



《计算机图形学基础》

习题课1

助教 杨国焘

2022年3月13日



联系方式

- 邮箱:

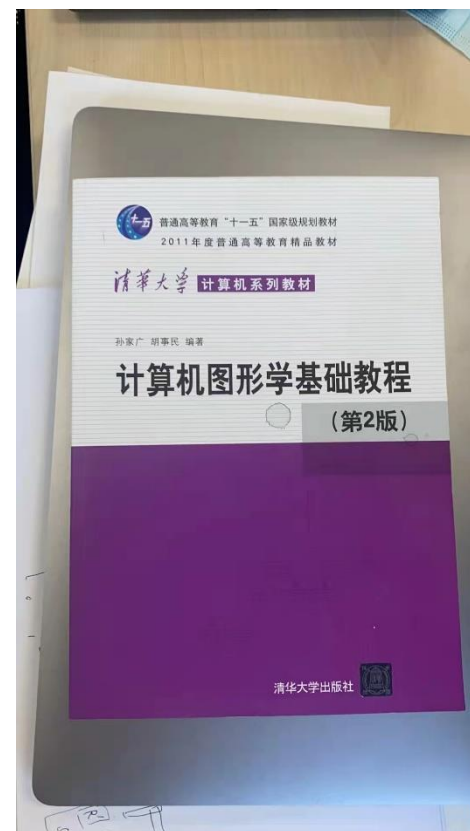
yanggy19@mails.tsinghua.edu.cn

lixl19@mails.tsinghua.edu.cn

cgc21@mails.tsinghua.edu.cn

- 位置:

— FIT楼3 区 523





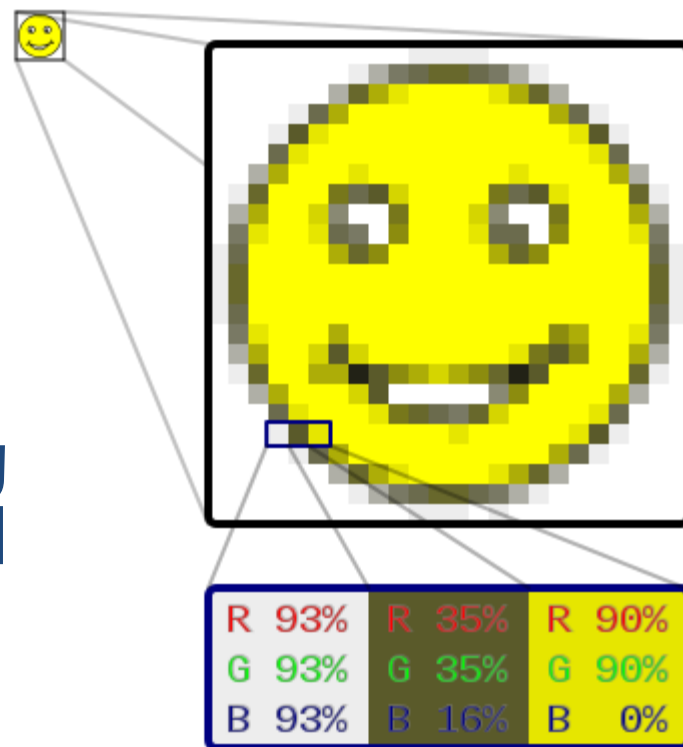
主要内容

- 光栅图形学
 - 直线段光栅化
 - 多边形光栅化
 - 字符光栅化
 - 裁剪
 - 反走样
 - 裁剪
- PA0:RasterGraphics
 - 实现细节
 - 环境配置



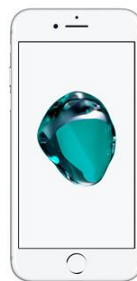
光栅图形学介绍

Raster Graphics



光栅图形学

- 全是像素点 (Pixels)
- 光栅化 Rasterizing
 - 确定最佳逼近图形的像素集合
- 裁剪 Cropping
 - 图形哪些部分该显示
- 反走样 (抗锯齿) Anti-aliasing
 - 有限屏幕分辨率下, 减少畸变
- 消隐 Blanking
 - 留近不留远



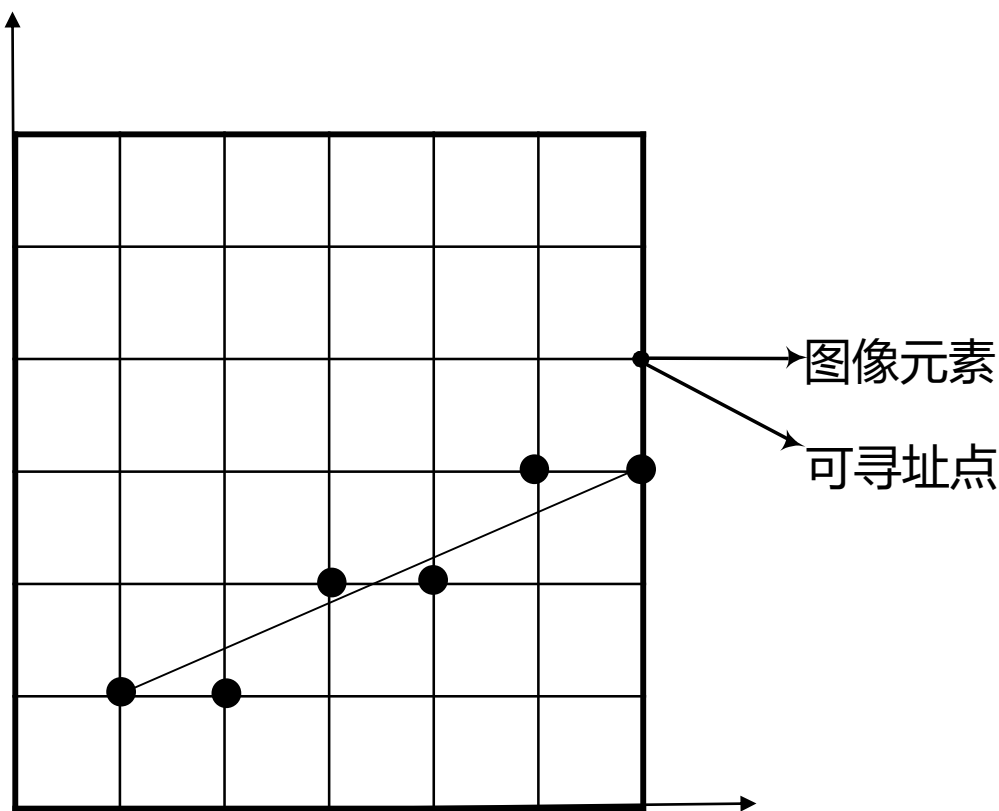
哇！我看不到像素点！
Wow! Retina!

_(: 3 」 ∠) _



光栅化算法(线)

- 确定图形的最佳像素逼近



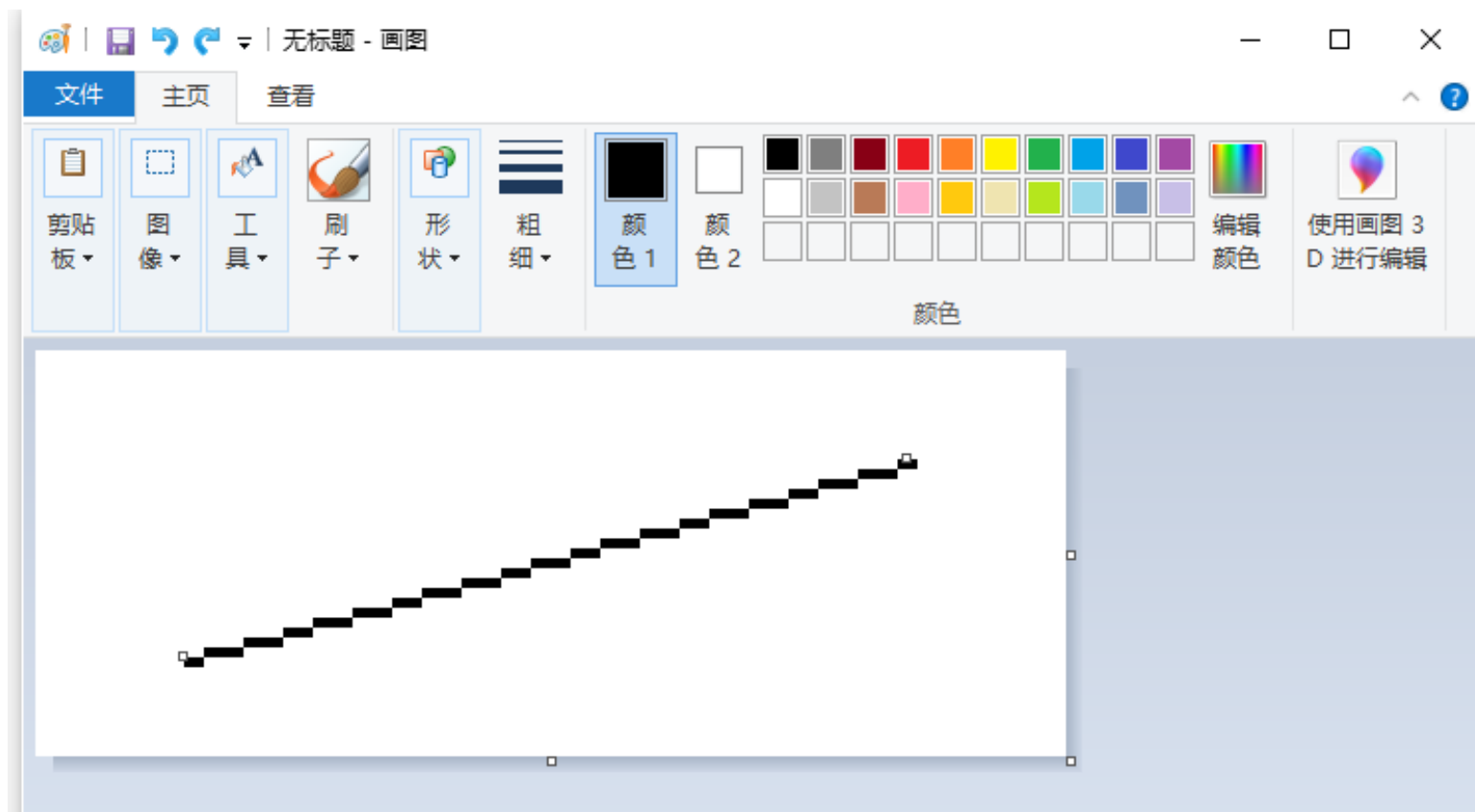
$$y = 0.4x + 0.6, x \in [1, 6]$$

(1,1), (2,1), (3,2)
(4,2), (5,3), (6,3)



光栅化算法(线)

- Windows – 画图工具





光栅化算法(线)

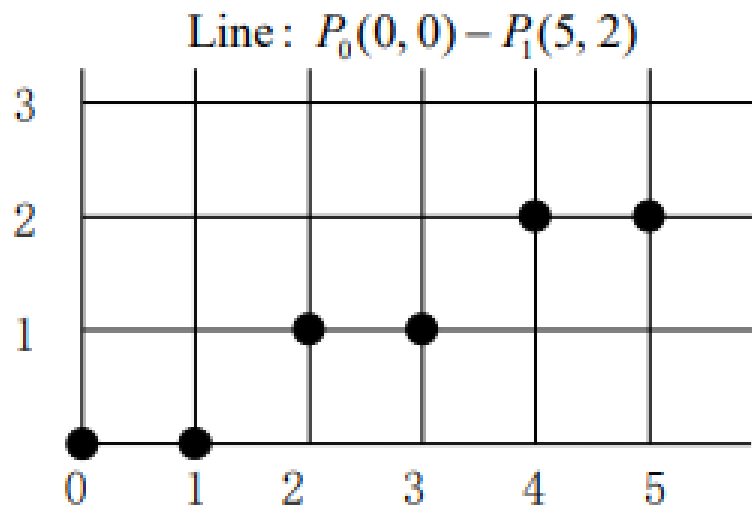
- 数值微分 (DDA)
- 中点画线
- Bresenham



光栅化算法(线) – 数值微分法

- 从 x 的左端点开始，向右端点步进。
- 步长为 1 像素，每步按 $y=kx+b$ 计算相应的 y 坐标。
- 并取像素点 $(x, \text{round}(y))$ 作为当前点的坐标。

x	$\text{int}(y+0.5)$	$y+0.5$
0	0	0
1	0	$0.4+0.5$
2	1	$0.8+0.5$
3	1	$1.2+0.5$
4	2	$1.6+0.5$
5	2	$2.0+0.5$





光栅化算法(线) – 数值微分法

```
void DDALine(int x0, int y0, int x1, int y1, int color) {  
    int x;  
    float dx, dy, y, k;  
    dx = x1-x0, dy=y1-y0;  
    k=dy/dx, y=y0;  
    for (x=x0; x<=x1, x++) {  
        drawpixel (x, int(y+0.5), color);  
        y=y+k;  
    }  
}
```



光栅化算法(线) – 数值微分法

- 此算法当 $|k| > 1$ 时应把 x 与 y 互换。
- 这个算法每一步都要进行浮点运算和四舍五入，不利于加速。



光栅化算法(线) – 中点画线法

- 通过观察可以发现，画直线段的过程中，当前像素点为 (x_p, y_p) ，下一个像素点有两种可选择点 $(x_p + 1, y_p)$ 或 $(x_p + 1, y_p + 1)$ 。
- 直线方程为 $F(x, y) = ax + by + c = 0$ 。
其中 $a = y_0 - y_1$ ， $b = x_1 - x_0$ ， $c = x_0 y_1 - x_1 y_0$
- 有：
$$\begin{cases} \text{线上: } F(x, y) = 0 \\ \text{上方: } F(x, y) > 0 \\ \text{下方: } F(x, y) < 0 \end{cases}$$

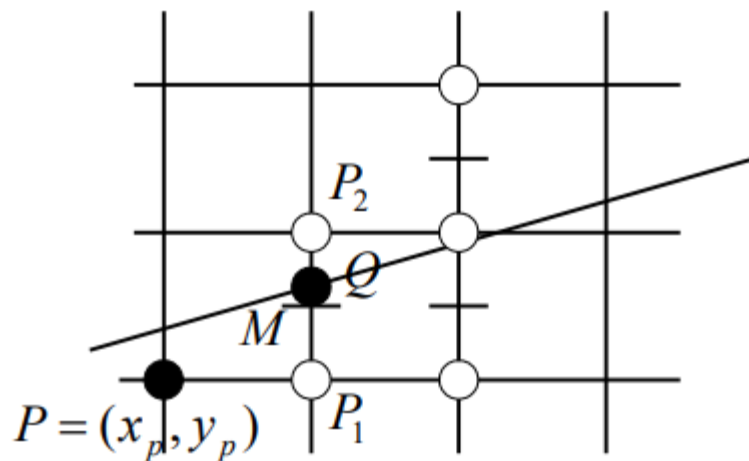


光栅化算法(线) – 中点画线法

- 通过判断 $(x_p + 1, y_p + 0.5)$ 与线的关系, 即可判断出下一个点绘制在哪一个点。
- 即判断下式的正负:

$$d = F(M) = F(x_p + 1, y_p + 0.5) = a(x_p + 1) + b(y_p + 0.5) + c$$

- 为减少浮点运算, 可以把该式乘2。





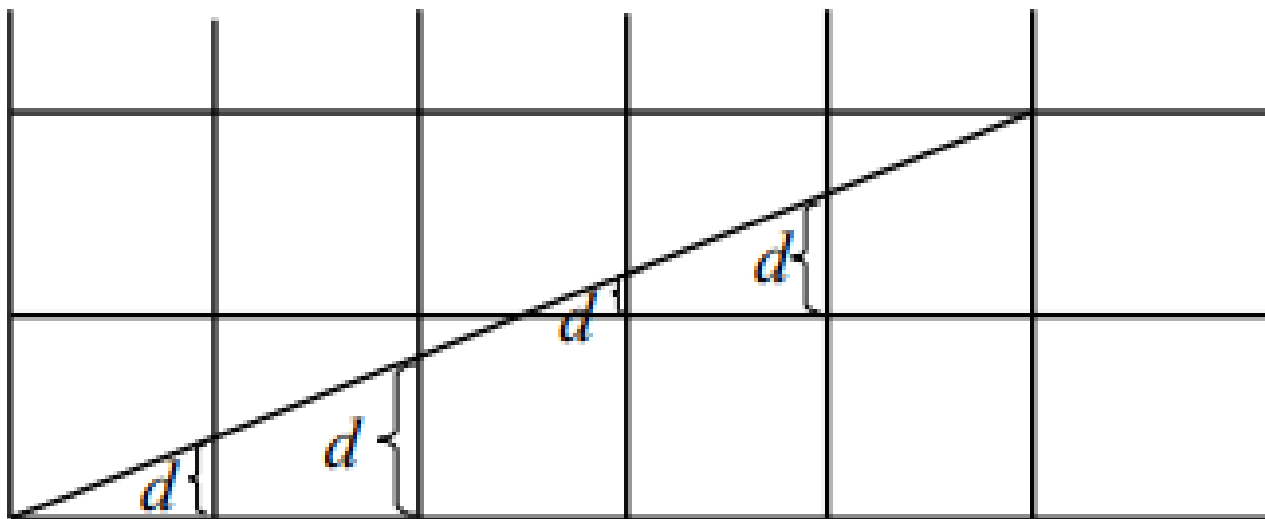
光栅化算法(线) – 中点画线法

```
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);  
    }  
}
```



光栅化算法(线) – Bresenham算法

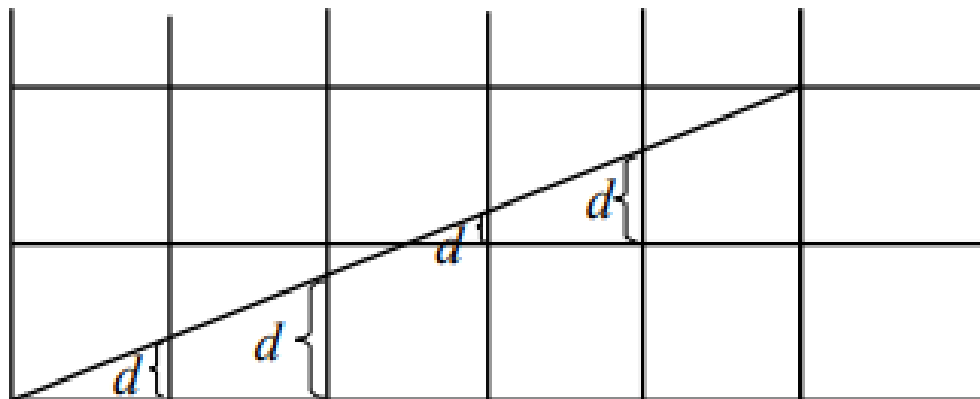
- 该方法类似于中点法，由误差项符号决定下一个像素取右边点还是右上点。





光栅化算法(线) – Bresenham算法

```
void Bresenhamline (int x0, int y0, int x1, int y1, int color)
{
    int x, y, dx, dy;
    float k, e;
    dx = x1-x0, dy = y1- y0, k=dy/dx;
    e=-0.5, x=x0, y=y0;
    for (i=0; i<=dx; i++) {
        drawpixel (x, y, color);
        x=x+1, e=e+k;
        if (e>=0) { y++; e=e-1;}
    }
}
```





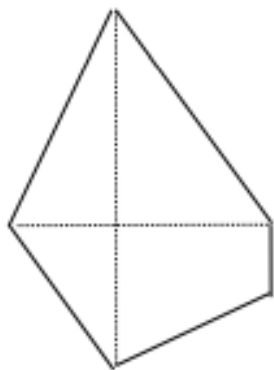
光栅化算法(线) – Bresenham算法

- 为避免除法与小数,
可以用 $e' = 2 \cdot dx \cdot e$ 替代。

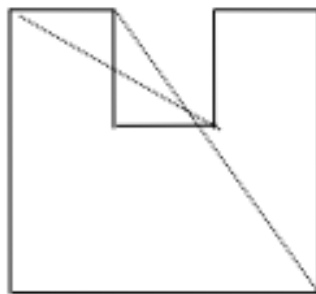
```
void IntegerBresenhamline (int x0, int y0, int x1, int y1, int color) {  
    int x, y, dx, dy, e;  
    dx = x1-x0, dy = y1- y0, e=-dx;  
    x=x0, y=y0;  
    for (i=0; i<=dx; i++) {  
        drawpixel (x, y, color);  
        x++, e=e+2*dy;  
        if (e>=0) { y++; e=e-2*dx;}  
    }  
}
```

光栅化算法(形)

- 多边形有凸多边形、凹多边形和含内环的多边形等。
- 算法主要研究多边形的填色问题。



(a) 凸多边形



(b) 凹多边形



(c) 含内环的多边形



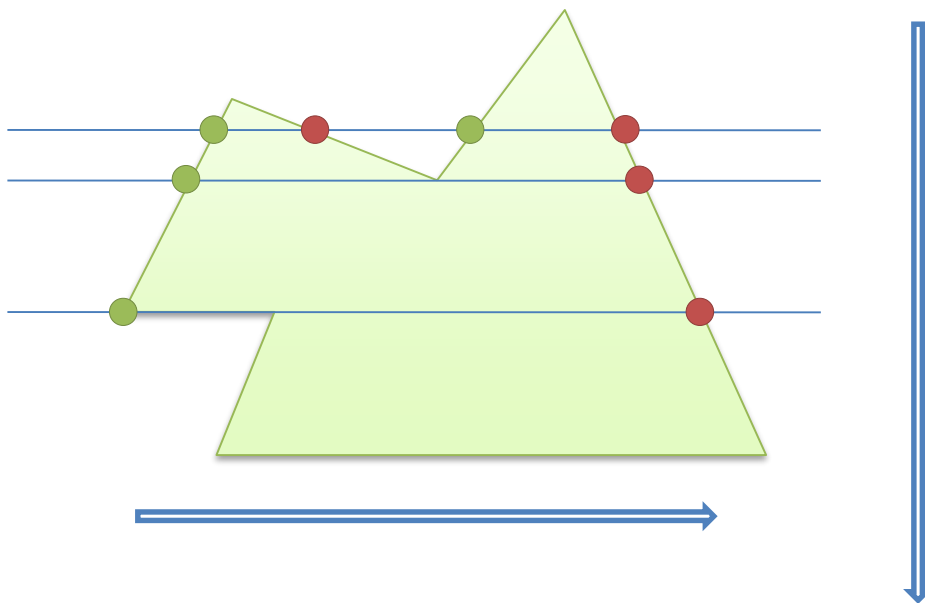
光栅化算法(形)

- 扫描线算法
- 边界标志算法
- 区域填充算法



光栅化算法(形) - 扫描线算法

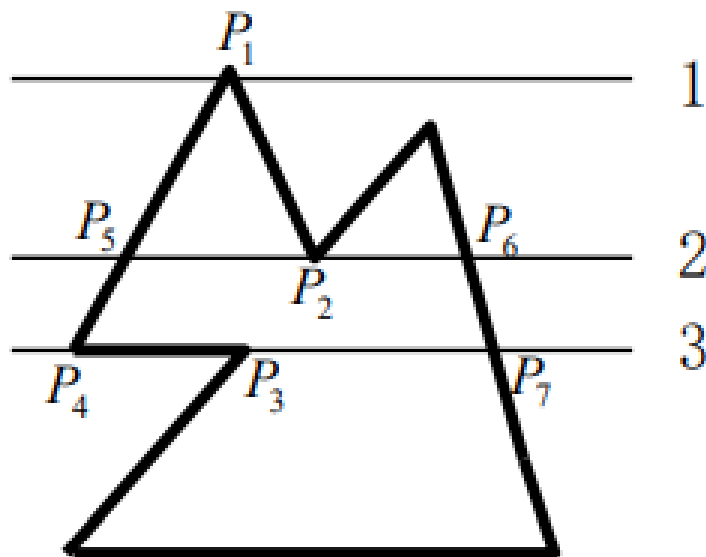
- 按扫描线顺序，计算扫描线与多边形的相交区间
- 再用要求的颜色显示这些区间的像素，以完成填充工作。





光栅化算法(形) - 扫描线算法

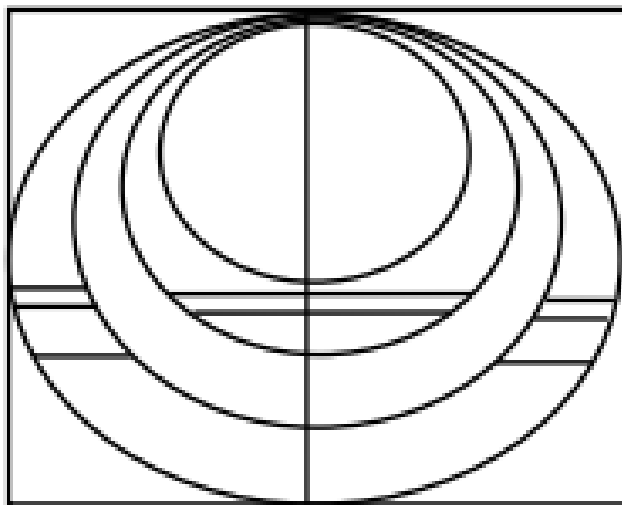
- 若扫描线与多边形相交的边分处扫描线的两侧，则计一个交点，如P5。
- 若扫描线与多边形相交的边分处扫描线同侧，则计零个交点，如P1, P2。
- 若扫描线与多边形边界重合，则计1个交点，如边P3P4。





光栅化算法(形) - 边界标志算法

- 对多边形边界所经过的像素打上标志。
- 对每条与多边形相交的扫描线依从左到右的顺序，逐个访问该扫描线上的像素。
- 遇到标志则将inside属性取反。
- 根据inside属性决定是否染色。





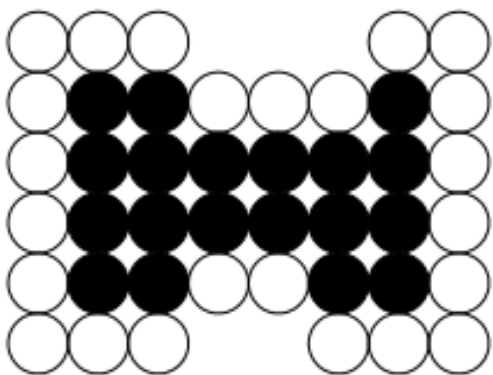
光栅化算法(形) - 边界标志算法

```
void edgemark_fill(多边形定义 polydef, int color) {  
    对多边形 polydef 每条边进行直线扫描转换;  
    for (每条与多边形 polydef 相交的扫描线 y) {  
        inside = FALSE;  
        for (扫描线上的每个像素 x) {  
            if (像素 x 被打上边标志) inside = !inside;  
            if (inside)  
                drawpixel (x, y, color);  
            else  
                drawpixel (x, y, background);  
        }  
    }  
}
```



光栅化算法(形) - 区域填充算法

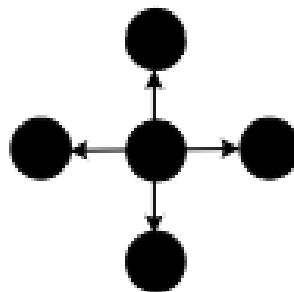
- 区域填充指先将区域的一点赋予指定的颜色，然后将该颜色扩展到整个区域的过程。
- 区域可分为四连通区域和八连通区域。



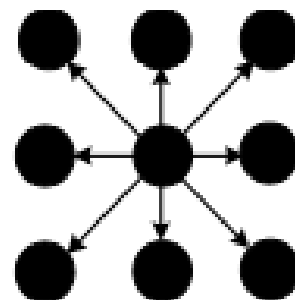
表示内点



表示边界点



四联通



八连通



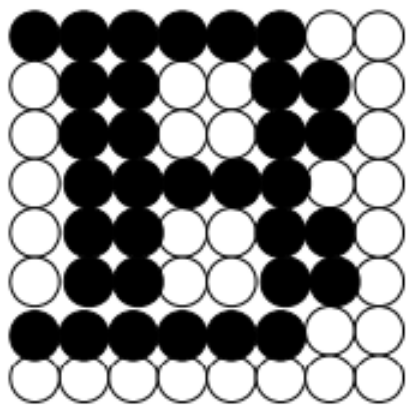
光栅化算法(形) - 区域填充算法

- 四联通填充算法

```
void FloodFill4(int x,int y,int oldcolor,int newcolor) {  
    if (getpixel(x, y)==oldcolor) {  
        drawpixel(x,y,newcolor);  
        FloodFill4(x, y+1, oldcolor,newcolor);  
        FloodFill4(x, y-1, oldcolor,newcolor);  
        FloodFill4(x-1, y, oldcolor,newcolor);  
        FloodFill4(x+1, y, oldcolor,newcolor);  
    }  
}
```

光栅化算法(字符)

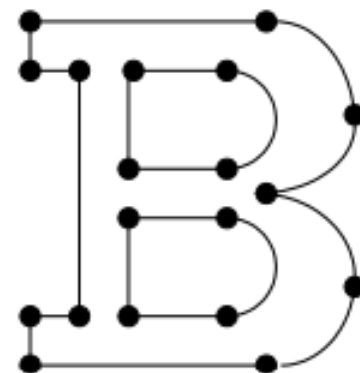
- 为了在显示器等输出设备上输出字符，系统中必须装备有相应的字库。
- 字库中存储了每个字符的形状信息，分为点阵和矢量型两种。



(a) 点阵字符

1	1	1	1	1	1	0	0
0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
0	1	1	1	1	1	0	0
0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0
1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0

(b) 点阵字库中的位图表示



(c) 矢量轮廓字符



光栅化算法(字符)

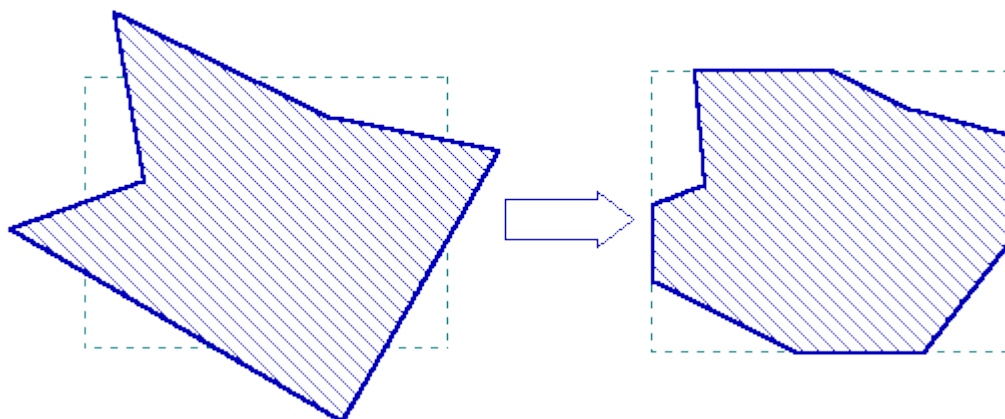
- 字符属性包括:

- 字体 例如: 宋体 仿宋体 楷体 黑体 隶书
- 字高 例如: 宋体 宋体 宋体 宋体
- 字宽因子(扩展/压缩) 例如: 大海 大海 大海 大海
- 字倾斜角 例如: 倾斜 倾斜
- 对齐 左对齐、中心对齐、右对齐
- 字色 对字符设置各种颜色
- 写方式 “替换”方式时, 对应字符掩膜中的空白区被置成背景色;
“与”方式时, 这部分区域的颜色不受影响。



裁剪

- 线段裁剪
- 多边形裁剪
- 字符裁剪

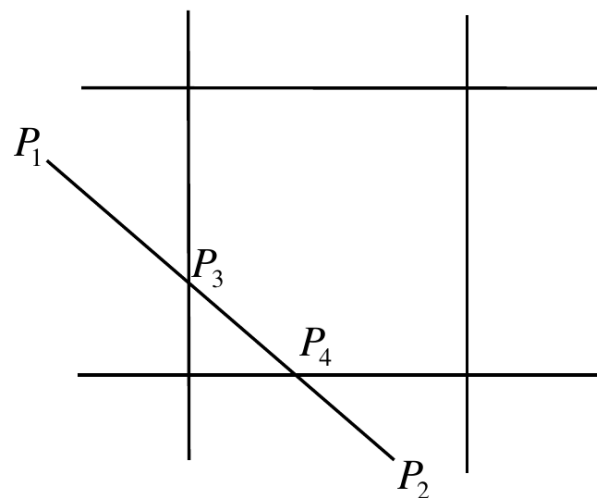




裁剪：直线段

- Cohen-Sutherland裁剪
- 三种情况：完全在外，完全在内，与边界相交
- $\text{code1} \mid \text{code2} == 0 \rightarrow$ 完全在内，保留
- $\text{code1} \& \text{code2} != 0 \rightarrow$ 完全在外，删除
- otherwise \rightarrow 求交点

100 1	100 0	101 0
000 1	000 0	001 0
010 1	010 0	011 0

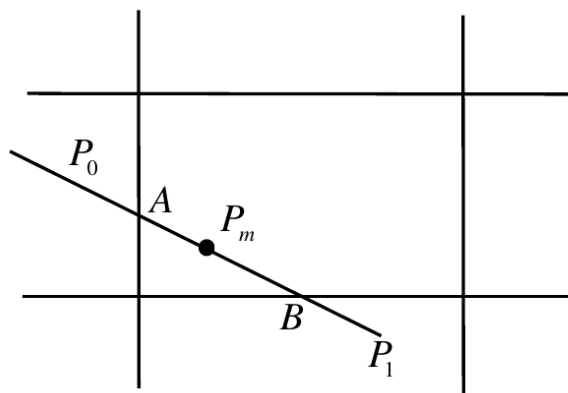




裁剪：直线段

- 中点分割裁剪算法

跟 Cohen-Sutherland 裁剪方法相似，但是使用另一种求交算法。

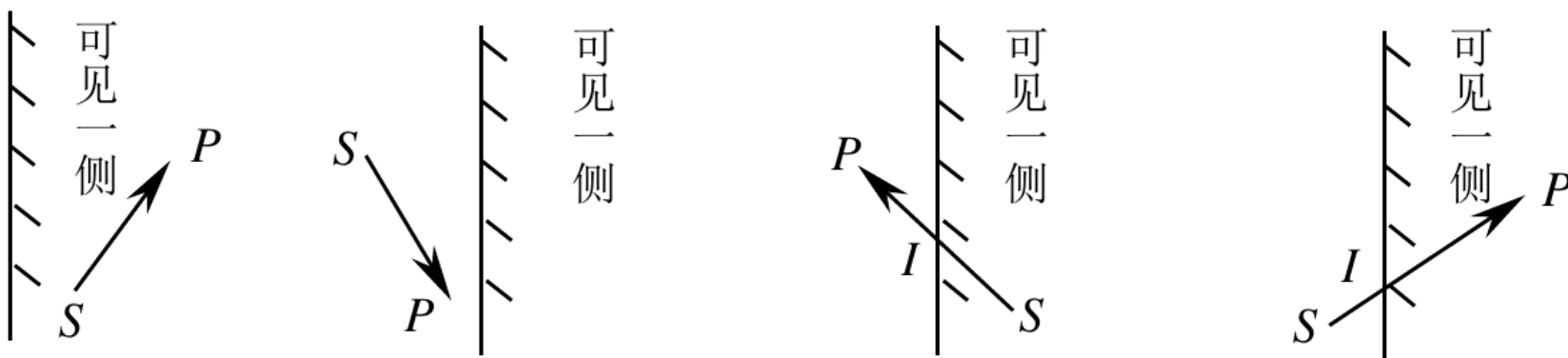


- 梁友栋 - Barskey算法
不做展开



裁剪：多边形

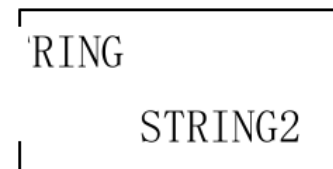
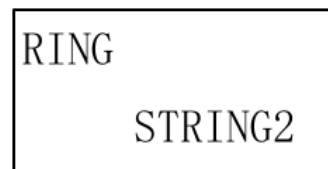
- Sutherland-Hodgeman 算法
 - 一次裁剪一个屏幕边界
 - 给多边形顶点规定顺序（顺时针或逆时针），按顺序裁剪边，得到新的顶点序列。





裁剪：字符

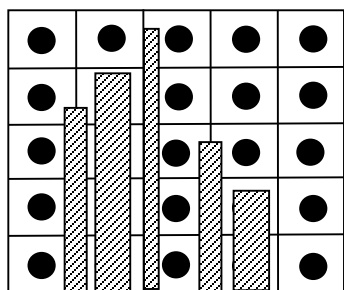
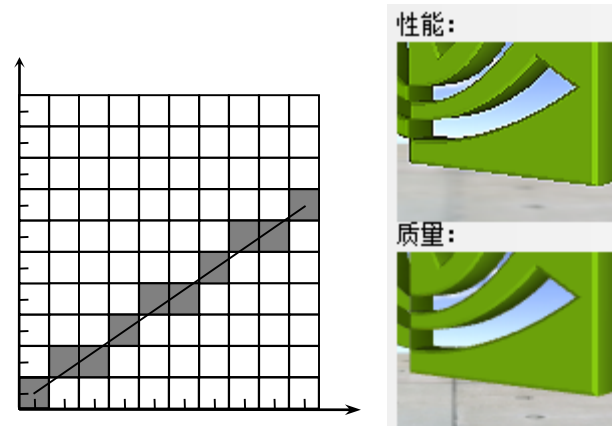
- 串精度裁剪
- 字符精度裁剪
- 笔画或像素精度裁剪



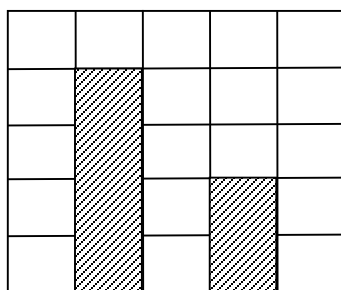


反走样：走样现象

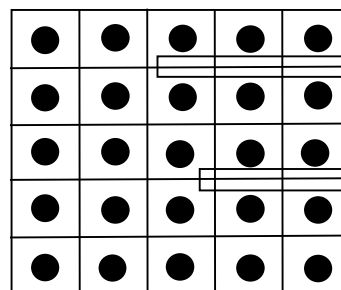
- 阶梯状边界
- 细节失真与遗失 (小于 1 pixel)



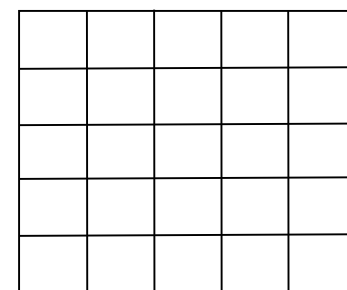
(a) 待显示的细小图形



(b) 显示结果



(a) 待显示的狭小矩形

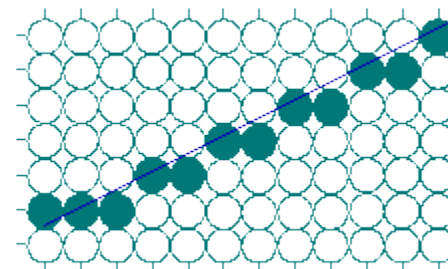
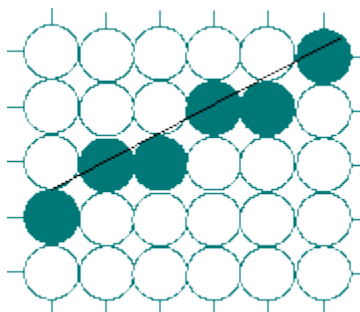


(b) 显示结果



反走样：提高分辨率

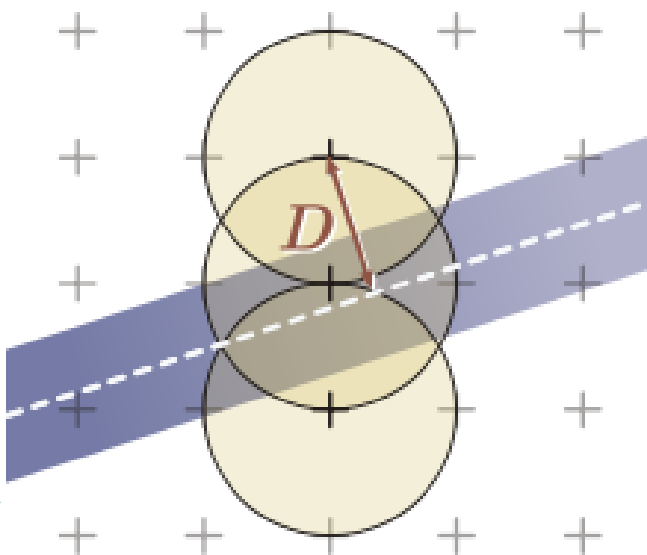
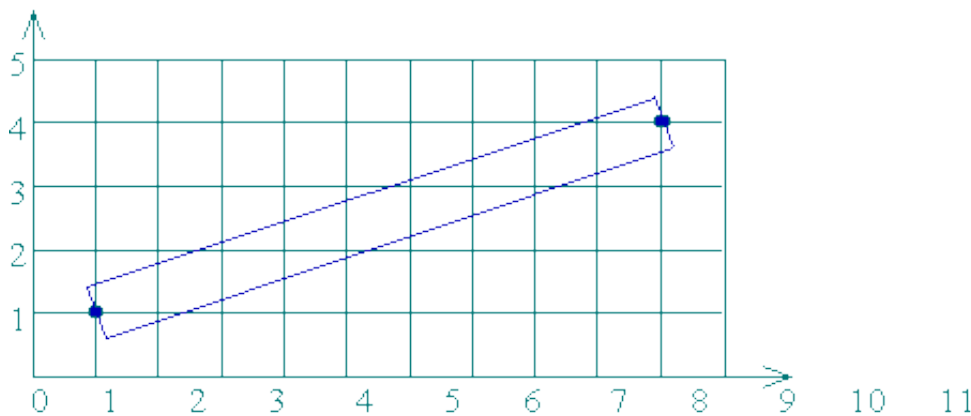
- 更高分辨率的屏
 - 4倍存储，2倍光栅化
- 高分辨率计算，低分辨率显示
 - SSAA [全视窗提高分辨率渲染]
 - MSAA [仅对模型边缘区域提高]





反走样：区域采样

- 像素有面积，根据覆盖面积或中心距离确定亮度值（公式见教材）





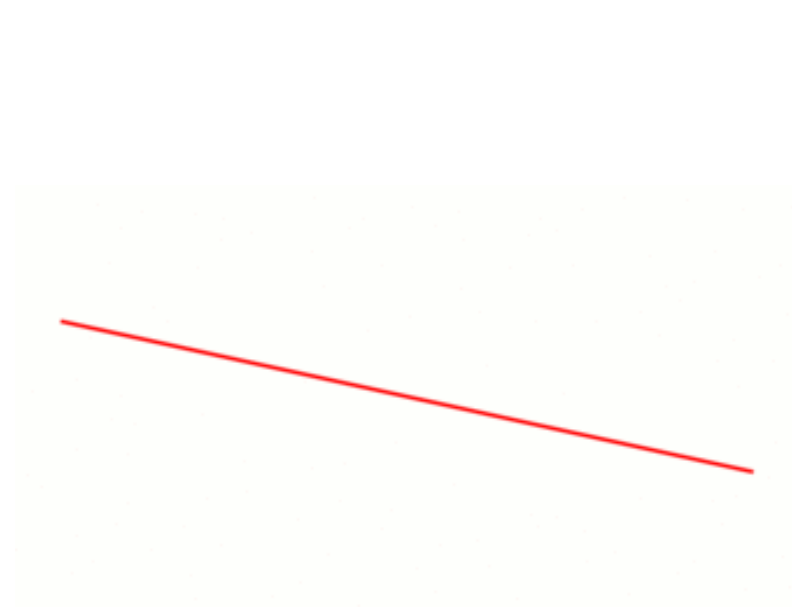
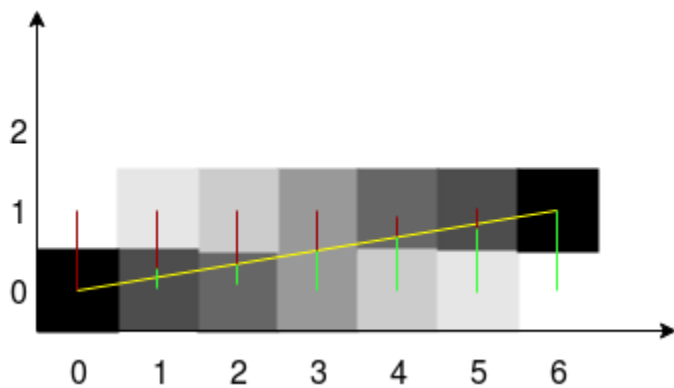
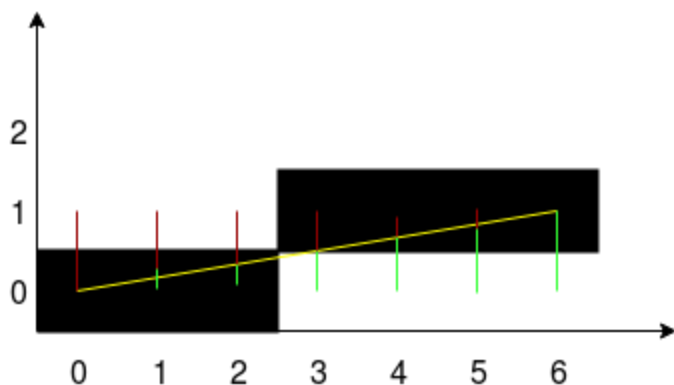
反走样：区域采样加速

- 将像素分成子像素，计算落入直线区域的子像素比例



反走样：区域采样加速

- Anti-aliasing version of Bresenham





反走样：加权区域采样

非加权区域采样方法的缺点

- 像素的亮度与相交区域位置无关，仍会导致锯齿效应
- 相邻两个像素有时会有较大的灰度差



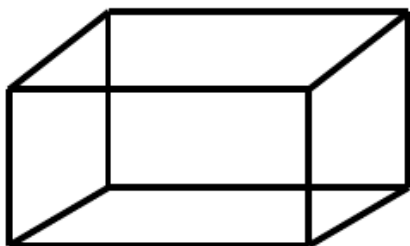
反走样：加权区域采样

- 使相交区域对像素亮度的贡献依赖于该区域与像素中心的距离，权重函数可以取高斯函数
- 积分计算量大，一般采用离散计算方法，将像素分成子像素计算

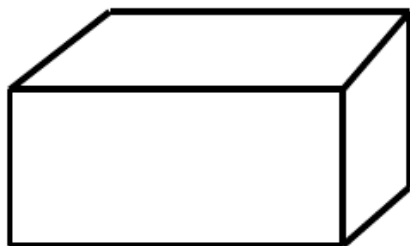


消隐

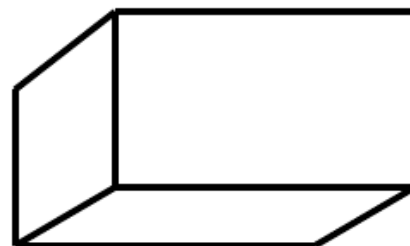
- 未经处理的投影图具有二义性，需要消除被遮挡的不可见的线或面，简称为 **消隐**。



(a)



(b)



(c)

具有二义性的长方体投影图



消隐：分类

- 按消隐对象分类：
 - 线消隐
 - 面消隐
- 按消隐空间分类：
 - 物体空间的消隐算法
 - 图像空间的消隐算法
 - 物体空间和图像空间的消隐算法



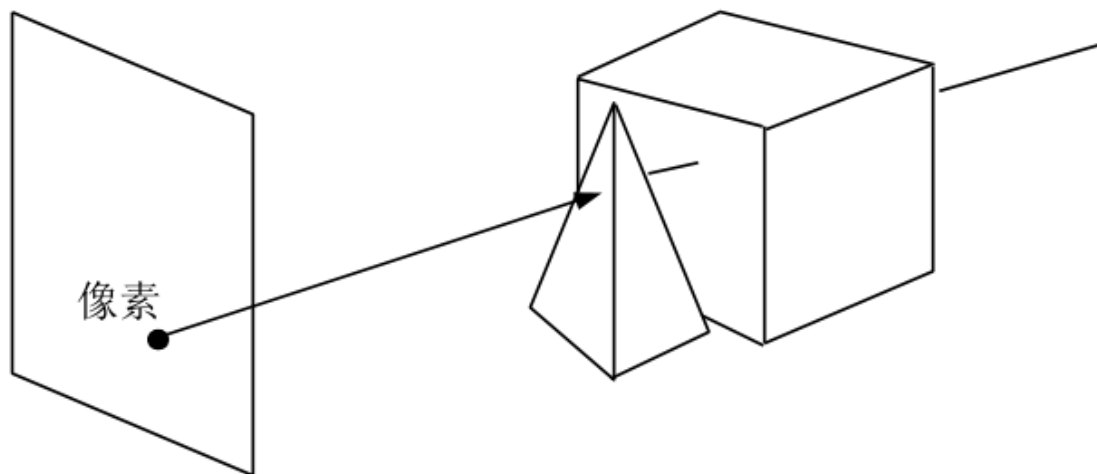
消隐：面消隐

- 画家算法
- **Z-buffer 算法**
- 扫描线 Z-buffer 算法
- 区间扫描线算法
- 区域子分割算法
- 光线投射算法
- ...



消隐：面消隐

- 光线投射算法





PA0: RasterGraphics



作业形式

- 给定一个代码框架，我们移除了一些核心部分的算法实现，需要你补充完整
- 每一个需要补充的位置都会加上TODO标记

```
class Line : public Element {  
  
public:  
    int xA, yA;  
    int xB, yB;  
    Vector3f color;  
    void draw(Image &img) override {  
        // TODO: Implement Bresenham Algorithm  
        printf("Draw a line from (%d, %d) to (%d, %d) using color (%f, %f, %f)\n", xA, yA, xB, yB,  
            color.x(), color.y(), color.z());  
    }  
};
```



代码讲解

- deps/vecmath: 矩阵和向量计算库
- image.hpp/cpp: 主要看图像是如何存储的

private:

```
    int width;
    int height;
    Vector3f *data;

    const Vector3f &GetPixel(int x, int y) const {
        assert(x >= 0 && x < width);
        assert(y >= 0 && y < height);
        return data[y * width + x];
    }

    void SetPixel(int x, int y, const Vector3f &color) {
        assert(x >= 0 && x < width);
        assert(y >= 0 && y < height);
        data[y * width + x] = color;
    }
}
```



代码讲解

- `canvas_parser.hpp/cpp`: 需要绘制的图形在 `testcases` 文件夹中以 `txt` 的形式描述, 其读取的代码已完整给出
- `element.hpp`: 定义了三种几何元素 (线段/圆/区域填充), 你需要实现其绘制算法。
 - 比如这个代码就画出了线段的两个端点。

```
class Line : public Element {  
  
public:  
    int xA, yA;  
    int xB, yB;  
    Vector3f color;  
    void draw(Image &img) override {  
        img.SetPixel(xA, yA, color);  
        img.SetPixel(xB, yB, color);  
    }  
};
```




框架代码环境配置

- Linux
 - Cmake/make
- Windows
 - 虚拟机/双系统安装Linux



Thank You !

Any Questions?