 对于光栅系统来说，用光栅网格上的像素近似地描绘平滑的直线、多边形和如圆、椭圆等曲 线图元的边界。
- 这会引起锯齿状与阶梯状边界 称为“走样”(aliasing)。
- 抑制或消除这种现象的技术称为“反走样” （antialiasing）。

图形光栅化的走样(Aliasing)

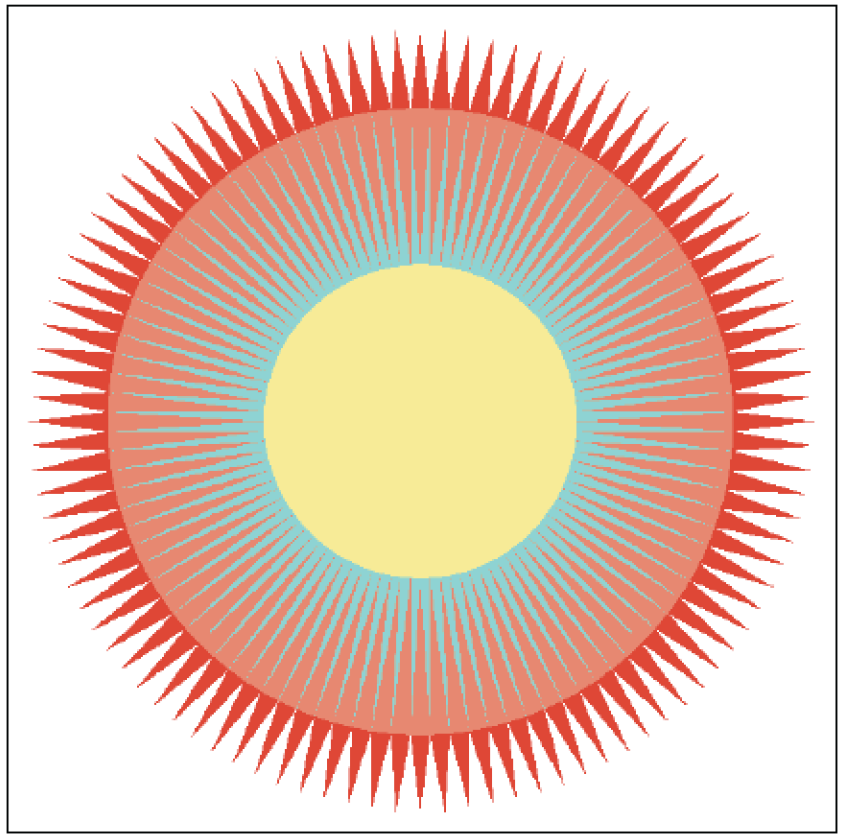
- 锯齿现象(Jaggies)

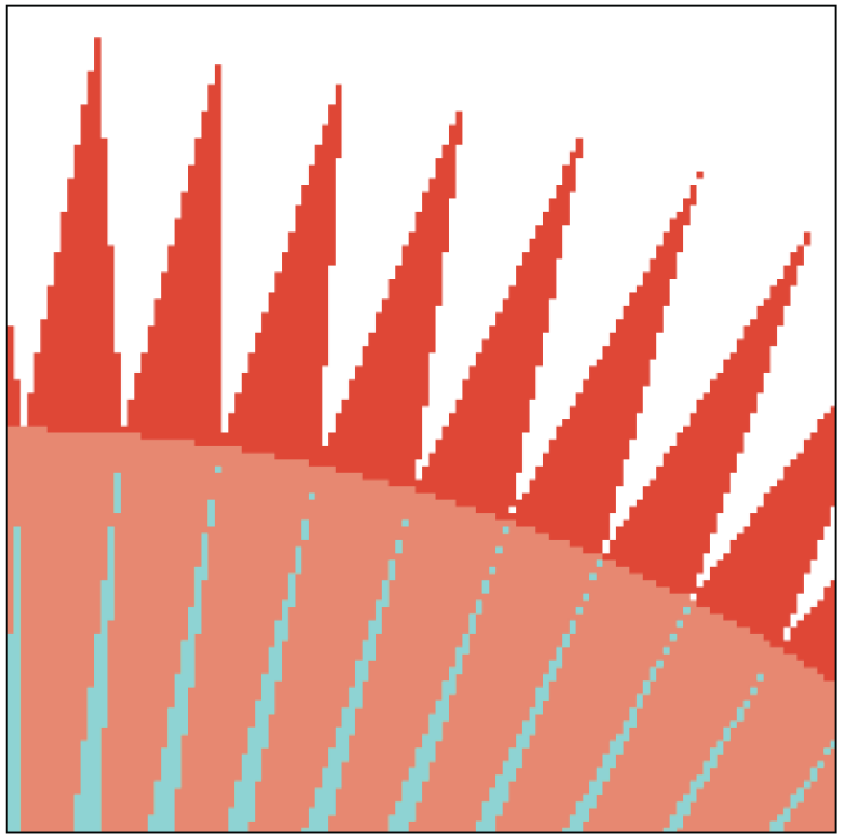




图形光栅化的走样(Aliasing)

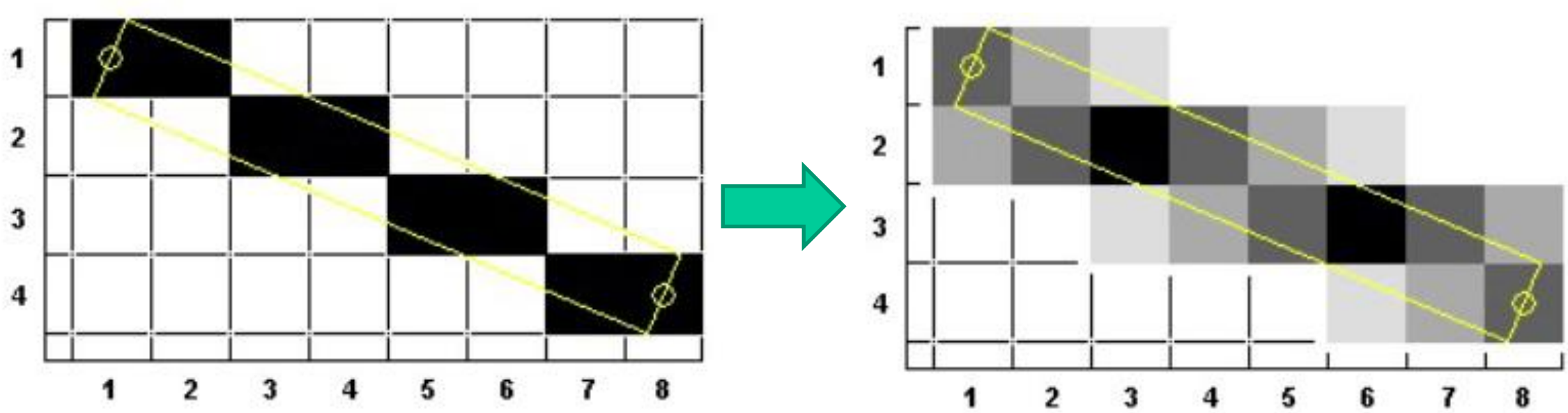
- 如何解决或者缓解?



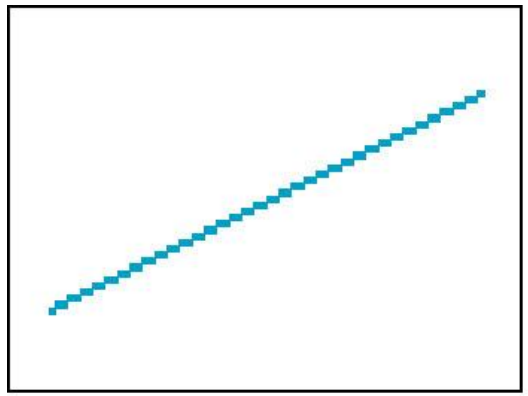


走样现象的缓解

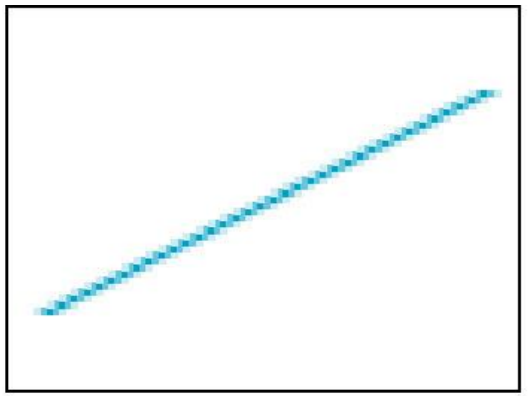
- 非0即1的颜色?
- 引入中间颜色值，缓解视觉突兀感



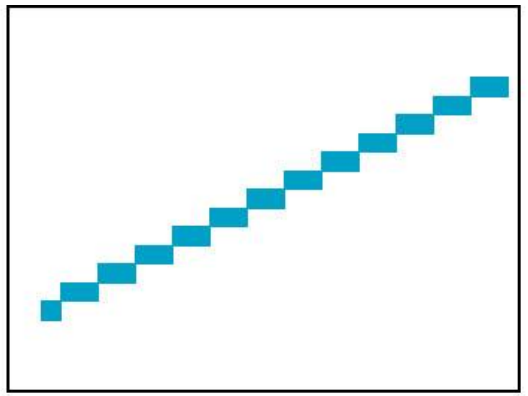
反走样(Anti-Aliasing)

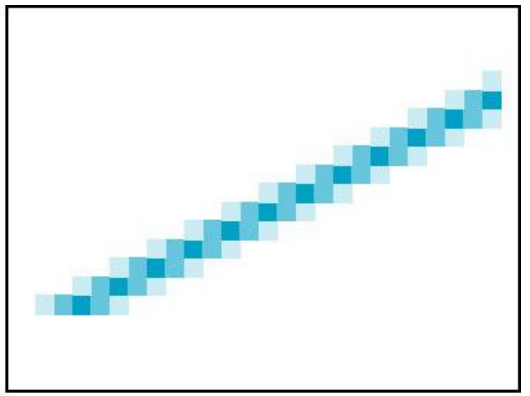


初始形状



反走样的结果

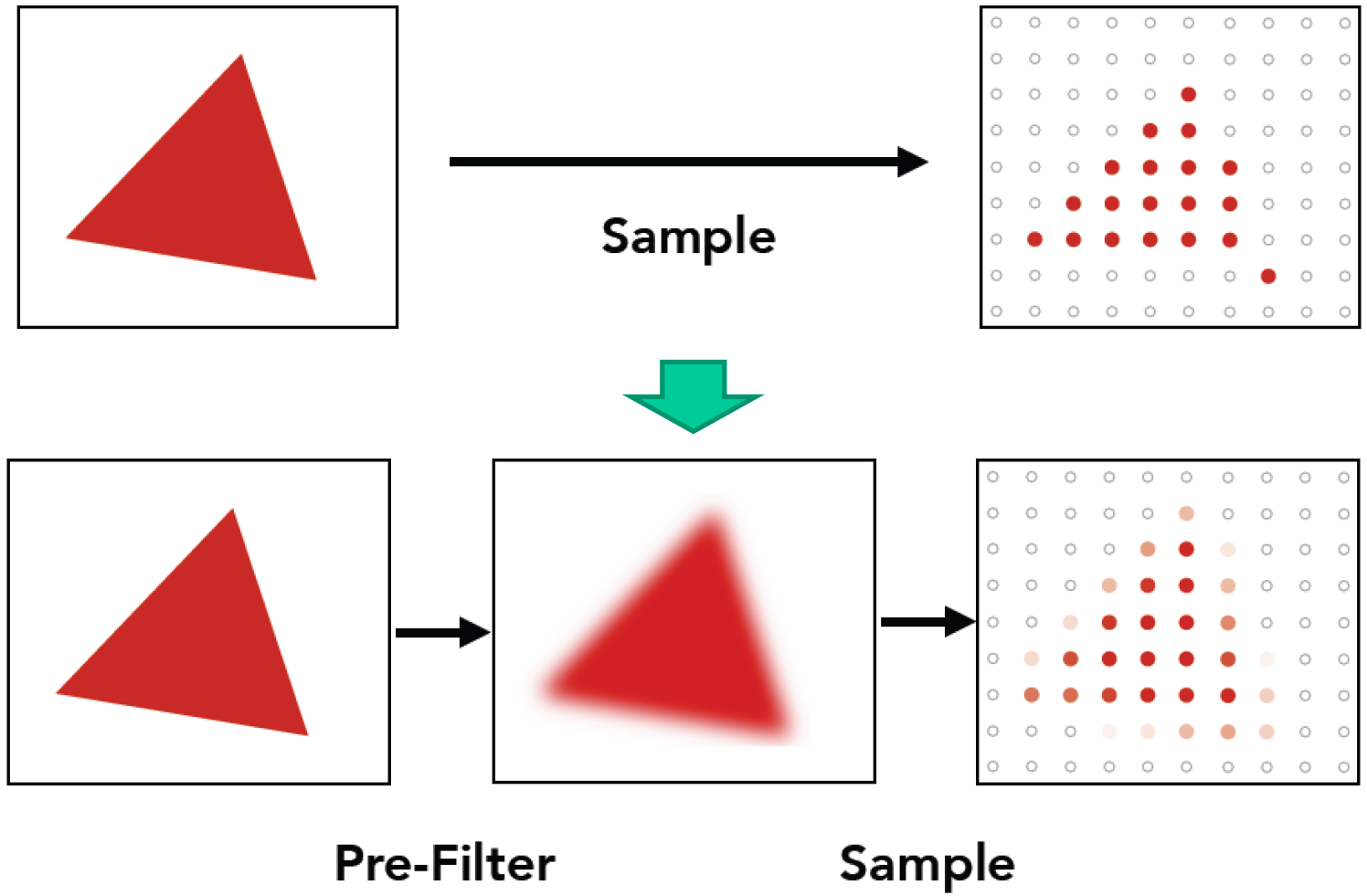




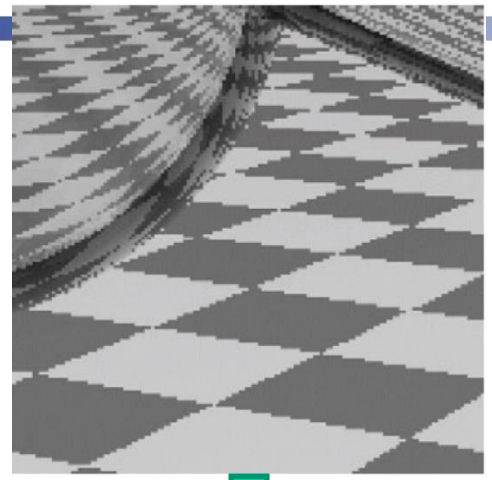
放大显示的结果

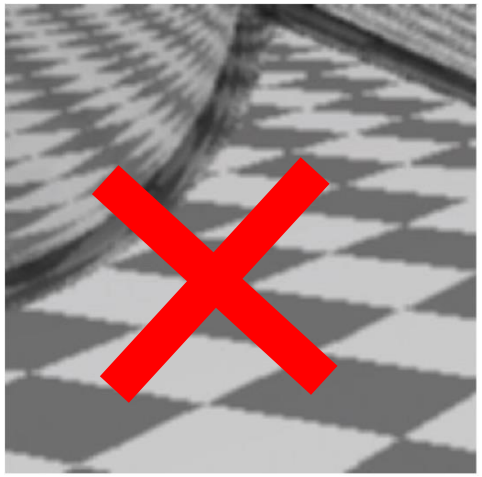
反走样方案

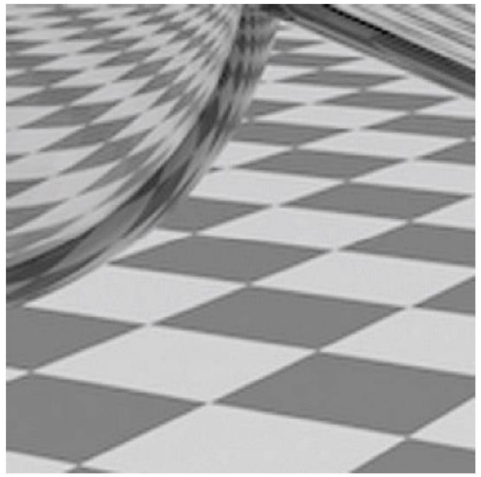
- 在采样前模糊（预过滤）



反走样样例







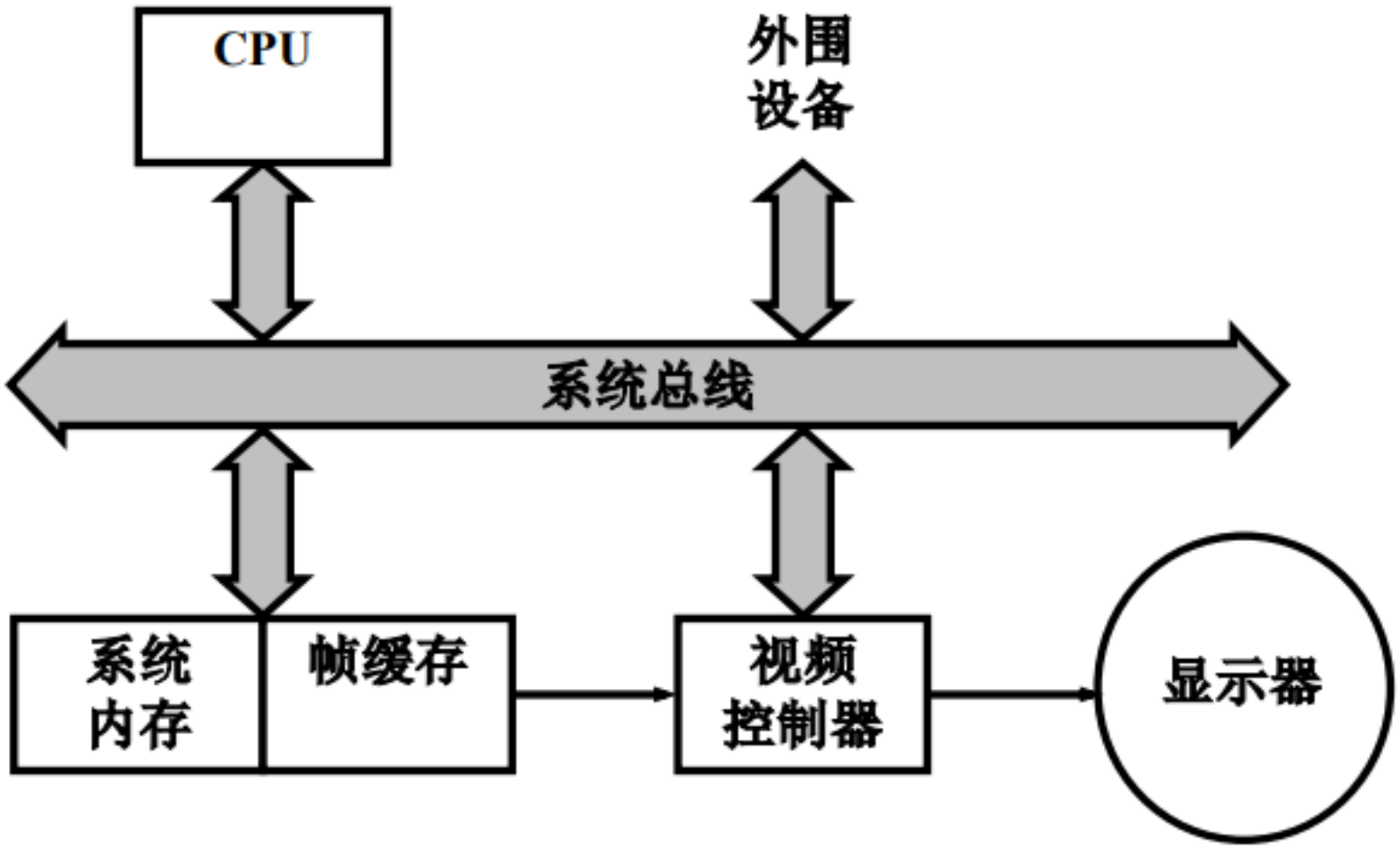
先采样后过滤

先过滤后采样

光栅化

- 如何使光栅图形最完美地逼近实际图形，是光栅图形学要研究的内容。
- 确定最佳逼近图形的像素集合，并用指定的颜色和灰度设置像素的过程叫做图形的扫描转换（scan converting），或者称作“光栅化”（rasterization）。

光栅显示系统的结构

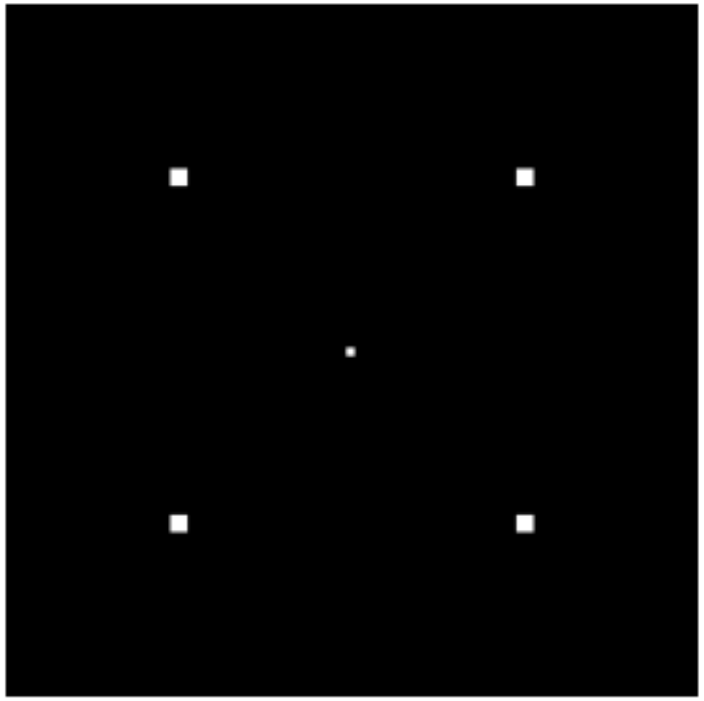


简单二维图元的生成

- 作为图形硬件的软件接口，OpenGL最主要的工作就是将二维及三维物体绘制到帧缓存;
- 绘制过程主要有以下3步:
  - 定义几何要素，构建物体在计算机内的表示；确定各物体在三维空间中的方位，选取场景观察点;
  - 计算物体表面需要显示的颜色，这些颜色可以直接赋值，或根据光照条件及表面纹理计算得到;
  - 光栅化，把物体的几何描述和颜色信息转换为屏幕的像素。

OpenGL环境下点的绘制

- OpenGL允许采用glPointSize()函数由用户自定义点在屏幕上的显示尺度（缺省宽度为1.0）。



OpenGL环境下直线的绘制

- 直线由两端点定义。只要把glBegin()的参数设为GL_LINES，OpenGL即可通过 glBegin()/glEnd()函数对来绘制直线。
- 此时函数对中的第一个顶点为第一条直线的起点，第二个顶点为该直线的终点，第三个顶点则为第二条直线的第一个顶点，依次类推。

直线的扫描转换算法

- 直线的扫描转换算法
  - 基本增量算法 (digital differential analyser, DDA)
  - Bresenham算法

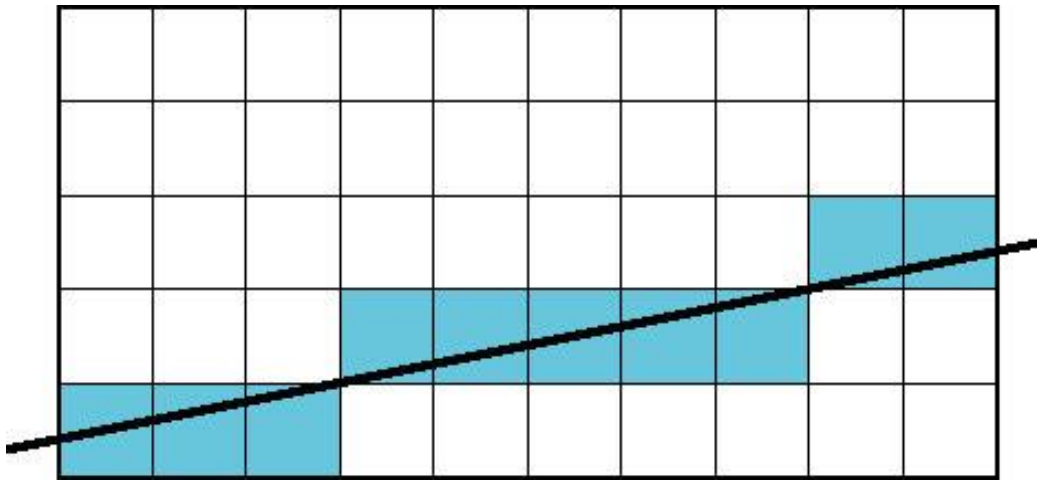
- 其中Bresenham算法是计算机图形学领域使用最为广泛的直线扫描转换算法

(1) 基本增量算法（DDA算法）

- DDA: Digital Differential Analyzer 数字微分分析法
- 直线 $y = mx + h$满足微分方程 $dy/dx = m = Dy/Dx = (y2 - y1)/(x2 - x1)$
- 沿扫描线 Dx = 1

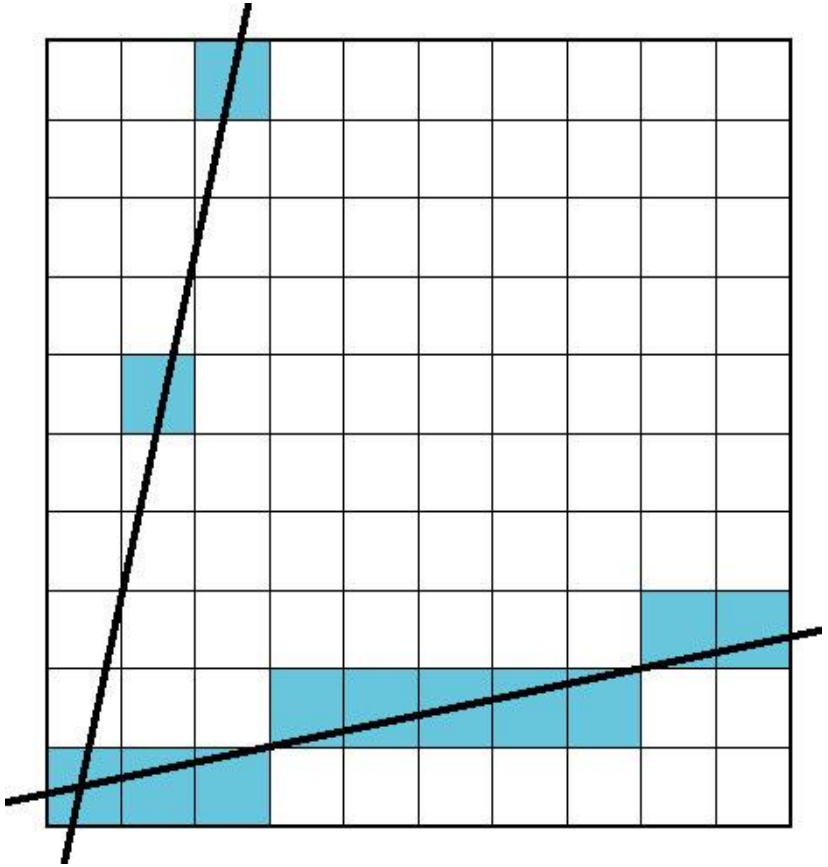
```
for(x = x1; x<=x2; x++) {
        y+=m;
        write_pixel(x, round(y));
}
```

- 对于每个x画出最接近的整数y



问题：斜率大的直线

- x增加1，y增加m



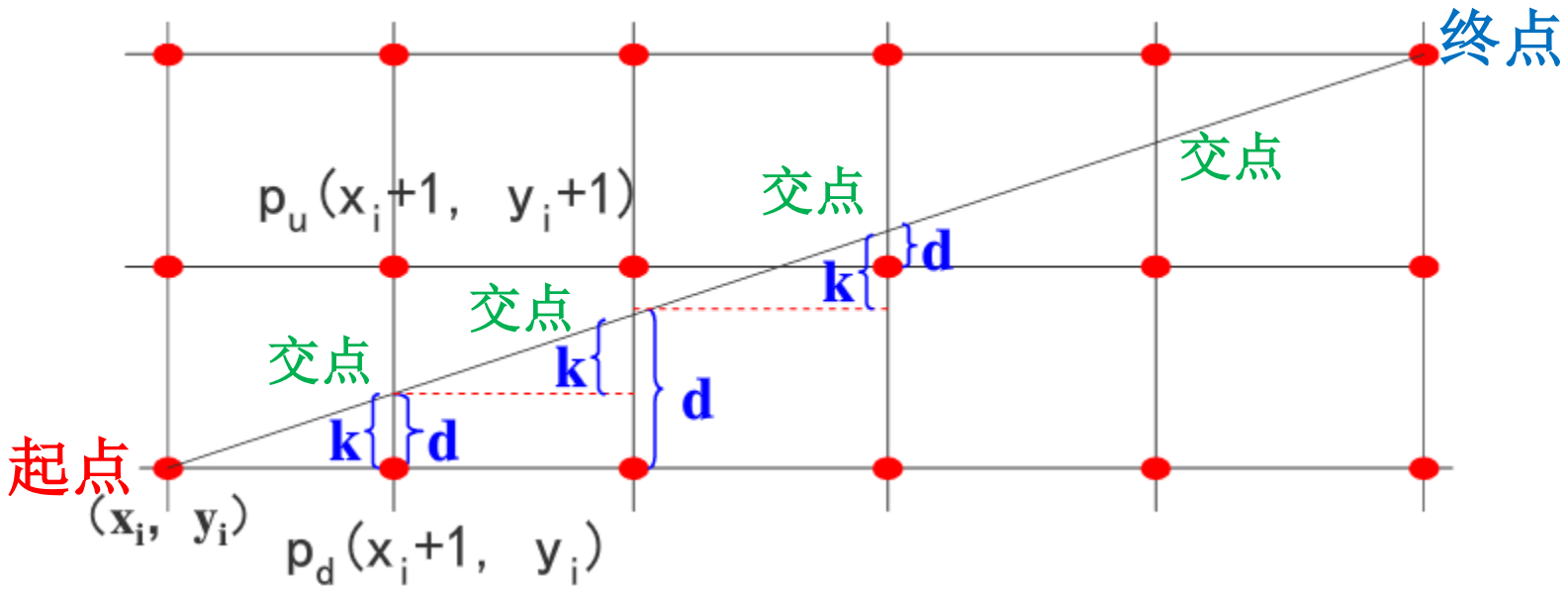
利用对称性，交换x与y的角色

(2) Bresenham算法

- 所有的计算都是简单的整数运算
  - 加法、减法、乘以 2（移位）
- 保证线是连续的，其中每个点都精确绘制 1 次
- 也称为中点线算法

(2) Bresenham算法

- 该算法的思想是通过各行、各列像素中心构造一组虚拟网络线。按照直线**起点**到**终点**的顺序，计算直线与各垂直网络线的**交点**，然后根据误差项的符号确定该列像素中于此交点最近的像素。



(2) Bresenham算法

- 每次x+1，y的递增（减）量为0或1，它取决于实际直线与最近光栅网格点的距离，这个距离最大误差为0.5。

- 误差项d的初值为0，d=d+k,一旦d>=1,就把它减去1，保证d的相对性，且在0、1之间。

$$\begin{cases} x_{i+1} = x_i + 1 \\ y_{i+1} = \begin{cases} y_i + 1 & (d>0.5) \\ y_i & (d\leqslant 0.5) \end{cases} \end{cases}$$

- 使用整数加法可以提高算法效率，如何实现?

(2) Bresenham算法（改进1）

- 令e=d-0.5

$$\begin{cases} x_{i+1} = x_i + 1 \\ y_{i+1} = \begin{cases} y_i + 1 & (e>0) \\ y_i & (e\leqslant 0) \end{cases} \end{cases}$$

- e>0，y方向递增1；e<0，y方向不递增。
- e=0时，可任取上下光栅点显示。
- $e_{init} = -0.5$
- 每走一步有$e = e + k, k = \frac{dy}{dx}$
- if $e > 0.5$: $e = e - 1$

由于计算机底层特性，判断e>0或e<0的速度快于判断d>0.5或d<0.5

(2) Bresenham算法（改进2，基于改进1）

- 算法中只用到误差项e的符号，于是用$e' = e * 2 * dx$来替换e
- 原因：e的初值和每一步的迭代都是浮点数，改成整数更方便运算
- $e'_{init} = -dx$

dx = Xend - Xstart, dy = Yend - Ystart

- 每走一步有：$e' = e' + 2 * dy$
  - 公式推导:
    - e = e + dy/dx
    - e' = e' + (dy/dx) * 2 * dx = e' + 2 * dy
- if $e' > dx$: $e' = e' - 2 * dx$

(2) Bresenham算法（改进2）

- 算法步骤

这里的e指代上一页的e’

  - 1. 输入直线的两端点$P_0(x_0, y_0)$和$P_1(x_1, y_1)$
  - 2. 计算初始值$dx, dy, e = -dx, x = x_0, y = y_0$
  - 3. 绘制点(x, y)
  - 4. e更新为$e + 2 * dy$，判断e的符号。若e>0，则(x, y)更新为(x+1, y+1)，否则更新为(x+1, y)；另判断e>dx，若成立，则e = e - 2 * dx
  - 5. 当直线没有画完时，重复步骤34

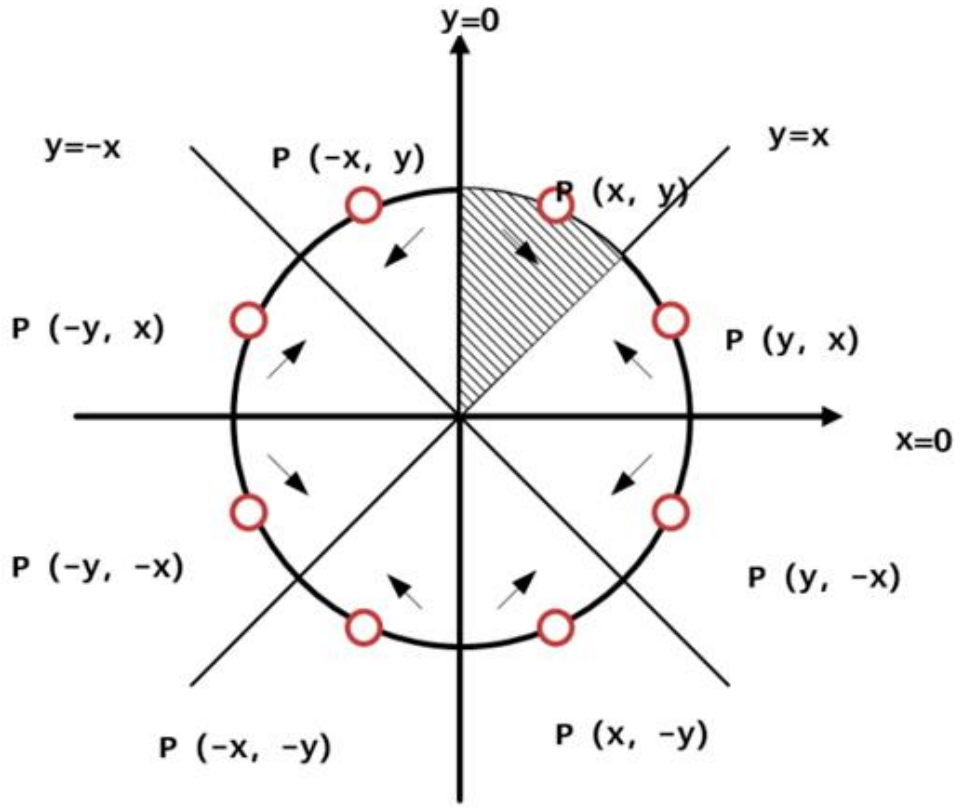
联系与区别

- Bresenham算法很像DDA算法，都是加斜率，
- 但DDA算法是每次求一个新的y以后取整来画，而Bresenham算法是判断符号来决定上下两个点。
- 所以Bresenham算法集中了DDA算法的优点，而且应用范围广泛。

圆的扫描转换

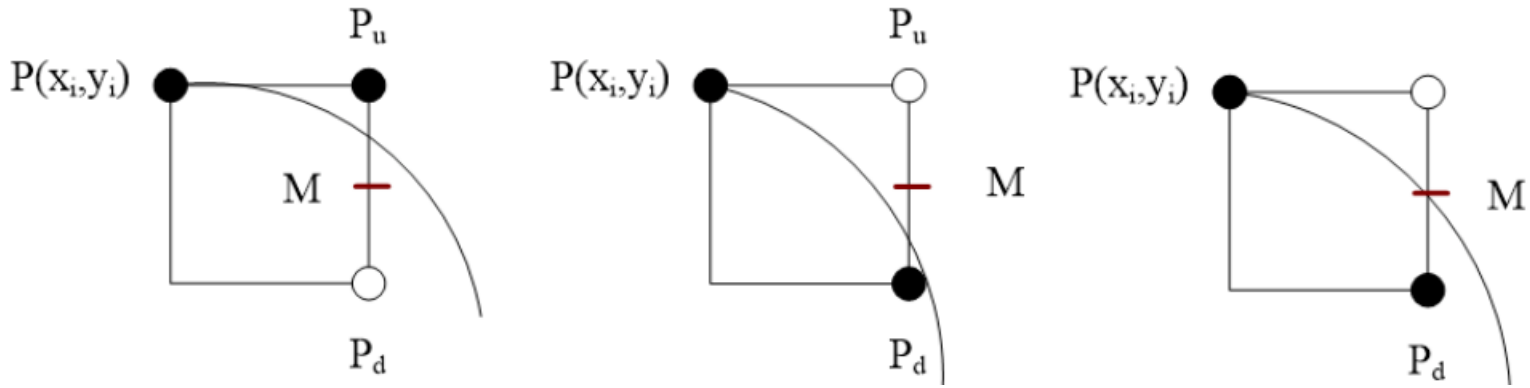
- 选取$x=0, y=0, y=x$和$y=-x$四条直线作为对称轴将圆八等分，只需画出一段圆弧其余七段可以直接对称获得。

- 对于第一象限$y=x$上方的圆弧，任意点
- 的斜率大小不超过1。
- 因此若当前圆弧在(x_i, y_i)像素上，那么
- 在x_i+1位置上，纵坐标只能是y_i或者
- y_i-1



圆的扫描转换

- 选取两种情况的中点$(x_i+1, y_i-0.5)$，若其在圆内说明纵坐标距离y_i更近，否则选取y_i-1



- 于是转而判断
- $d_i=(x_i+1)^2+(y_i-0.5)^2-r^2$与0的关系

$$y_{i+1}=\begin{cases} y_i, & d<0 \\ y_i-1, & d\geq 0 \end{cases}$$

- 继续考虑递推：
- 若$d_i<0, d_{i+1}=(x_i+2)^2+(y_i-0.5)^2-r^2=d_i+2x_i+3$
- 若$d_i\geq 0, d_{i+1}=(x_i+2)^2+(y_i-1.5)^2-r^2=d_i+2(x_i-y_i)+5$

圆的扫描转换

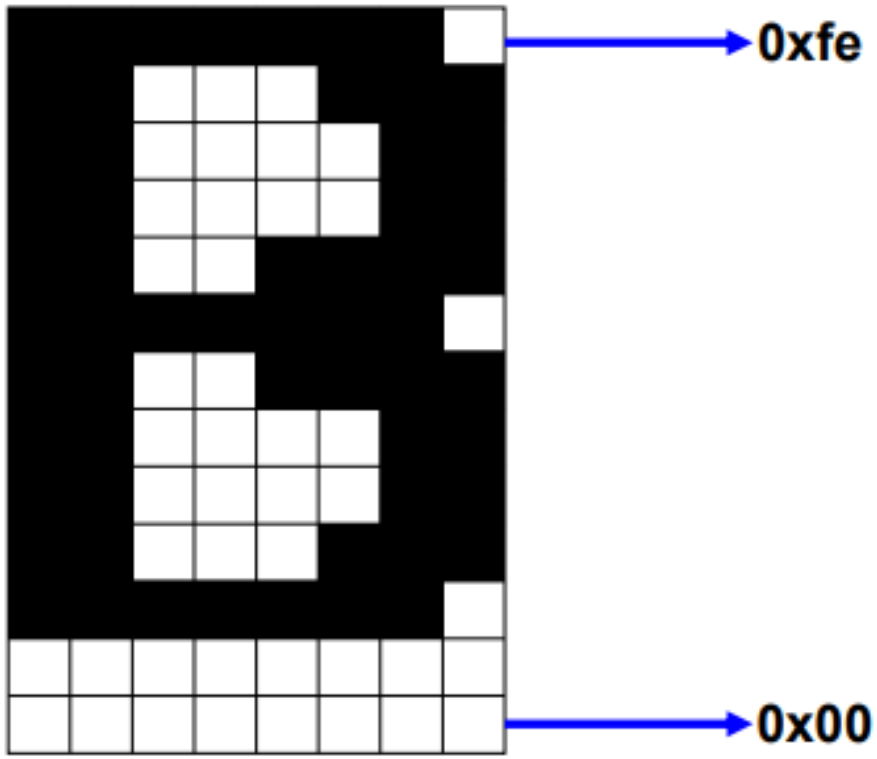
- 初始条件为：$x_0 = 0, y_0 = r, d_0 = 1 + (r - 0.5)^2 - r^2 = 1.25 - r$

- 可得算法步骤：
- 1. 设置x, y, d, r初始值如上
- 2. 若横坐标x已经大于纵坐标y，说明脱离八分之一圆弧，退出。
- 3. 绘制点(x, y)
- 4. 判断d是否小于0，若是，则迭代为d+2*x+3，否则迭代为d+2*(x-y)+5并且将y减一，回到步骤2

字符的生成

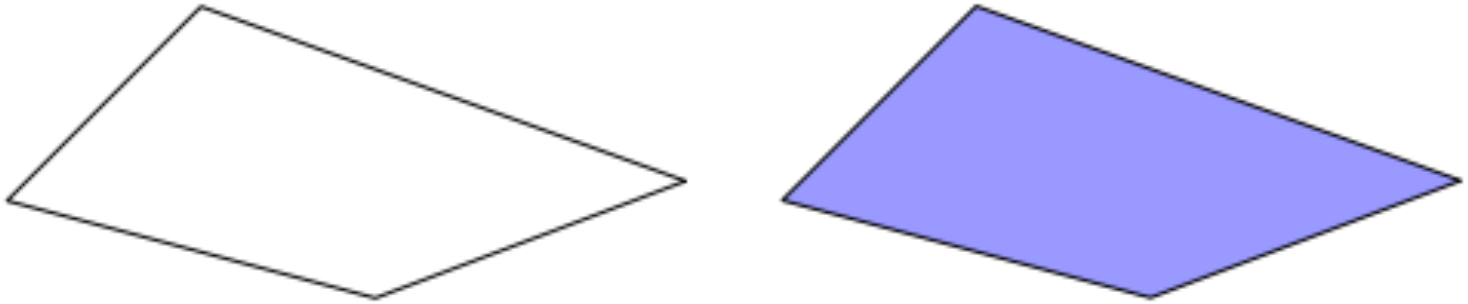
- 生成字符有两种基本的方法。
  - 将字符定义为一条曲线或多边形的轮廓，然后进行扫描转换，计算开销很大。
  - 对于给定某种字体的每一个字符，生成一个小型的矩形位图。位图中该位为1表示字符的笔画经过此位， 该位为0表示字符的笔画不经过此位。
  - 字库设计。

字母“B”的位图显示



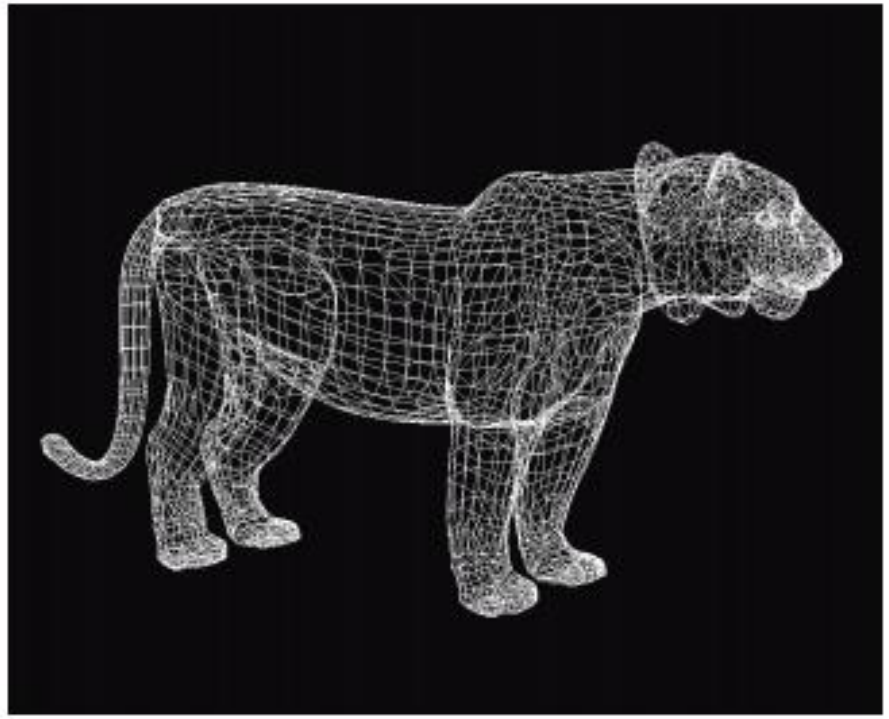
光栅图形的基本概念

- 关于光栅图形
  - 本质：点阵表示
  - 特点：面着色，画面明暗自然、色彩丰富
  - 与线框图相比：更加生动、直观、真实感强

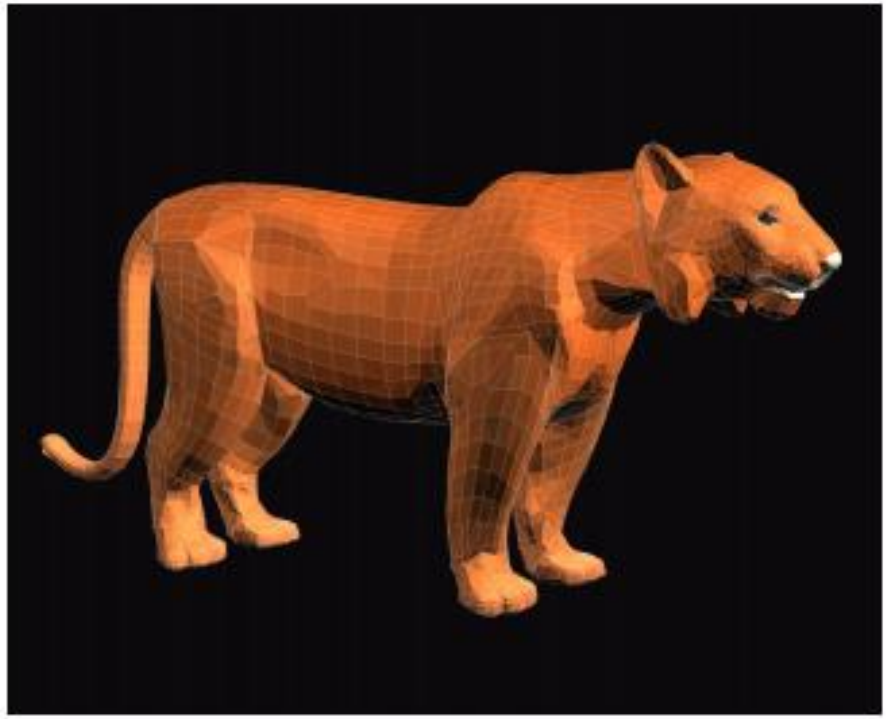


线框平面多边形　　着色的平面多边形

光栅图形的基本概念



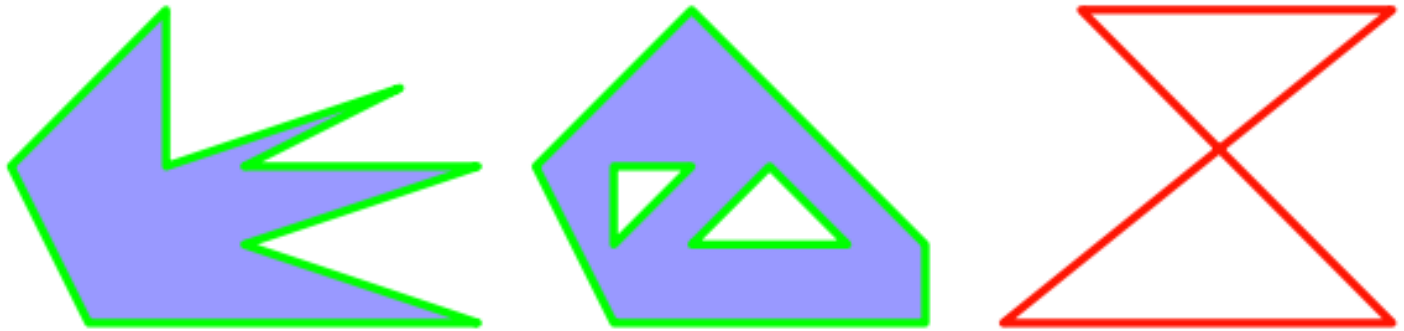
线框多边形物体



填充多边形物体

光栅图形的基本概念

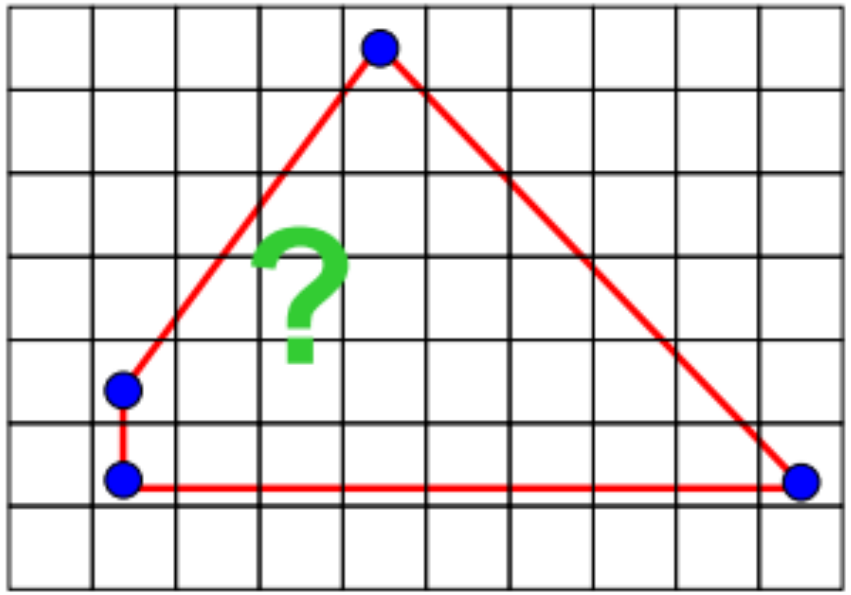
- 关于平面多边形
  - 图形学中的多边形：无自相交的简单多边形



  - 图形学中多边形的两种表示方式
    - 顶点表示：用多边形的有序顶点序列表示多边形
    - 点阵表示：用位于多边形内部的像素集合来表示多边形

顶点表示

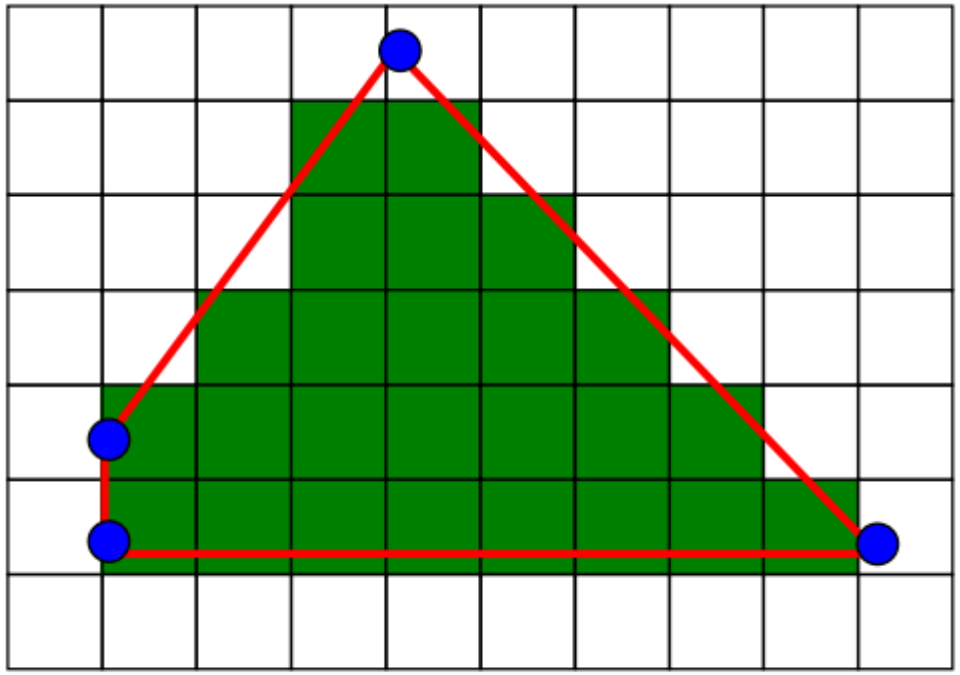
- 优点
  - 直观
  - 几何意义明显
  - 存贮量小
- 不足
  - 难以判断哪些像素
  - 位于多边形内部
  - 不能直接用于多边形着色



多边形的顶点表示

点阵表示

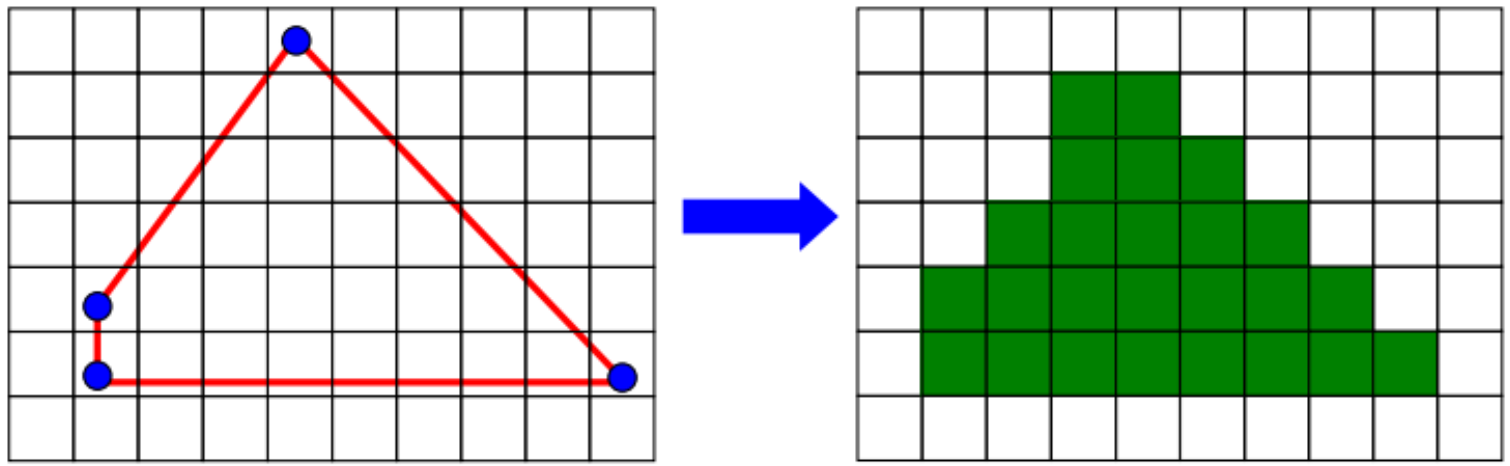
- 优点
  - 便于用帧缓冲器 (frame buffer) 表示图形
  - 面着色所需的图形表示
- 缺点
  - 丢失了几何信息
  - 占用存储空间多



多边形的点阵表示

多边形的扫描转换

- 多边形的扫描转换：
- 把顶点表示转换为点阵表示
  - 从多边形的给定边界出发，求出其内部的各个像素
  - 并给帧缓冲器中各个对应元素设置相应灰度或颜色



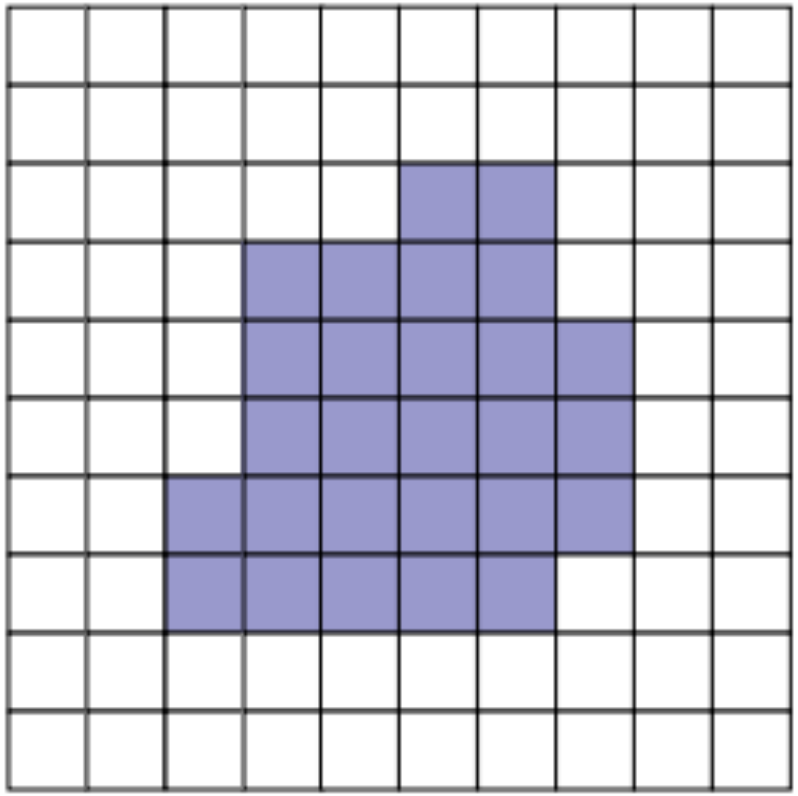
多边形的顶点表示　　多边形的点阵表示

点阵表示的区域填充

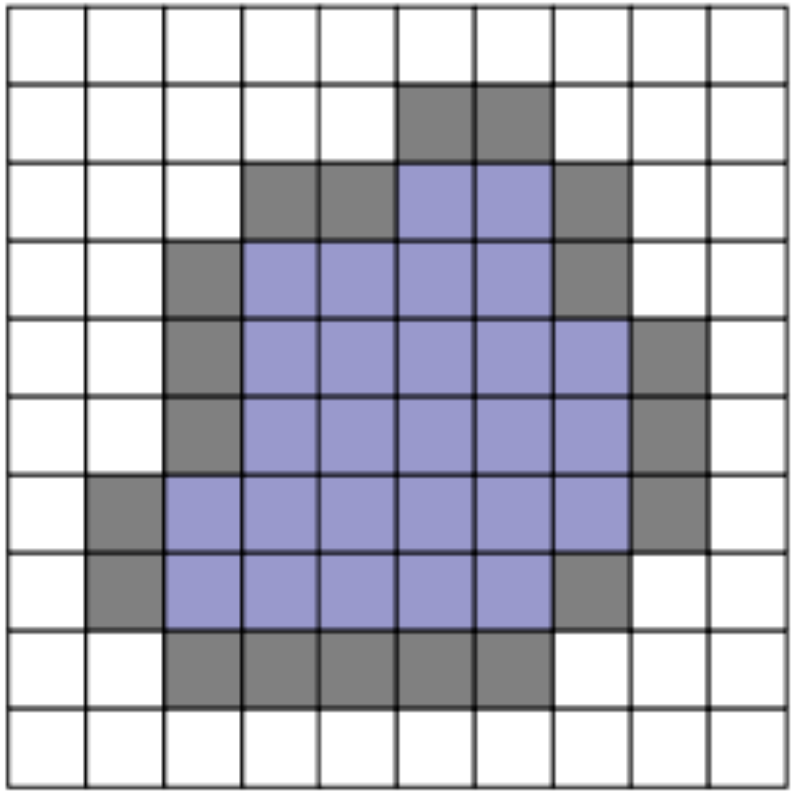
- 区域的定义：已经表示成点阵的像素集合
- 区域的表示：
  - 内部表示：把给定区域内的像素枚举出来
    - 区域内所有像素都着同一种颜色
    - 区域边界像素不能着上述颜色
  - 边界表示：把区域边界上的像素枚举出来
    - 边界上所有像素都着同一种颜色
    - 区域内部像素不能着上述颜色

区域表示

- 内部表示

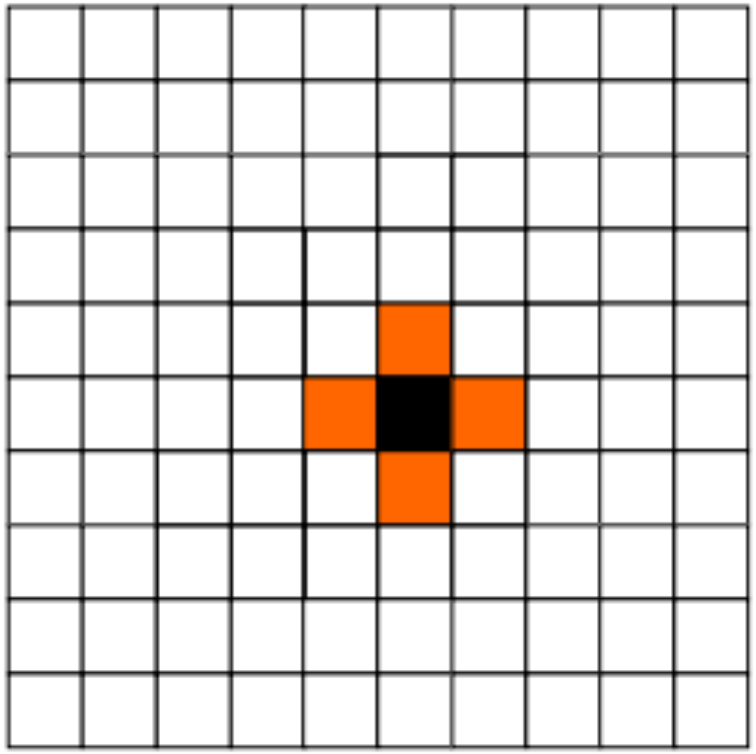


- 边界表示

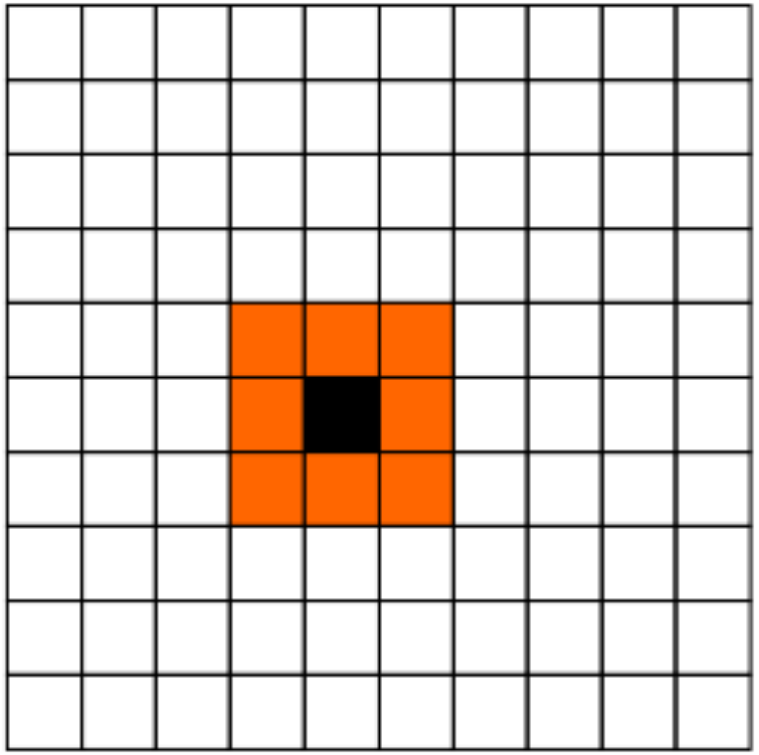


区域的类型

- 四连通邻域



- 八连通邻域

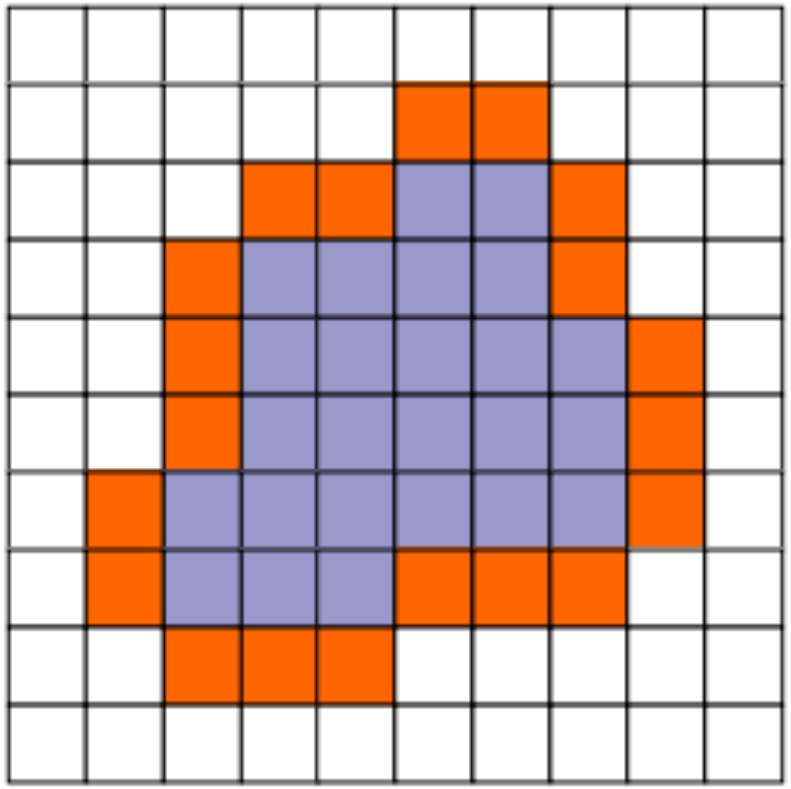


区域的类型

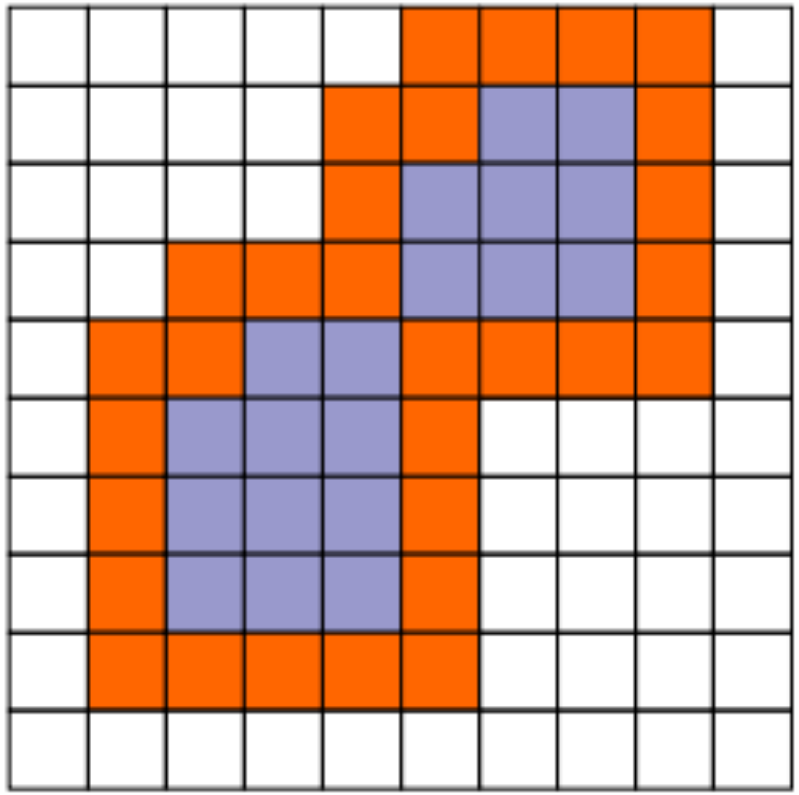
- 四连通区域：区域内任意两个像素，从一个像素出发，可以通过上、下、左、右四种运动， 到达另一个像素
- 八连通区域：区域内任意两个像素，从一个像素出发，可以通过水平、垂直、正对角线、反对角线八种运动，到达另一个像素

区域的类型

- 四连通区域实例



- 八连通区域实例



内部表示区域种子填充算法

- 假设内部表示区域为G，其中的像素原有颜色为 G0，需要填充的颜色为G1。
- 算法需要提供一个种子点(x,y)，它的颜色为G0。
- 具体算法如下(四连通区域)。

内部表示区域种子填充算法

```
Flood_Fill_4(x, y, G0, G1)
{
        if(GetPixel(x,y) ==G0 ) // GetPixel(x,y) 返回(x,y)的颜色
        {
                SetPixel(x, y, G1); // 将(x,y)的添上颜色G1
                Flood_Fill_4(x-1, y, G0, G1);
                Flood_Fill_4(x, y+1, G0, G1);
                Flood_Fill_4(x+1, y, G0, G1);
                Flood_Fill_4(x, y-1, G0, G1);
        }
}
```

边界表示区域种子填充算法

```
Fill_Boundary_4_Connnected(x, y, BoundaryColor, InteriorColor)
// (x,y) 种子像素的坐标;
// BoundaryColor 边界像素颜色;  InteriorColor 需要填充的内部像素颜色
{
        if(GetPixel(x,y) != BoundaryColor && GetPixel(x,y)!= InteriorColor ) // GetPixel(x,y): 返回像素(x,y)颜色
        {
                SetPixel(x, y, InteriorColor); // 将像素(x, y)置成填充颜色
                Fill_Boundary_4Connnected(x, y+1, BoundaryColor, InteriorColor);
                Fill_Boundary_4Connnected(x, y-1, BoundaryColor, InteriorColor);
                Fill_Boundary_4Connnected(x-1, y, BoundaryColor, InteriorColor);
                Fill_Boundary_4Connnected(x+1, y, BoundaryColor, InteriorColor);
        }
}
```

多变形扫描转换

- 逐点判断算法：

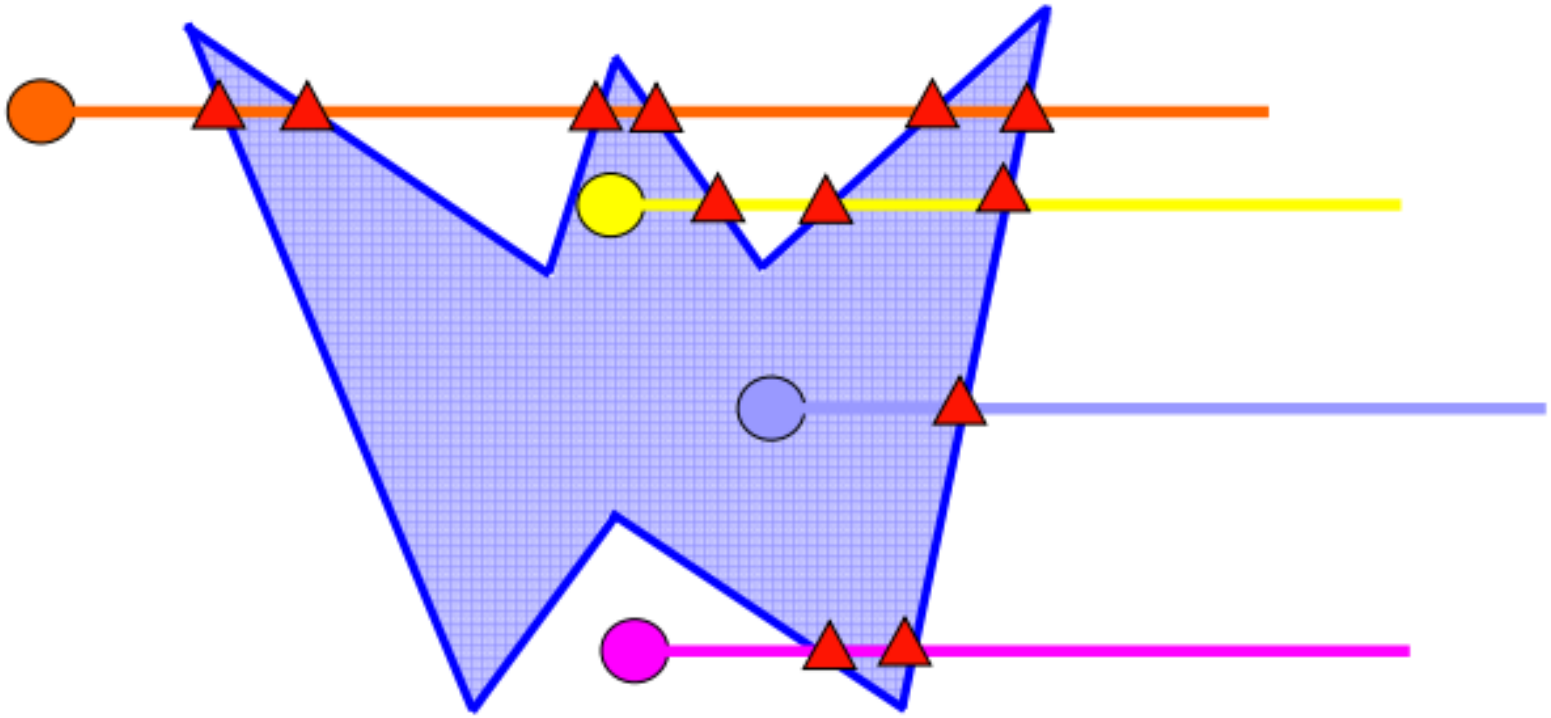
逐个像素判别其是否位于多边形内部 一个典型的计算几何问题

- 判断一个点是否位于多边形内部：射线法

从当前像素发射一条射线，计算射线与多边形的交点个数

  - 内部：奇数个交点
  - 外部：偶数个交点

逐点判断算法



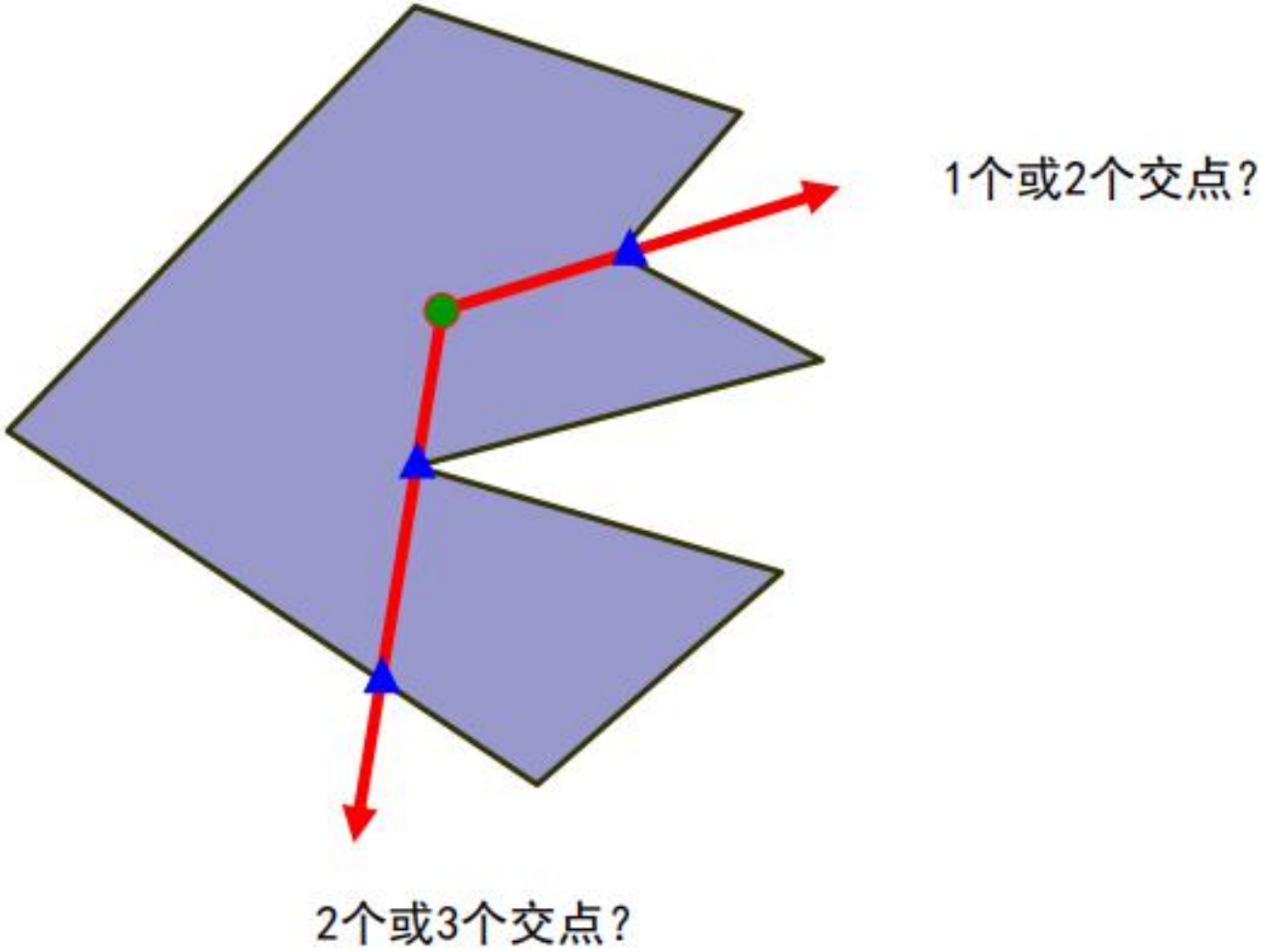
判断一点是否位于多边形内部？

逐点判断算法

- 算法描述

```
for(y=0; y<=y_resolution; y++)
for(x=0; x<=x_resolution; x++)
{
        if(inside(polygon, x, y))
                setpixel(framebuffer,x,y,polygon_color)
        else
                setpixel(framebuffer,x,y,background_color)
}
```

逐点判断算法中的奇异情况



逐点判断算法的不足

- 速度慢：几十万甚至几百万像素的多边形内外判断，大量的求交、乘除运算
- 没有考虑像素之间的联系

- 结论：逐点判断算法需要改进

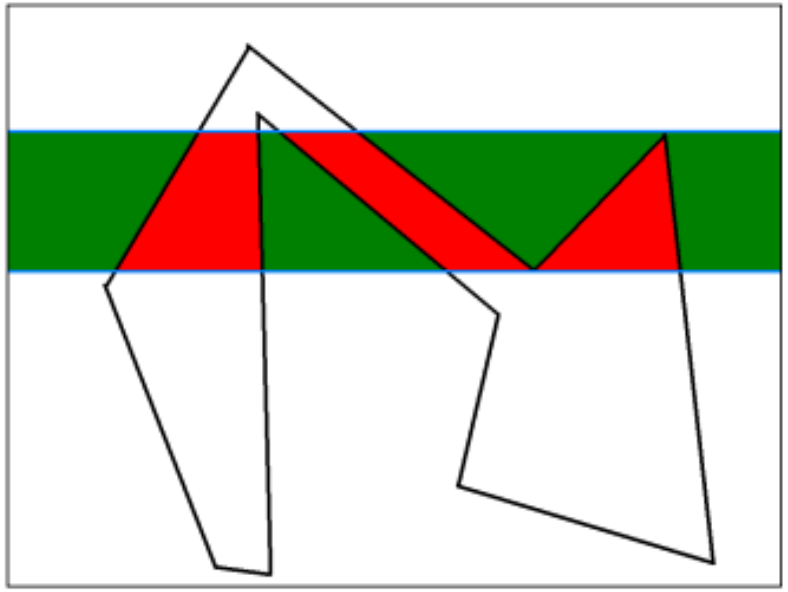
相邻像素之间的连贯性

- 扫描线算法充分利用了相邻像素之间的连贯性， 避免了对像素的逐点判断和求交运算，提高了算法效率

- 连贯性（coherence）
  - 区域连贯性
  - 扫描线连贯性
  - 边的连贯性

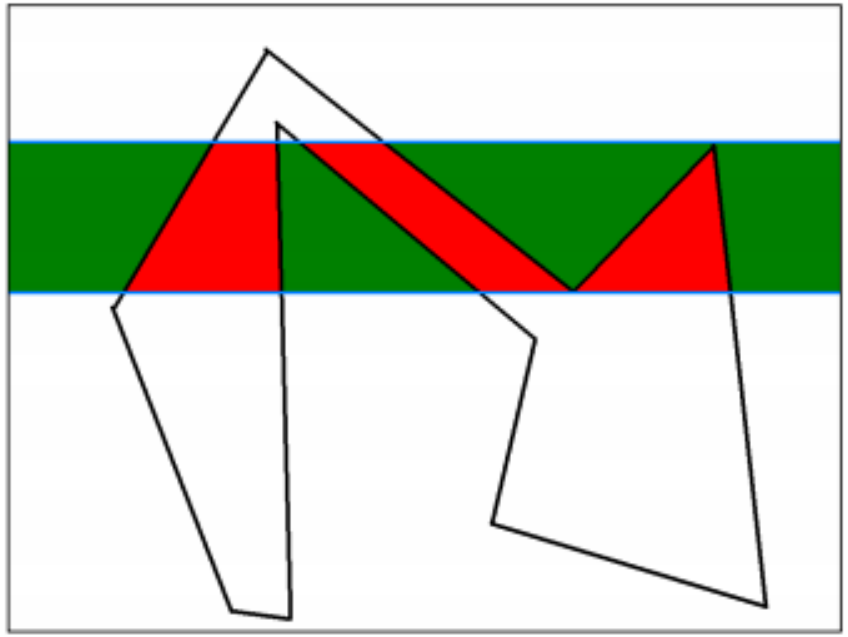
区域连贯性

- 区域的连贯性是指多边形定义的区域内部相邻的像素具有相同的性质。例如具有相同的颜色。



区域的连贯性

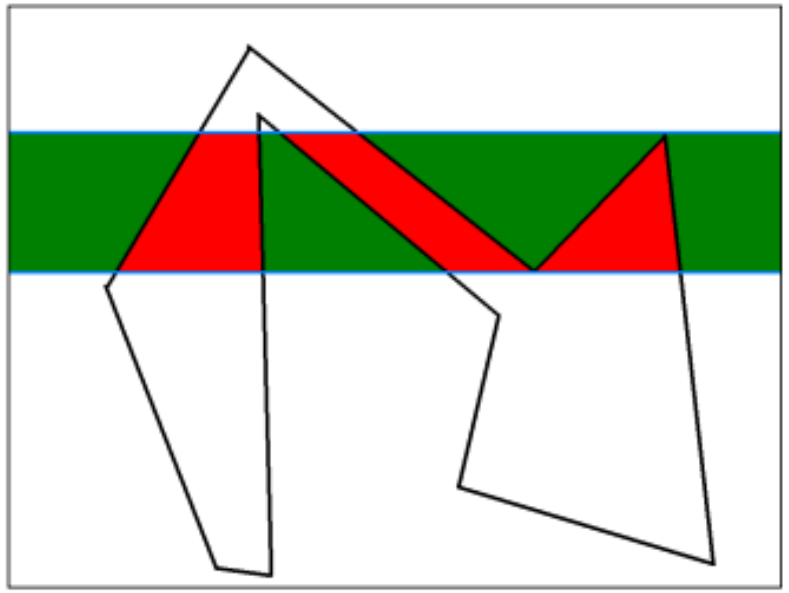
区域连贯性



区域的连贯性

- 两条扫描线之间的长 方形区域被所处理的 多边形分割成若干梯 形(三角形可以看作 退化梯形)
- 梯形的底边为扫描线， 梯形的腰为多边形的边或窗口边缘

区域连贯性



区域的连贯性

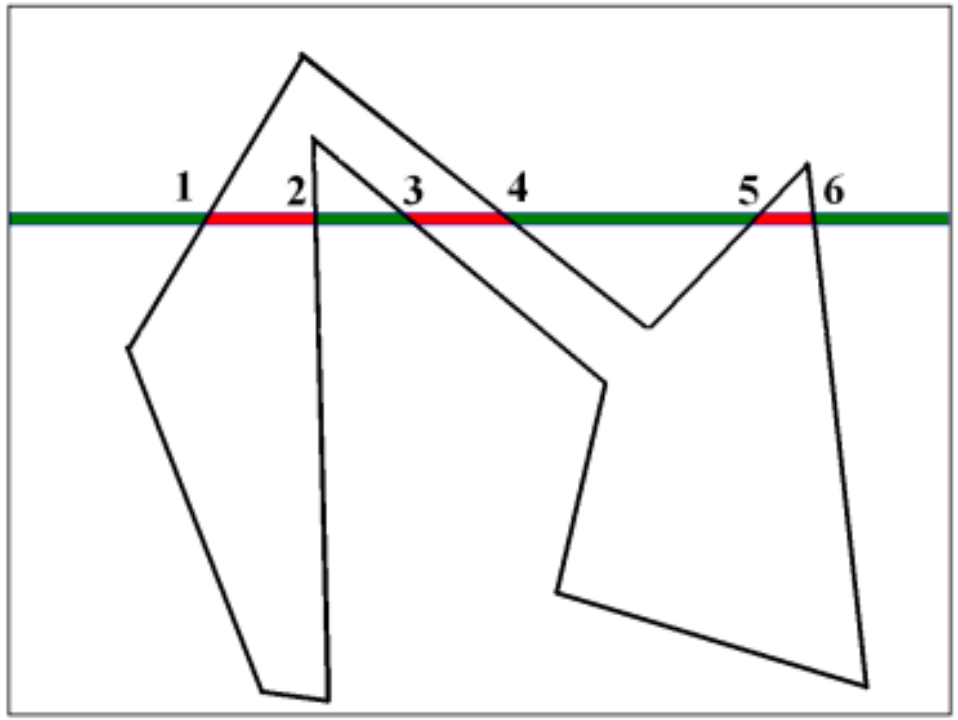
- 梯形分为两类：
  - 多边形内部和外部
- 两类梯形相间排列
  - 相邻梯形必有一个位 于多边形内部，另一 个在多边形外部

区域连贯性

- 推论：如果上述梯形属于多边形内(外)，那么 该梯形内所有点的均属于多边形内(外)。
- 逐点判断->区域判断

扫描线连贯性

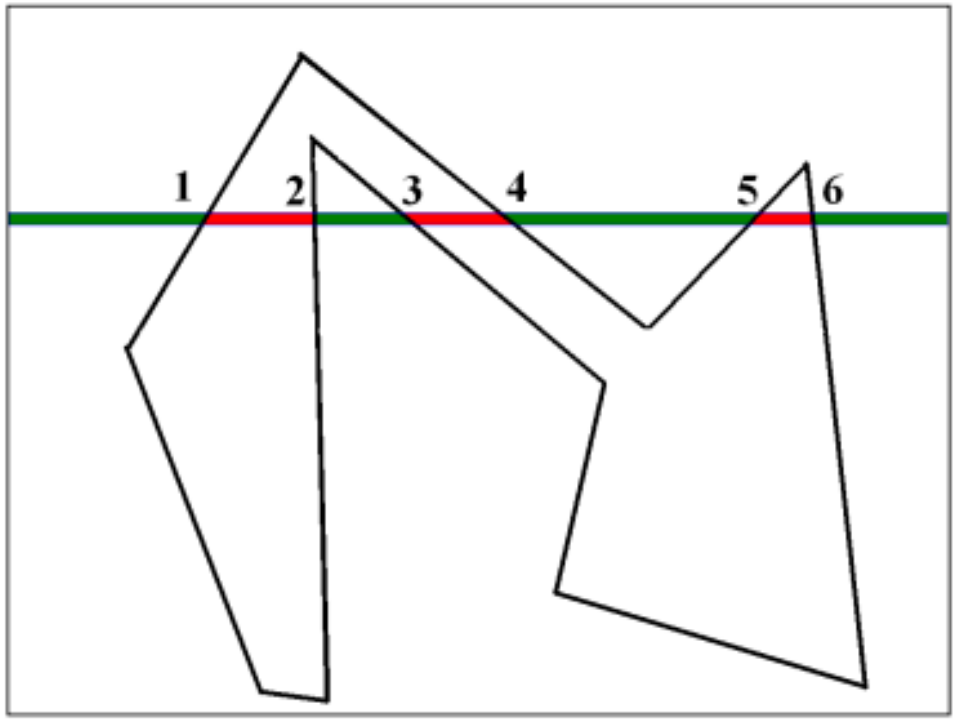
- 区域连贯性在一条扫描线上的反映



区域的连贯性

扫描线连贯性

- 区域连贯性在一条扫描线上的反映



区域的连贯性

- 交点序列：扫描线与多边形的交点个数为偶数(1,2,3,4,5,6)
- 红色区间(1,2)、 (3,4)、(5,6)位于多边形内部
- 其余绿色区间位于多边形外部
- 两类区间相间排列

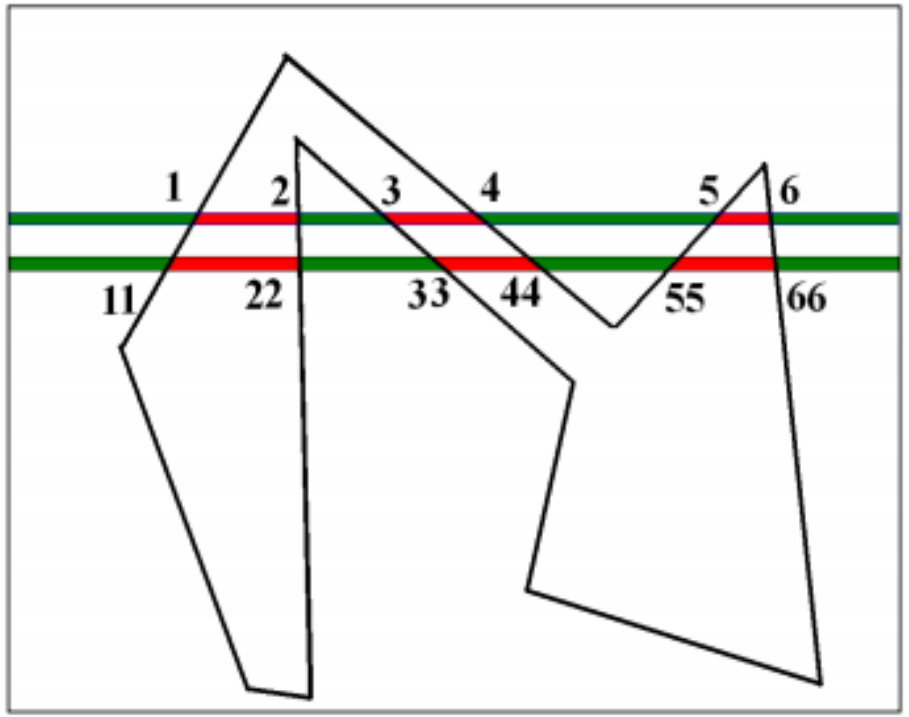
扫描线连贯性

- 推论：如果上述交点区间属于多边形内(外)， 那么该区间内所有点均属于多边形内(外)。

- 逐点判断->区间判断

边的连贯性

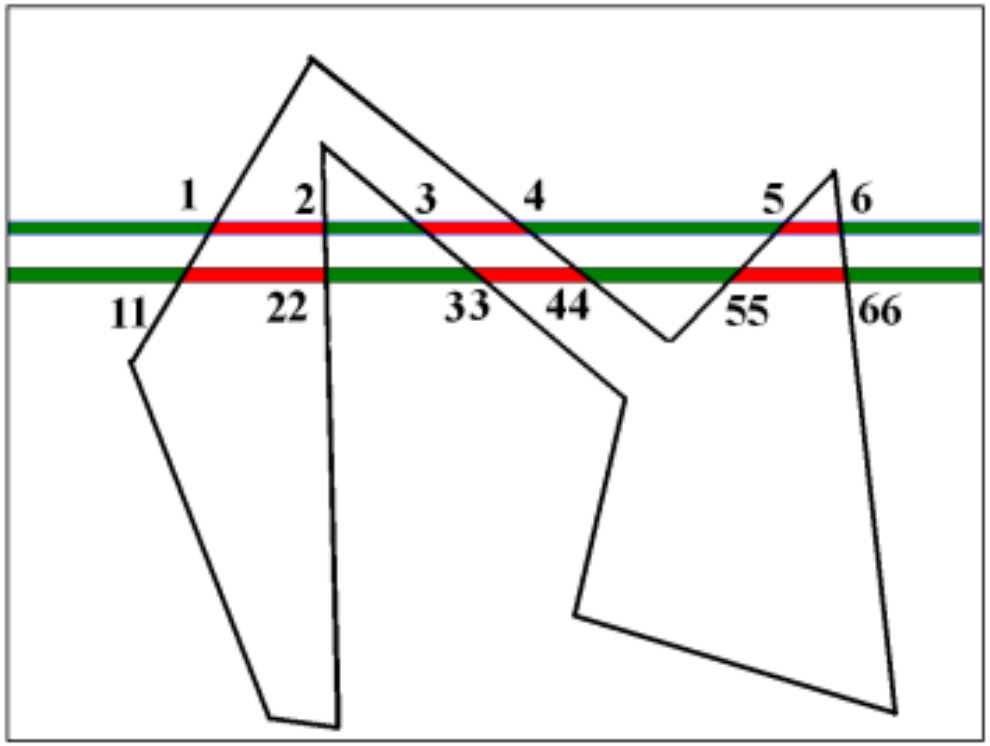
- 边的连贯性：直线的线性性质在光栅上的表现



区域的连贯性

- 扫描线与边的交点
  - 1 (x_1, y_1)
  - 2 (x_2, y_2)
  - 11 (x_{11}, y_{11})
  - 22 (x_{22}, y_{22})

边的连贯性



边的连贯性

- 相邻扫描线($y_1 = y_{11} + 1$)与多边形的同一条边的交点存在如下关系：
- $\frac{y_1 - y_{11}}{x_1 - x_{11}} = k \quad \rightarrow \quad x_1 = x_{11} + \frac{1}{k}$
- 当知道扫描线与一条边 的一个交点之后， 通过上述公式可以通过增量算法迅速求出其它交点

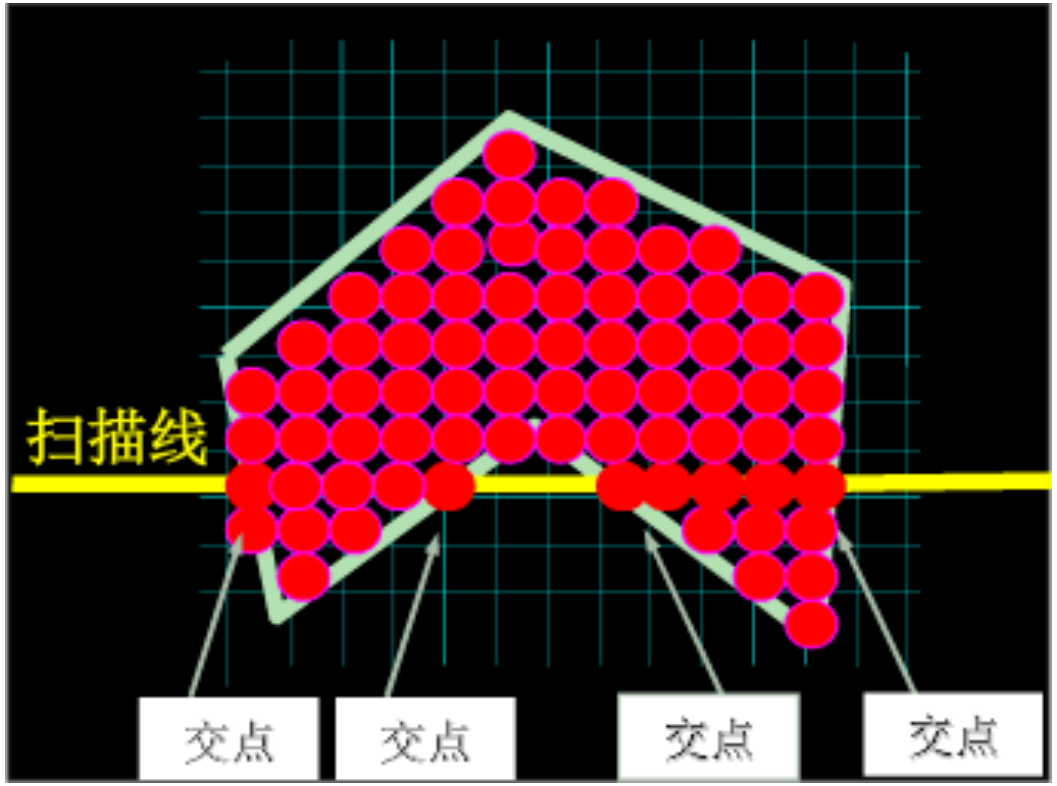
边的连贯性

- 推论：边的连贯性是连接区域连贯性和扫描线连贯性的纽带。

- 扫描线连贯性“+”边连贯性“=”区域连贯性。

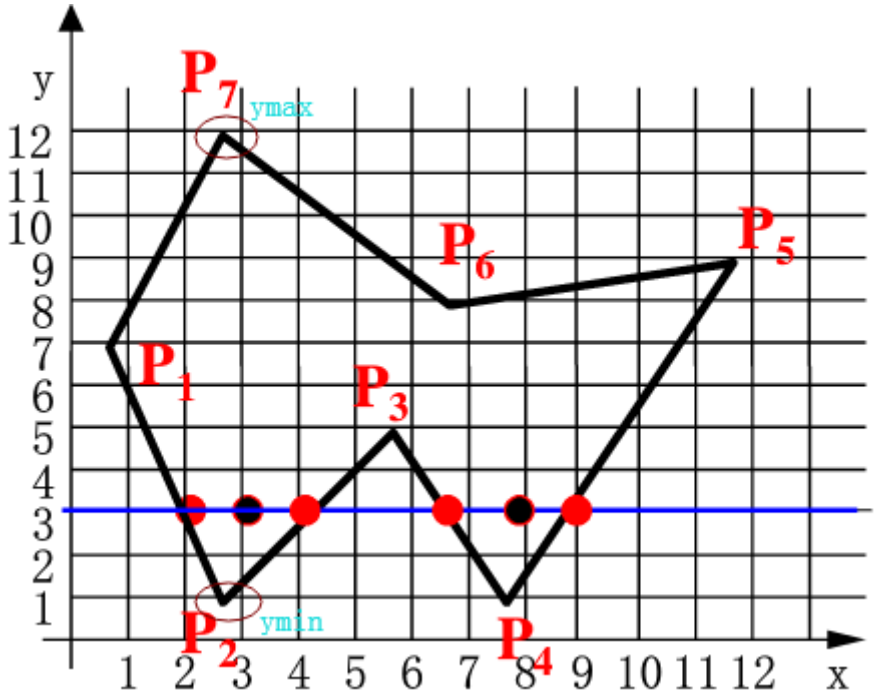
多边形扫描转换算法

- 核心思想(从下到上扫描)
  - 扫描线算法填充多边形的基本思想是按扫描线顺序，计算扫描线与多边形的相交区间，再用要求的颜色显示这些区间的象素，即完成填充工作。



多边形扫描转换算法

- 区间的端点可以通过计算扫描线与多边形边界线的交点获得。
  - 如扫描线y=3与多边形的边界相交于4点（2,3）、（4,3）、（7,3）、（9,3）
  - 这四个点定义了扫描线从x=2到x=4，从x=7到x=9两个落在多边形内的区间，该区间内像素应取填充色。

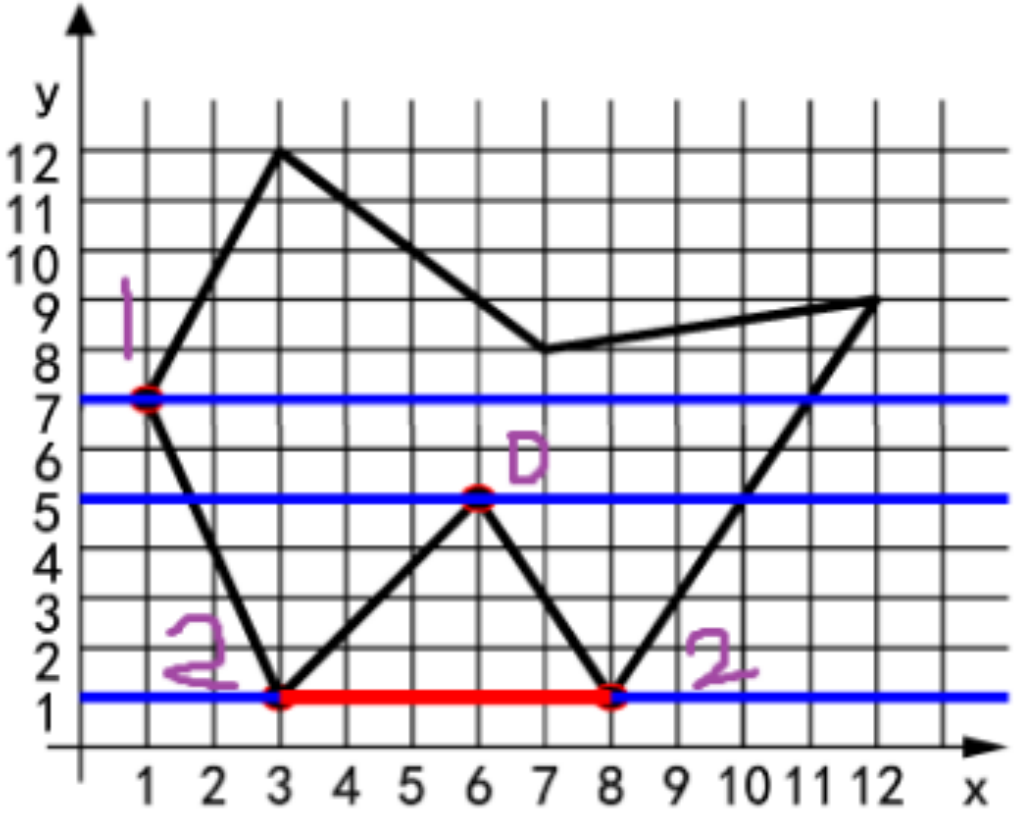


多边形扫描转换算法

- 算法步骤：
  - 1.确定多边形所占有的最大扫描线数，得到多边形定点的最小最大值（ymin和ymax）；
  - 2.从ymin到ymax每次用一条扫描线进行填充；
  - 3.对一条扫描线填充的过程分为四个步骤：
    - a)求交点；
    - b)把所有交点按递增顺序排序；
    - c)交点配对（第一个和第二个，第三个和第四个）；
    - d)区间填色。把相交区间内的像素置成多边形的颜色，相交区间外的像素置成背景色。

多边形扫描转换算法

- 扫描线与多边形顶点相交时，交点的取舍问题。
  - 交点应保证为偶数个



多边形扫描转换算法

- 解决方案
  - 若共享顶点的两条边分别落在扫描线的两边，交点只能算一个。
  - 若共享顶点的两条边在扫描线的同一边，这时交点作为0个或者2个
- 检查共享顶点的两条边的另外两个端点的y值，按这两个y值中大于交点y值的个书来决定交点数：
  - 两端点y值小于交点y值，交点作为0个；
  - 两端点y值大于交点y值，交点作为2个。

小结

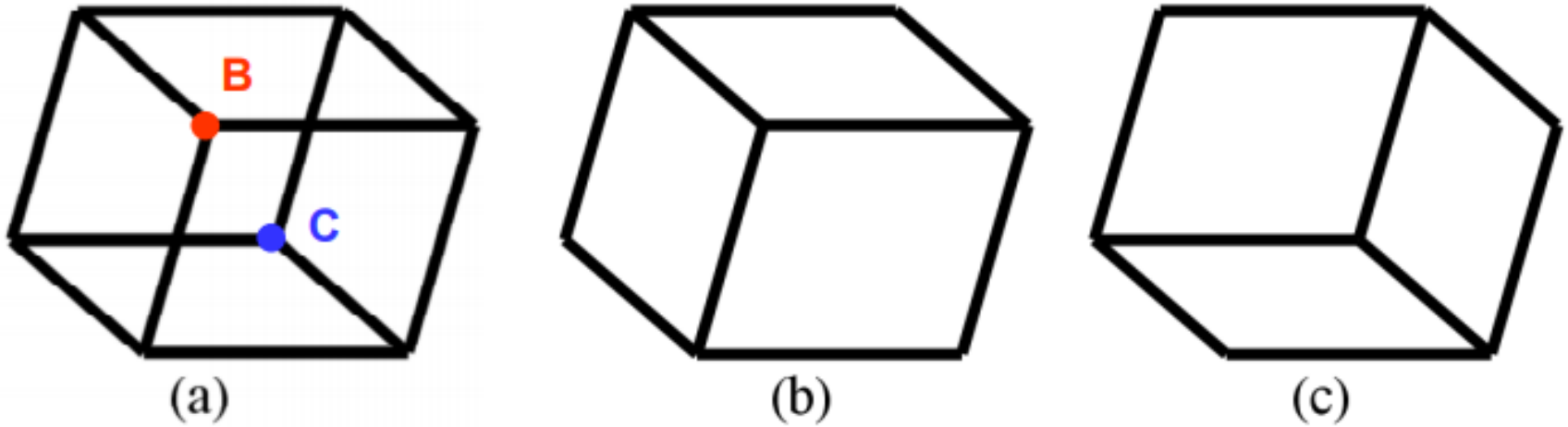
- 基本概念
- 区域填充
  - 四连通区域和八连通区域
  - 连通区域的种子填充算法
- 多边形的扫描转换
  - 逐点判断算法
  - 扫描线算法
    - 连贯性：区域、扫描线、边
    - 奇异点的处理
- 多边形的扫描转换与区域填充的比较

隐藏线隐藏面消除算法

隐藏线隐藏面消除算法

- 消隐的基本概念(隐藏线或面消除)：
  - hidden line/face remove
  - 相对于观察者，确定场景中哪些物体是可见或部分可见的，哪些物体是不可见的
- 消隐可以增加图形的真实感
  - 投影：三维空间→二维平面
  - 二维平面：通过确定物体的前后关系，可以获得更多信息
- 消隐是图形学中非常重要的一个基本问题

消隐的基本概念



- 没有消隐的图形具有二义性：
  - (a) 立方体的线框图；
  - (b) 顶点B离视点最近时的消隐；
  - (c) 顶点C离视点最近时的消隐.

消隐的基本概念

- 消隐问题的复杂性导致许多精巧的算法

不同算法适合于不同的应用环境

  - 在实时模拟过程中，要求消隐算法速度快，通常生成的图形质量一般
  - 在真实感图形生成过程中，要生成高质量的图形， 通常消隐算法速度较慢

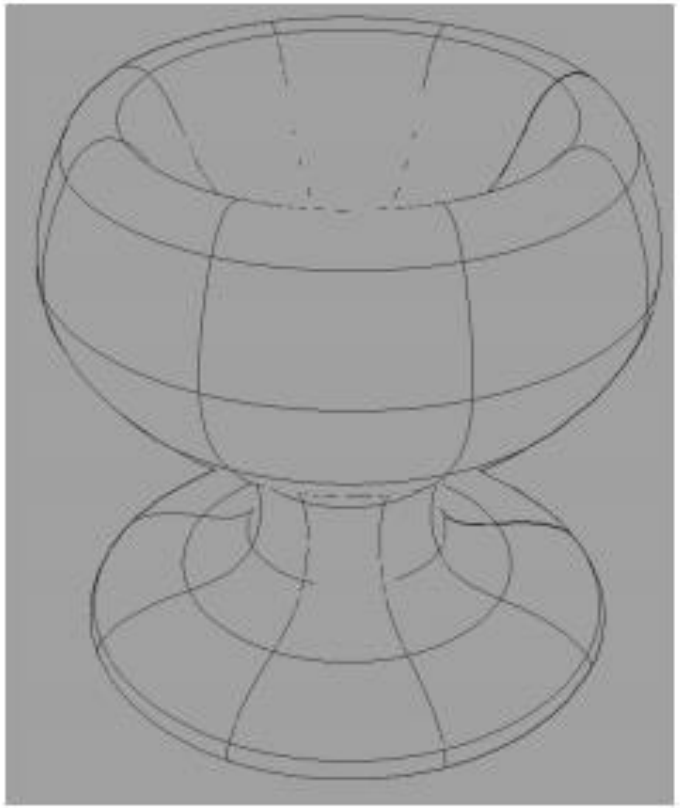
- 消隐算法的权衡：
  - 计算效率、图形质量

消隐与排序、连贯性

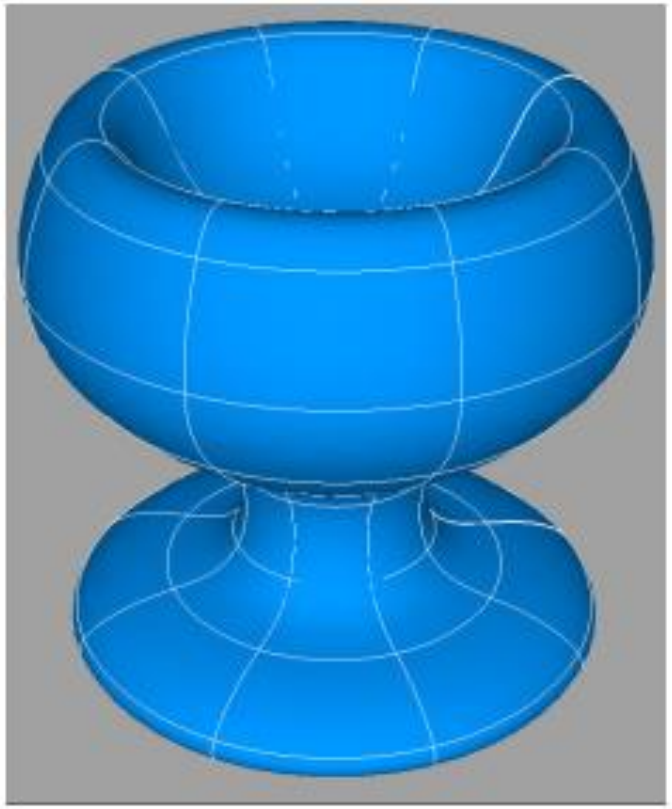
- 消隐与物体排序、连贯性密切相关
  - 排序： 判断场景中的物体全部或者部分与视点之间的远近
  - 连贯性： 场景中物体或其投影所表现出来的相似程度
- 消隐算法的效率很大程度上取决于
  - 排序的效率
  - 各种连贯性的利用

消隐的分类 - 对象与输出

- 根据消隐对象和输出结果



线消隐：输出线框图



面消隐：输出着色图

消隐的分类：坐标空间

- 算法实现时所在的坐标系(空间)进行分类：
  - 图像空间消隐
  - 物体空间消隐

图像空间消隐

- 描述

for(图像中每一个像素) {

确定由投影点与像素连线穿过的距离观察点 最近的物体;

用适当的颜色绘制该像素;

}

- 特点：在屏幕坐标系中进行的，生成的图像一般受限于显示器的分辨率
- 算法复杂度为$O(nN)$：场景中每一个物体要和屏幕中每一个像素进行排序比较， n为物体个数，N为像素个数
- 代表方法：z缓冲器算法，扫描线算法等

物体空间消隐

- 描述

for(世界坐标系中的每一个物体) {

确定未被遮挡的物体或者部分物体;

用恰当的颜色绘制出可见部分;

}

- 特点：算法精度高，与显示器的分辨率无关，适合于精密的CAD工程领域
- 算法复杂度为$O(n^2)$：场景中每一个物体都要和场景中其他的物体进行排序比较，n为物体个数
- 代表方法：背面剔除、表优先级算法等

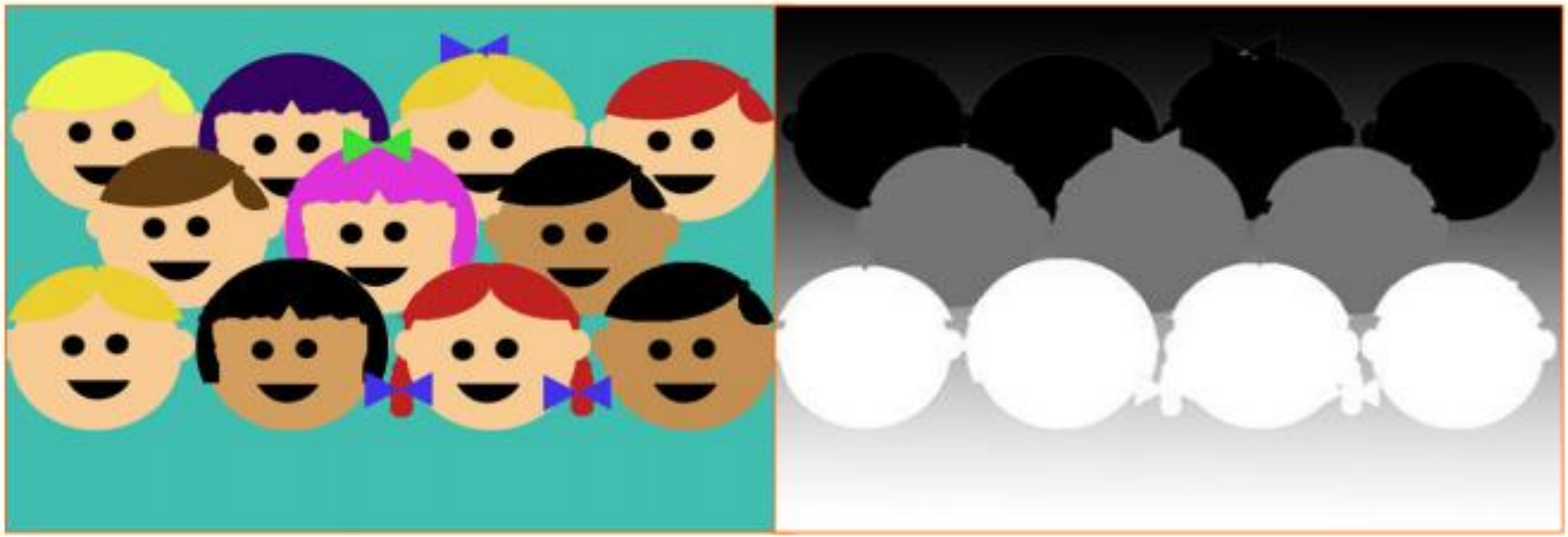
物体空间和图像空间消隐方法的比较

- 理论上
  - 如果 n(物体数)<N(像素数)，则景物空间算法的计算量$O(n^2)$小于图像空间算法$O(n*N)$
- 实际应用中通常会考虑画面的连贯性，所以图像空间算法的效率有可能更高
- 物体空间和图像空间的混合消隐算法

z缓冲器算法

- z (深度)缓冲器算法属于图像空间算法
- z缓冲器是帧缓冲器的推广
  - 帧缓冲器：存储的是像素的颜色属性
  - z缓冲器：存储的是对应像素的z值
    - 假设在视点坐标系$(oxyz)$中，投影平面为z=0，视线方向沿 (z)轴方向，投影为平行投影
    - 深度值就是物体沿着视线(z)方向、与视点的距离
    - 离视点近的物体遮挡离视点远的物体：z值越小，离视点越近

颜色与深度缓冲举例



颜色缓冲　　深度缓冲

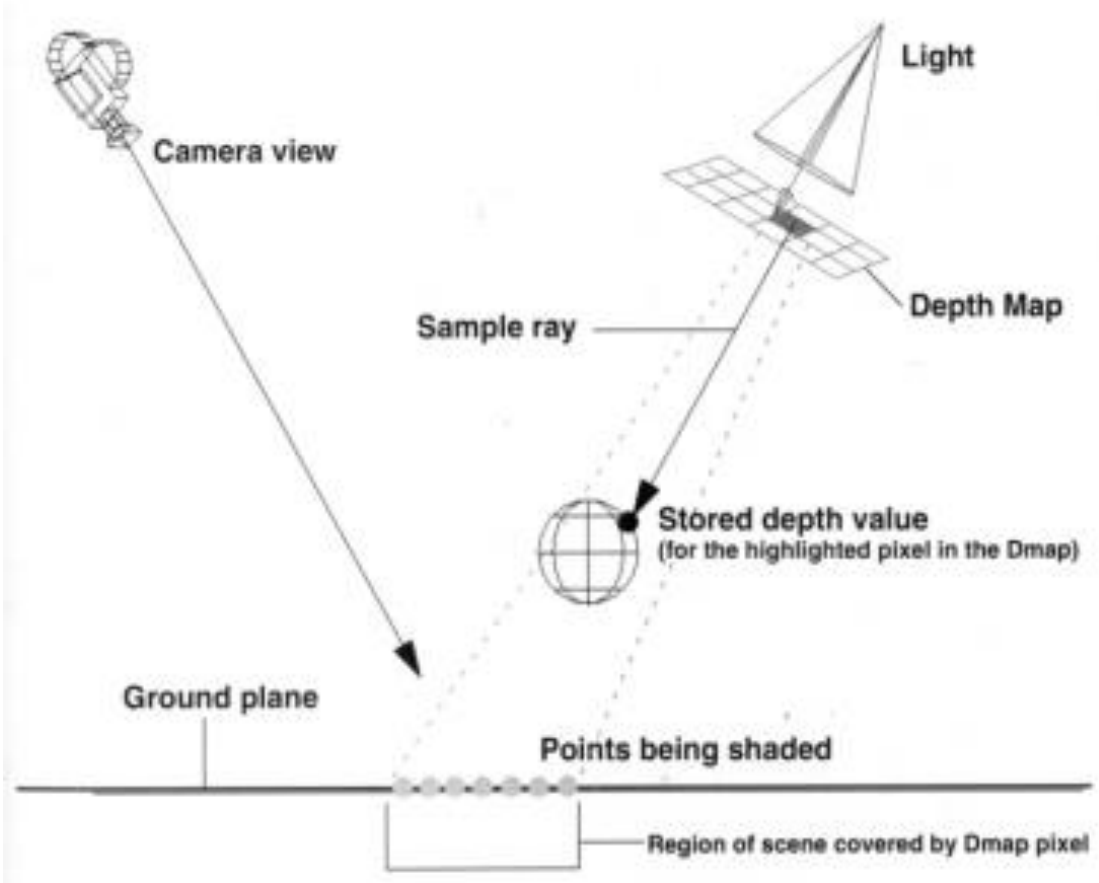
z缓冲器算法描述

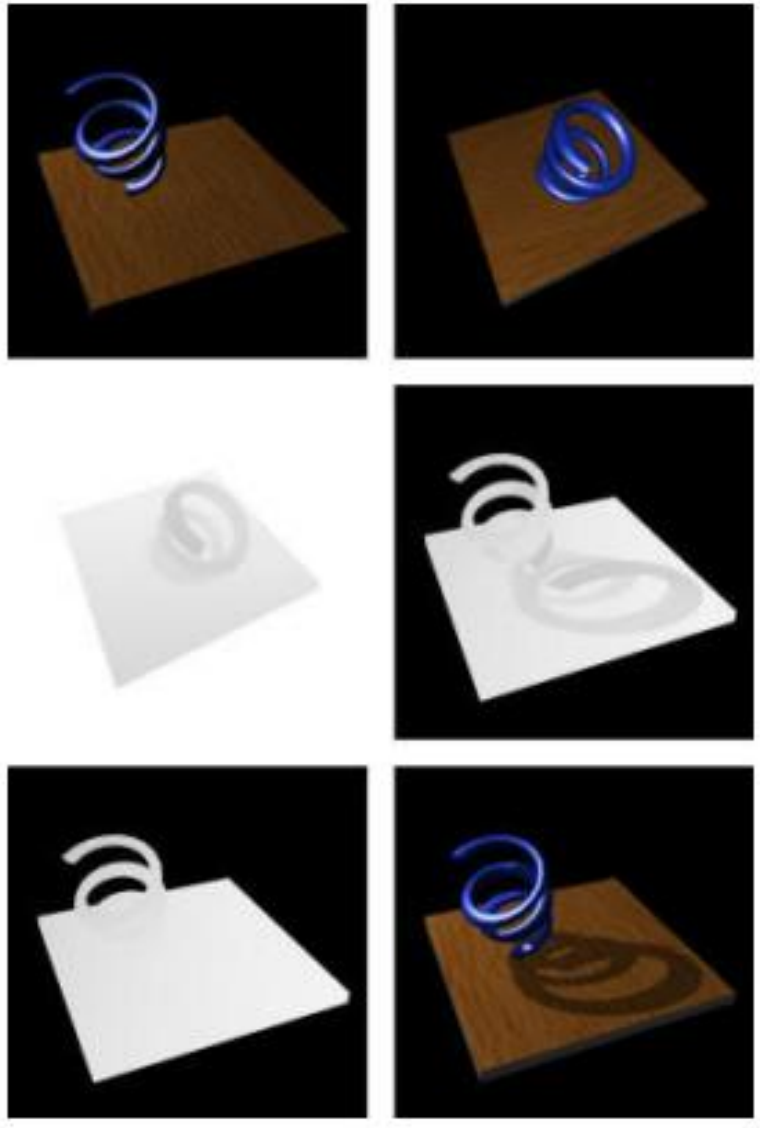
- (1)帧缓冲器中的颜色置为背景颜色
- (2) z缓冲器中的z值置成最小值(离视点最远)
- (3) 以任意顺序扫描各多边形
  - a) 对于多边形中的每一像素，计算其深度值$z(x,y)$
  - b) 比较$z(x,y)$与z缓冲器中已有的值$zbuffer(x,y)$

    如果$z(x,y) < zbuffer(x,y)$，那么 计算该像素(x, y)的光亮值属性并写入帧缓冲器更新z缓冲器 $zbuffer(x,y) = z(x,y)$

z缓冲器的其它应用

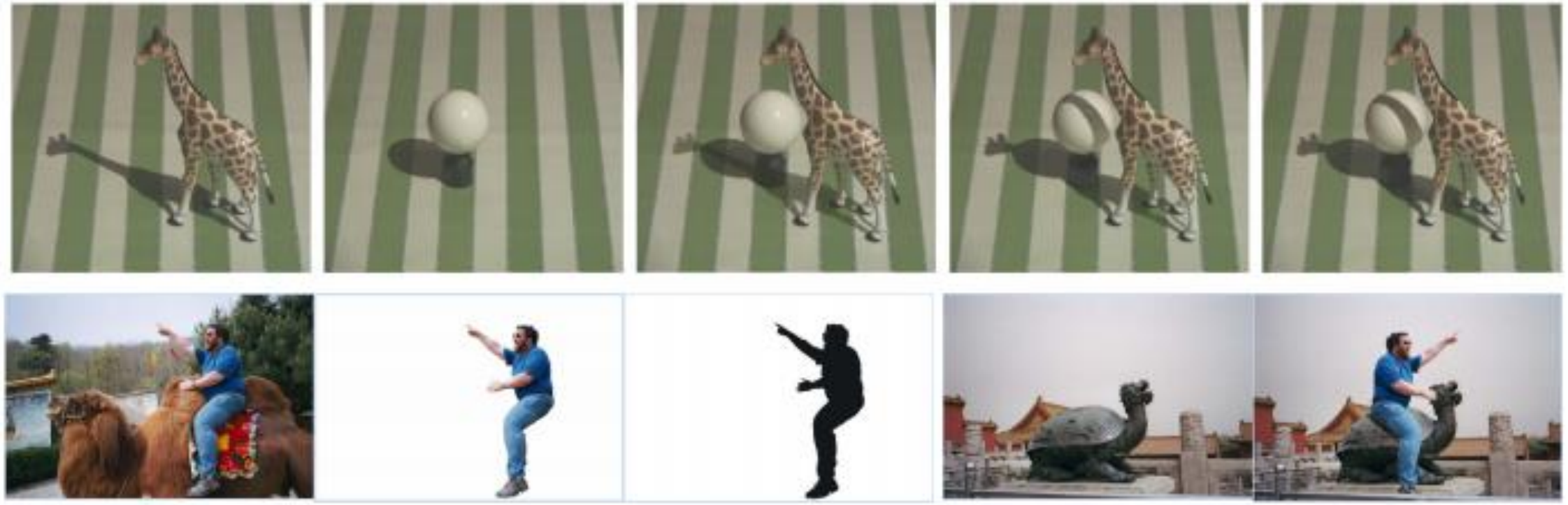
- 阴影算法：以光源为视点的z缓冲器





z缓冲器的其它应用

- rgbα和z缓冲器相结合，实现图像的合成



z缓冲器算法分析

- 优点
  - 算法复杂度($O(n*N)$)：对于给定的图像空间，N是固定的，所以算法复杂度只会随着场景的复杂度线性地增加
  - 无须排序：场景中的物体是按任意顺序写入帧缓冲器和z缓冲器的，无须对物体进行排序，从而节省了排序的时间
  - 适合于任何几何物体：能够计算与直线交点
  - 适合于并行实现(硬件加速)

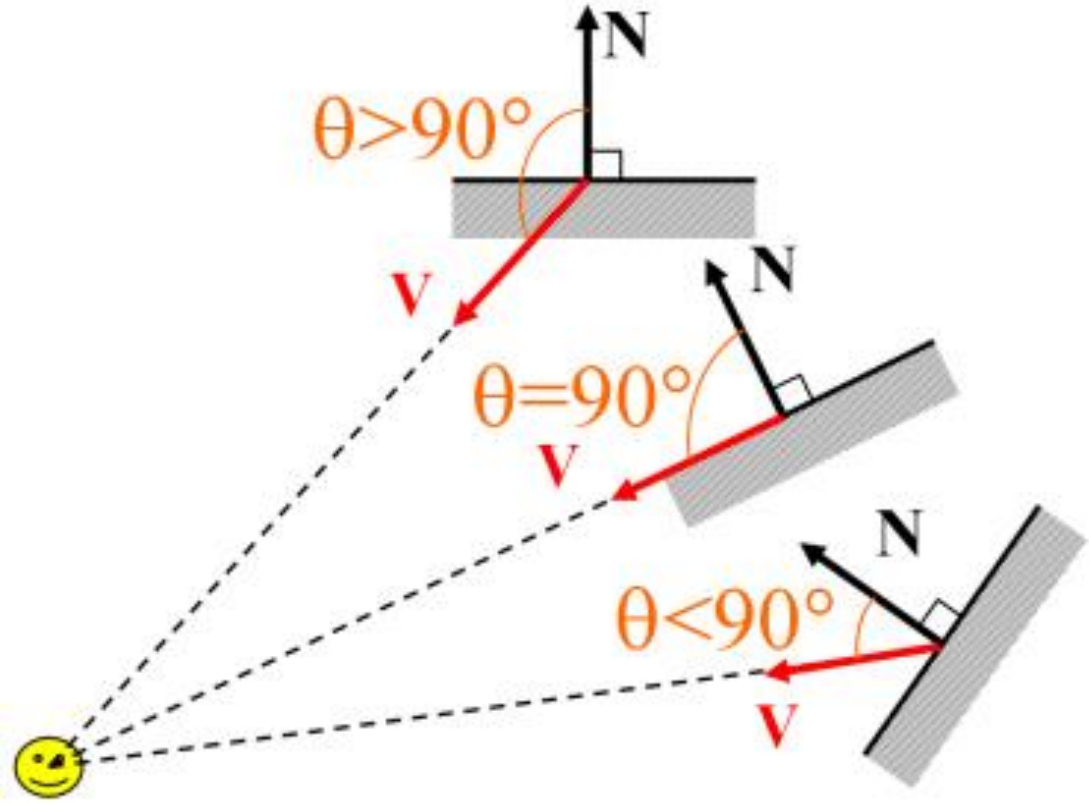
z缓冲器算法分析

- 不足
  - z缓冲器需要占用大量的存储单元

    一个大规模复杂场景中：深度范围可能为106，一个 像素需要24bit来存储其深度信息。如果显示分辨率 为1280×1024，那么深度缓冲器需要4MB存储空间
  - 深度的采样与量化带来走样现象
  - 难以处理透明物体
- 解决存储问题：逐区域进行z缓冲器消隐

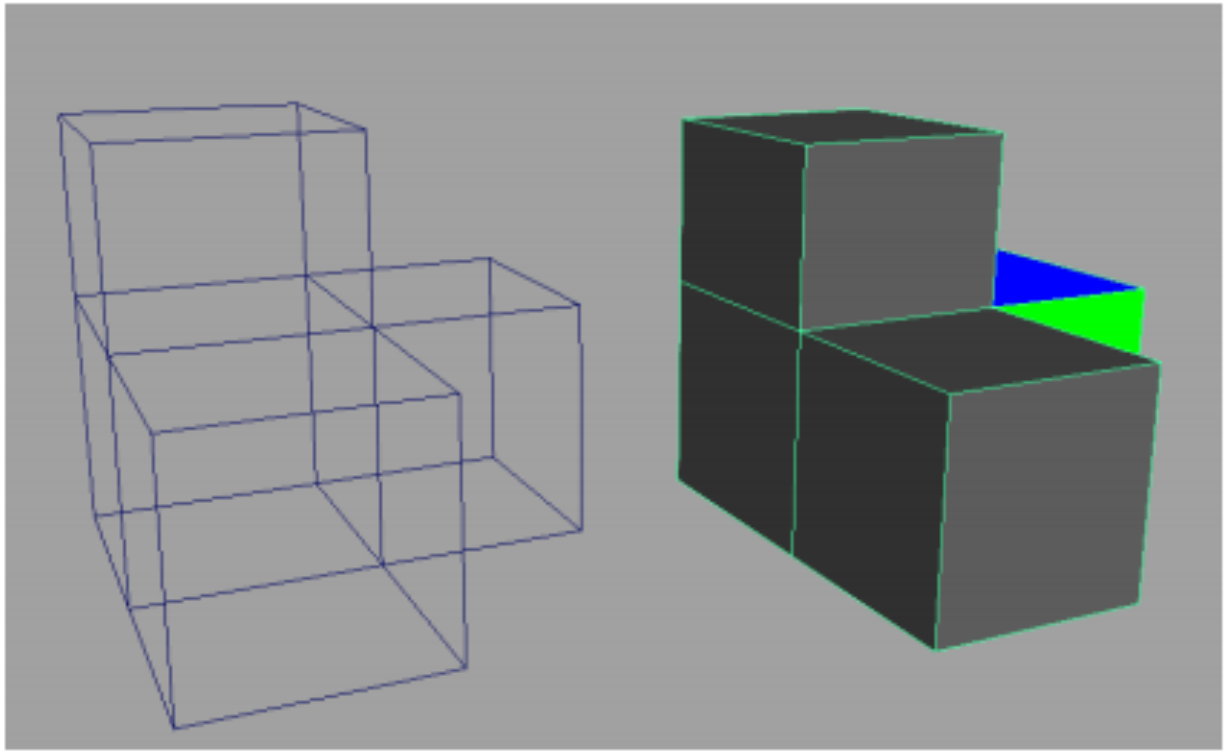
物体空间消隐：背面剔除算法

- 利用视线方向V和物体表面法向N之间的关系
- $N \cdot V < 0$: 不可见
- $N \cdot V \geq 0$:可见



背面剔除算法

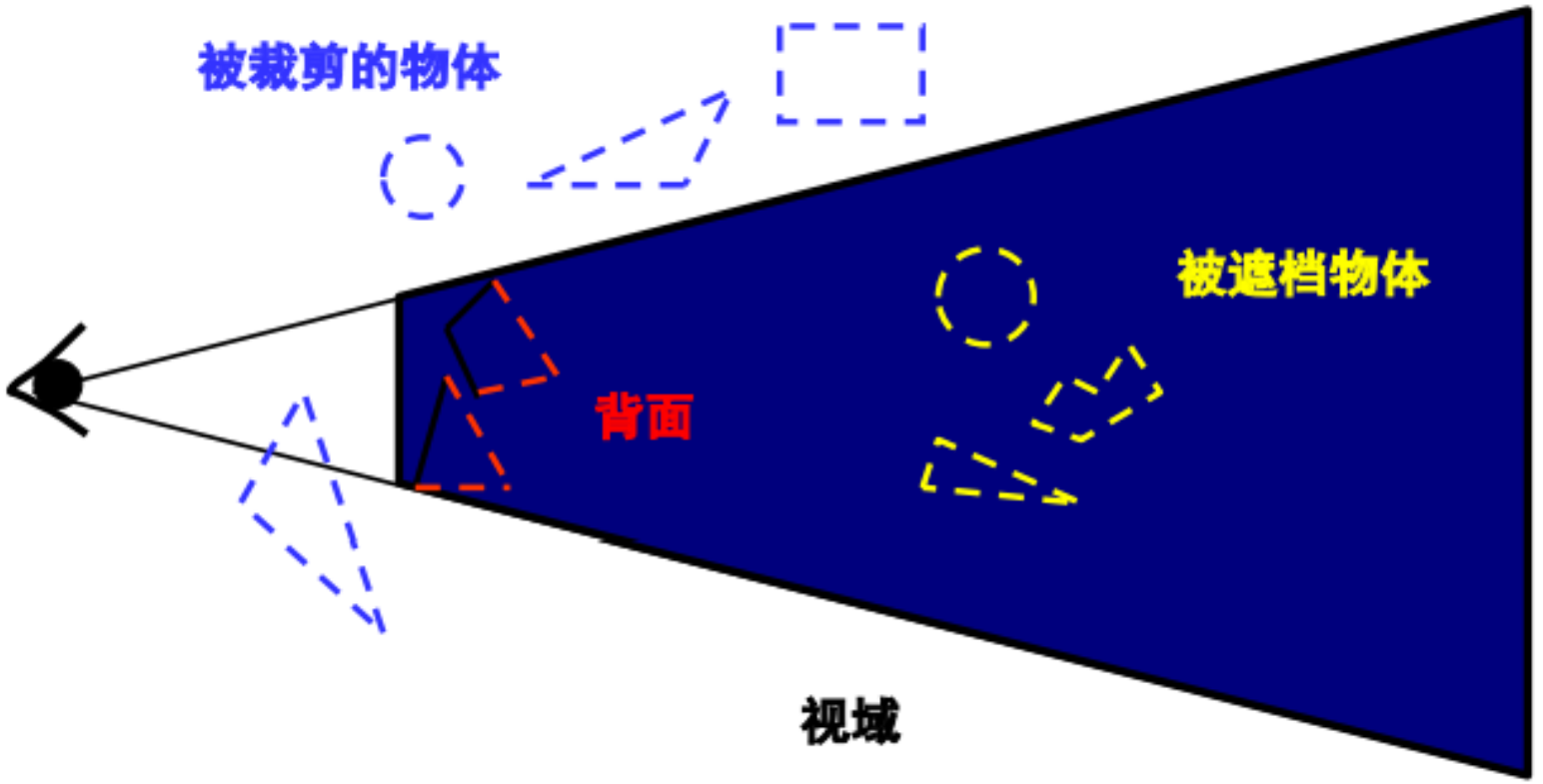
- 作为消隐算法，背面剔除适用于凸多面体，不 适用于凹多面体或其它复杂物体



对于蓝色与绿色的面，简单的背面剔除不能实现完全消隐

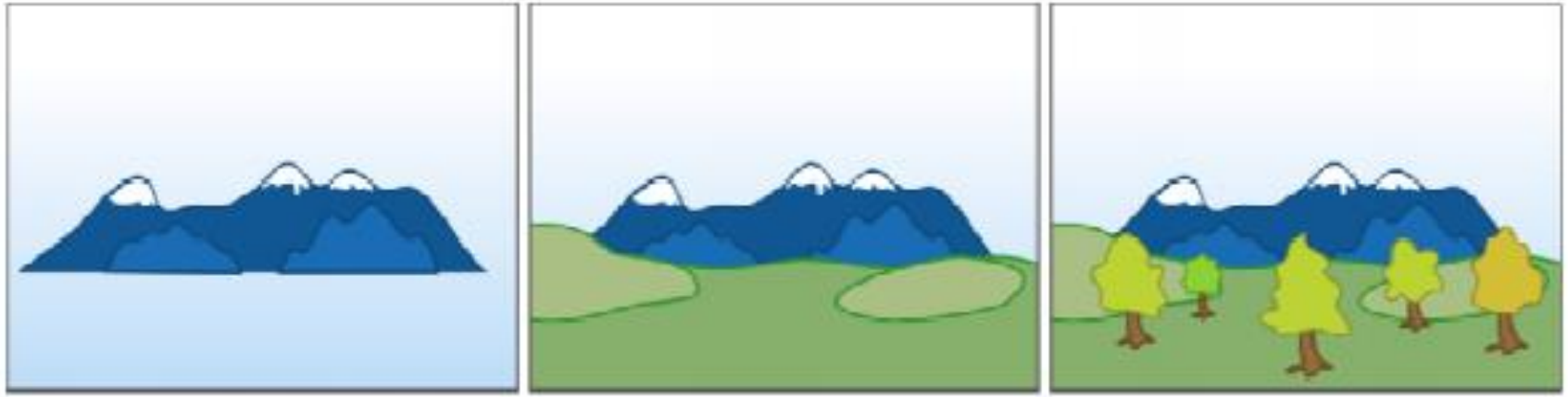
背面剔除算法

- 适用于场景消隐的预处理：消除一些显然不可见表面，从而提高其它消隐算法的效率。



物体空间中的表优先级算法

- 原理：离视点近的物体可能遮挡离视点远的物体
- 在物体空间确定物体之间的可见性顺序(物体离 视点远近)，由远及近地绘制出正确的图像结果——油画家算法



- 条件：场景中物体在z方向上没有相互重叠

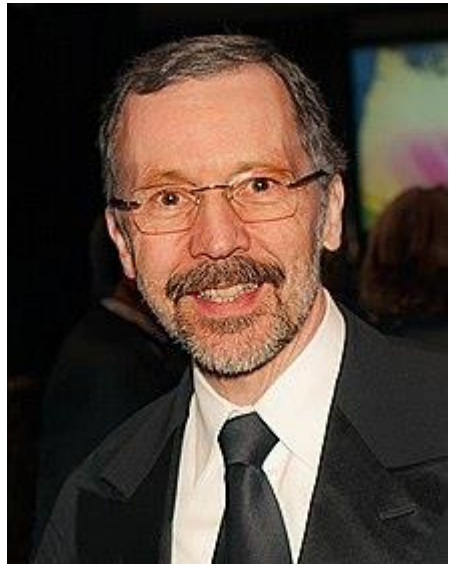
二维半物体的深度排序

- 二维半物体的深度值是常数：卡通动画、窗口管理、VLSI设计、图像合成

- 二维半物体的深度排序算法只要简单地比较其z 值即可

Z-buffer算法发明人

- Edwin Catmull
  - 皮克斯的创始人之一。
  - 迪士尼动画工作室和皮克斯动画工作室现任总裁。
  - 81届奥斯卡金像奖。
  - 2019年图灵奖获得者。



OpenVDB发明人

- Ken Museth
  - 发明OpenVDB，并因此奥斯卡奖。
  - 该技术成为电影行业中表示火、爆炸、烟、水等视觉效果的行业标准。
  - 阿凡达、复仇者联盟、功夫熊猫等电影使用了该技术。
  - 创办Voxel Tech公司。

K. Museth, “VDB: High-Resolution Sparse Volumes with Dynamic Topology”, ACM Transactions on Graphics 32(3), 2013. Presented at SIGGRAPH 2013.

小结

- 消隐的基本概念
- 图像空间消隐：z缓冲器(z-buffer)算法
- 物体空间消隐
  - 背面剔除算法
  - 表优先级算法