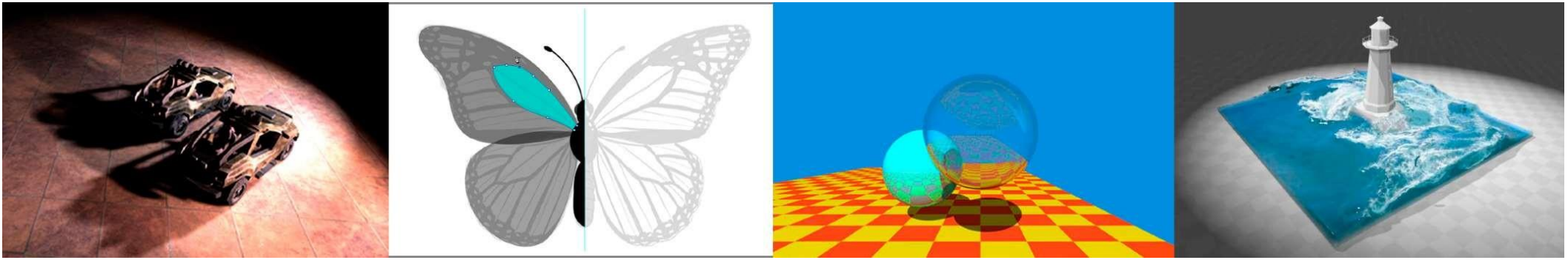
Computer Graphics

Rasterization 3
(Clipping, Hidden Removal)



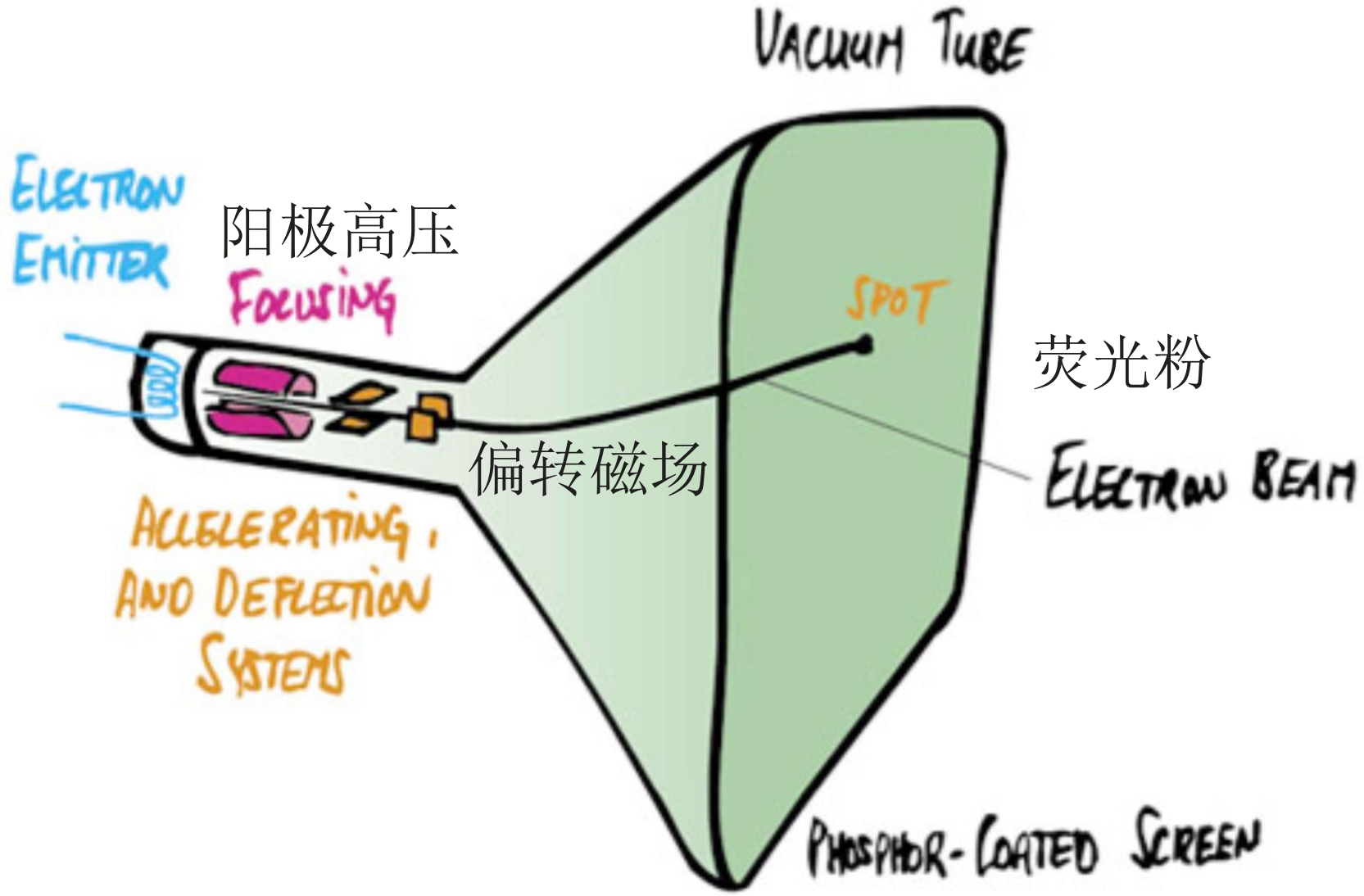
Last Lecture

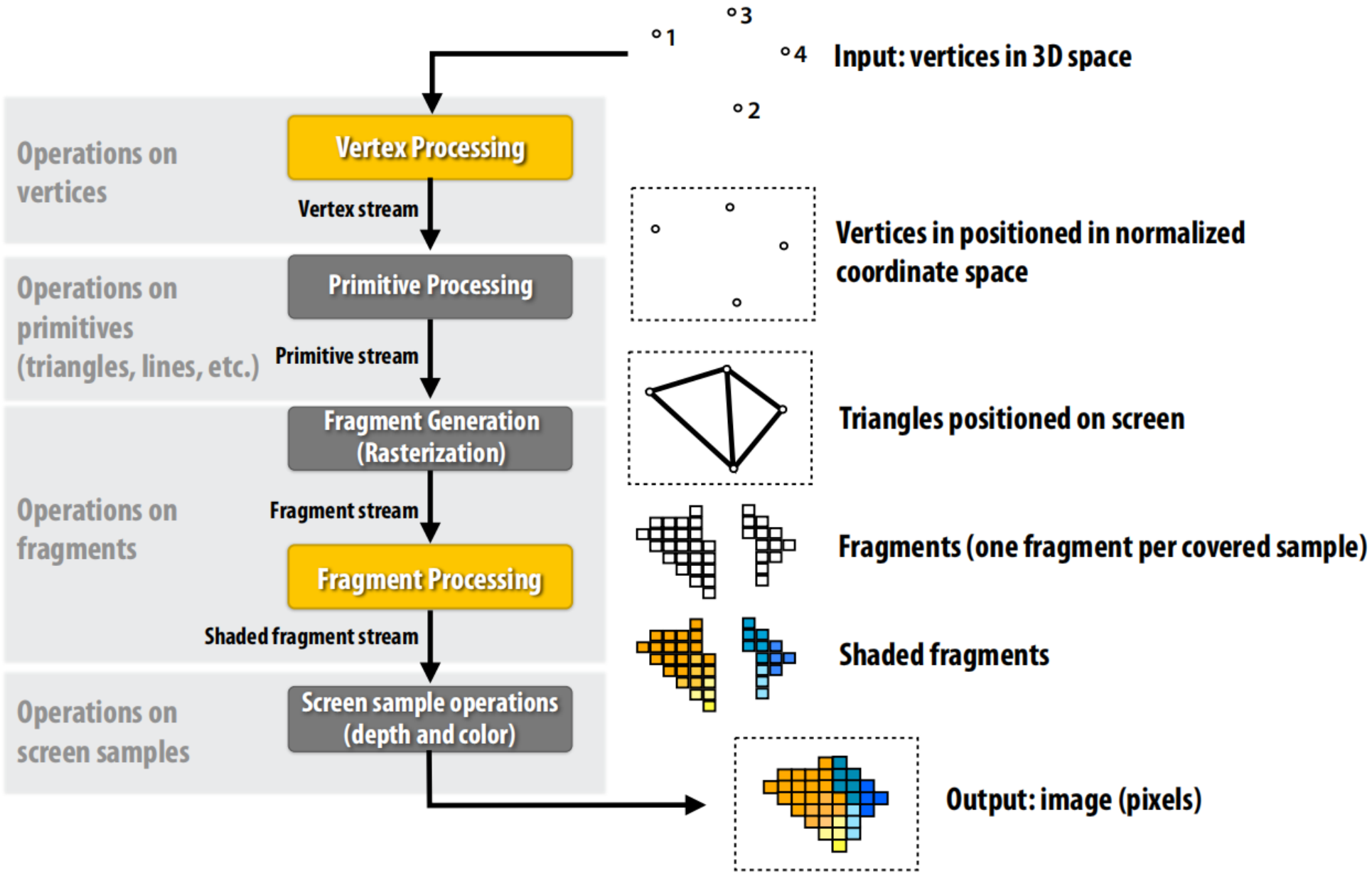
- Raster Display (光栅化显示器)
- Rasterization pipeline
- Rasterization (Line and Triangle)
- Antialiasing (反走样)

What is screen?

- What is a screen?
  - An array of pixels
  - Size of the array: resolution
  - A typical kind of raster display
- Raster == screen in German
  - Rasterize == drawing onto the screen
- Pixel (FYI, short for “picture element”)
  - For now: A pixel is a little square with uniform color
  - Color is a mixture of (red, green, blue)

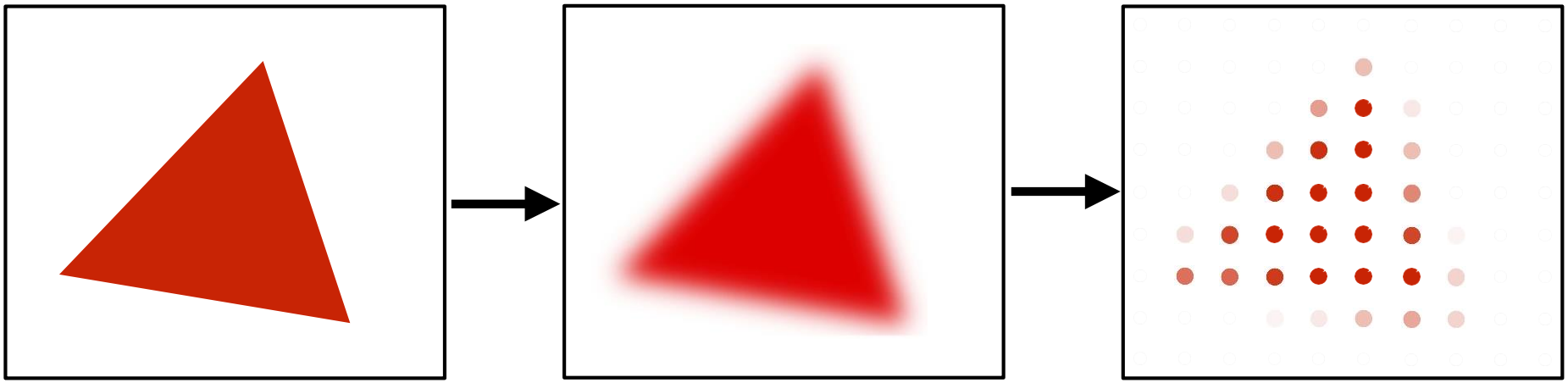
Cathode Ray Tube （阴极射线管）




Operations on vertices
Vertex Processing
Vertex stream
Operations on primitives (triangles, lines, etc.)
Primitive Processing
Primitive stream
Operations on fragments
Fragment Generation (Rasterization)
Fragment stream
Fragment Processing
Shaded fragment stream
Operations on screen samples
Screen sample operations (depth and color)
1
2
3
4
Input: vertices in 3D space
Vertices in positioned in normalized coordinate space
Triangles positioned on screen
Fragments (one fragment per covered sample)
Shaded fragments
Output: image (pixels)

Sampling Artifacts (Errors / Mistakes / Inaccuracies) in Computer Graphics

Rasterization: Antialiased Sampling



Pre-Filter

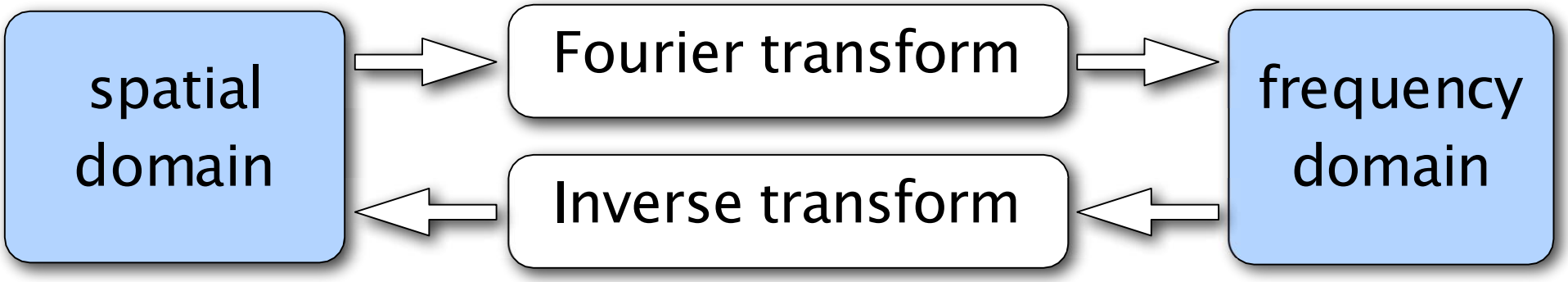
(remove frequencies above Nyquist) (?)

Sample

Note antialiased edges in rasterized triangle where pixel values take intermediate values

Fourier Transform Decomposes A Signal Into Frequencies

$f(x)$ $\quad F(\omega) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i \omega x} dx \quad$ $F(\omega)$



$$f(x) = \int_{-\infty}^{\infty} F(\omega) e^{2\pi i \omega x} d\omega$$

Recall $\quad e^{ix} = \cos x + i \sin x$

Convolution Theorem

Convolution in the spatial domain is equal to multiplication in the frequency domain, and vice versa

Option 1:

- Filter by convolution in the spatial domain

Option 2:

- Transform to frequency domain (Fourier transform)
- Multiply by Fourier transform of convolution kernel
- Transform back to spatial domain (inverse Fourier)

Convolution Theorem

Spatial Domain

$*$

$\frac{1}{9}$

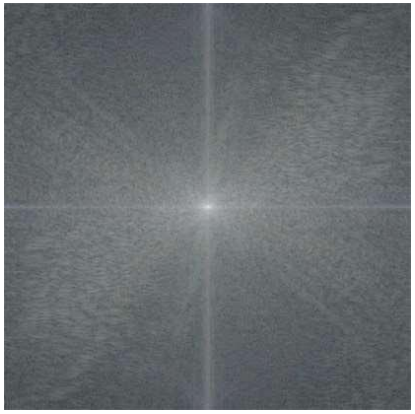
1	1	1
1	1	1
1	1	1

$=$

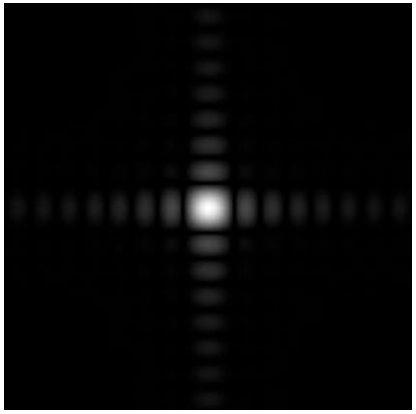
Fourier Transform

Inv. Fourier Transform

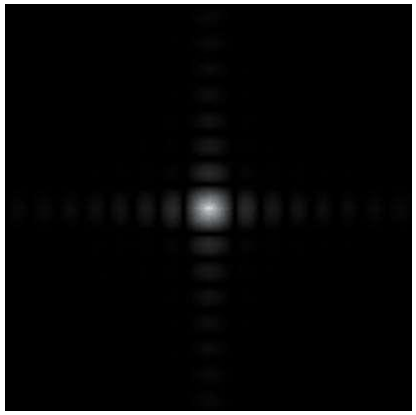
Frequency Domain



$\times$



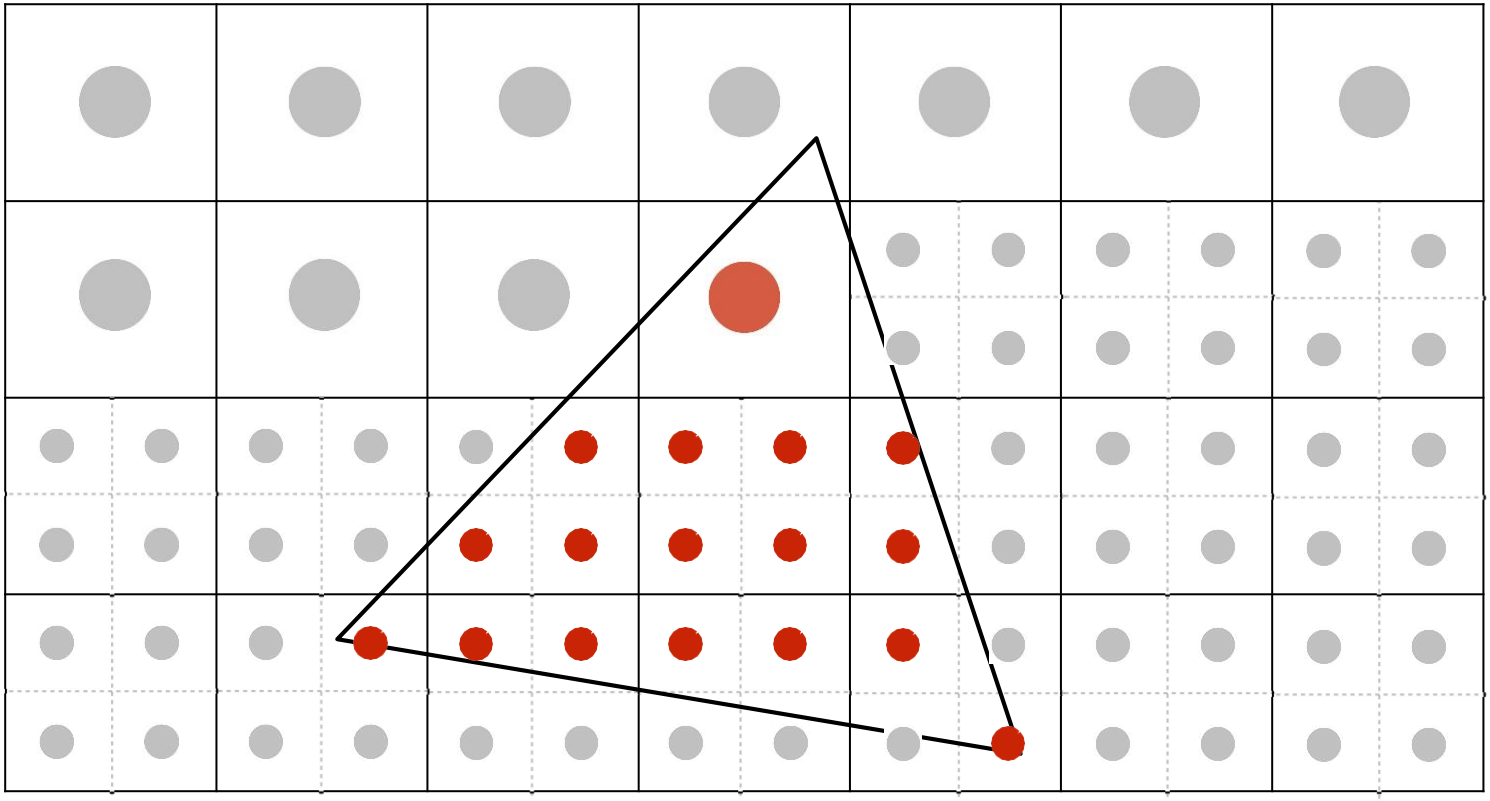
$=$



Multisample Anti-Aliasing
多重采样抗锯齿

Supersampling: Step 2

Average the NxN samples “inside” each pixel.



Averaging down

单选题 1分

设置

三角形的光栅化过程，属于光栅化流水线的哪一个环节？

A Vertex Processing

B Primitive Processing

C Fragment Generation

D Fragment Processing

提交

多选题 1分

设置

改善走样现象的正确方法有?

A Filter then sample

B Sample then filter

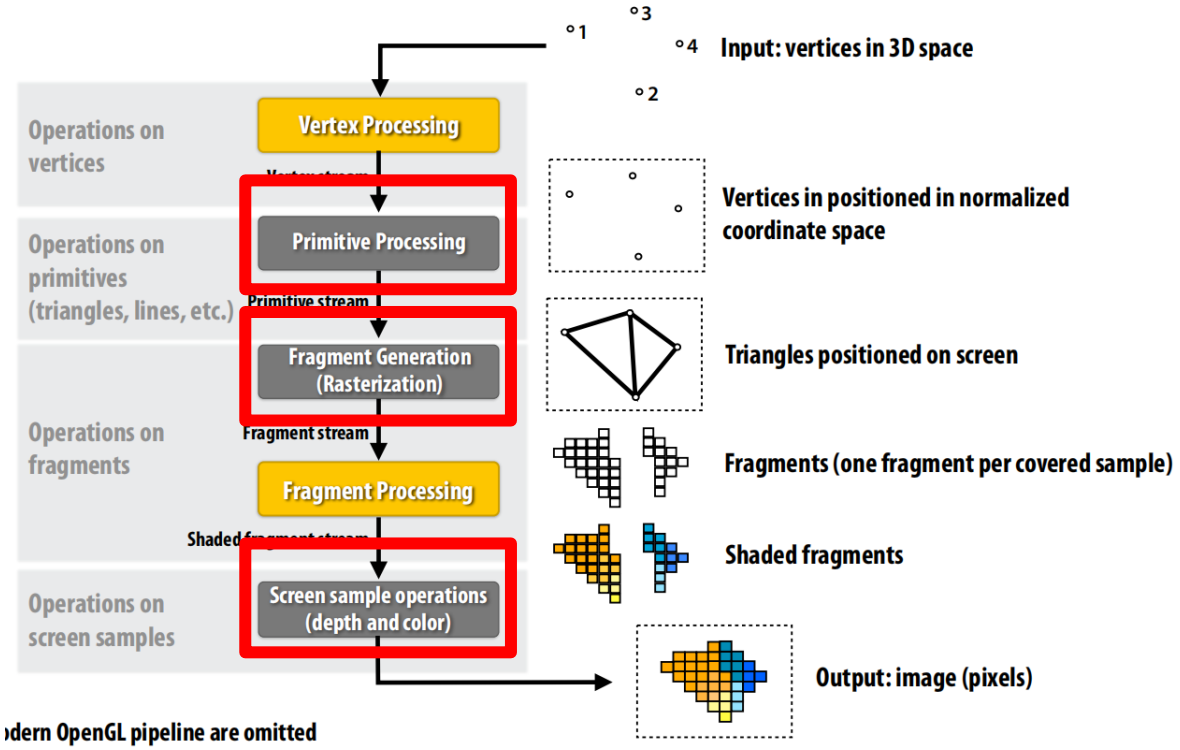
C Increase sampling rate

D MSAA

提交

Today

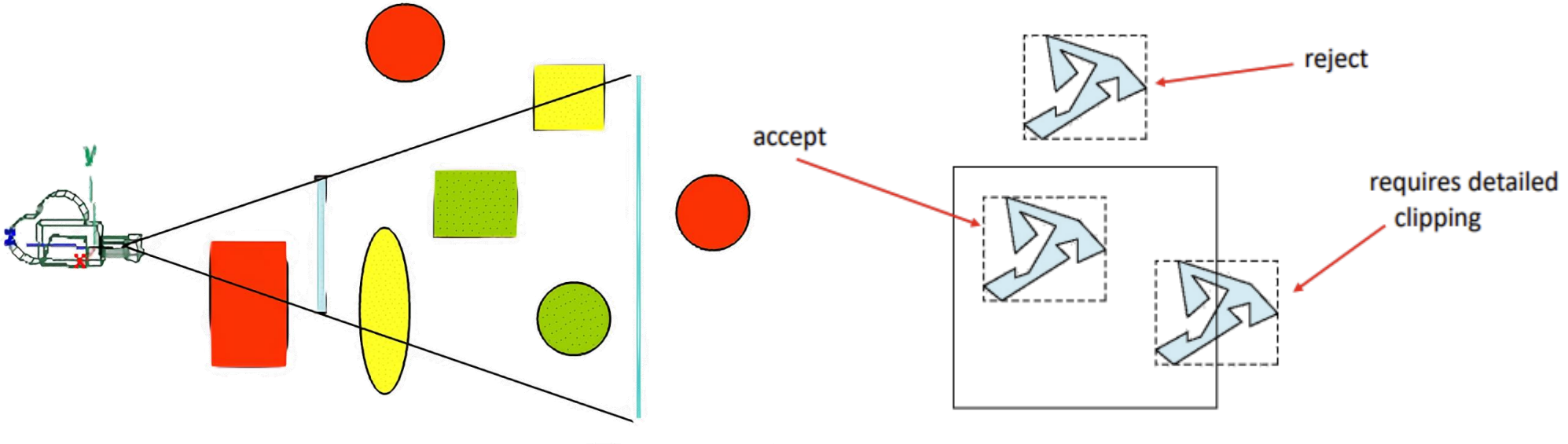
- Clipping (裁剪)
- Scan Conversion of Line Segments
  - 直线段的扫描转化
- Scan Line Polygon Filling
  - 扫描线多边形填充
- Hidden Removal (消隐)



Clipping

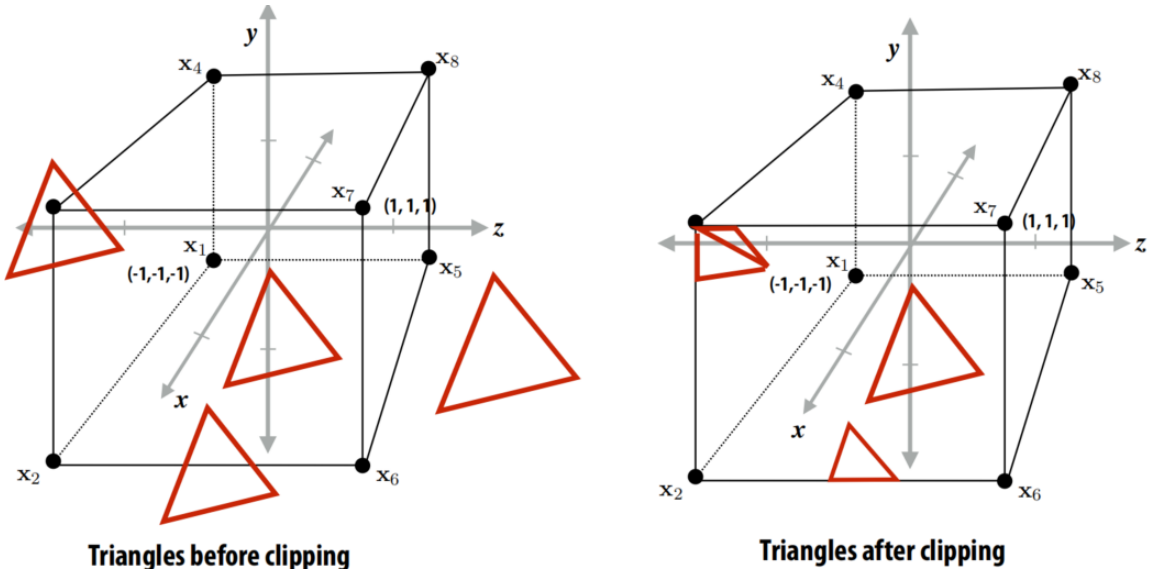
Clipping

- Some objects may go out of the frustum
- These objects (or part of objects) are invisible from the camera
- Clipping: eliminate regions that aren't visible from the camera



Clipping in normalized device coordinates (NDC)

- **Accept** primitives inside the view frustum
- **Reject** primitives outside the view frustum
- **Clip** primitives that extend beyond the view frustum



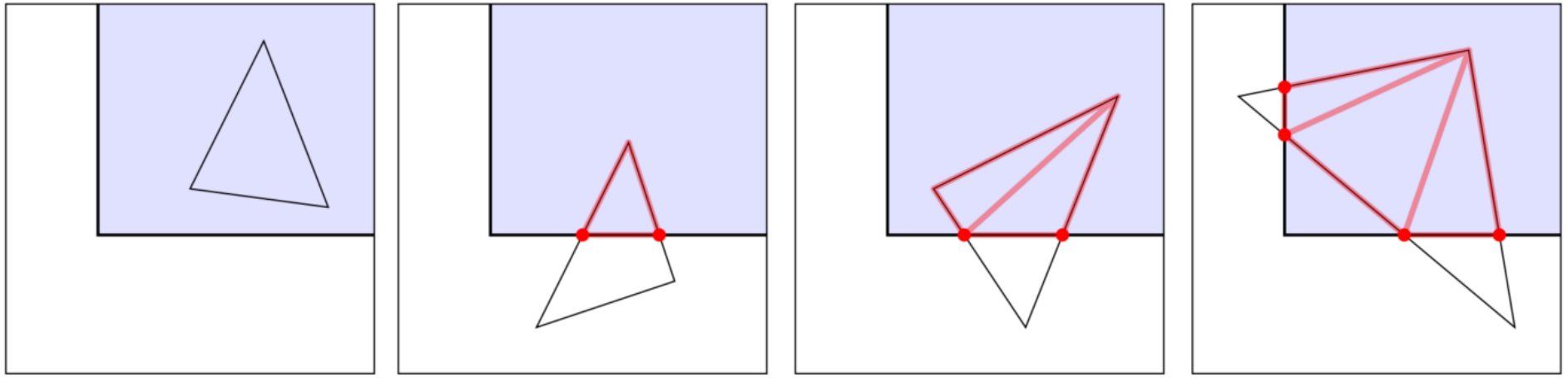
Triangles before clipping

Triangles after clipping

*Figures are correct: NDC for OpenGL (a popular graphics API) is left-handed coordinate space

Clipping in normalized device coordinates (NDC)

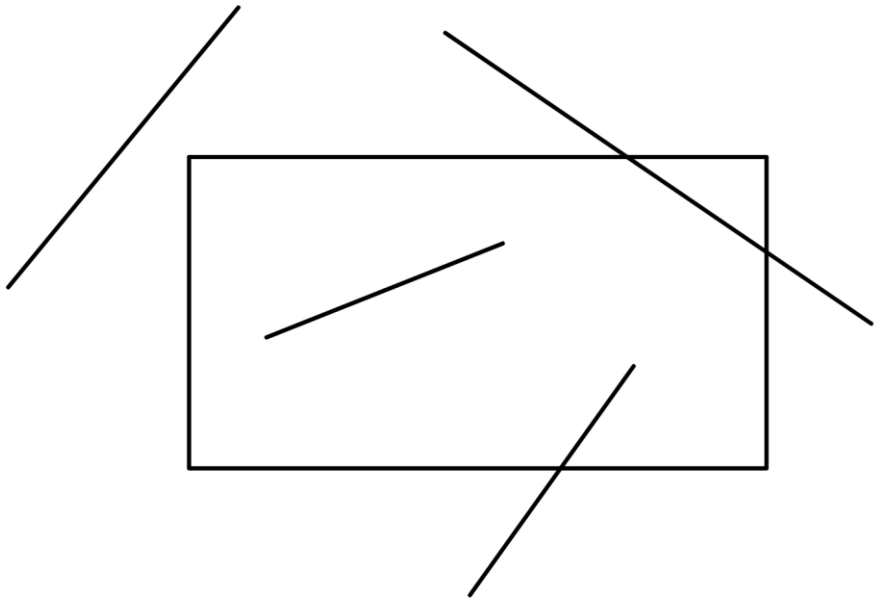
- **Accept** primitives inside the view frustum
- **Reject** primitives outside the view frustum
- **Clip** primitives that extend beyond the view frustum



https://paroj.github.io/gltut/

Clipping algorithms

- **There are many different primitives: segments, triangles, polygons**
- **Line segments:**
  - **Sutherland-cohen algorithm: consider 4 cases**

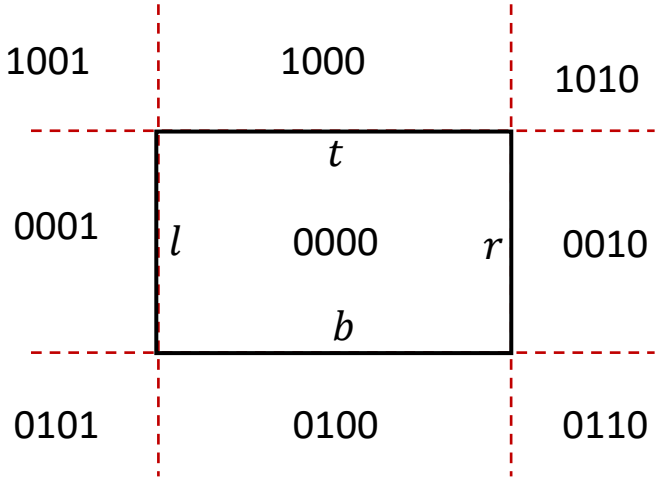


- 算法基本思想：问题分解，**Coarse2Fine**
  - 通过分类讨论简化问题
  - 解耦：非判定问题和求交问题

Sutherland-Cohen Algorithm

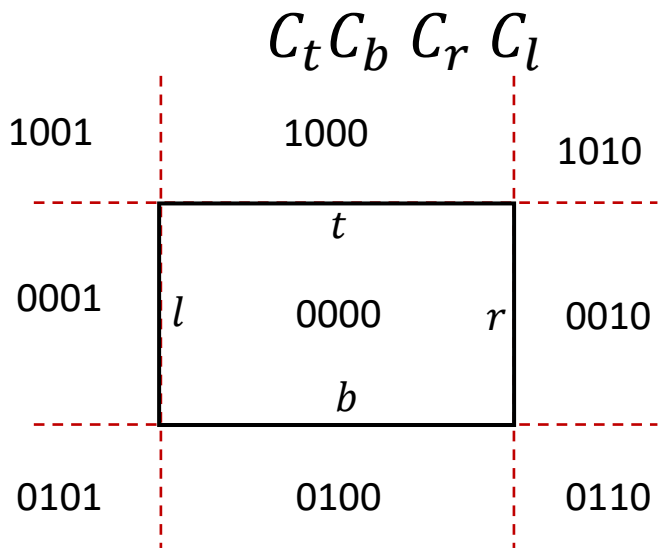
Step 1: Intersecion test

- **Divide the window into 9 regions with binary codes**



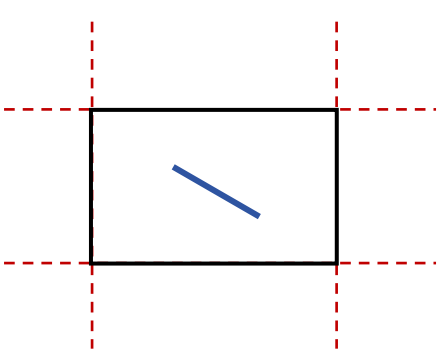
$$C_t C_b\ C_r\ C_l$$

Sutherland-Cohen Algorithm

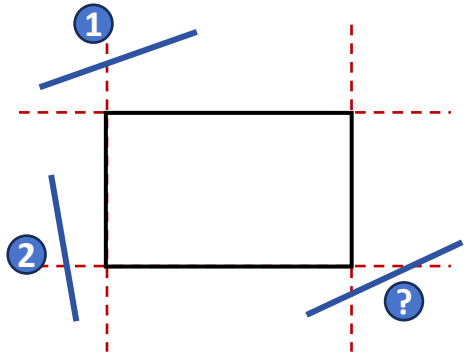


Step 1: Intersecion test

- **Divide the window into 9 regions with binary codes**
- **Query the code for each endpoint**
  (1) Two codes are 0000 and 0000: inside
  (2) "AND" operation and not obtain 0000: outside
  (3) Otherwise

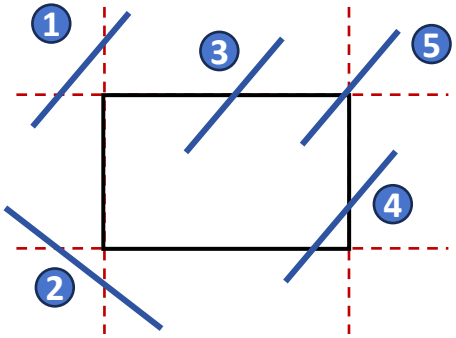


(1)



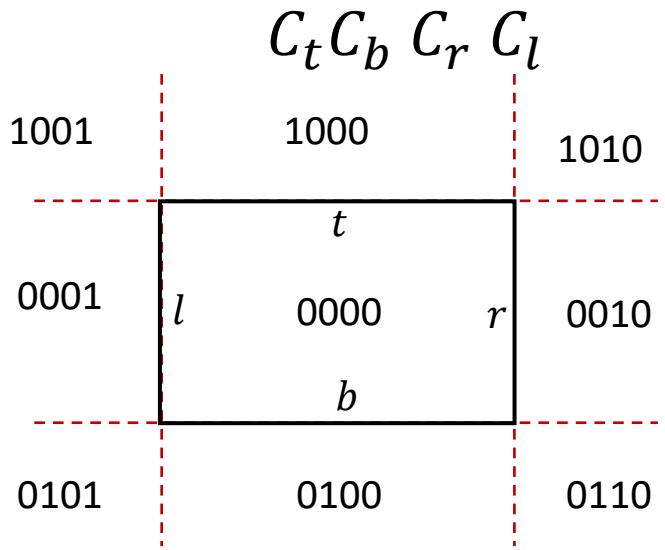
(2)

Whether both two endpoints are on top, bottom, left, right of the window



other

Sutherland-Cohen Algorithm



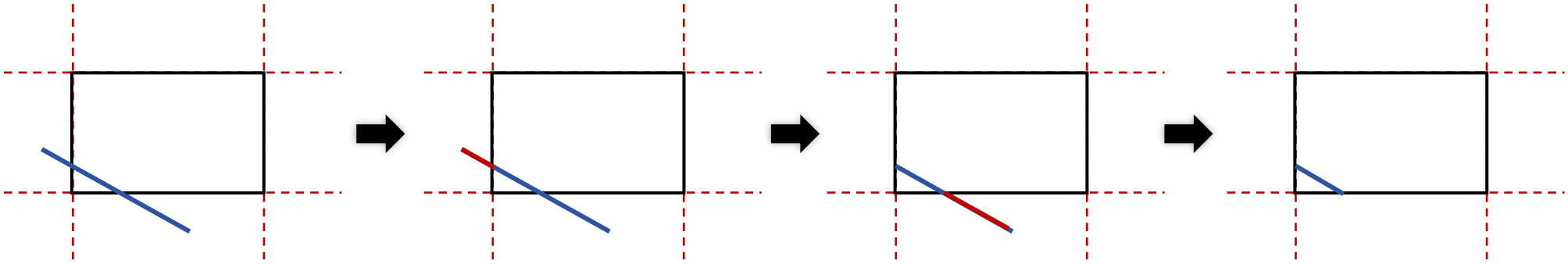
Step 1: Intersecion test

Step 2: Find the part of segment inside the viewing region

(4) Query the horizontal/vertical line ("or" operation)

(5) Compute the intersection point

(6) Remove the part outside window, repeat step1 for the other part



Simple to implement!

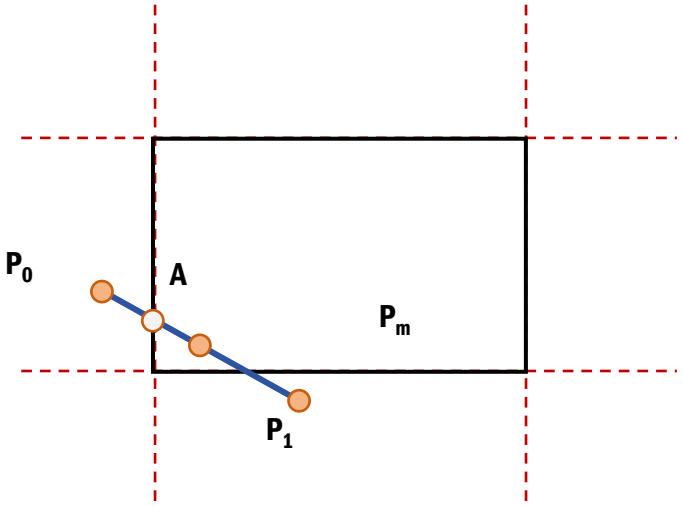
中点分割裁剪算法

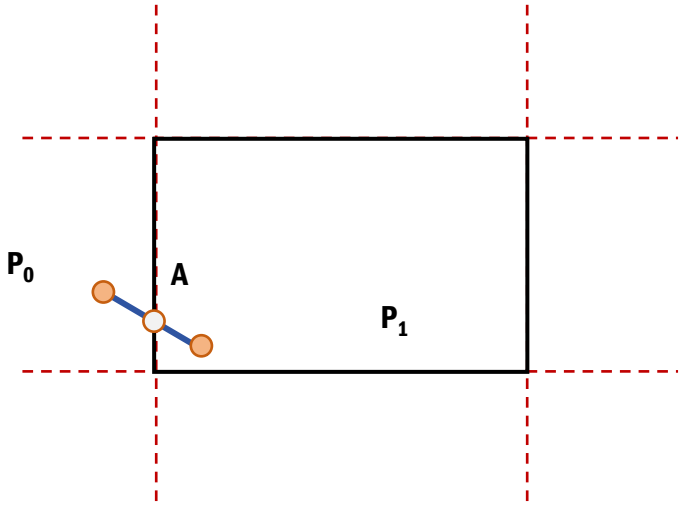
Step 1: Same to Sutherland-Cohen Algorithm

Step 2: Find the part of segment inside the viewing region

Another option: looking for A from P_0, and looking for B from P_1

- **Midpoing partition: Compute midpoint P_m, if P_0P_m visible, replace P_0P_1 by P_0P_m**





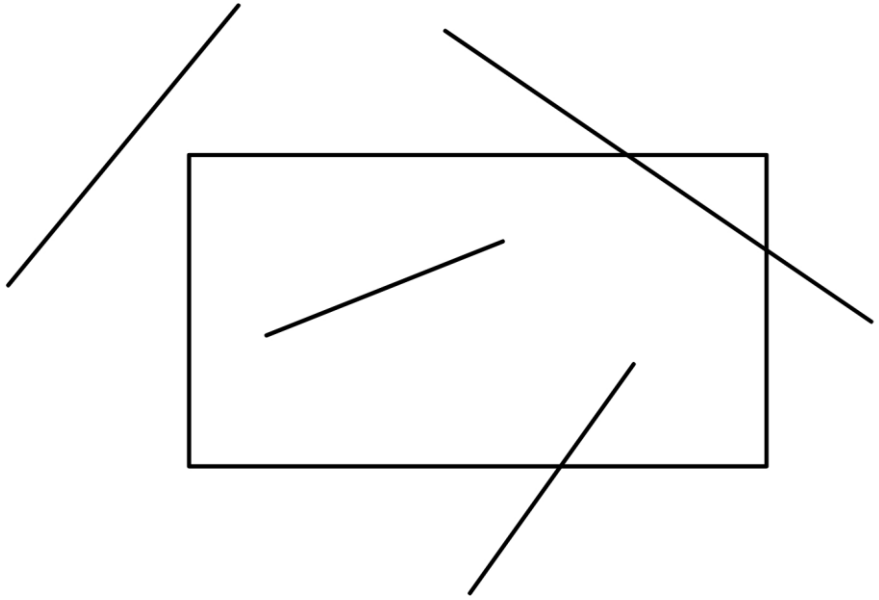
.....

Only addition and division by 2, easy to implement in the hardware!

递归式的求交点方法
避免进行求交计算

Clipping algorithms

- **There are many different primitives: segments, triangles, polygons**
- **Line segments:**
  - **Sutherland-cohen algorithm: consider 4 cases**

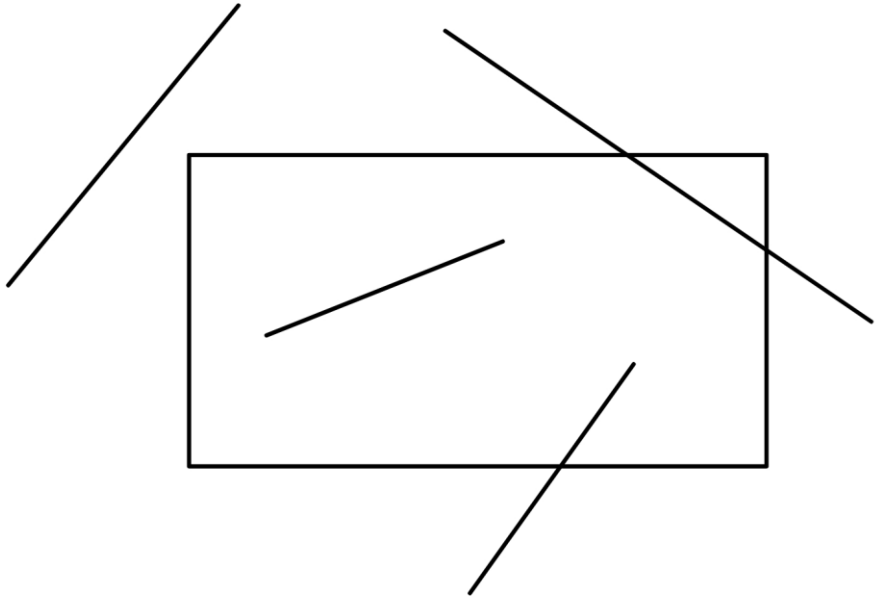


- 算法基本思想：问题分解，**Coarse2Fine**
  - 通过分类讨论简化问题
  - 解耦：非判定问题和求交问题

优缺点是什么？大家想一想？

Clipping algorithms

- **There are many different primitives: segments, triangles, polygons**
- **Line segments:**
  - **Sutherland-cohen algorithm: consider 4 cases**



- 算法基本思想：问题分解
  - 通过分类讨论简化问题
  - 解耦：非判定问题和求交问题

- 优点：思路简单，易于实现
- 缺点：效率低，两步通常比单步慢，没有充分利用判定问题和求交问题中的重合部分

Liang-baskey algorithm

梁友栋先生出生于1935年7月，福建福州人。1956-1960年于复旦大学作为研究生师从苏步青先生学习几何理论，1960年研究生毕业后任教于浙江大学数学系，1984-1990年任数学系主任。80年代初梁友栋先生提出了著名的Liang-Barsky裁剪算法，通过线段的参数化表示实现快速裁剪，至今仍是计算机图形学中最经典的算法之一。

加州大学伯克利分校的Brian A. Barsky(柏世奇)教授是计算机图形学领域的先行者，他先后从康奈尔大学和犹他大学获得硕士和博士学位。计算机辅助设计与建模、交互式真实感三维计算机图形学、科学计算可视化、计算机辅助角膜建模与可视化、医学摄影以及用于手术模拟的虚拟空间建模等。

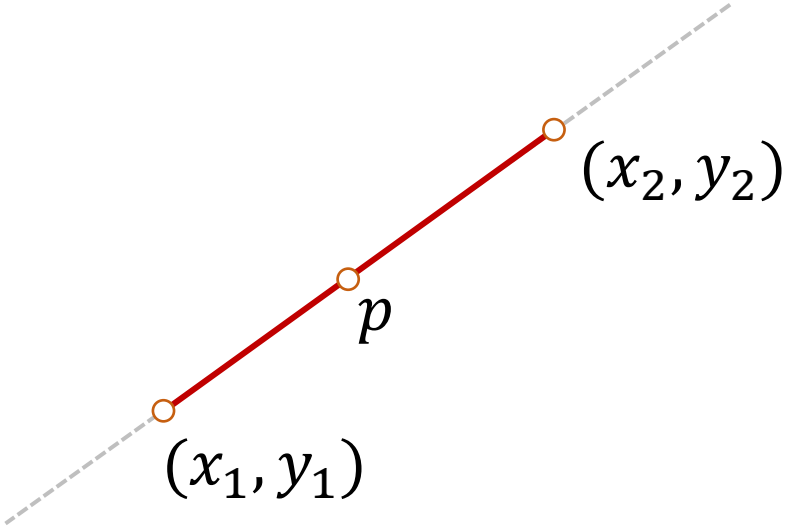
他长期致力于spline曲线和曲面造型研究，并被广泛应用于计算机图形学和几何建模。1984年与梁友栋先生提出著名的Liang-Barsky算法。

Liang-baskey algorithm

- **Which part of the line segment is visible?**
- **How do we present a segment?** 目标segment如何表达？参数化!

$$\Delta x = x_2 - x_1,\ \Delta y = y_2 - y_1$$

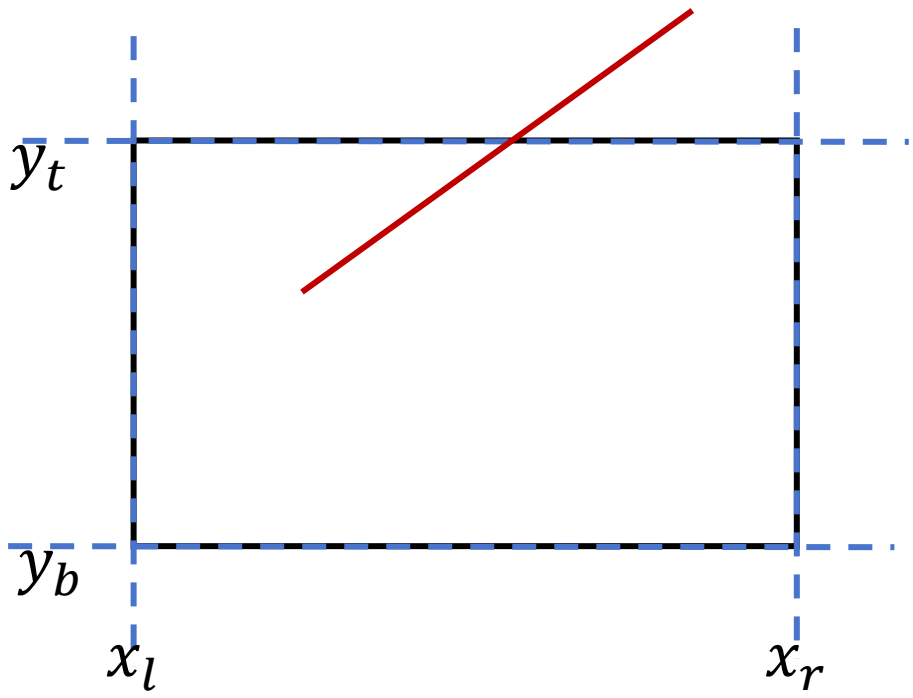
$$(x_1, y_1) + u(\Delta x, \Delta y)$$



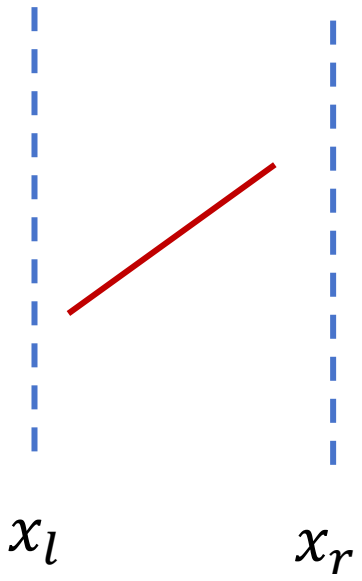
Liang-baskey algorithm

- **Which part of the line segment is visible?**

$$x_l \le x_1 + u\Delta x \le x_r$$
$$y_b \le y_1 + u\Delta y \le y_t$$
$$\Delta x = x_2 - x_1,\ \Delta y = y_2 - y_1$$



Liang-baskey algorithm



- **Which part of the line segment is visible?**

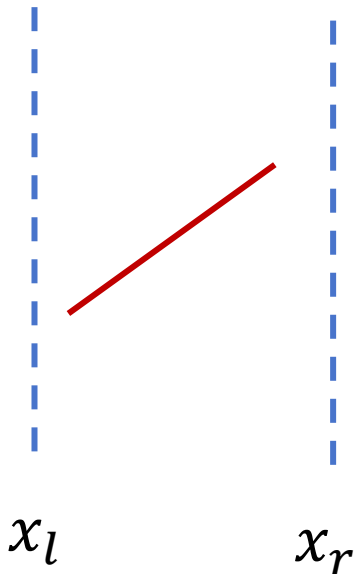
$$x_l \overset{1}{\leq} x_1 + u\Delta x \overset{2}{\leq} x_r$$

$$\Delta x = x_2 - x_1,\ \Delta y = y_2 - y_1$$

- **Rewrite as** $up_k \leq q_k$

1. $-u\Delta x \leq x_1 - x_l$ 1. $\begin{cases} p_1 = -\Delta x \\ q_1 = x_1 - x_l \end{cases}$
2. $u\Delta x \leq x_r - x_1$ 2. $\begin{cases} p_2 = \Delta x \\ q_2 = x_r - x_1 \end{cases}$

Liang-baskey algorithm



- **Which part of the line segment is visible?**

$$x_l \overset{①}{\leq} x_1 + u\Delta x \overset{②}{\leq} x_r$$

$$\Delta x = x_2 - x_1,\ \Delta y = y_2 - y_1$$

- **Rewrite as** $up_k \leq q_k$

① $-u\Delta x \leq x_1 - x_l$ ① $\begin{cases} p_1 = -\Delta x \\ q_1 = x_1 - x_l \end{cases}$

② $u\Delta x \leq x_r - x_1$ ② $\begin{cases} p_2 = \Delta x \\ q_2 = x_r - x_1 \end{cases}$

如果$p_k < 0$： **入边**

如果$p_k > 0$： **出边**

1. 如果Δx=0，线段平行于X
   1. $q_k < 0$, **outside**
   2. $q_k \geq 0, inside$
2. 如果$\Delta x \neq 0$, 线段不平行
   - **Compute all $r_k = q_k / p_k$**
   - **u_1 is max(0, r_{in})**
   - **u_2 is min(1, r_{out})**
   - **If $u_1 > u_2$, outside the window**
   - **else, the part is from u_1 -> u_2**

u_1 and u_2 defines the part of segment that is inside the window

Liang-baskey algorithm

- **Which part of the line segment is visible?**

$$x_l \overset{1}{\leq} x_1 + u\Delta x \overset{2}{\leq} x_r$$

$$y_b \overset{3}{\leq} y_1 + u\Delta y \overset{4}{\leq} y_t$$

$$\Delta x = x_2 - x_1,\ \Delta y = y_2 - y_1$$

- **Rewrite as** $up_k \leq q_k$

$$\text{1)}\begin{cases} p_1 = -\Delta x \\ q_1 = x_1 - x_l \end{cases} \quad \text{2)}\begin{cases} p_2 = \Delta x \\ q_2 = x_r - x_1 \end{cases} \quad \text{3)}\begin{cases} p_3 = -\Delta y \\ q_3 = y_1 - y_b \end{cases} \quad \text{4)}\begin{cases} p_4 = \Delta y \\ q_4 = y_t - y_1 \end{cases}$$

Liang-baskey algorithm

$$x_l \le x_1 + u\Delta x \le x_r$$

$$y_b \le y_1 + u\Delta y \le y_t$$

$$\Delta x = x_2 - x_1,\ \Delta y = y_2 - y_1$$

$$up_k \le q_k$$

$$\begin{cases} p_1 = -\Delta x \\ q_1 = x_1 - x_l \end{cases} \quad \begin{cases} p_2 = \Delta x \\ q_2 = x_r - x_1 \end{cases} \quad \begin{cases} p_3 = -\Delta y \\ q_3 = y_1 - y_b \end{cases} \quad \begin{cases} p_4 = \Delta y \\ q_4 = y_t - y_1 \end{cases}$$

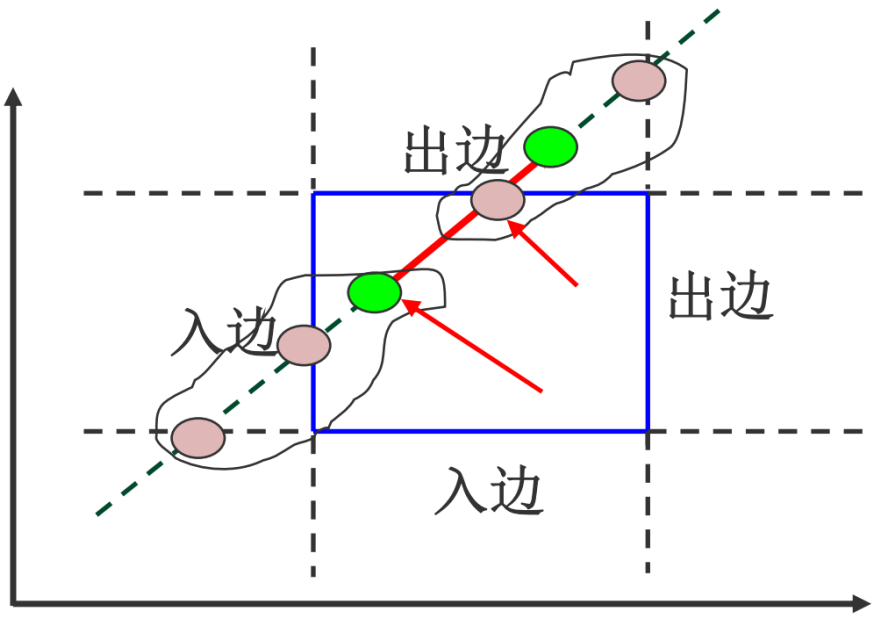
- u_1 and u_2 defines the part of segment that is inside the window
  - Compute all $r_k = q_k / p_k$
  - u_1 is max(0, r_{in1}, r_{in2})
  - u_2 is min(1, r_{out1}, r_{out2})
  - If $u_1 > u_2$, the segment is completely outside the window
  - else, the part is from u_1 -> u_2

Liang-baskey algorithm

梁算法的主要思想：

（2）把被裁剪的红色直线段看成是一条有方向的线段，把窗口的四条边分成两类：

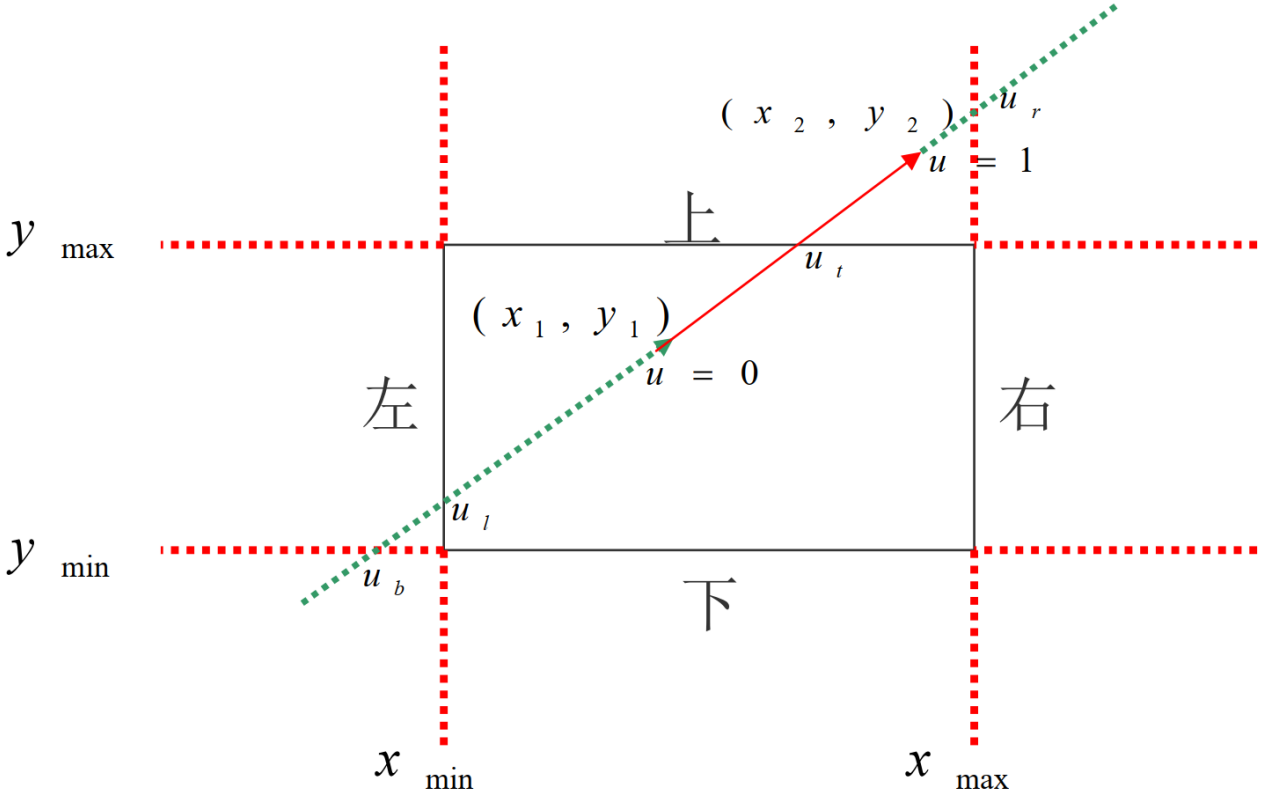
入边和出边



裁剪结果的线段起点是直线和两条入边的交点以及始端点三个点里最前面的一个点，即参数u最大的那个点；

裁剪线段的终点是和两条出边的交点以及端点最后面的一个点，取参数u最小的那个点。

Liang-baskey algorithm



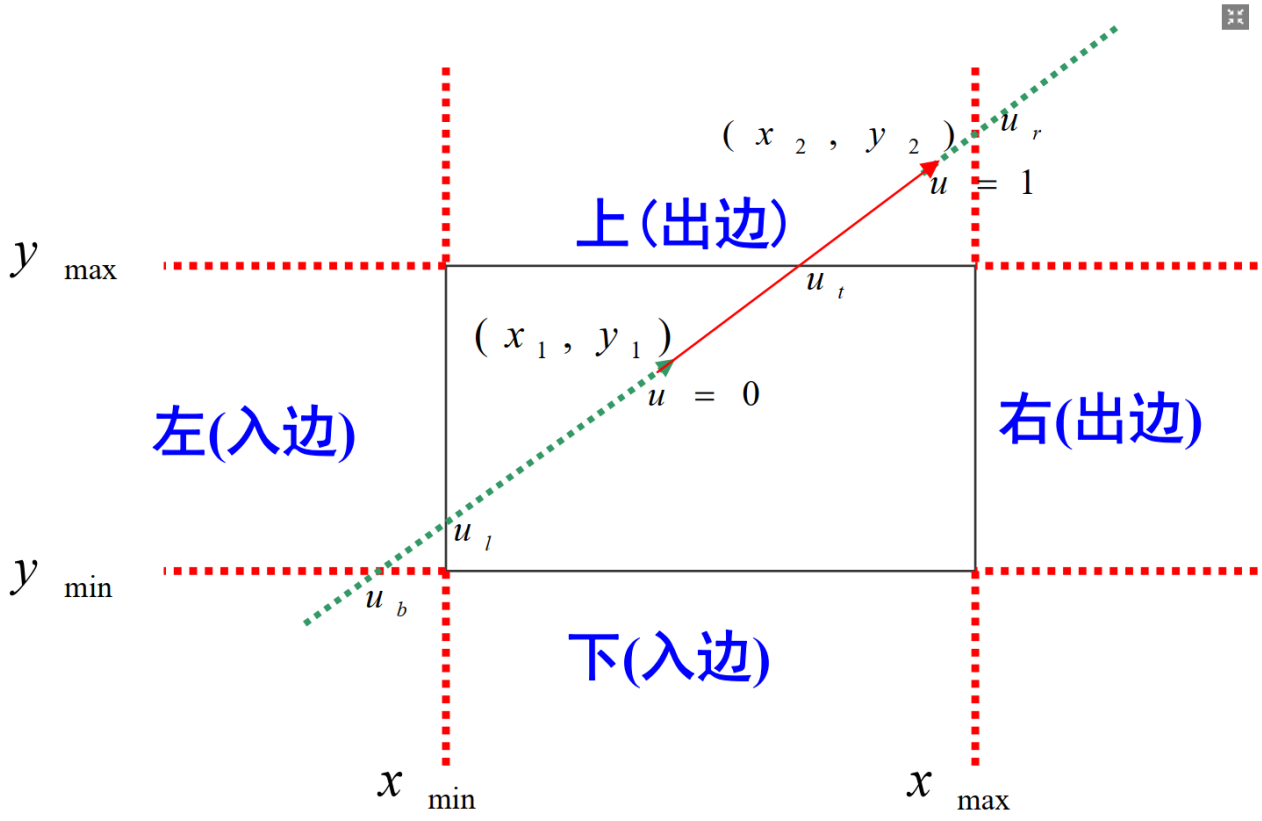
值得注意的是，当u从−∞到+∞遍历直线时，首先对裁剪窗口的两条边界直线（下边和左边）从外面向里面移动，再对裁剪窗口两条边界直线（上边和右边）从里面向外面移动。

Liang-baskey algorithm

如果用u_1，u_2分别表示线段（$u_1 \leqslant u_2$）可见部分的开始和结束

$$u_1 = \max(0, u_l, u_b)$$

$$u_2 = \min(1, u_t, u_r)$$



这就是梁先生的重大发现！

Liang-baskey algorithm

3. 梁友栋-Barskey 裁剪算法

梁友栋和 Barskey 提出了更快的参数化裁剪算法。首先按参数化形式写出裁剪条件：

$$\begin{cases} XL \leqslant x_1 + u\Delta x \leqslant XR, & \Delta x = x_2 - x_1 \\ YB \leqslant y_1 + u\Delta y \leqslant YT, & \Delta y = y_2 - y_1 \end{cases}$$

这 4 个不等式可以表示为统一的形式：$up_k \leqslant q_k$。

其中，参数 p_k, q_k 定义为：

$$p_1 = -\Delta x, \quad q_1 = x_1 - XL; \quad p_2 = \Delta x, \quad q_2 = XR - x_1$$

$$p_3 = -\Delta y, \quad q_3 = y_1 - YB; \quad p_4 = \Delta y, \quad q_4 = YT - y_1$$

对于任何平行于裁剪边界之一的直线 $p_k = 0$，其中 k 对应于裁剪边界（$k = 1, 2, 3, 4$，对应于左、右、下、上边界）；如果还满足 $q_k < 0$，则线段完全在边界外，舍弃该线段；如果 $q_k \geqslant 0$，则该线段平行于裁剪边界并且在窗口内。

当 $p_k < 0$ 时，线段从裁剪边界所在直线的外部指向内部[1]。当 $p_k > 0$ 时，线段从裁剪边界所在直线的内部指向外部。当 $p_k \neq 0$ 时，可以计算出线段与边界 k 的延长线交点的 u 值：$u = q_k / p_k$。

对于每条直线段，可以计算出参数 u_1 和 u_2，它们定义了在裁剪矩形内的线段部分。u_1 的值由线段从外到内遇到的矩形边界所决定（$p < 0$）。对这些边界计算 $r_k = q_k / p_k$，u_1 取 0 和各个 r_k 值之中的最大值。u_2 的值由线段从内到外遇到的矩形边界所决定（$p > 0$）。对这些边界计算 $r_k = q_k / p_k$，u_2 取 1 和各个 r_k 值之中的最小值。如果 $u_1 > u_2$，则线段完全落在裁剪窗口之外，被舍弃；否则裁剪线段由参数 u 的两个值 u_1, u_2 计算出来。

梁友栋-Barskey 算法程序如下。

算法程序 2.12　梁友栋-Barskey 裁剪算法

```
void LB_LineClip(x1,y1,x2,y2,XL,XR,YB,YT)
float x1,y1,x2,y2,XL,XR,YB,YT;
{
    float dx,dy,u1,u2;
    t1=0;tu=1;
    dx =x2-x1;
    dy =y2-y1;
    if(ClipT(-dx,x1-Xl,&u1,&u2)
      if(ClipT(dx,XR-x1, &u1,&u2)
        if(ClipT(-dy,y1-YB, &u1,&u2)
          if(ClipT(dy,YT-y1, &u1,&u2)
            displayline(x1+u1 * dx,y1+u1 * dy, x1+u2 * dx,y1+u2 * dy);
}
bool ClipT(p,q,u1,u2)
float p,q, * u1, * u2;
{
  float r;
  if(p<0)
  {
    r=q/p;
    if(r> * u2)
      return FALSE;
    if(r> * u1)
      * u1=r;
  }
  else if(p>0)
  {
    r=q/p;
    if(r< * u1)
      return FALSE;
    if(r< * u2)
      * u2=r;
  }
  else return (q>=0);
  return TRUE;
}
```

2.5.2　多边形裁剪

对于一个多边形，可以把它分解为边界的线段逐段进行裁剪，但这样做会使原来封闭的多边形变成不封闭的或者一些离散的线段。当多边形作为实区域考虑时，封闭的多边形裁剪后仍应当是封闭的多边形，以便进行填充。为此，可以使用 Sutherland-Hodgman 算法，该算法的基本思想是一次用窗口的一条边裁剪多边形。

在算法的每一步中，仅考虑窗口的一条边以及延长线构成的裁剪线。该线把平面分成两个部分：一部分包含窗口，称为可见一侧；另一部分称为不可见一侧。依序考虑多边形各条边的两端点 S、P，它们与裁剪线的位置关系只有如下 4 种（如图 2.20 所示）。

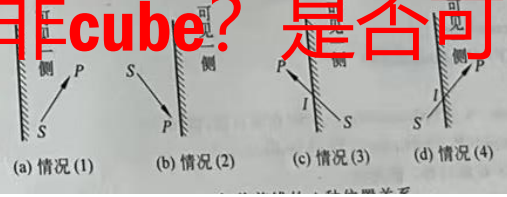
(1) S、P 均在可见一侧。

(2) S、P 均在不可见一侧。

(3) S 可见，P 不可见。

(4) S 不可见，P 可见。

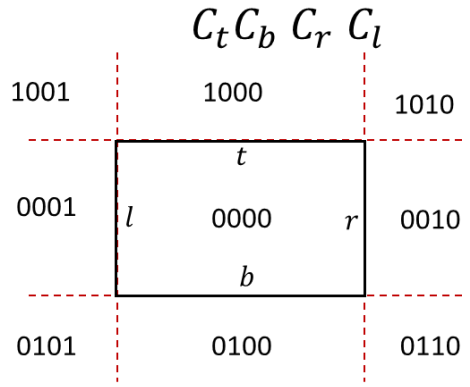
将每条线段端点 S、P 与裁剪线比较之后，可输出 0～2 个顶点。对于情况(1)仅输出顶点 P；对于情况(2)输出 0 个顶点；对于情况(3)输出线段 SP 与裁剪线的交点 I；对于情况(4)输出线段 SP 与裁剪线的交点 I 和终点 P。



思考：是否可以推广到非cube？是否可以推广到三维？

两种算法的比较

- **There are many different primitives: segments, triangles, polygons**
- **Line segments:**
  - **Sutherland-cohen algorithm: consider 4 cases**
    - 视角：**TOP2Down**，从问题整体出发
    - **Motivation problem**：我必须进行复杂的求交计算吗?
  - **Liang-baskey algorithm: faster!**
    - 视角：**Bottom2Up**，从求解目标出发
    - **Motivation problem:** 求解目标应该如何表达呢?



- **Which part of the line segment is visible?**

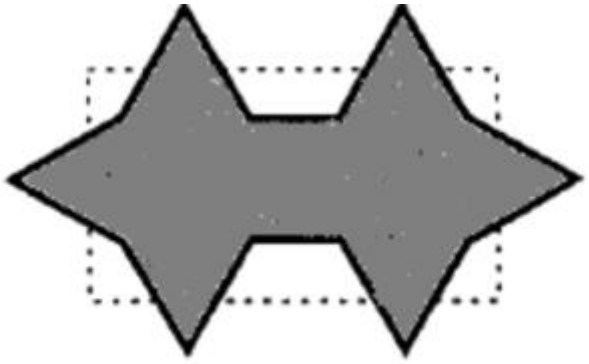
$$x_l \leq x_1 + u\Delta x \leq x_r$$
$$y_b \leq y_1 + u\Delta y \leq y_t$$
$$\Delta x = x_2 - x_1, \Delta y = y_2 - y_1$$

- **Rewrite as** $up_k \leq q_k$

$$\begin{cases} p_1 = -\Delta x \\ q_1 = x_1 - x_l \end{cases} \quad \begin{cases} p_2 = \Delta x \\ q_2 = x_r - x_1 \end{cases} \quad \begin{cases} p_3 = -\Delta y \\ q_3 = y_1 - y_b \end{cases} \quad \begin{cases} p_4 = \Delta y \\ q_4 = y_t - y_1 \end{cases}$$

Clipping algorithms

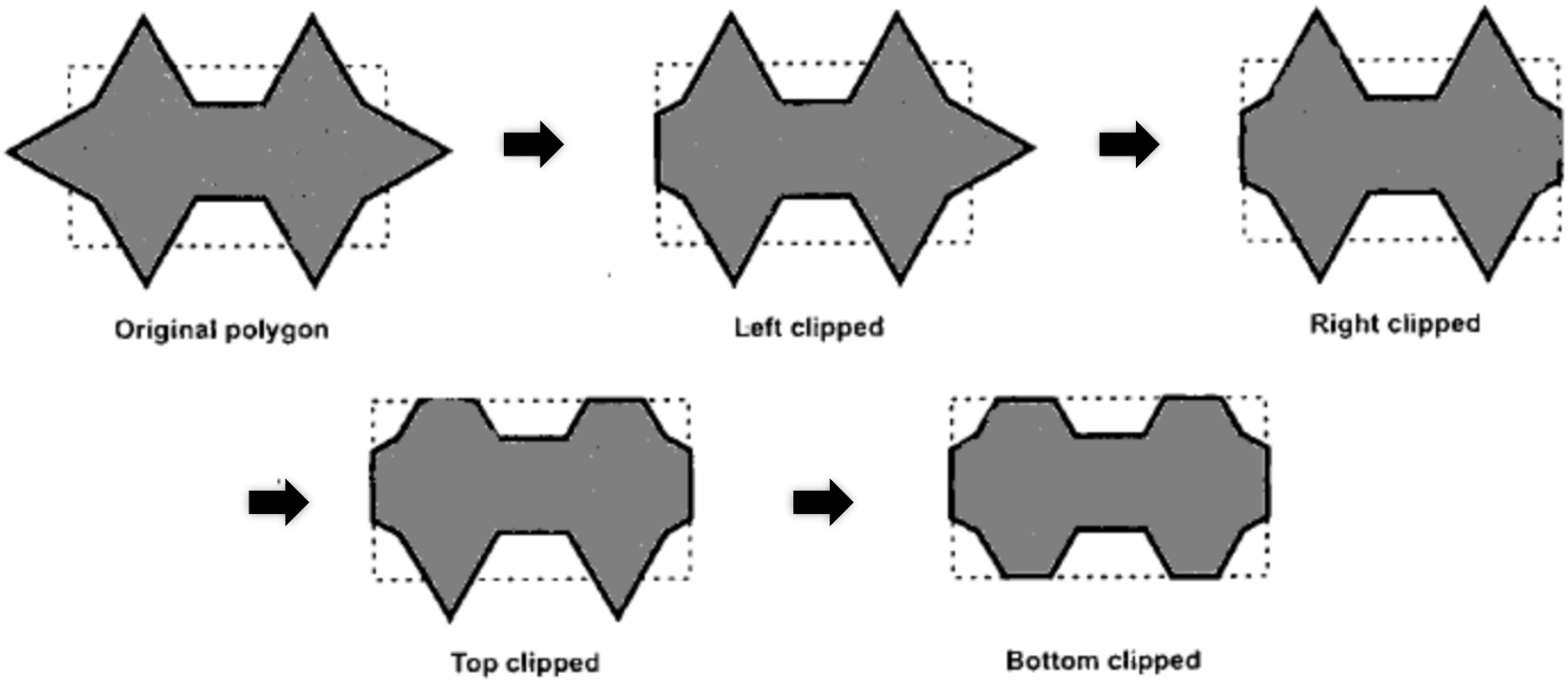
- There are many different primitives: segments, triangles, polygons
- Line segments:
  - Sutherland-cohen algorithm: consider 4 cases
  - Liang-baskey algorithm: faster!
- Polygons:
  - The above line segment clipping can be used for wireframe polygons, not solid polygons
    - What if? An example?
  - Sutherland-hodgman algorithm



Original polygon

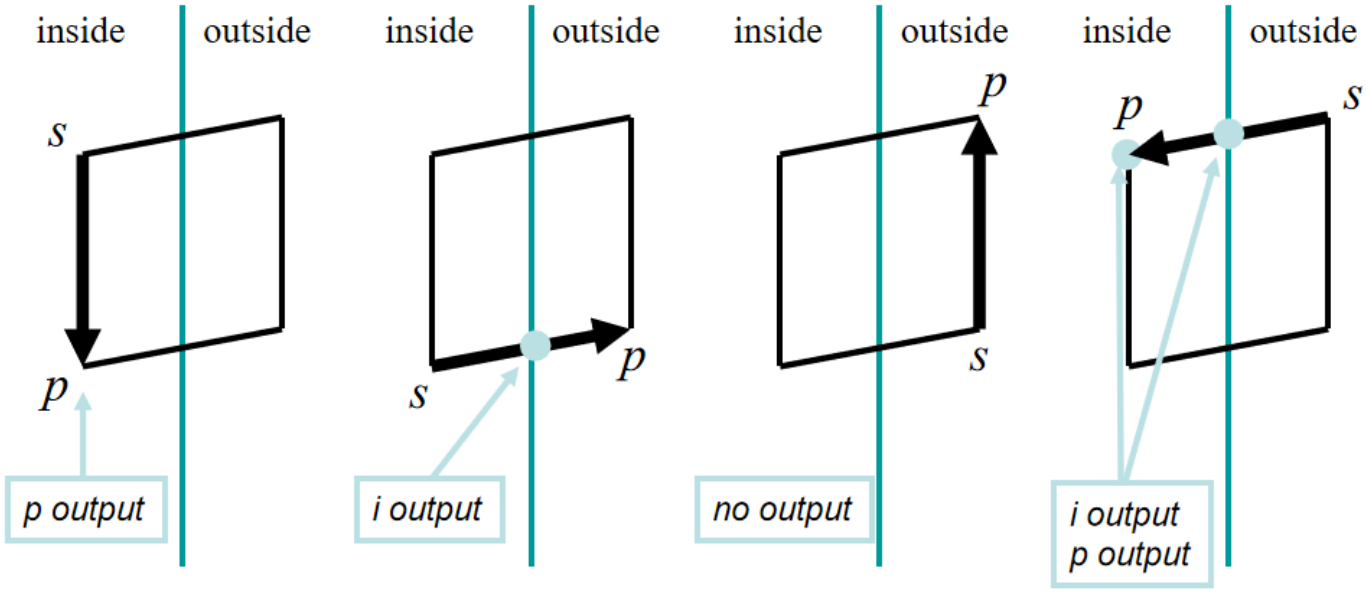
Sutherland-hodgman algorithm

- Idea: clipping with each line of the convex clip polygon in turn



Sutherland-hodgman algorithm

- Let's consider one step
  - The boundary divides the space into two parts: inside and outside
  - Consider a segment SP of the polygon, their relationship has 4 cases



  - It outputs the points for the next steps.

Sutherland-hodgman algorithm

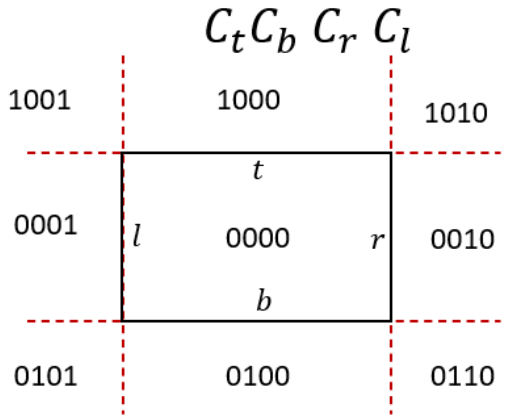
To clip an area against an individual boundary:

- **Consider each vertex in turn against the boundary (顺序)**
- **Vertices inside the boundary are saved for clipping against the next boundary** （内部）
- **Vertices outside the boundary are clipped** （外部）
- **If we proceed from a point inside the boundary to one outside, the intersection of the line with the boundary is saved** （从内向外）
- **If we cross from the outside to the inside, intersection point and the vertex are saved** （从外向内）

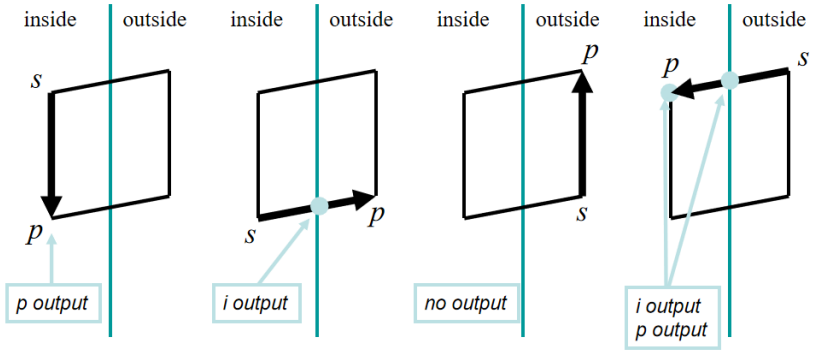
思考：与边裁剪算法有什么关系？算法思想是什么？

Last Lectures

- Line Clipping (裁剪)
  - Sutherland-cohen algorithm
  - Liang-baskey algorithm
- Polygon Clipping (裁剪)
  - Sutherland-hodgman algorithm

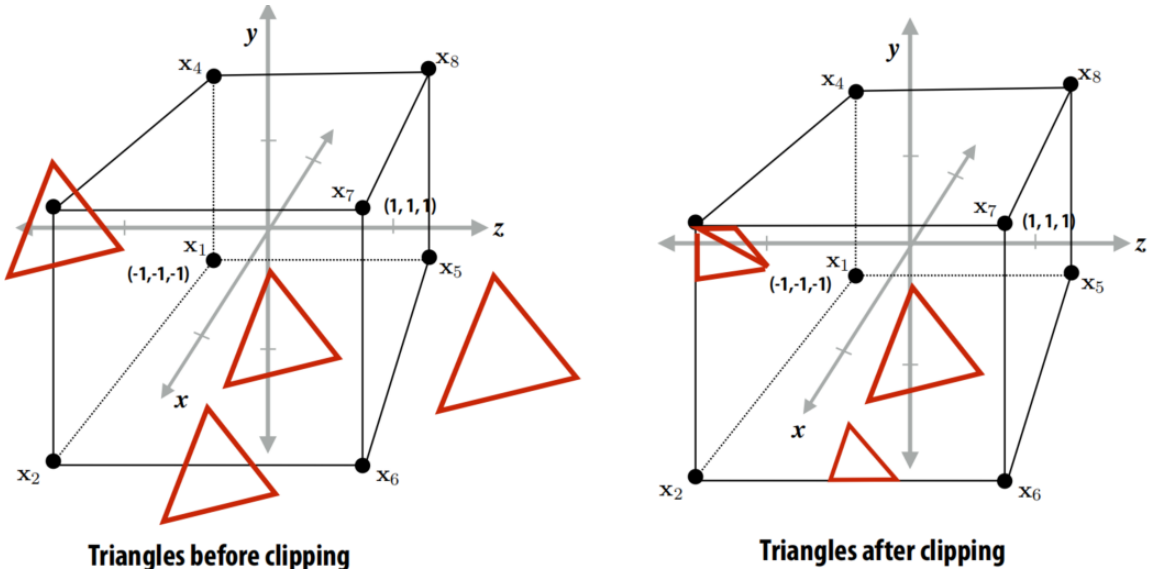


1. 如果Δx=0，线段平行于X
   1. $q_k < 0$, **outside**
   2. $q_k \geq 0$, *inside*
2. 如果$\Delta x \neq 0$, 线段不平行
   - **Compute all $r_k = q_k / p_k$**
   - **u_1 is max(0, min(r_1, r_2))**
   - **u_2 is min(1, max(r_1, r_2))**
   - **If $u_1 > u_2$, outside the window**
   - **else, the part is from u_1 -> u_2**



Clipping in normalized device coordinates (NDC)

- **Accept** primitives inside the view frustum
- **Reject** primitives outside the view frustum
- **Clip** primitives that extend beyond the view frustum

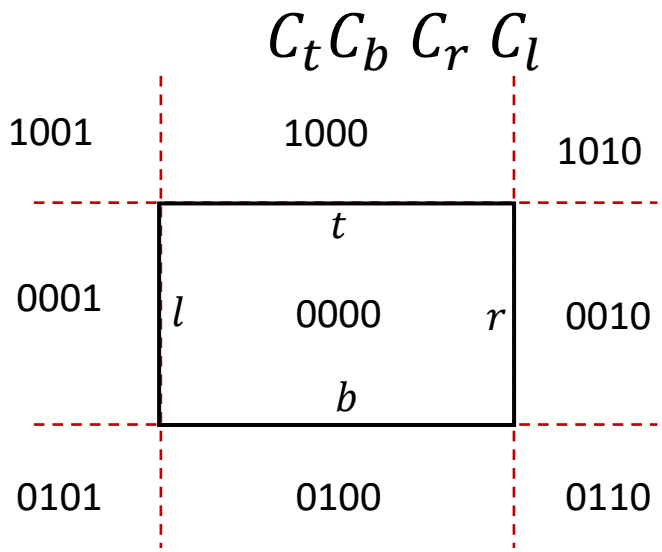


Triangles before clipping

Triangles after clipping

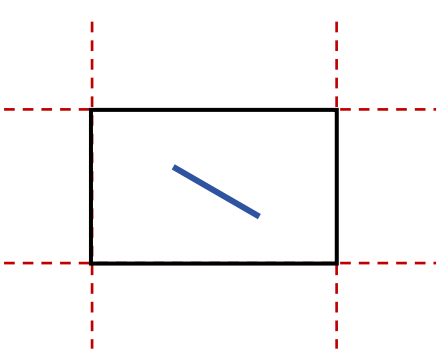
*Figures are correct: NDC for OpenGL (a popular graphics API) is left-handed coordinate space

Sutherland-Cohen Algorithm

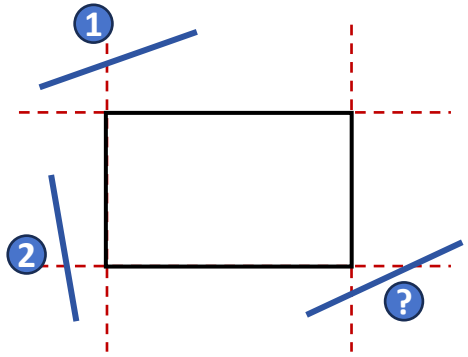


Step 1: Intersecion test

- **Divide the window into 9 regions with binary codes**
- **Query the code for each endpoint**
  (1) Two codes are 0000 and 0000: inside
  (2) "AND" operation and not obtain 0000: outside
  (3) Otherwise

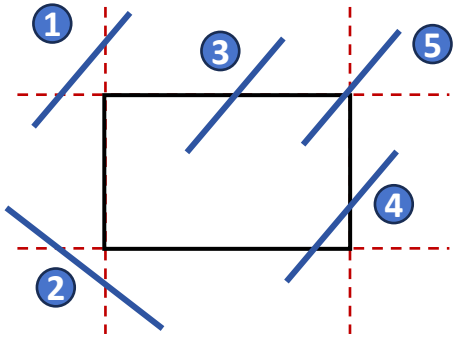


(1)



(2)

Whether both two endpoints are on top, bottom, left, right of the window



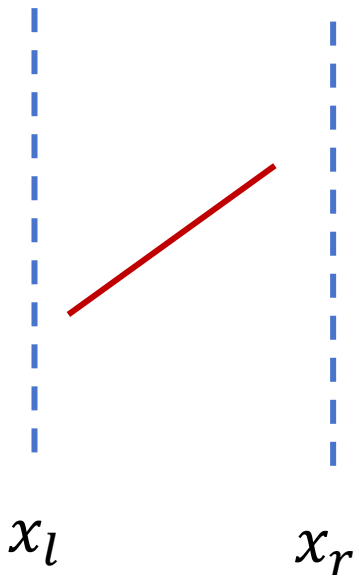
other

Liang-baskey algorithm

- **Which part of the line segment is visible?**

$$\overset{①}{x_l} \leq x_1 + u\Delta x \overset{②}{\leq} x_r$$

$$\Delta x = x_2 - x_1,\ \Delta y = y_2 - y_1$$



- **Rewrite as** $up_k \leq q_k$

① $-u\Delta x \leq x_1 - x_l$ ① $\begin{cases} p_1 = -\Delta x \\ q_1 = x_1 - x_l \end{cases}$

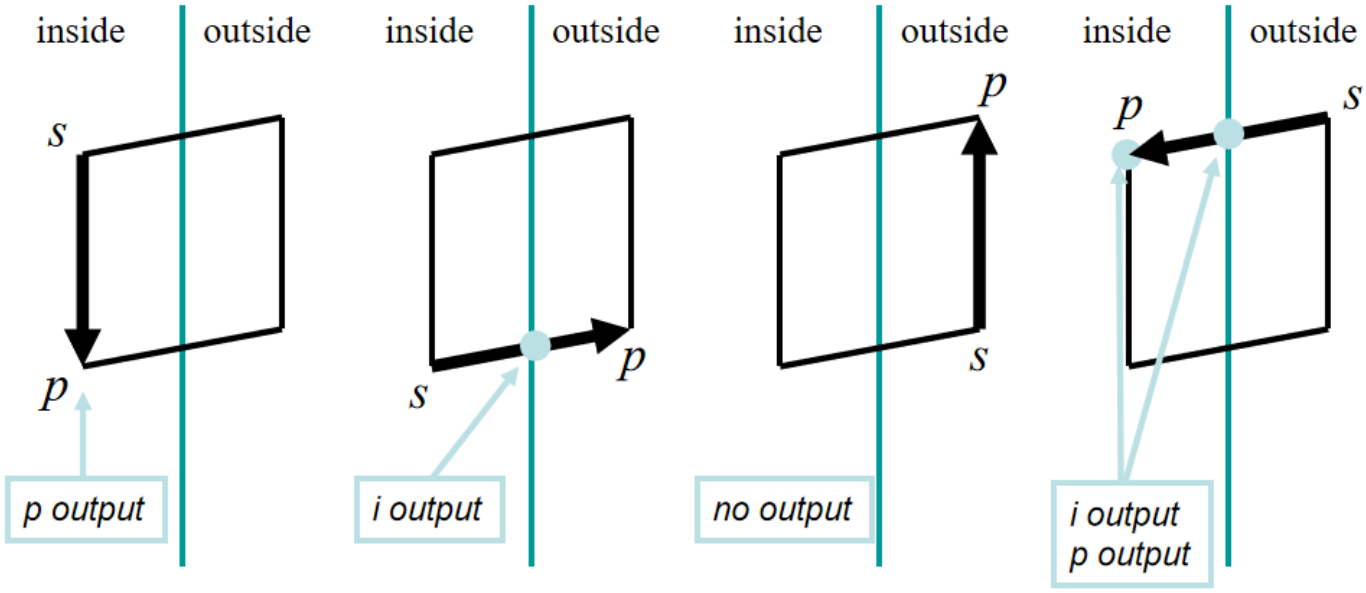
② $u\Delta x \leq x_r - x_1$ ② $\begin{cases} p_2 = \Delta x \\ q_2 = x_r - x_1 \end{cases}$

1. 如果Δx=0，线段平行于X
   1. $q_k < 0$, **outside**
   2. $q_k \geq 0, inside$
2. 如果$\Delta x \neq 0$, 线段不平行
   - **Compute all $r_k = q_k / p_k$**
   - **u_1 is max(0, min(r_1, r_2))**
   - **u_2 is min(1, max(r_1, r_2))**
   - **If $u_1 > u_2$, outside the window**
   - **else, the part is from u_1 -> u_2**

u_1 and u_2 defines the part of segment that is inside the window

Sutherland-hodgman algorithm

- Let's consider one step
  - The boundary divides the space into two parts: inside and outside
  - Consider a segment SP of the polygon, their relationship has 4 cases

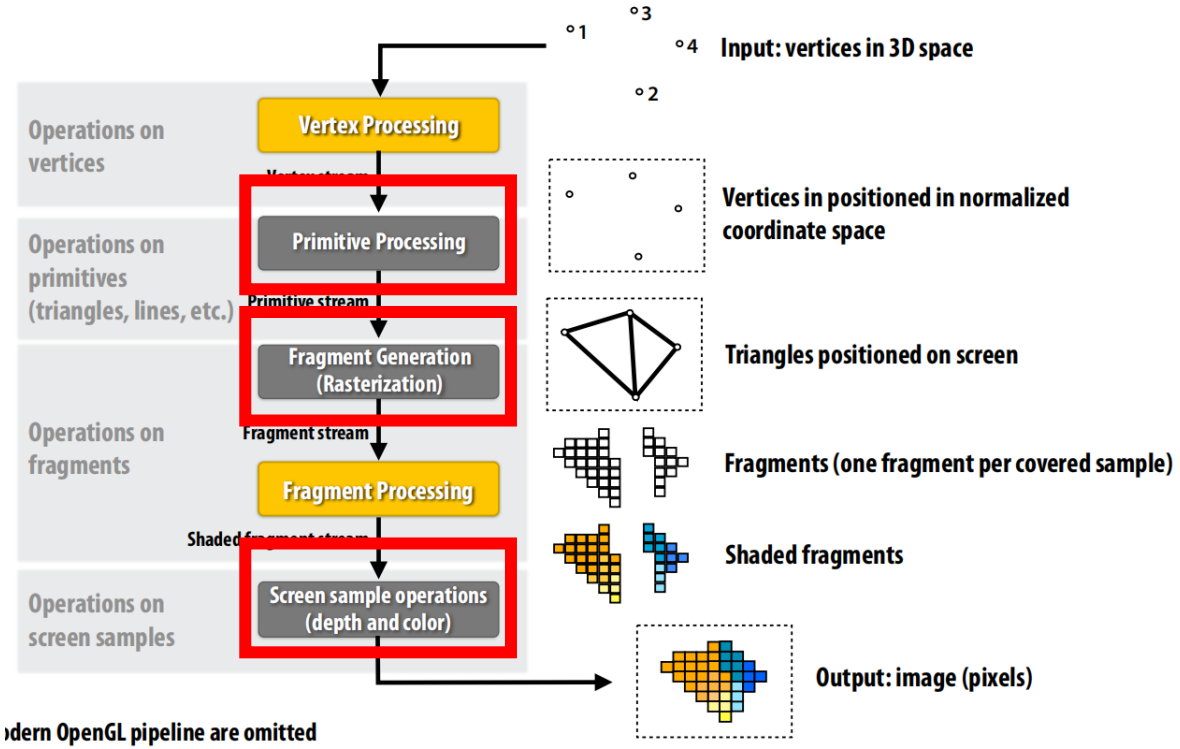


  - It outputs the points for the next steps.

拍照

Today

- Liang-baskey algorithm
- Scan Conversion of Line Segments
  - 直线段的扫描转化
- Scan Line Polygon Filling
  - 扫描线多边形填充
- Hidden Removal (消隐)

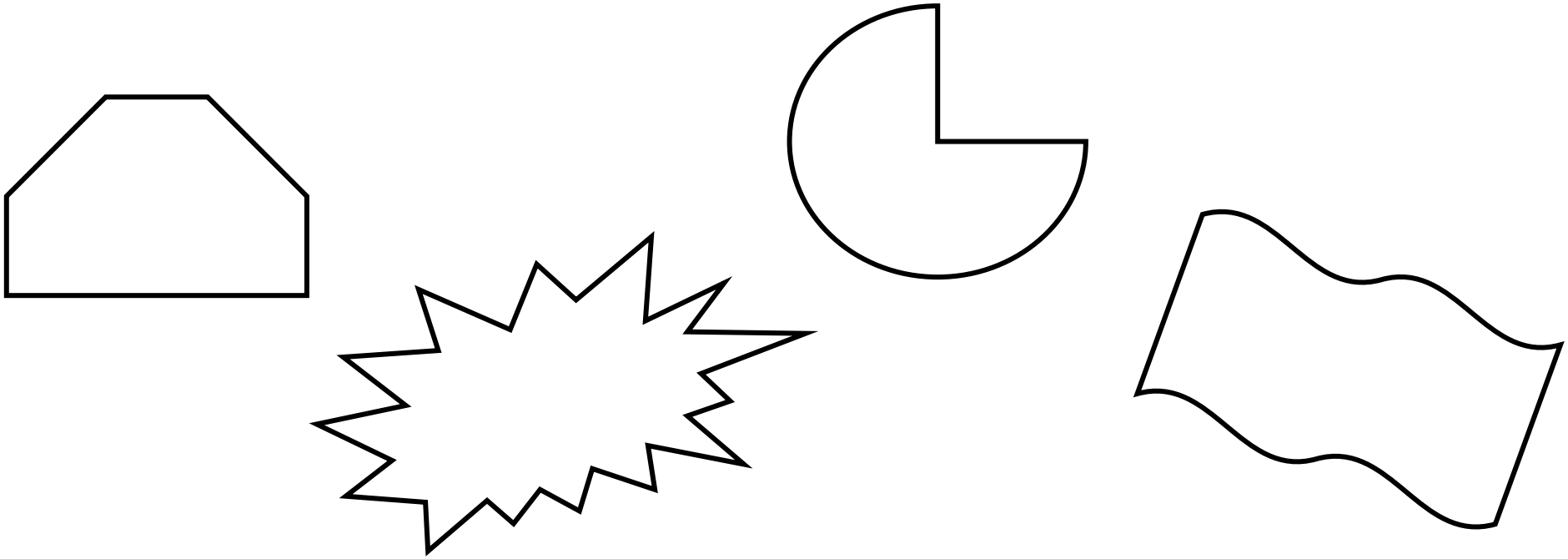


2D primitives to rasterized screen

Rasterization: a continuous object to a discrete representation on a raster grid

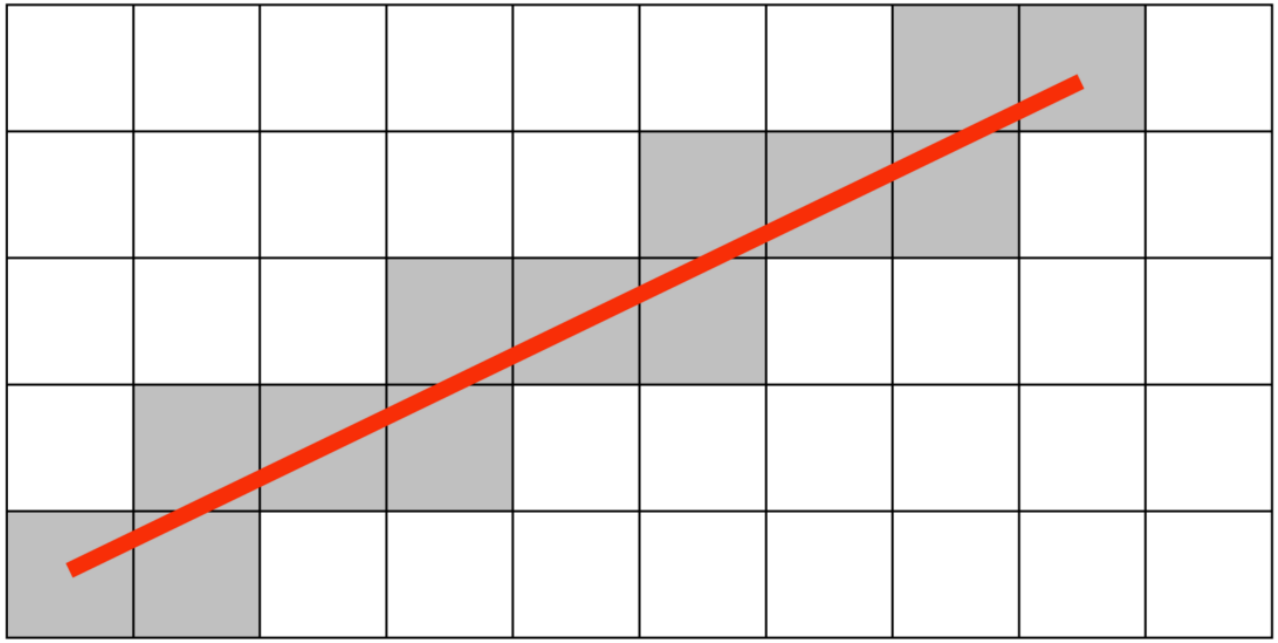
How to draw the 2D primitives on the screen?

- **2D Primitives: a set of projected vertices and edges**
- **Screen: a grid of pixels**



How to draw a line segment?

Light up all pixels intersected by the line

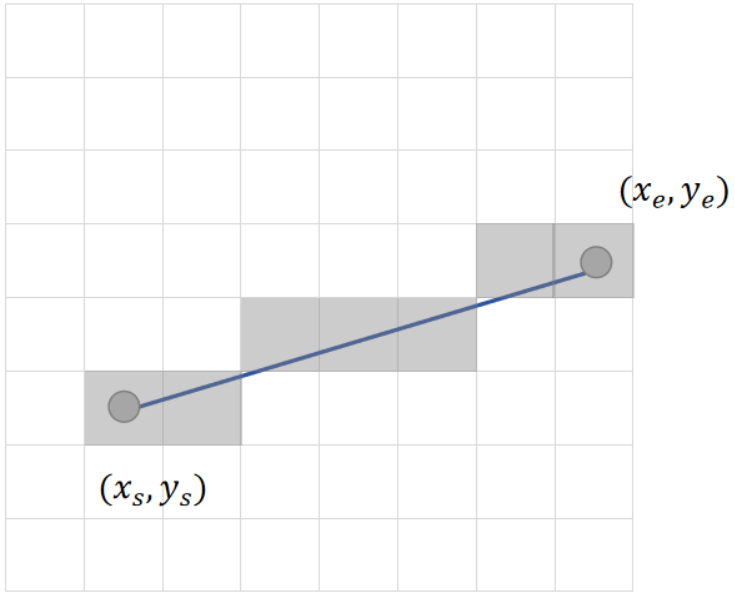


How to draw a line segment?

DDA (Digital Differential Analyzer)

Light up all pixels intersected by the line

- **Digital Differential Analyzer (DDA, 数值微分法): find the pixels along x or y axis**



Move along x axis

$$\Delta t = 1/\Delta x$$

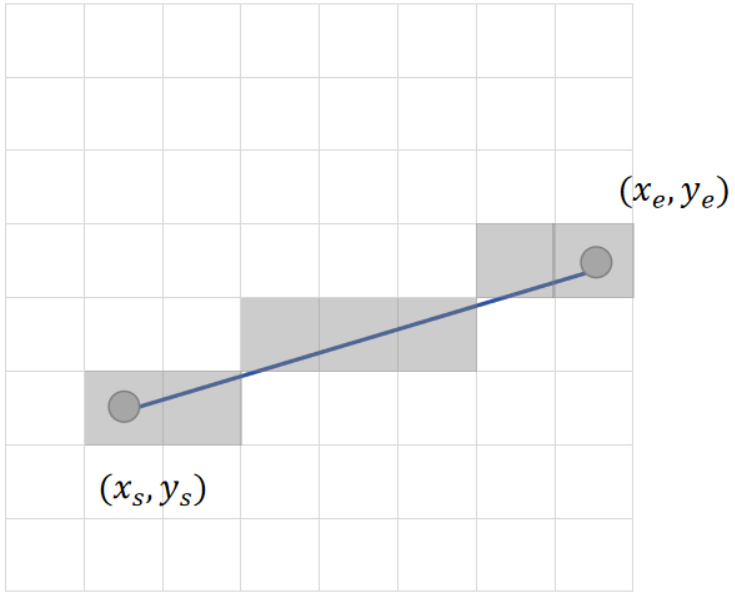
$$x = x_s + \Delta x \cdot \Delta t$$
$$y = y_s + \Delta y \cdot \Delta t$$

How to draw a line segment?

Light up all pixels intersected by the line

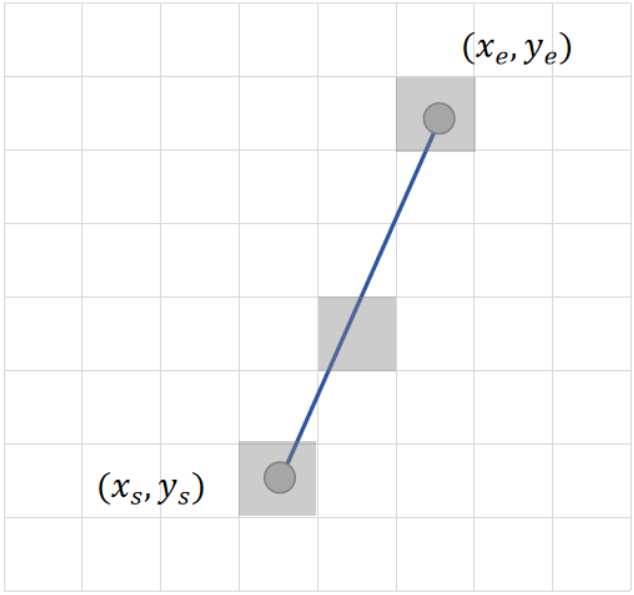
- **DDA algorithm: find the pixels along x or y axis**

$$x = x_s + \Delta x \cdot \Delta t$$
$$y = y_s + \Delta y \cdot \Delta t$$



Move along x axis

$$\Delta t = 1/\Delta x$$



Move along y axis

$$\Delta t = 1/\Delta y$$

How to draw a line segment?

Light up all pixels intersected by the line

- **DDA algorithm: find the pixels along x or y axis**
  - **select x axis if $\Delta x \geq \Delta y \geq 0$; select y axis otherwise**
  - **high computation cost:**
    - **Requires division operation, e.g. $\Delta t = 1/\Delta x$**
    - **Two sum and round every time**

$$x = x_s + \Delta x \cdot \Delta t$$
$$y = y_s + \Delta y \cdot \Delta t$$

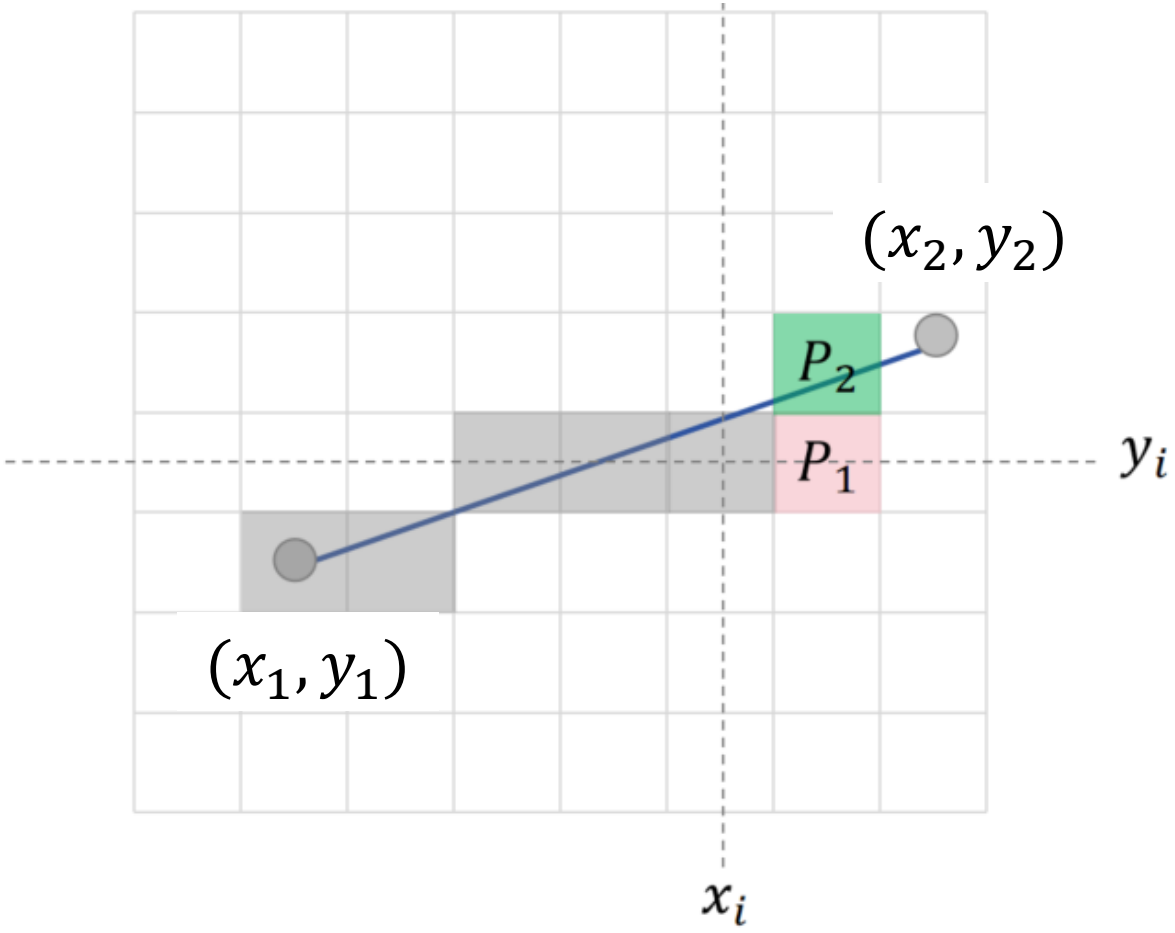
```
x+=xinc;
y+=yinc;

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

How to draw a line segment?

Light up all pixels intersected by the line

- **DDA algorithm: find the pixels along x or y axis**
- **Incremental Bresenham algorithm: integer arithmetric （中点画线法）**



基本思想：
问题简化（二选一）
距离判断
中点判断

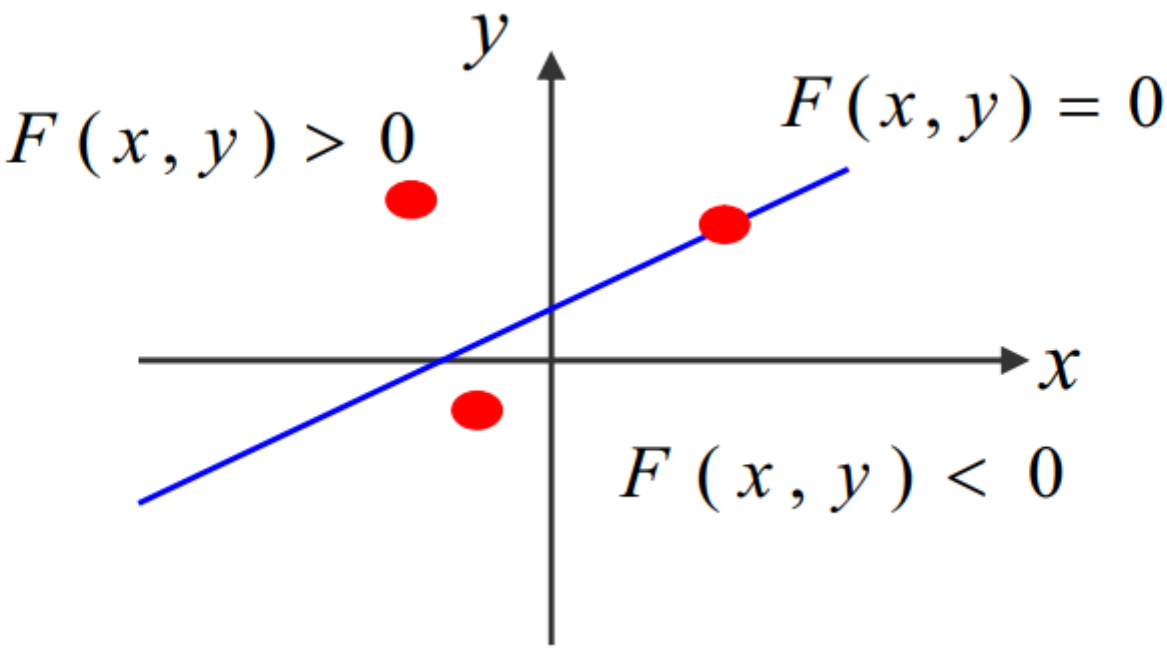
中点画线法

直线的一般式方程：

$$F(x, y) = 0$$

$$Ax + By + C = 0$$

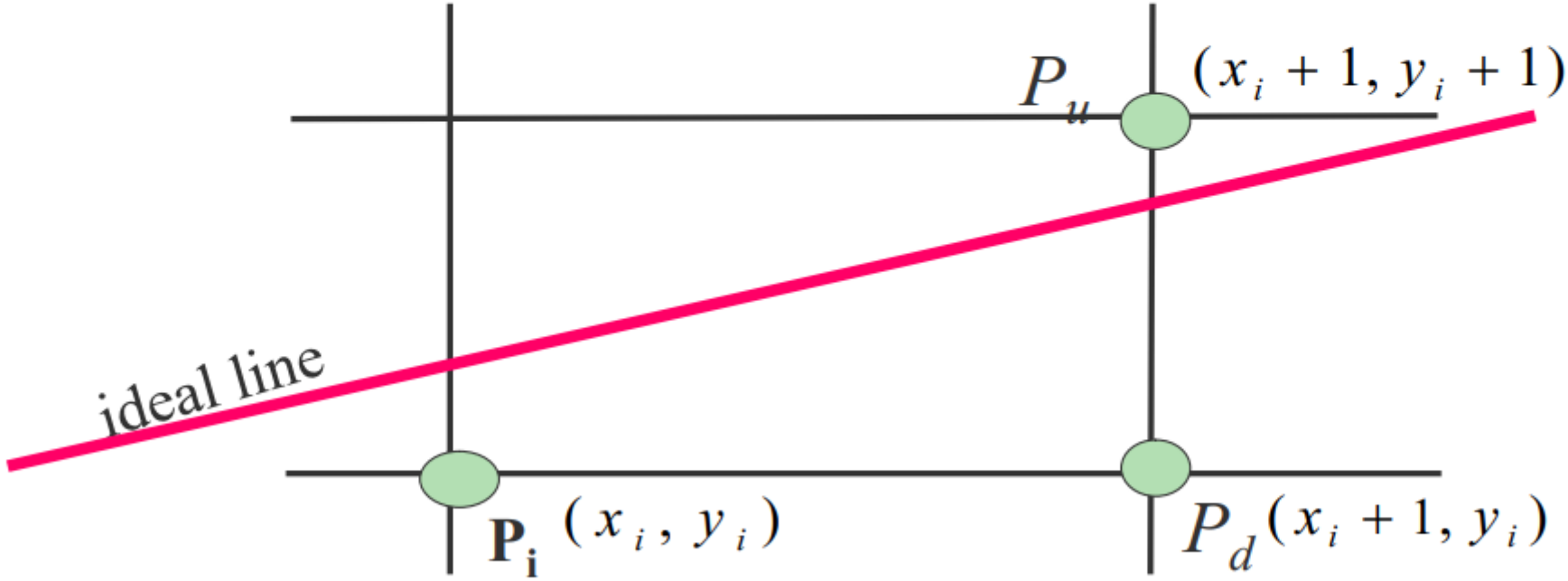
其中：$A = -(\Delta y);\quad B = (\Delta x);\quad C = -B(\Delta x)$

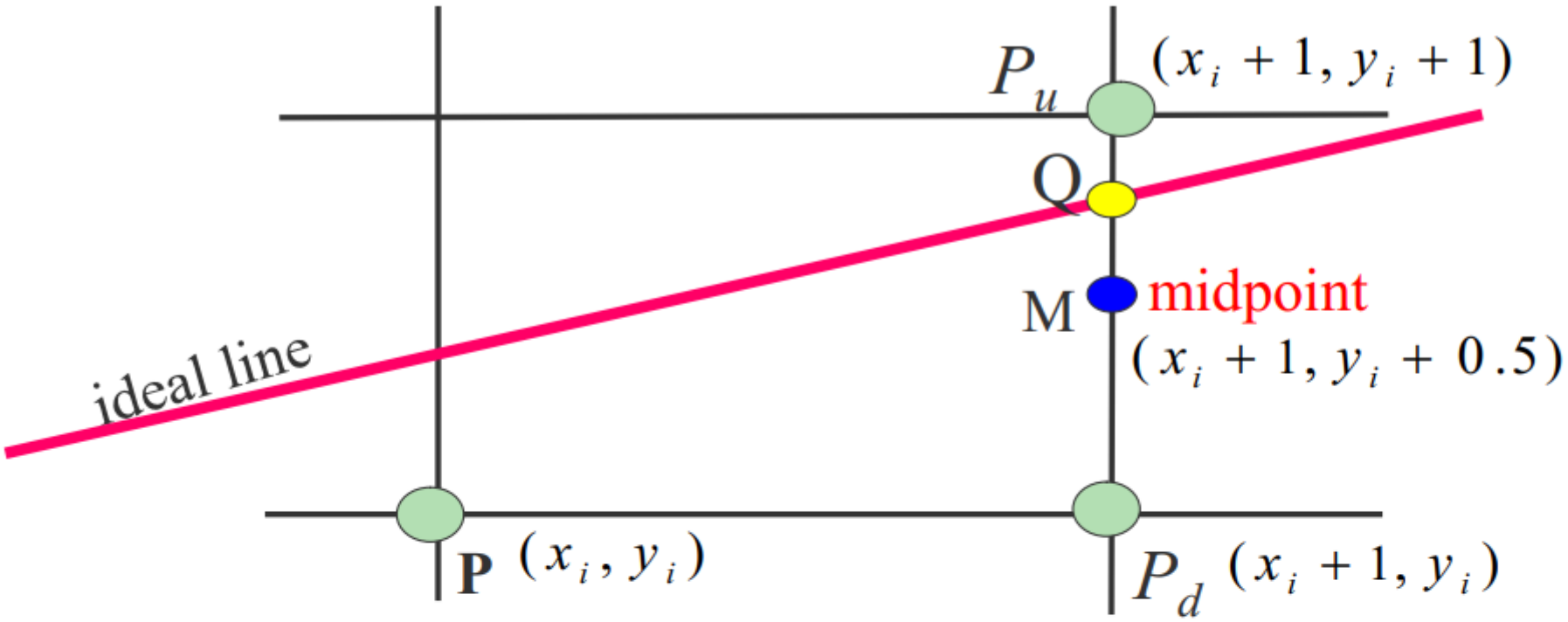


- 对于直线**上**的点：$F(x, y) = 0$
- 对于直线**上方**的点：$F(x, y) > 0$
- 对于直线**下方**的点：$F(x, y) < 0$

每次在最大位移方向上走一步，而另一个方向是走步还是不走步要取决于中点误差项的判断。

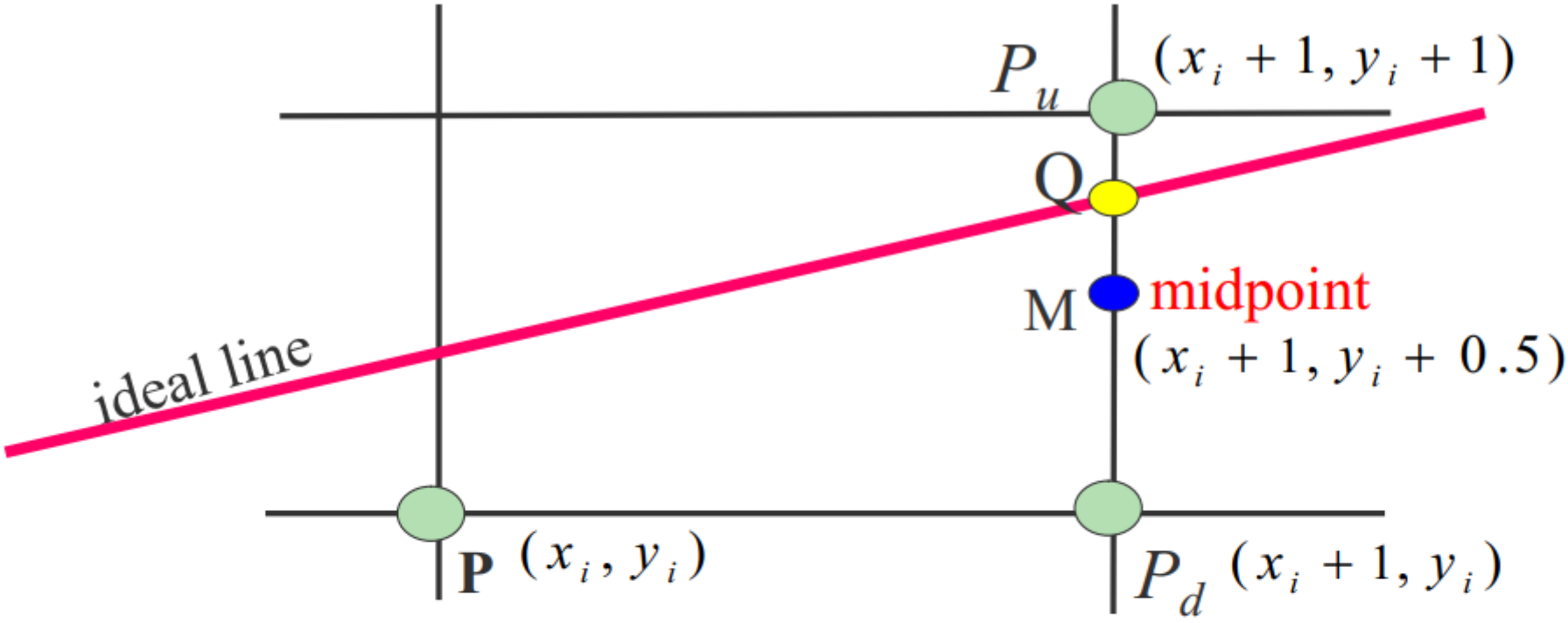
假定：0≤|k|≤1。因此，每次在x方向上加1，y方向上加1或不变需要判断。



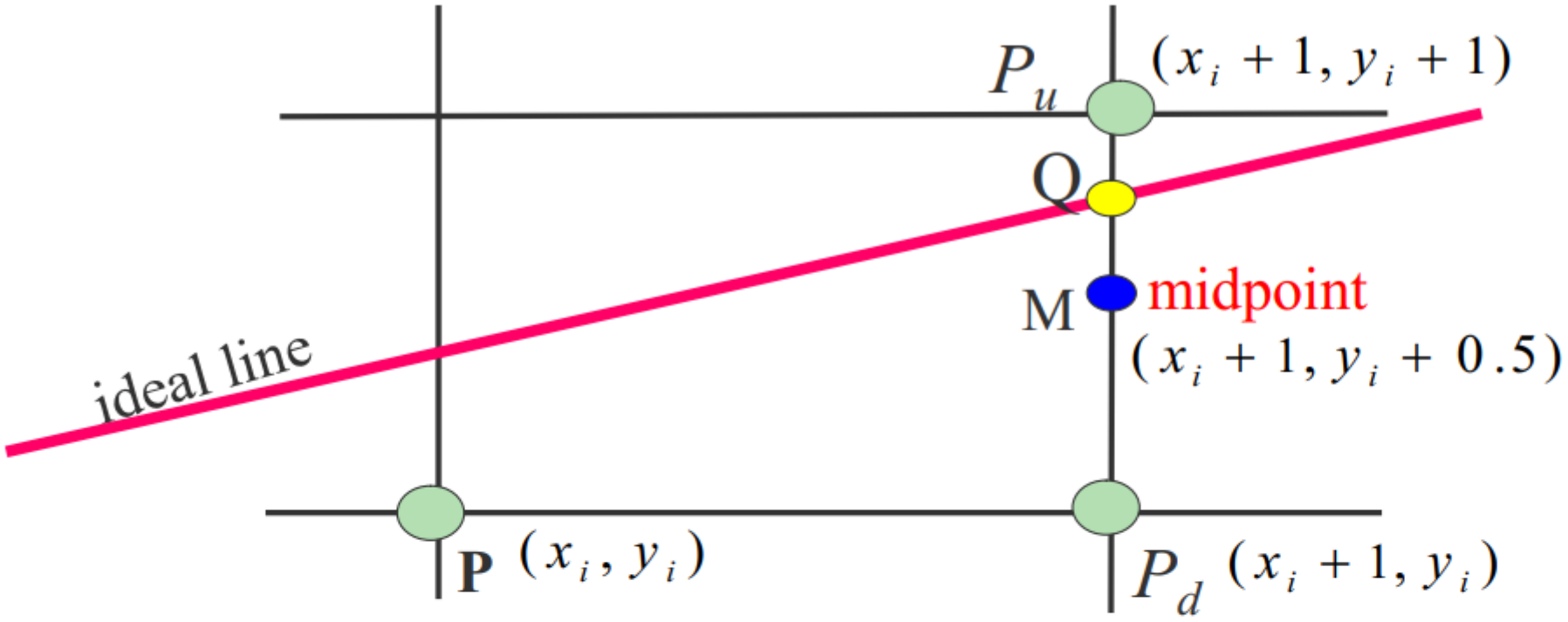


当M在Q的下方，则P_u离直线近，应为下一个象素点

当M在Q的上方，应取P_d 为下一点。



如何判断Q在M的上方还是下方？

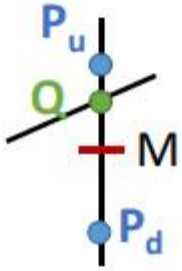


把M代入理想直线方程：

$$F(x_m, y_m) = Ax_m + By_m + C$$

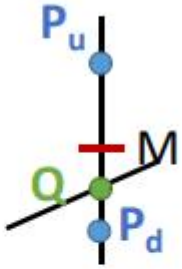
$$d_i = F(x_m, y_m) = F(x_i + 1, y_i + 0.5)$$
$$= A(x_i + 1) + B(y_i + 0.5) + C$$

当d<0时：



M在Q下方，应取P_u

当d>0时：



M在Q上方，应取P_d

当d=0时：

M在直线上，选P_u或P_d均可。

$$d_i = F(x_m, y_m) = F(x_i + 1, y_i + 0.5)$$
$$= A(x_i + 1) + B(y_i + 0.5) + C$$

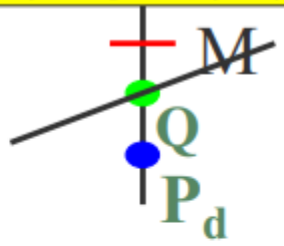
当d＜

$$y = \begin{cases} y + 1 & (d < 0) \\ y & (d \geq 0) \end{cases}$$

P_u

这就是中点画线法的基本原理

当d＞0 时：



M在Q上方，应取P_d

当d＝ 0时： M在直线上，选p_d或p_u均可。

下面来分析一下中点画线算法的计算量？

$$y = \begin{cases} y + 1 & (d < 0) \\ y & (d \geq 0) \end{cases}$$

$$d_i = A(x_i + 1) + B(y_i + 0.5) + C$$

为了求出d值，需要两个乘法，四个加法

能否也采用增量计算，提高运算效率呢？

$$d_{i+1} = d_i + ?$$

增量

How to draw a line segment?

Light up all pixels intersected by the line

- **DDA algorithm: find the pixels along x or y axis**
- **Incremental Bresenham algorithm: integer arithmetric （中点画线法）**

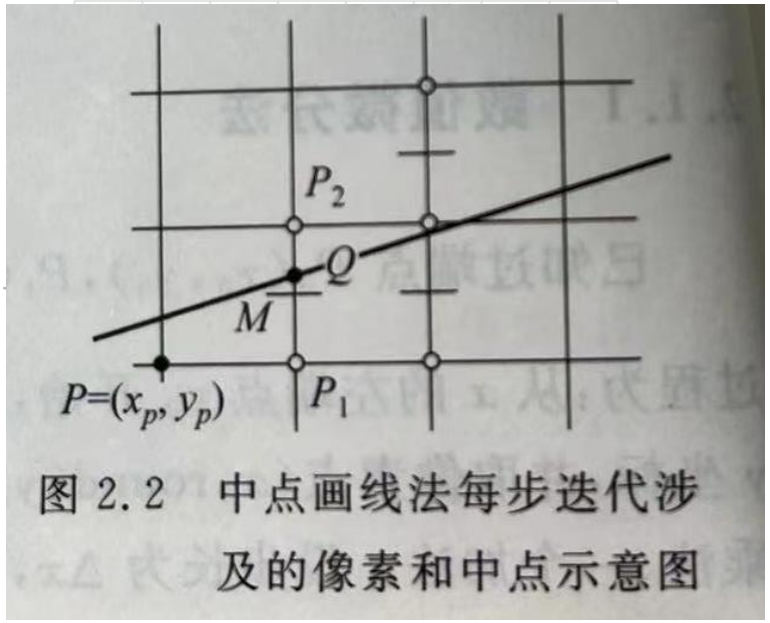


图 2.2 中点画线法每步迭代涉及的像素和中点示意图

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: $d_2 = F(x_p + 2, y_p + 0.5) = d + a$
  - $a = y_0 - y_1,\ b = x_1 - x_0 \quad c = x_0 y_1 - x_1 y_0$

$$F(M) = ax + by + c$$

$$d_{new} = \begin{cases} d_{old} + A + B & d < 0 \\ d_{old} + A & d \geq 0 \end{cases} \qquad d_0 = A + 0.5B$$

至此，中点算法至少可以和DDA算法一样好！

可以用2d代替d来摆脱浮点运算，写出仅包含整数运算的算法。

这样，中点生成直线的算法提高到整数加法，优于DDA算法。

DDA把算法效率提高到每步只做一个加法。

中点算法进一步把效率提高到每步只做一个整数加法

Bresenham提供了一个更一般的算法。该算法不仅有好的效率，而且有更广泛的适用范围

E.Jack Bresenham

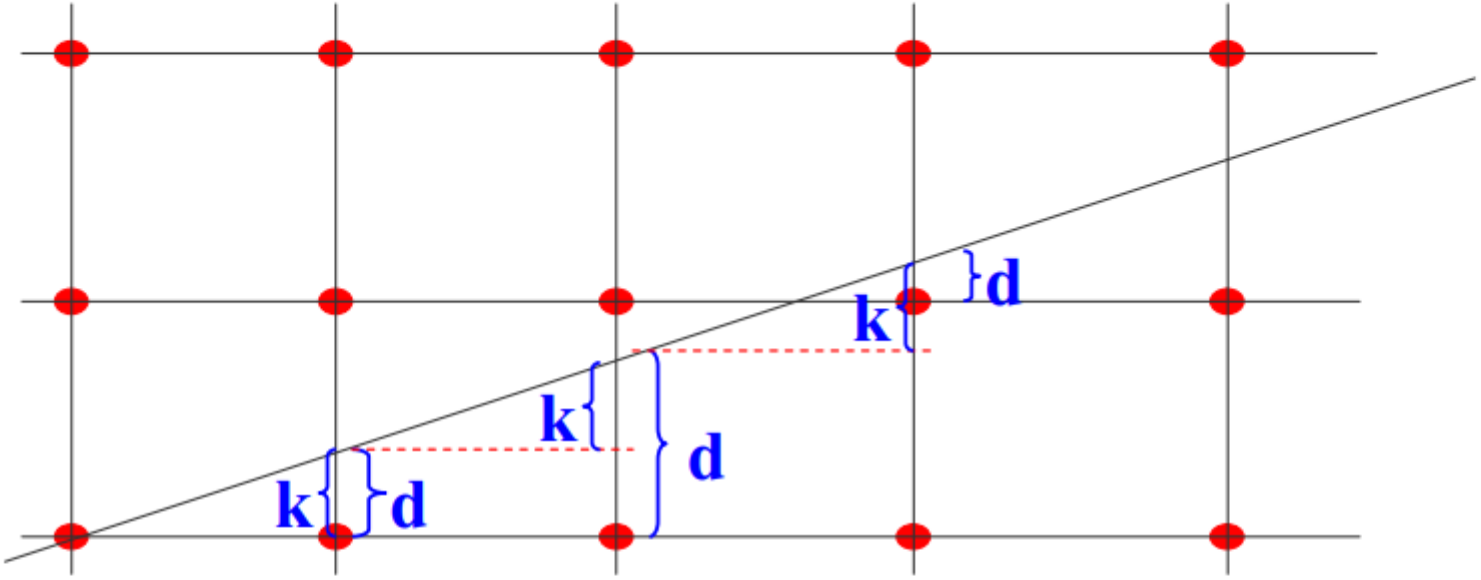
Biography [edit]

He retired from 27 years of service at IBM as a Senior Technical Staff Member in 1987. He taught for 16 years at Winthrop University and has nine patents[1]. He has three children: Janet, Linda, and David.

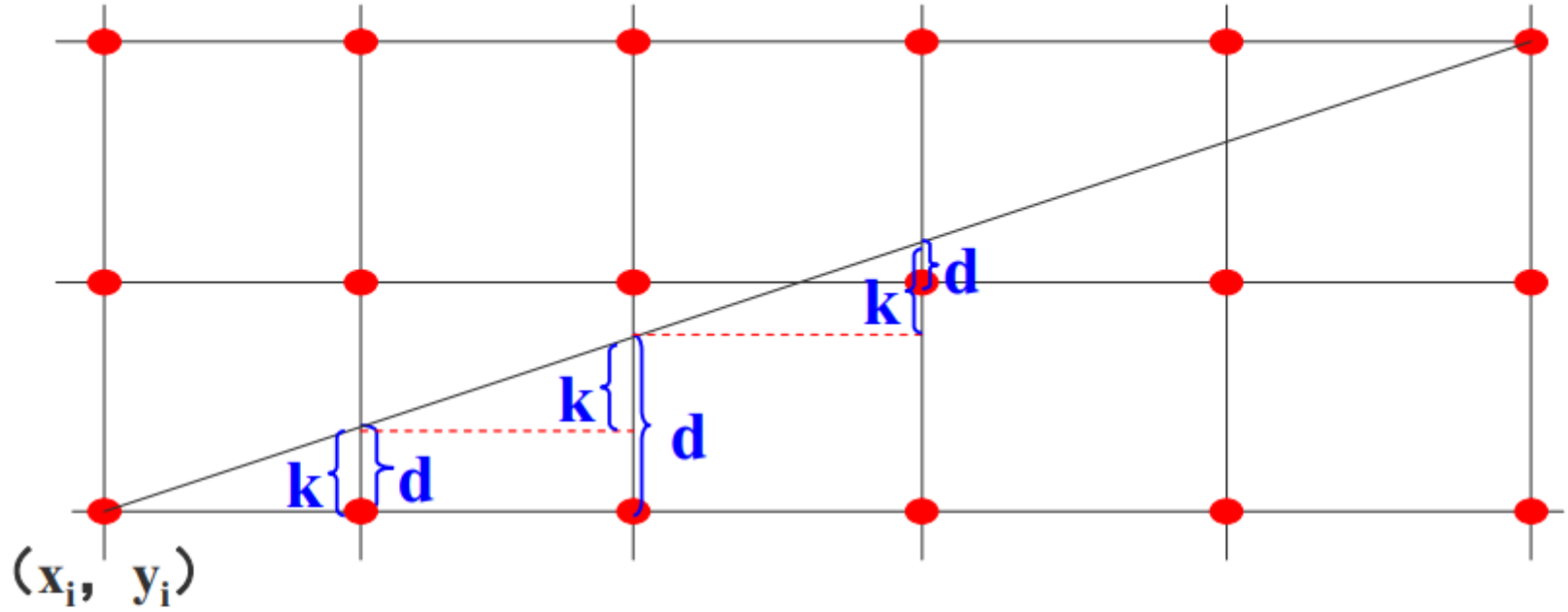
Bresenham's line algorithm, developed in 1962, is his most well-known innovation. It determines which points on a 2-dimensional raster should be plotte in order to form a straight line between two given points, and is commonly used to draw lines on a computer screen. It is one of the earliest algorithm discovered in the field of computer graphics. The Midpoint circle algorithm shares some similarities to his line algorithm and is known as *Bresenham's circle algorithm*.

- Ph.D., Stanford University, 1964
- MSIE, Stanford University, 1960
- BSEE, University of New Mexico, 1959

Bresenham算法的基本思想



该算法的思想是通过各行、各列像素中心构造一组虚拟网格线，按照直线起点到终点的顺序，计算直线与各垂直网格线的交点，然后根据误差项的符号确定该列象素中与此交点最近的象素。

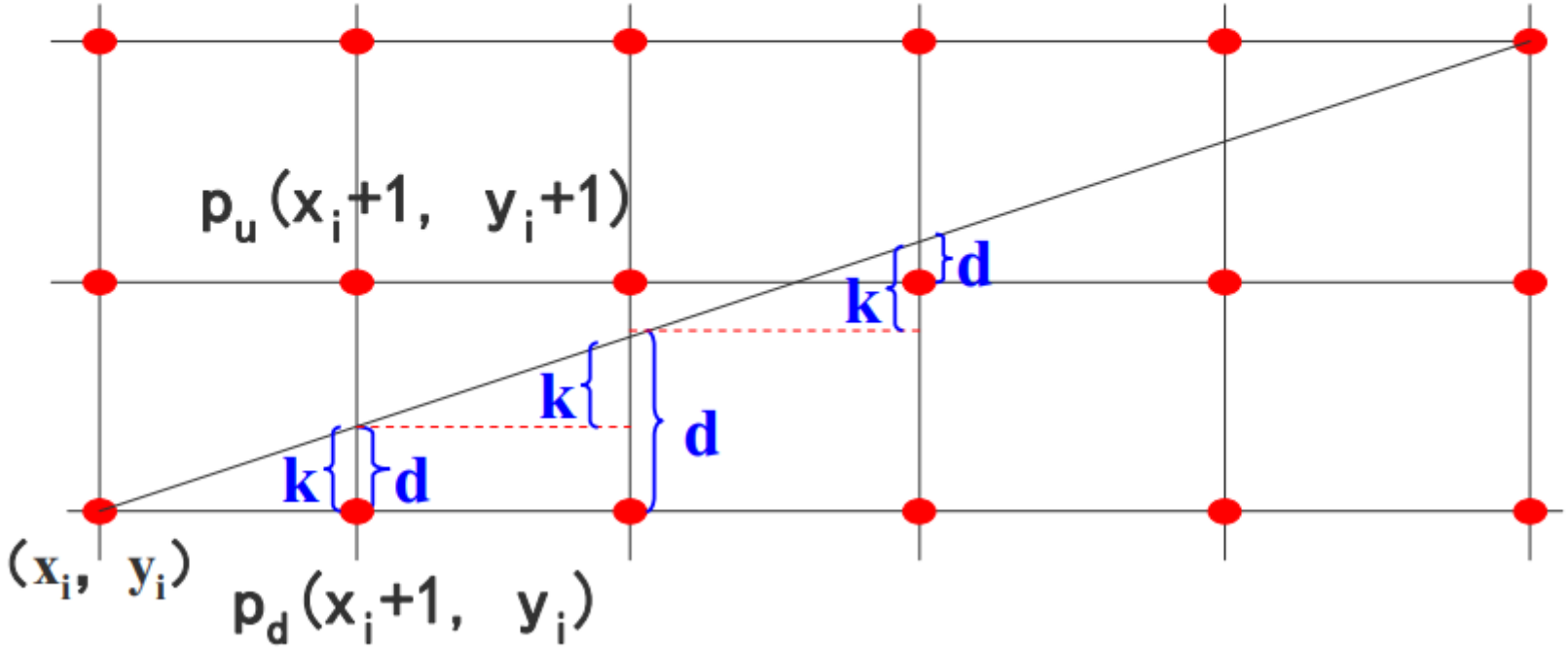


假设每次x+1，y的递增（减）量为0或1，它取决于实际直线与最近光栅网格点的距离，这个距离的最大误差为0.5。

误差项d的初值$d_0=0$

$$d=d+k$$

一旦d≥1，就把它减去1，保证d的相对性，且在0、1之间。



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

关键是把这个算法的效率也搞到整数加法，否则就是失败。如何提高到整数加法？

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

如何把这个算法的效率也提高到整数加法？

改进1：令 e = d−0.5

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

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

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

- $e_{初} = -0.5$
- 每走一步有e = e+k
- if (e>0) then e=e-1

0.5?

$e_{初} = -0.5$ $k = \frac{dy}{dx}$

改进2：由于算法中只用到误差项的符号，于是可以用e*2*△x来替换e。

- $e_{初}$ = −△x，
- 每走一步有：e = e+2△y。
- if （e>0） then e = e−2△x

算法步骤为：

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△x、△y、**e=−△x**、$x=x_0$、$y=y_0$。
3. 绘制点(x, y)。
4. e更新为**e+2△y**，判断e的符号。若e>0，则(x, y)更新为(x+1, y+1)，同时将e更新为**e−2△x**；否则(x, y)更新为(x+1, y)。
5. 当直线没有画完时，重复步骤3和4。否则结束。

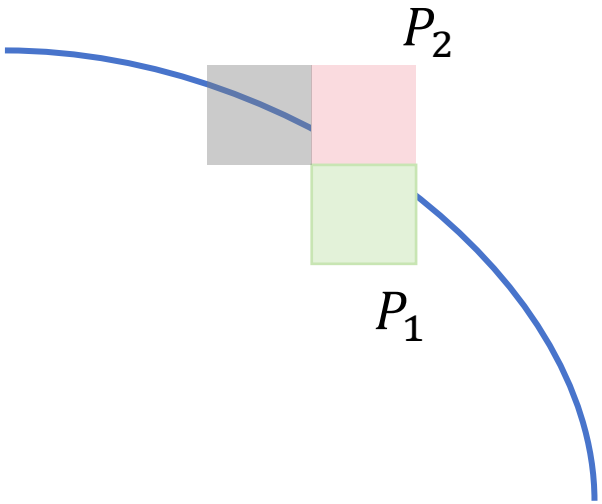
Bresenham算法很像DDA算法，都是加斜率

但DDA算法是每次求出一个新的y以后取整来画；而Bresenham算法是判符号来决定上下两个点。所以该算法集中了DDA和中点两个算法的优点，而且应用范围广泛

How to draw a curve segment?

Again we can use the Bresenham algorithm

- **Circle function:** $D(P) = (x^2 + y^2) - R^2 = 0$
- **We only need to select** P_1 **or** P_2



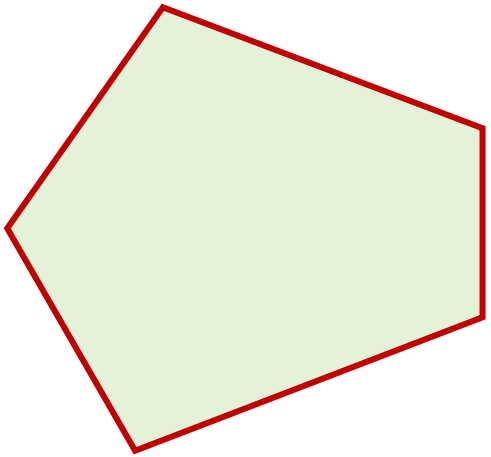
- Incremental computation:
  - $d_i < 0$, select P_2
    - Next: $d_{i+1} = d_i + 4x_i + 6$
  - $d \geq 0$, select P_1
    - Next: $d_{i+1} = d_i + 4(x_i - y_i) + 10$

A short break

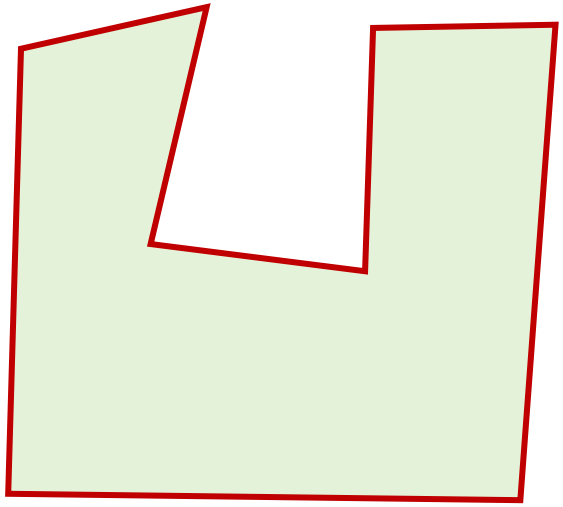
What about the other shapes?

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

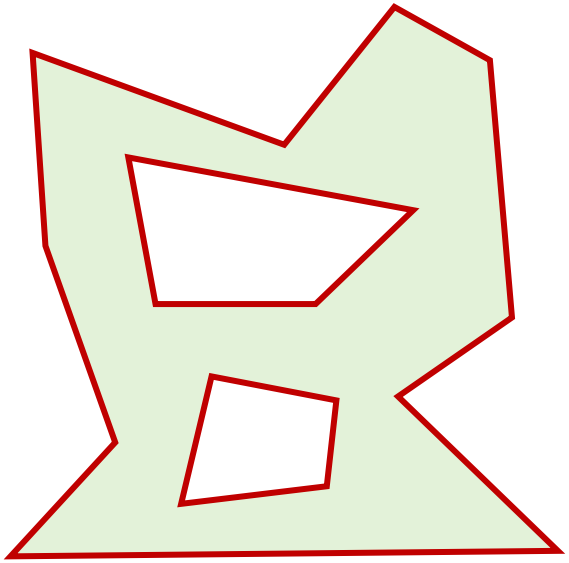
Now we can draw the boundary of any shape!



凸多边形



凹多边形

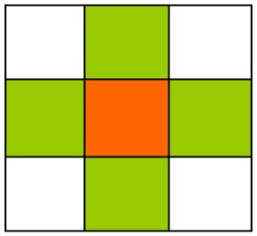


带洞（含内环）多边形

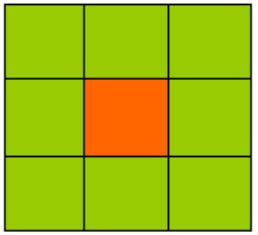
How to draw the filled shapes?

Simple filling algorithms

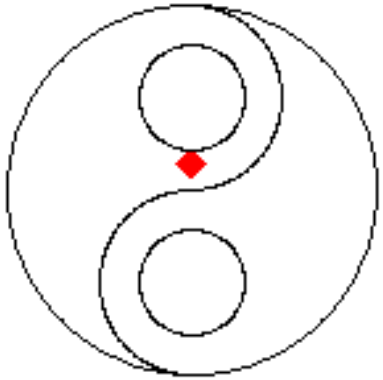
- Brute force rasterizer
  - Exhaustively test all the pixels
  - Is each pixel inside or outside the shape?
- Flood/Seed fill algorithm
  - Four-neighbors and eight-neighbors
  - Initialize a seed and fill the neighbors



Four-neighbors



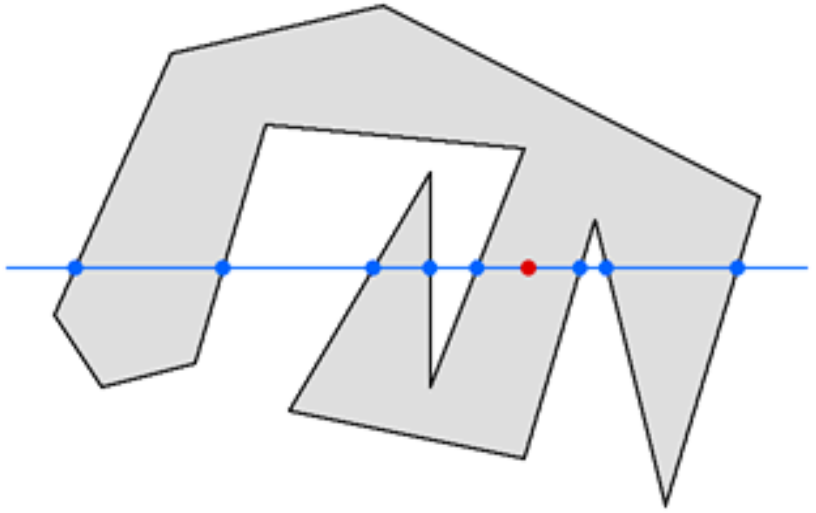
Eight-neighbors



Flood-fill algorithm

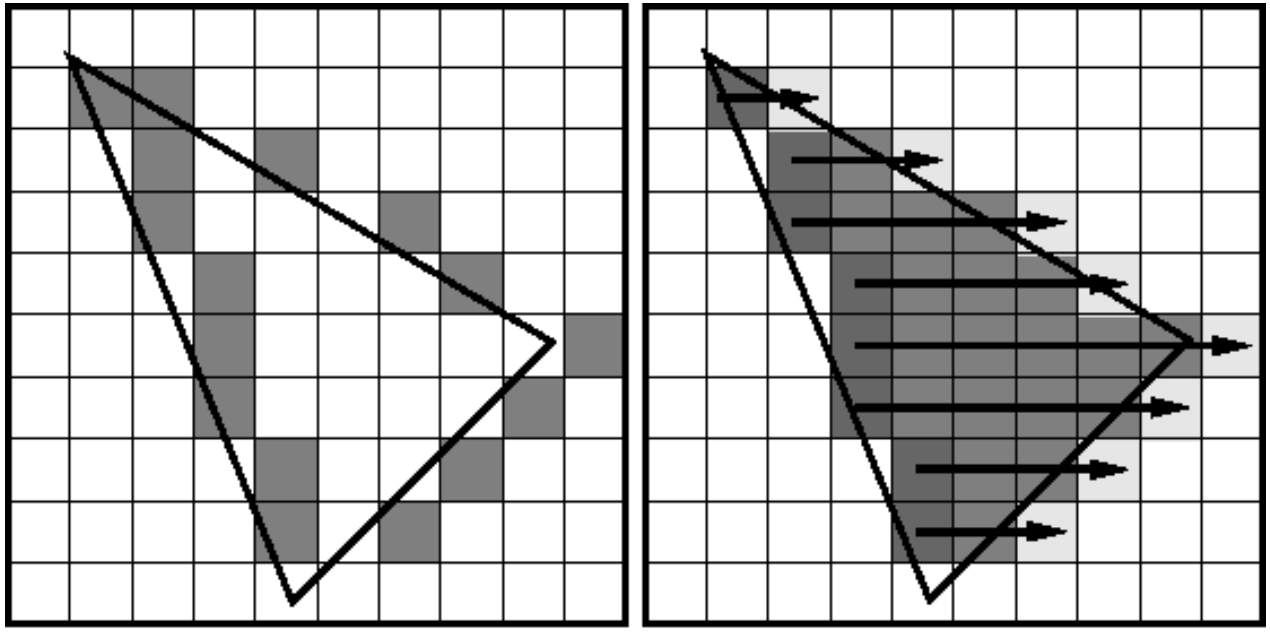
Brute force test

- Odd-even Rule
  - 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



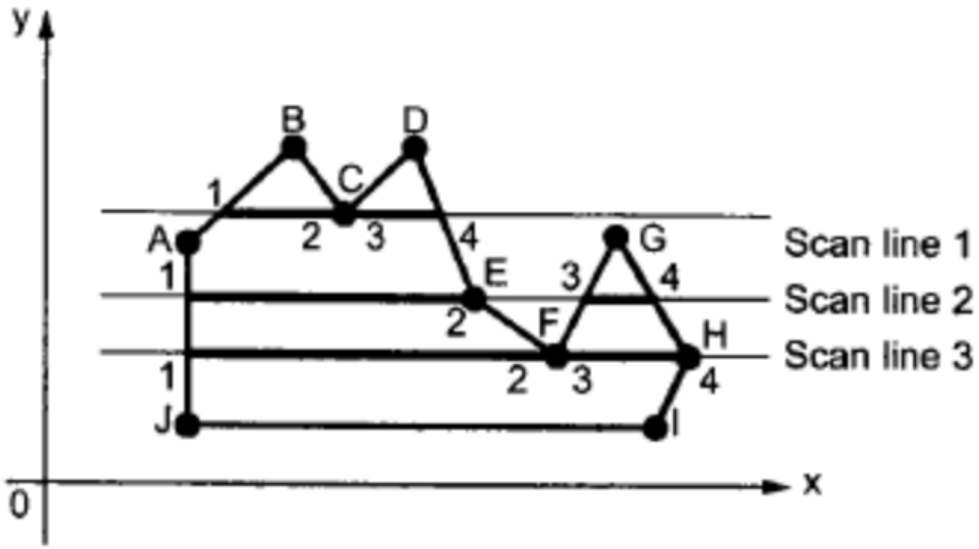
Scanline fill algorithm （扫描线算法）

- **Basic idea:**
  - **Compute the intersection of the polygon boundary and the scanlines**
  - **Sort the orders of intersections for each scanline**
  - **Fill up the pixels along the scanline**



Scanline fill algorithm

- Scanline: scan along horizontal or vertical directions



For each scanline:

(1) Intersection. Compute the intersection between the scanline and polygon boundaries

(2) Sorting. Sort the intersections based on their x values.

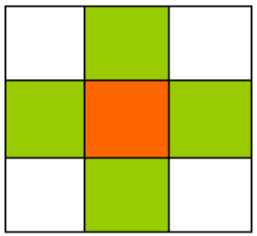
(3) Matching. Build points between <1st, 2nd>, <3rd, 4th>, ... intersections.

(4) Filling. Set the color for the pixels inside the paired interval.

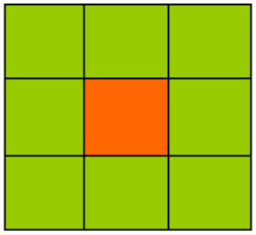
Easy to think, but low efficiency (intersection and sorting)

Flood fill algorithm （区域填充算法）

- **Suppose we already know one pixel inside the region**



Four-neighbors



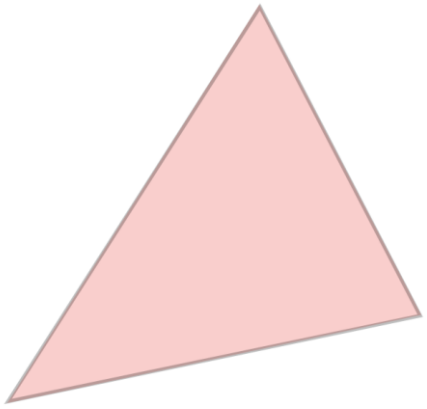
Eight-neighbors

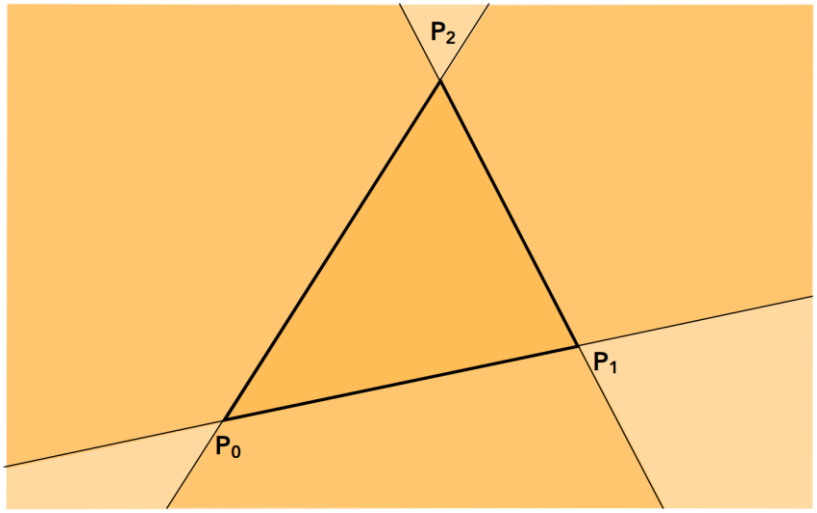
```
void boundaryfill(int x,int y,int fill_color,int
boundary_color)
{
    if(getpixel(x,y)!=boundary_color &&
getpixel(x,y)!=fill_color)
    {
       putpixel(x,y,fill_color);
       boundaryfill(x+1,y,fill_color,boundary_color);
       boundaryfill(x-1,y, fill_color,boundary_color);
       boundaryfill(x,y+1, fill_color,boundary_color);
       boundaryfill(x,y-1, fill_color,boundary_color);
    }
}
```

If we only have triangles

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





"Negative side" of line: $L(x,y) < 0$

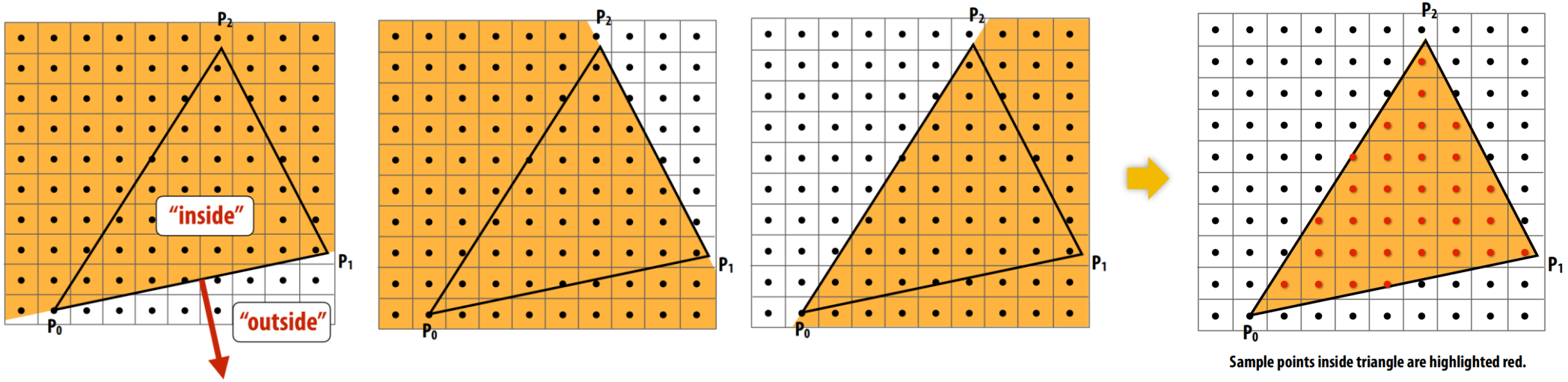
"Positive" side of line: $L(x,y) > 0$

If we only have triangles

Point-in-triangle test

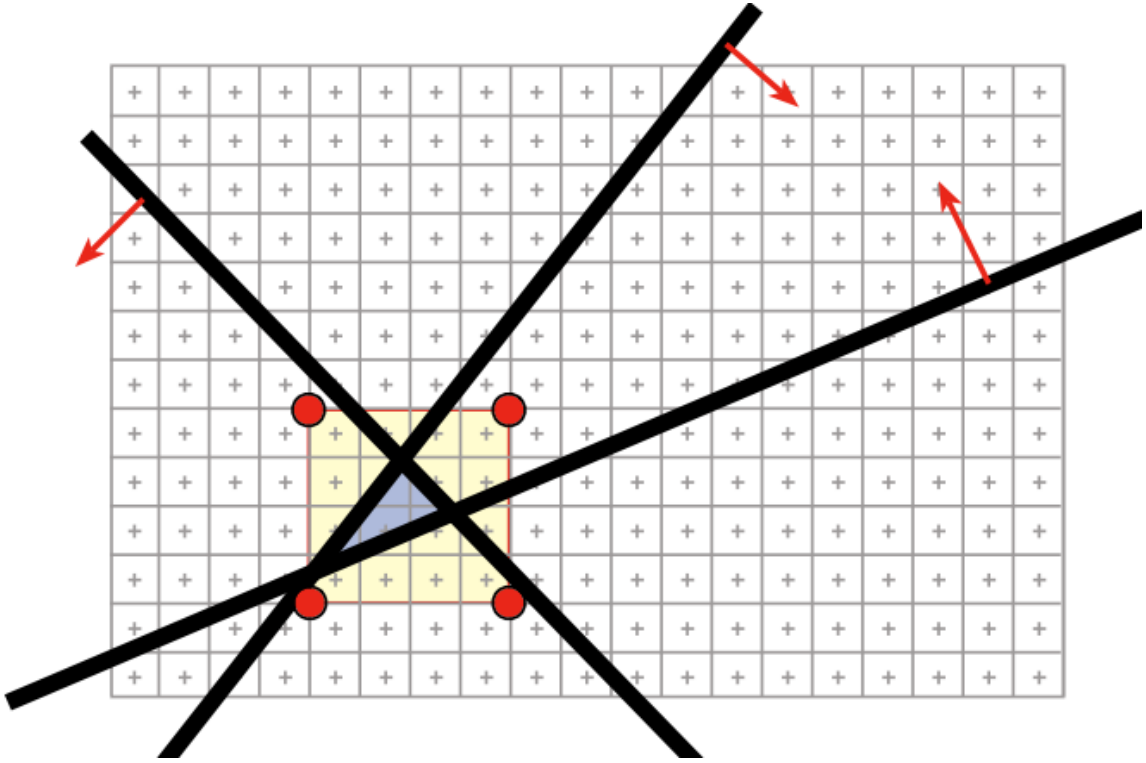
Step 1. Triangle setup (compute E1, E2 , E3 coefficients from projected vertices)

Step 2. Evaluate edge functions at pixel center



Speed up the rasterizer

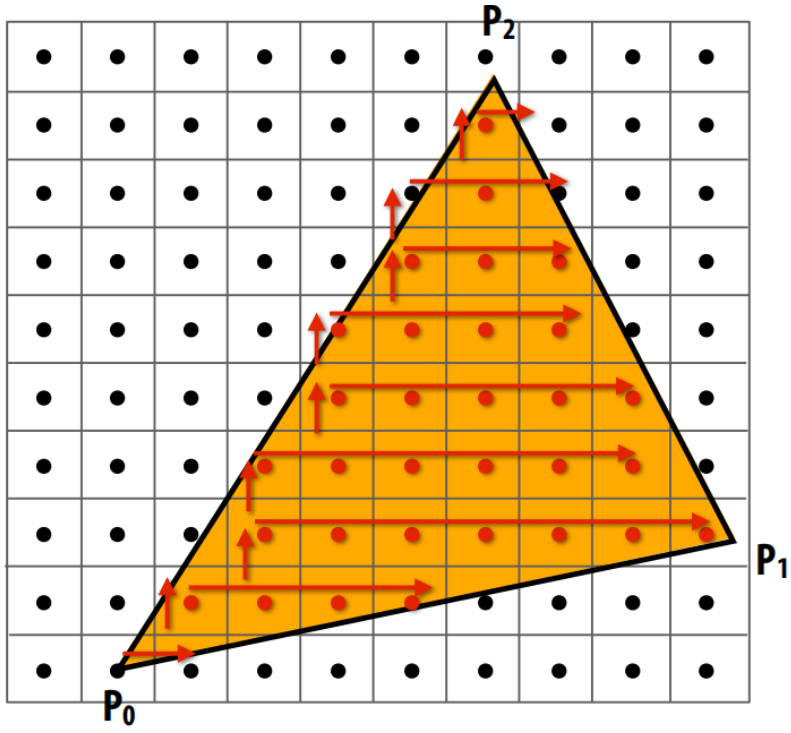
- Axis-aligned bounding box
  - Scan over only the pixels that overlap the screen bounding box of the triangle



Speed up the rasterizer

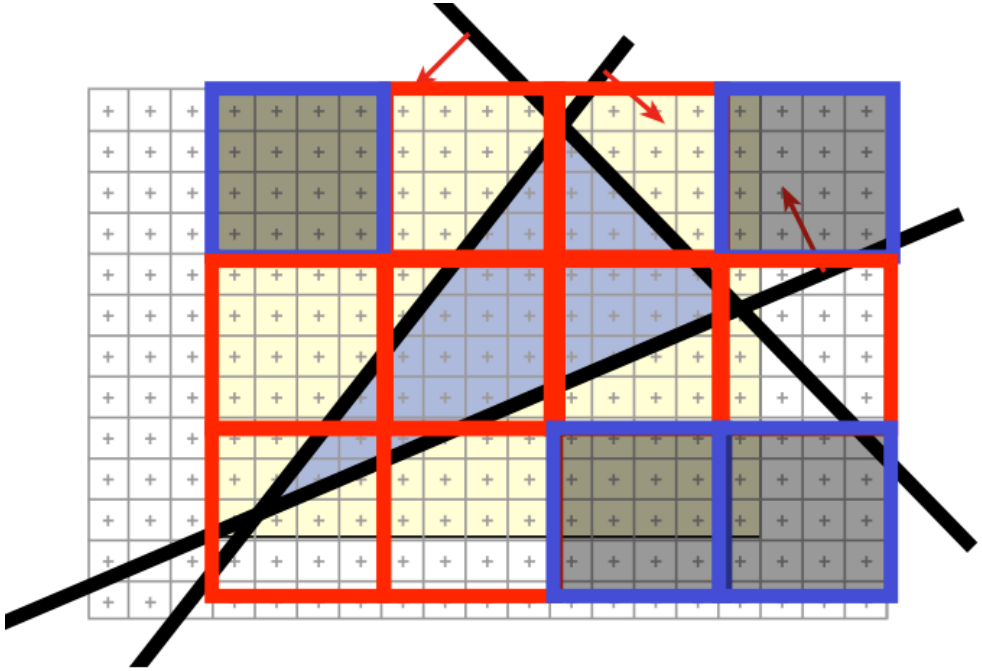
- **Axis-aligned bounding box**
- **Incremental Edge Functions** （扫描线算法）

```
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
```



Speed up the rasterizer

- Axis-aligned bounding box
- Incremental Edge Functions
- Hierarchical Rasterization
  - Conservatively test blocks before going to per-pixel level (can skip large blocks at once)



What if there are many triangles in the scene?

主要讲述的内容：

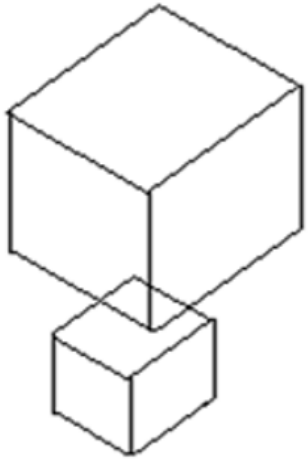
消隐的分类，如何消除隐藏线、隐藏面，主要介绍以下几个算法：

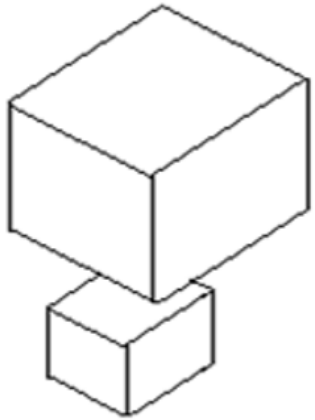
- Z缓冲区(Z-Buffer)算法
- 扫描线Z-buffer算法
- 区域子分割算法

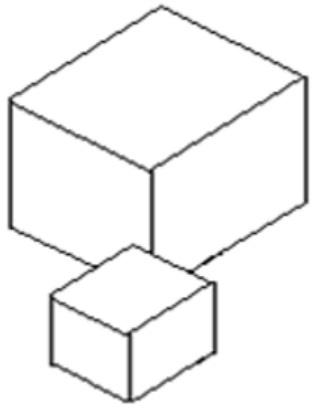
一、消隐

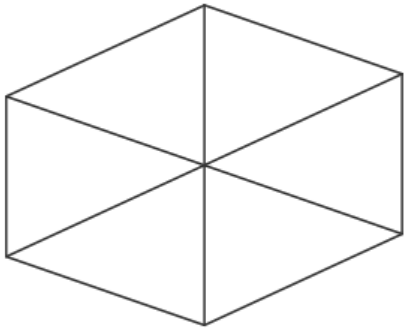
当我们观察空间任何一个不透明的物体时，只能看到该物体朝向我们的那些表面，其余的表面由于被物体所遮挡我们看不到

如果把可见和不可见的线都画出来，对视觉会造成多义性

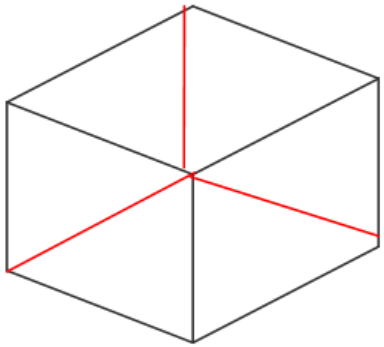




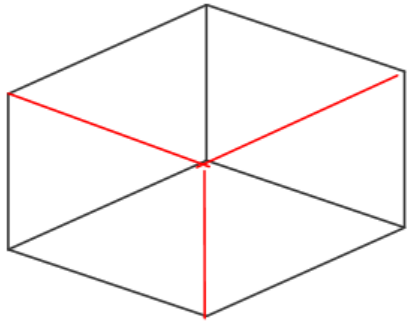




（a）



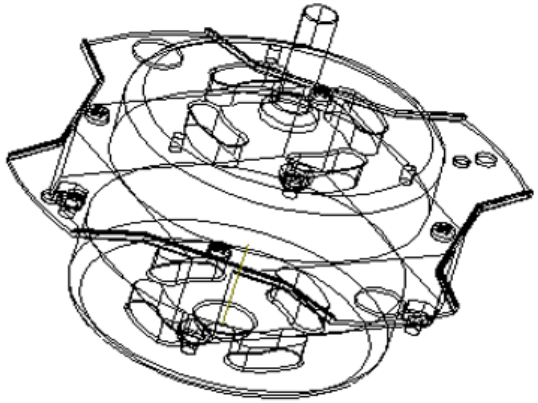
（b）



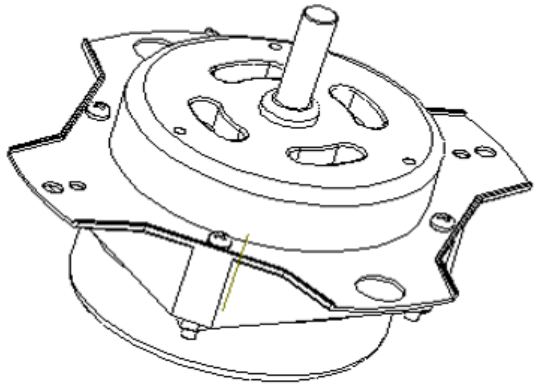
（c）

要消除二义性，就必须在绘制时消除被遮挡的不可见的线或面，习惯上称作消除隐藏线和隐藏面，简称为**消隐**。

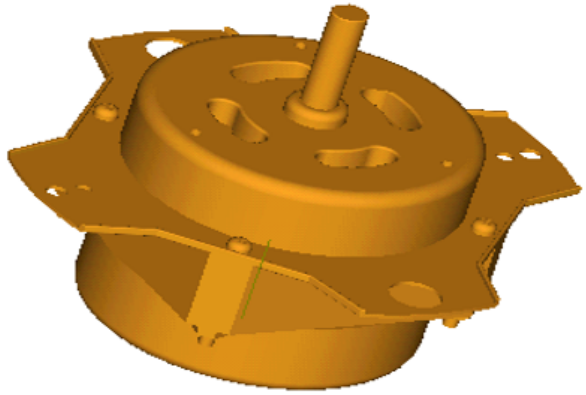
要绘制出意义明确的、富有真实感的立体图形，首先必须消去形体中的不可见部分，而只在图形中表现可见部分



线框图



消隐图

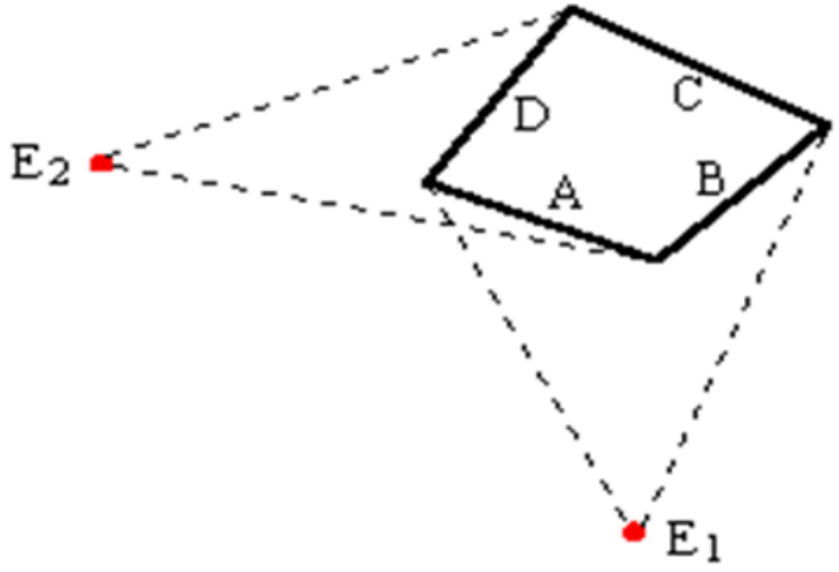


真实感图形

消隐包括消除“隐藏线”和“隐藏面”两个问题

到目前为止，虽然已有数十种算法被提出来了，但是由于物体的形状、大小、相对位置等因素千变万化，因此至今它仍吸引人们作出不懈的努力去探索更好的算法

消隐不仅与消隐对象有关，还与观察者的位置有关



消隐处理从原理上讲并不复杂，但是消隐处理的具体实现并不那么简单，它要求适当的算法及大量的运算。在60年代，消隐问题曾被认为是计算机图形学中的几大难题之一

二、消隐的分类

1、按消隐对象分类

（1）线消隐

消隐对象是物体上的边，消除的是物体上不可见的边

（2）面消隐

消隐对象是物体上的面，消除的是物体上不可见的面，通常做真实感图形消隐时用面消隐

2、按消隐空间分类

（1）物体空间的消隐算法

以场景中的物体为处理单元。假设场景中有k个物体，将其中一个物体与其余k-1个物体逐一比较，仅显示它可见表面以达到消隐的目的

此类算法通常用于线框图的消隐！

```
for（场景中的每一个物体）
  { 将该物体与场景中的其
    它物体进行比较，确定
    其表面的可见部分；
    显示该物体表面的可见
    部分；
  }
```

在物体空间里典型的消隐算法有两个： Roberts算法和光线投射法

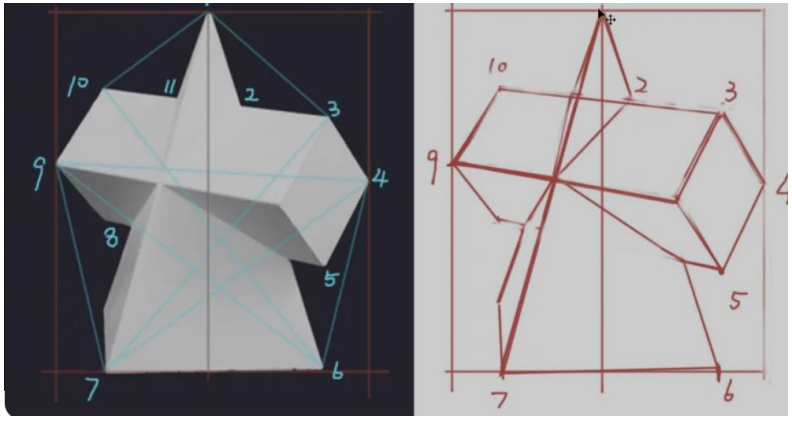
Roberts算法数学处理严谨，计算量甚大。算法要求所有被显示的物体都是凸的，对于凹体要先分割成多个凸体的组合

Roberts算法基本步骤：

- 逐个的独立考虑每个物体自身，找出为其自身所遮挡的边和面（自消隐）；
- 将每一物体上留下的边再与其它物体逐个的进行比较，以确定是完全可见还是部分或全部遮挡（两两物体消隐）；
- 确定由于物体之间的相互贯穿等原因，是否要形成新的显示边等，从而使被显示各物体更接近现实

Roberts算法基本步骤：

- 逐个的独立考虑每个物体自身，找出为其自身所遮挡的边和面（自消隐）；
- 将每一物体上留下的边再与其它物体逐个的进行比较，以确定是完全可见还是部分或全部遮挡（两两物体消隐）；
- 确定由于物体之间的相互贯穿等原因，是否要形成新的显示边等，从而使被显示各物体更接近现实



（2）图像空间的消隐算法

以屏幕窗口内的每个像素为处理单元。确定在每一个像素处，场景中的k个物体哪一个距离观察点最近，从而用它的颜色来显示该像素

```
for（窗口中的每一个像素）
  {确定距视点最近的物体，
   以该物体表面的颜色来显
   示像素；
  }
```

这类算法是消隐算法的主流！

因为最后看到的图像是在屏幕上的，所以就拿屏幕作为处理对象。针对屏幕上的像素来进行处理，算法的思想是围绕着屏幕的。对屏幕上每个象素进行判断，决定哪个多边形在该象素可见。

从物体出发 vs 从屏幕出发

三、图像空间的消隐算法

这类算法是消隐算法的主流！

Z－buffer算法

扫描线算法

Warnock消隐算法

1、Z缓冲区(Z-Buffer)算法

1973年，犹他大学学生艾德·卡姆尔（Edwin Catmull）独立开发出了能跟踪屏幕上每个像素深度的算法 Z-buffer

Z-buffer让计算机生成复杂图形成为可能。Ed Catmull目前担任迪士尼动画和皮克斯动画工作室的总裁

Edwin Catmull

Catmull in 2015

Born	Edwin Earl Catmull March 31, 1945 (age 79) Parkersburg, West Virginia, U.S.
Alma mater	University of Utah (B.S. Physics and Computer Science; Ph.D. Computer Science)

Z缓冲器算法也叫深度缓冲器算法，属于图像空间消隐算法

该算法有帧缓冲器和深度缓冲器。对应两个数组：

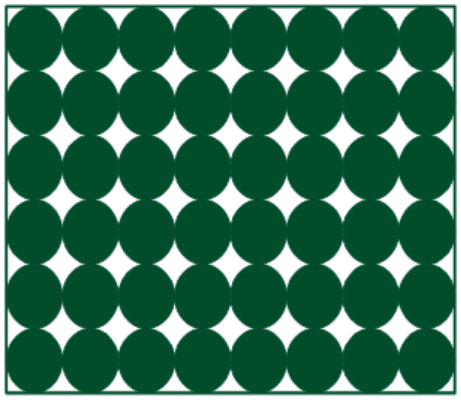
intensity（x，y）——属性数组（帧缓冲器）

存储图像空间每个可见像素的光强或颜色

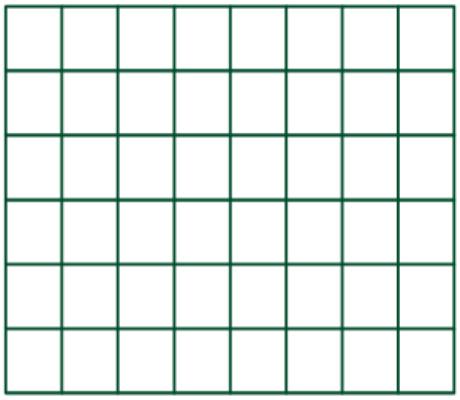
depth（x，y）——深度数组（z-buffer）

存放图像空间每个可见像素的z坐标

屏幕



帧缓冲器



每个单元存放对应
象素的颜色值

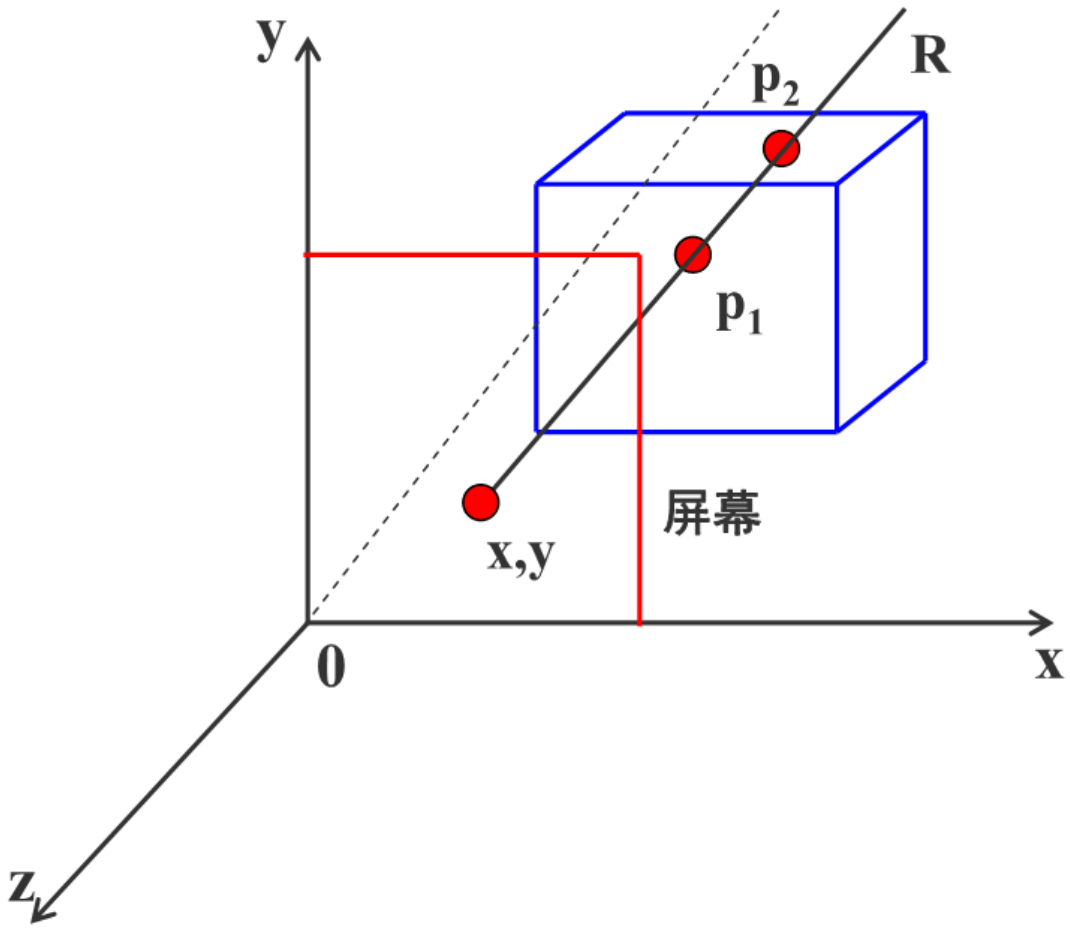
Z缓冲器

每个单元存放对应
象素的深度值

假定xoy面为投影面，z轴为观察方向

过屏幕上任意像素点（x, y）作平行于z轴的射线R，与物体表面相交于p_1和p_2点

p_1和p_2点的z值称为该点的深度值

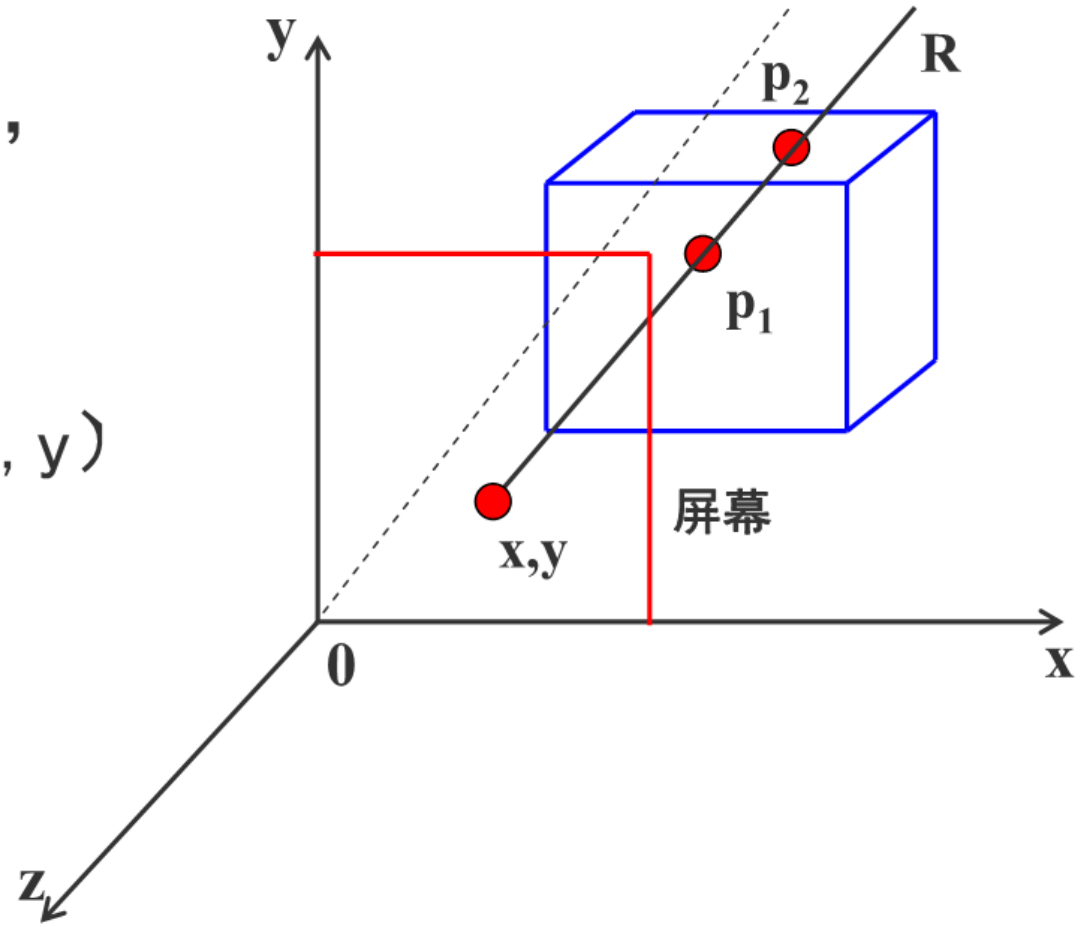


z-buffer算法比较p_1和p_2的z值，

将最大的z值存入z缓冲器中

显然，p_1在p_2前面，屏幕上（x, y）

这一点将显示p_1点的颜色



算法思想：先将Z缓冲器中各单元的初始值置为最小值。当要改变某个像素的颜色值时，首先检查当前多边形的深度值是否大于该像素原来的深度值（保存在该像素所对应的Z缓冲器的单元中）

如果大于原来的z值，说明当前多边形更靠近观察点，用它的颜色替换像素原来的颜色

```
Z-Buffer算法（）
 { 帧缓存全置为背景色
     深度缓存全置为最小z值
     for(每一个多边形)
       { 扫描转换该多边形
         for(该多边形所覆盖的每个象素(x,y) )
          {计算该多边形在该象素的深度值Z(x,y);
            if(z(x,y)大于z缓存在(x,y)的值)
              {把z(x,y)存入z缓存中(x,y)处
                把多边形在(x,y)处的颜色值存入帧缓存的(x,y)处
              }
          }
       }
 }
```

z-Buffer算法的优点：

（1）Z-Buffer算法比较简单，也很直观

（2）在象素级上以近物取代远物。与物体在屏幕上的出现顺序是无关紧要的，有利于硬件实现

z-Buffer算法的缺点：

（1）占用空间大

（2）没有利用图形的相关性与连续性，这是z-buffer算法的严重缺陷

（3）更为严重的是，该算法是在像素级上的消隐算法

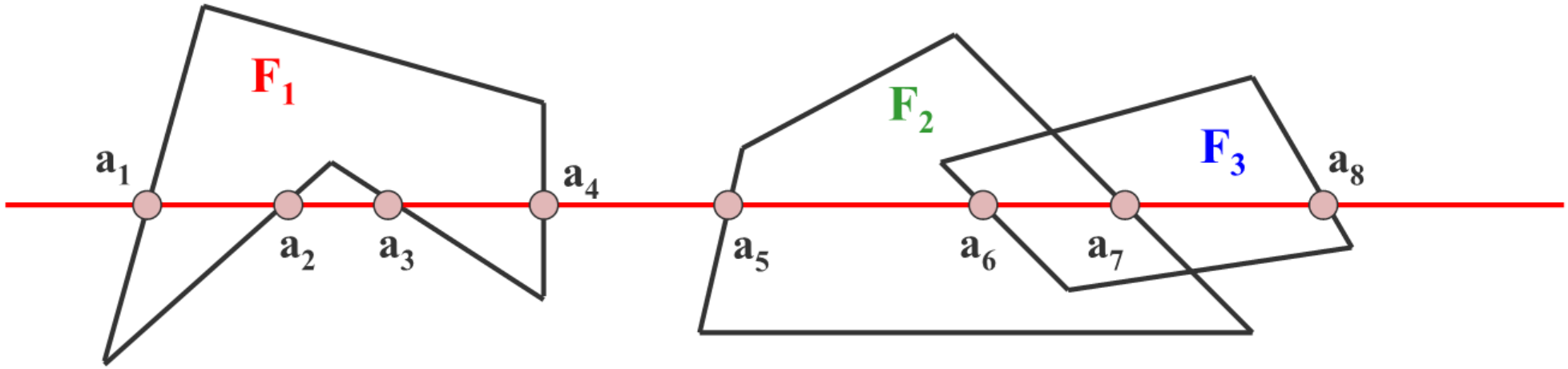
二、区间扫描线算法

前面介绍了经典的z-buffer算法，思想是开一个和帧缓存一样大小的存储空间，利用空间上的牺牲换取算法上的简洁

还介绍了只开一个缓存变量的z-buffer算法，是把问题转化成判别点在多边形内，通过把空间多边形投影到屏幕上，判别该像素是否在多边形内

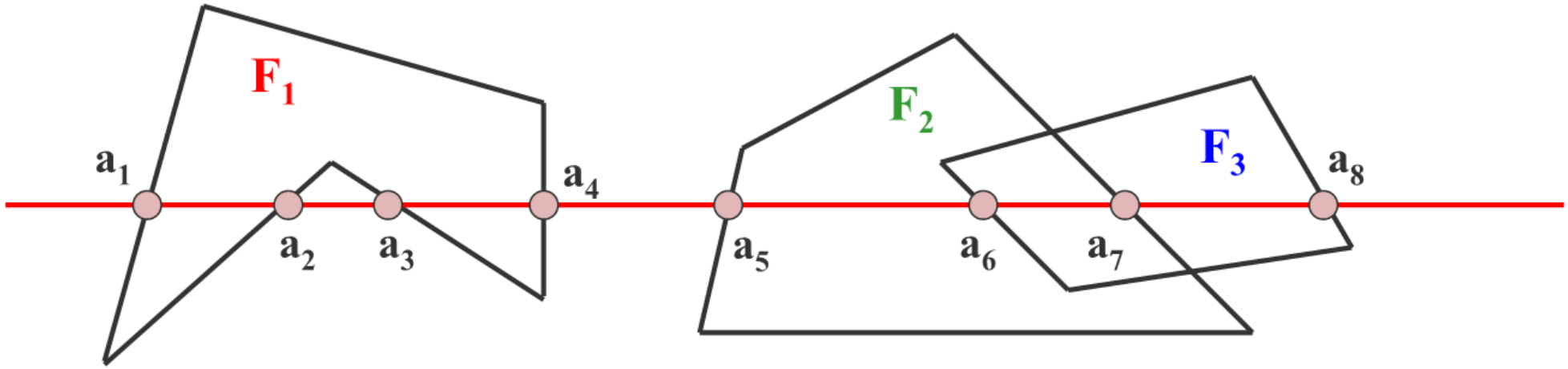
下面介绍区间扫描线算法。该算法放弃了z-buffer的思想，是一个新的算法，这个算法被认为是消隐算法中最快的

因为不管是哪一种z-buffer算法，都是在像素级上处理问题，要进行消隐，每个像素都要进行计算判别，甚至一个像素要进行多次（一个像素可能会被多个多边形覆盖）



扫描线的交点把这条扫描线分成了若干个区间，每个区间上必然是同样一种颜色

对于有重合的区间，如a6a7这个区间，要么显示F2的颜色，要么显示F3的颜色，不会出现颜色的跳跃

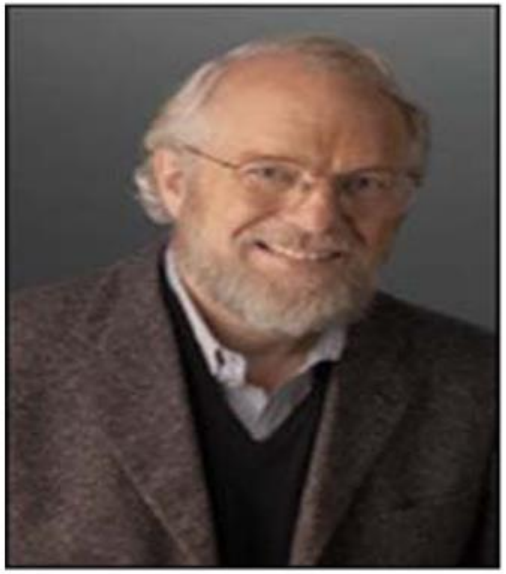


如果把扫描线和多边形的这些交点都求出来，对每个区间，只要判断一个像素的要画什么颜色，那么整个区间的颜色都解决了，这就是区间扫描线算法的主要思想

算法的优点：将象素计算改为逐段计算，效率大大提高！

三、区域子分割算法（ Warnock算法）

John E. Warnock 博士，Adobe创始人之一，曾担任董事会主席



In his 1969 doctoral thesis, Warnock invented the Warnock algorithm for hidden surface determination in computer graphics

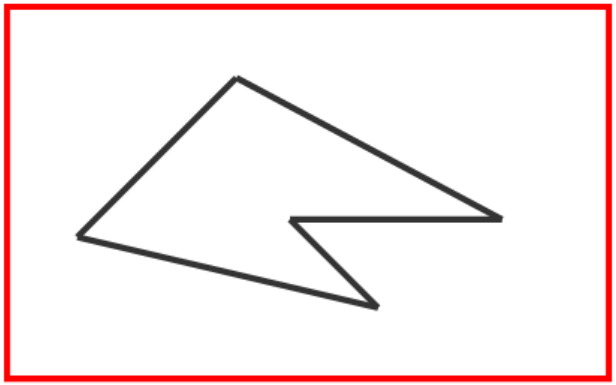
Warnock算法是图像空间中非常经典的一个算法

Warnock算法的重要性不在于它的效率比别的算法高，而在于采用了分而治之的思想，利用了堆栈的数据结构

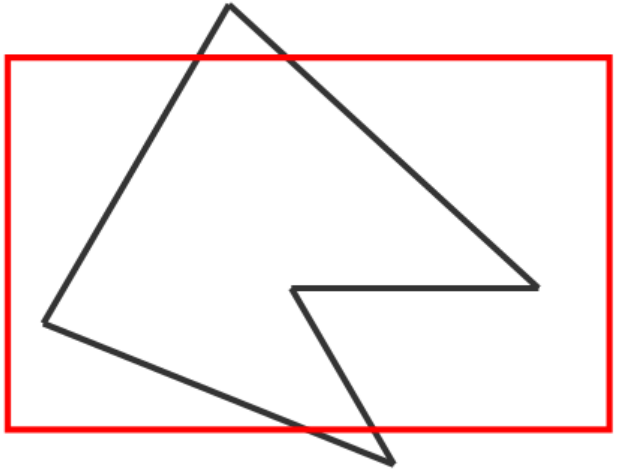
把物体投影到全屏幕窗口上，然后递归分割窗口，直到窗口内目标足够简单，可以显示为止

一、什么样的情况下，画面足够简单可以立即显示？

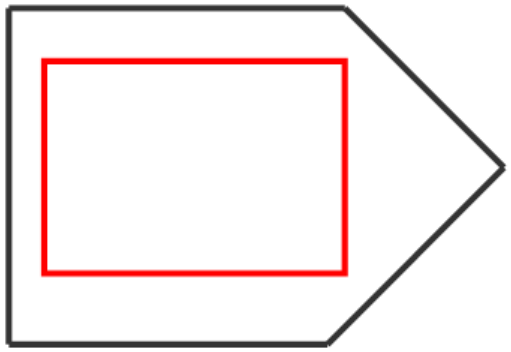
（1）窗口中仅包含一个多边形



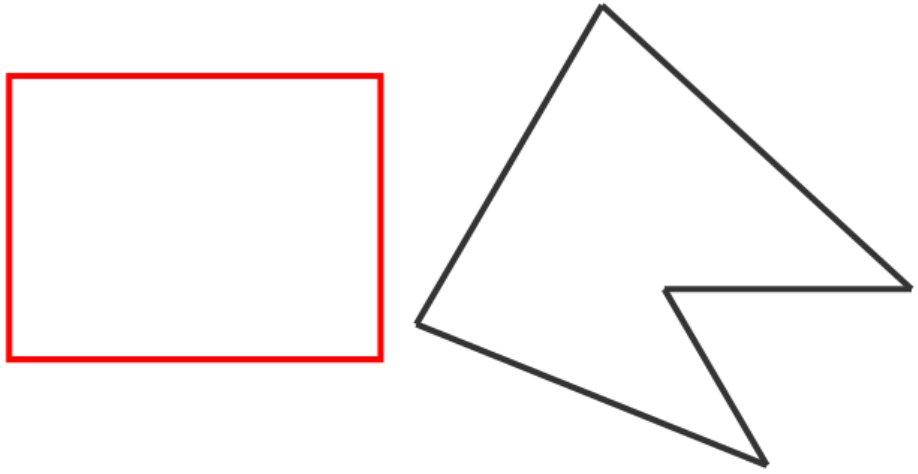
（2）窗口与一个多边形相交，且窗口内无其它多边形



（3）窗口为一个多边形所包围



（4）窗口与一个多边形相分离

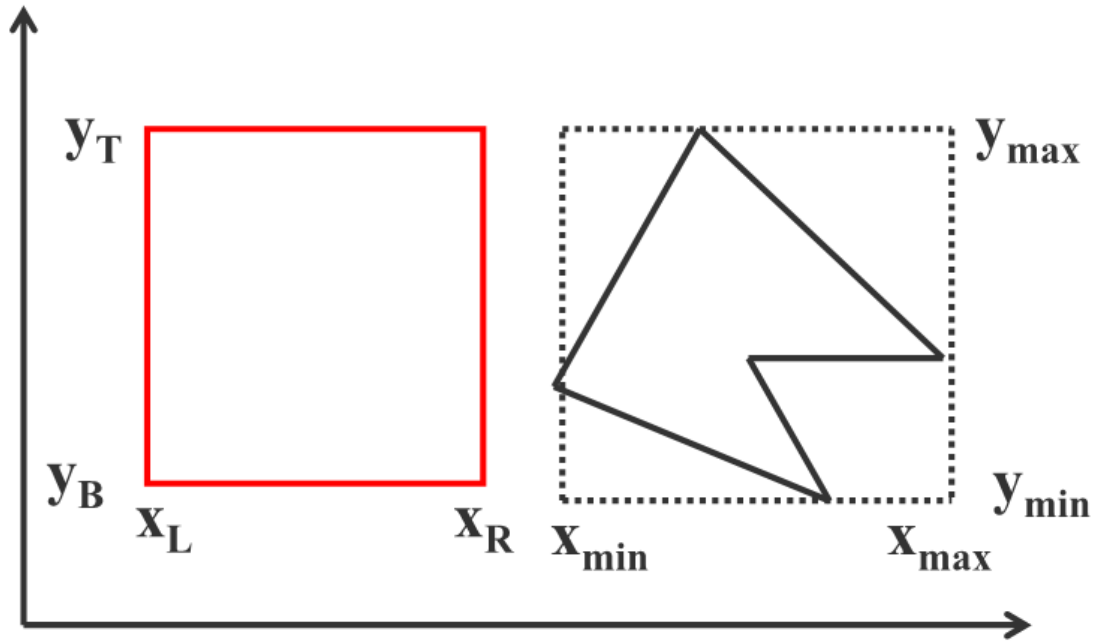


如何判别一个多边形和窗口是分离的？

当满足下列条件时，多边形和窗口分离：

$$x_{\min} > x_R \quad \text{or} \quad x_{\max} < x_L$$

$$y_{\min} > y_T \quad \text{or} \quad y_{\max} < y_B$$

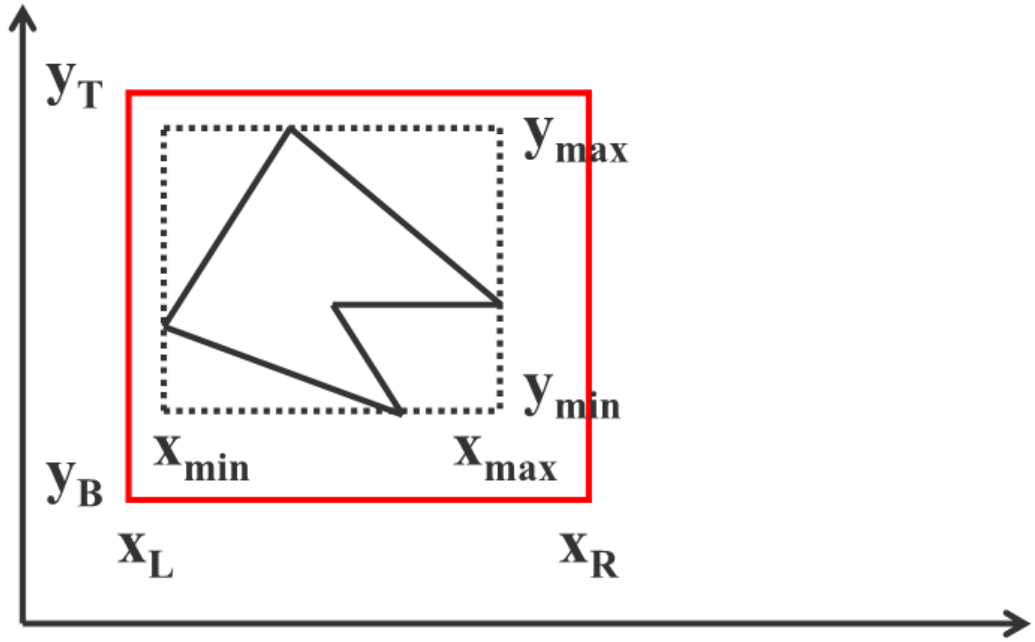


如何判别一个多边形在窗口内？

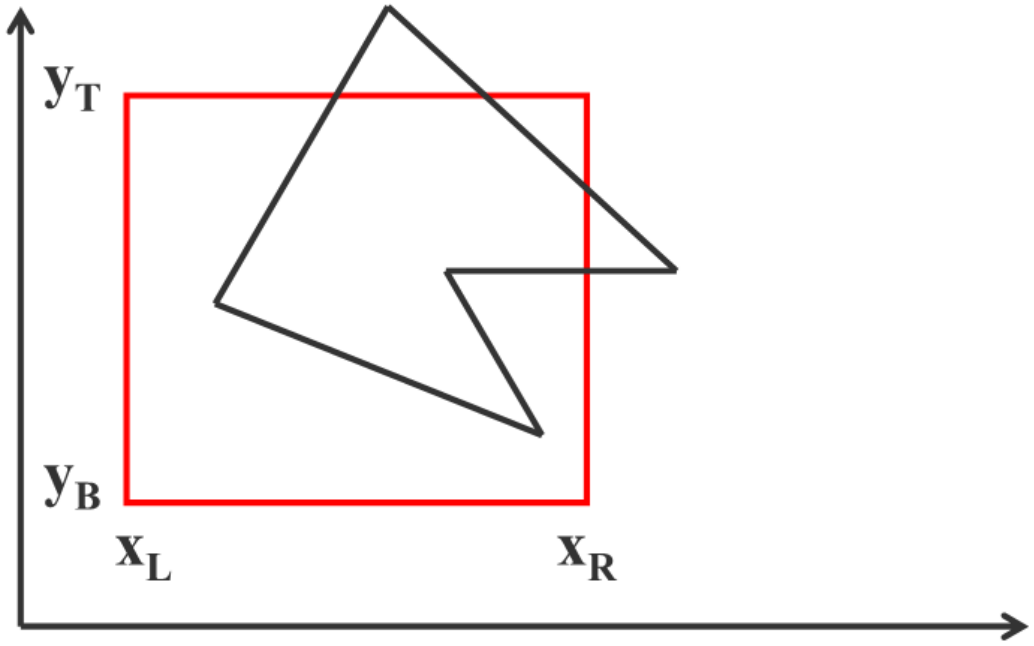
当满足下列条件时，多边形被窗口包含：

$$x_{\min} \geq x_L \quad \& \quad x_{\max} \leq x_R$$

$$y_{\min} \geq y_B \quad \& \quad y_{\max} \leq y_T$$



多边形与窗口相交的判别，可以采用直线方程作为判别函数来判定一个多边形是否与窗口相交

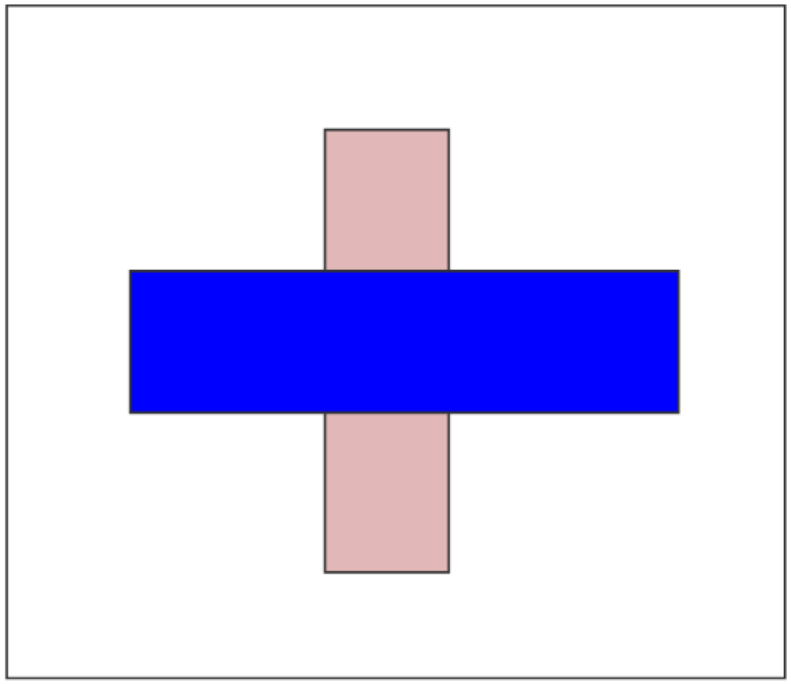


二、窗口有多个多边形投影面，如何显示？

Warnock算法的重要性不在于它的效率比别的算法高，而在于采用了分而治之的思想，利用了堆栈的数据结构

把物体投影到全屏幕窗口上，然后递归分割窗口，直到窗口内目标足够简单，可以显示为止

算法步骤：

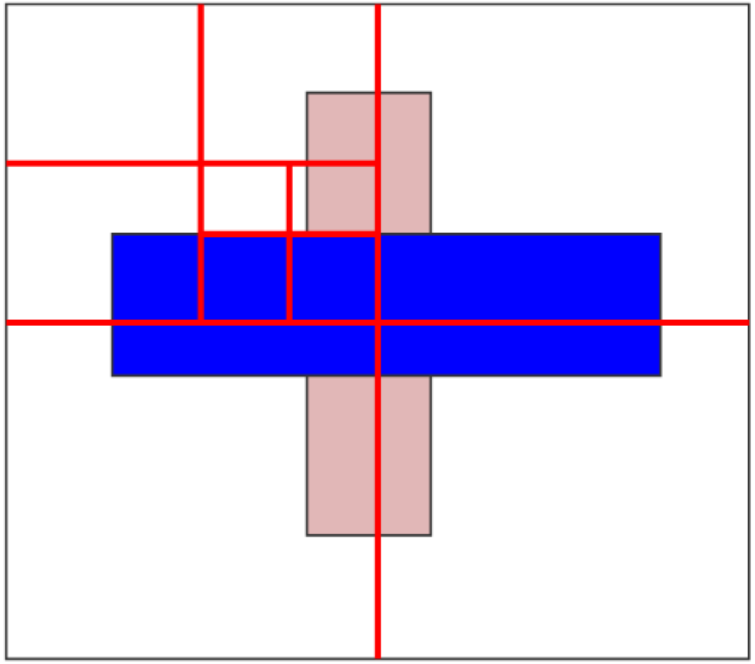


（1）如果窗口内没有物体则按背景色显示

（2）若窗口内只有一个面，则把该面显示出来

（3）否则，窗口内含有两个以上的面，则把窗口等分成四个子窗口。对每个小窗口再做上述同样的处理。这样反复地进行下去

（3）窗口内含有两个以上的面，则把窗口等分成四个子窗口。对每个小窗口再做上述同样的处理。这样反复地进行下去



把四个子窗口压在一个堆栈里（后进先出）。

核心思想

（1）增量思想：通过增量算法可以减少计算量

（2）编码思想

（3）符号判别—> 整数算法 尽可能的提高底层算法的效率，底层上提高效率才是真正解决问题

核心思想

（4）图形连贯性：利用连贯性可以大大减少计算量

（5）分而治之：把一个复杂对象进行分块，分到足够简单再进行处理

Thank you!