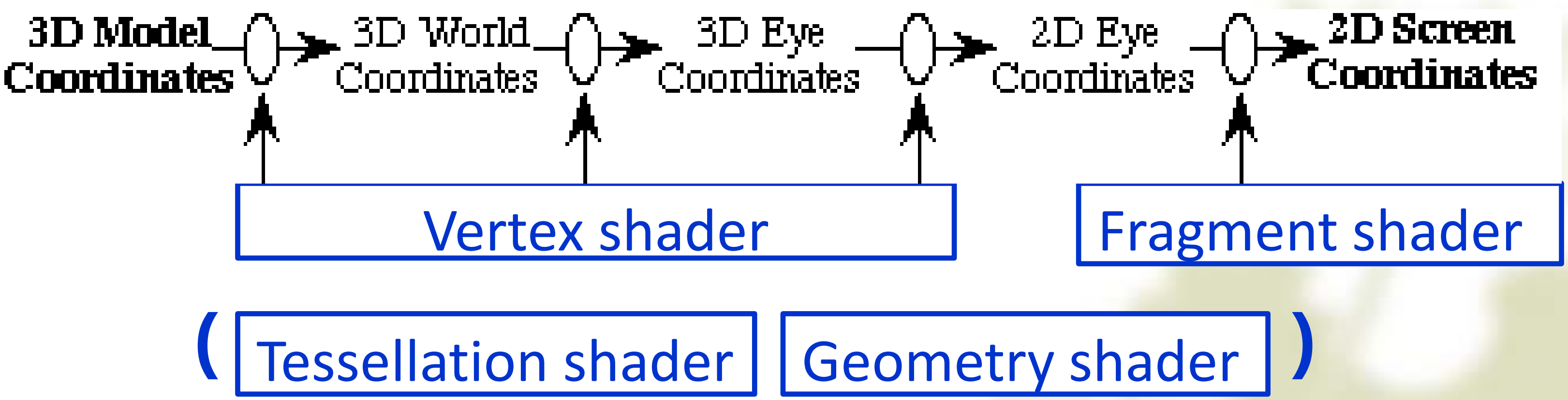
Chapter 8: From Vertices to Fragments

How the OpenGL system creates the image from your modeling

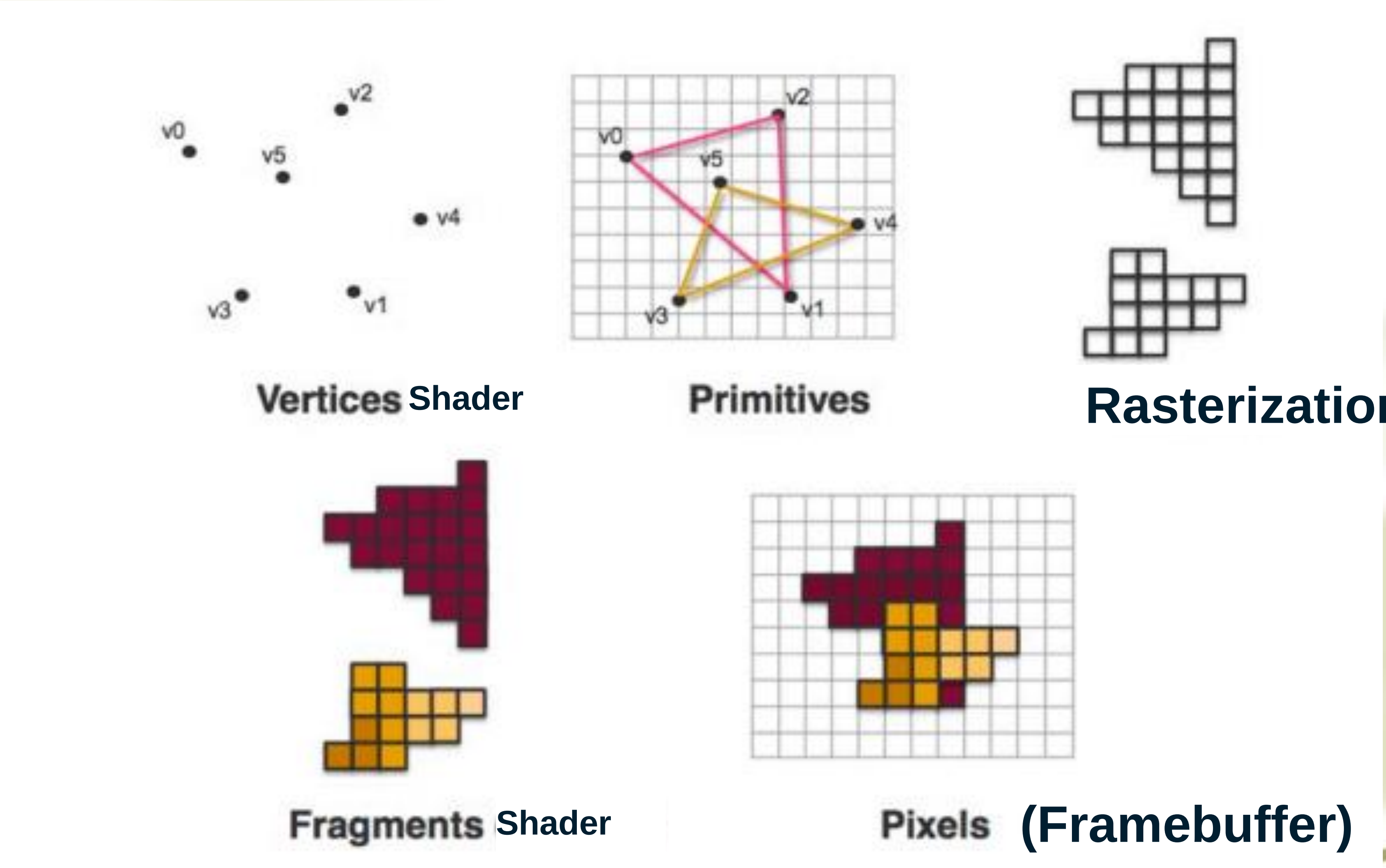
Modeling Pipeline

The modeling pipeline only maps vertices between the various spaces.



Rendering Pipeline

All vertices are processed to form an image in framebuffer.



OpenGL 4.5

The OpenGL 4.5 and OpenGL Shading Language 4.50 Specifications were released on August 11, 2014.
https://www.opengl.org/documentation/current_version/

New features of OpenGL 4.5 include:

•*Direct State Access (DSA)*

object accessors enable state to be queried and modified without binding objects to contexts, for increased application and middleware efficiency and flexibility;

•*Flush Control*

applications can control flushing of pending commands before context switching – enabling high-performance multithreaded applications;

OpenGL 4.5

- *Robustness*
  providing a secure platform for applications such as WebGL browsers, including preventing a GPU reset affecting any other running applications;
- *OpenGL ES 3.1 API and shader compatibility*
  to enable the easy development and execution of the latest OpenGL ES applications on desktop systems;
- *DX11 emulation features*
  for easier porting of applications between OpenGL and Direct3D.

New extensions to OpenGL 4.5

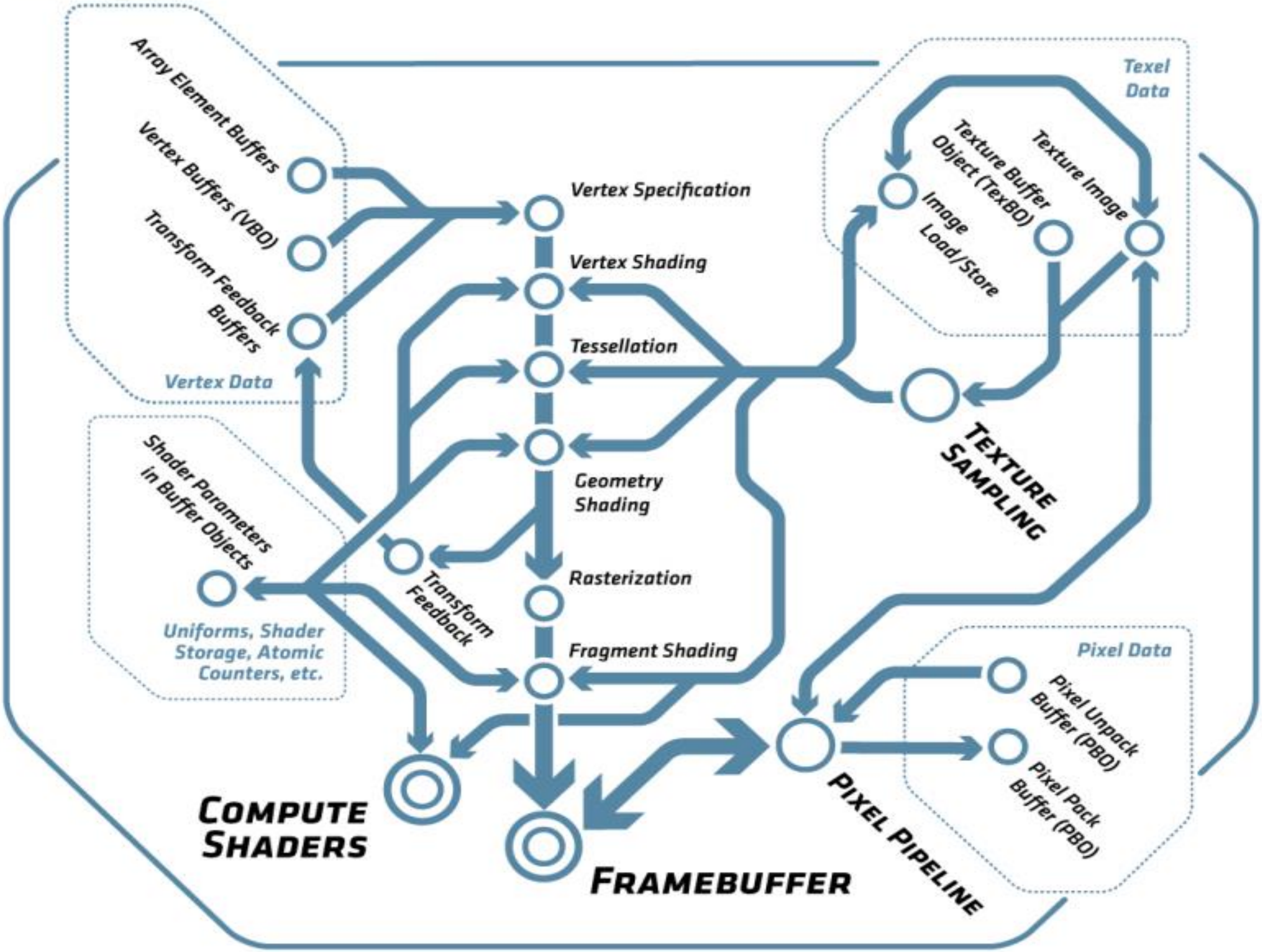
Vulkan是新的跨平台图形库

GL_ARB_clip_control
GL_ARB_cull_distance
GL_ARB_ES3_1_compatibility
GL_ARB_conditional_render_inverted
GL_KHR_context_flush_control
GL_ARB_derivative_control
GL_ARB_direct_state_access
GL_ARB_get_texture_sub_image
GL_KHR_robustness
GL_ARB_shader_texture_image_samples
GL_ARB_texture_barrier

ARB：Architecture Review Board

Khronos royalty-free, open standards for 3D graphics, Virtual and Augmented Reality, Parallel Computing, Neural Networks, and Vision Processing

OpenGL 4.6 Core Profile



OpenGL Objects Model

- Buffer Objects
- Shader Objects
- Program Objects
- Program Pipeline Objects
- Texture Objects
- Sampler Objects
- Renderbuffer Objects
- Framebuffer Objects
- Vertex Array Objects
- Transform Feedback Objects
- Query Objects
- Sync Objects

Key Contents

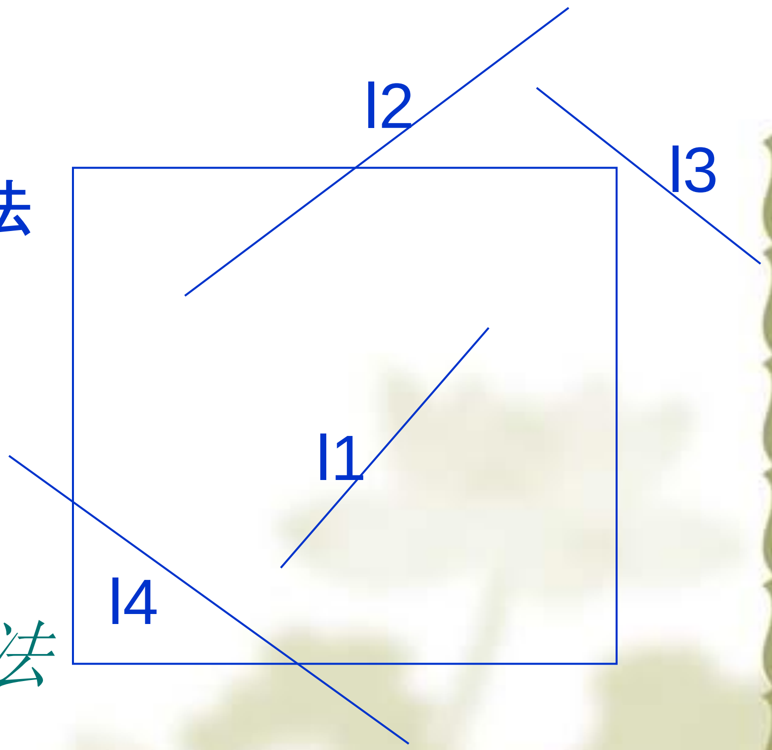
1. Clipping: line, polygon
2. Rasterization – Scan Conversion
3. Hidden-Surface Removal
4. Antialiasing
5. Color Model

1. Line Clipping Algorithms

Two algorithms will be explained:

1.1 Cohen-Sutherland线段裁剪算法

1.2 梁友栋-Barsky线段裁剪算法



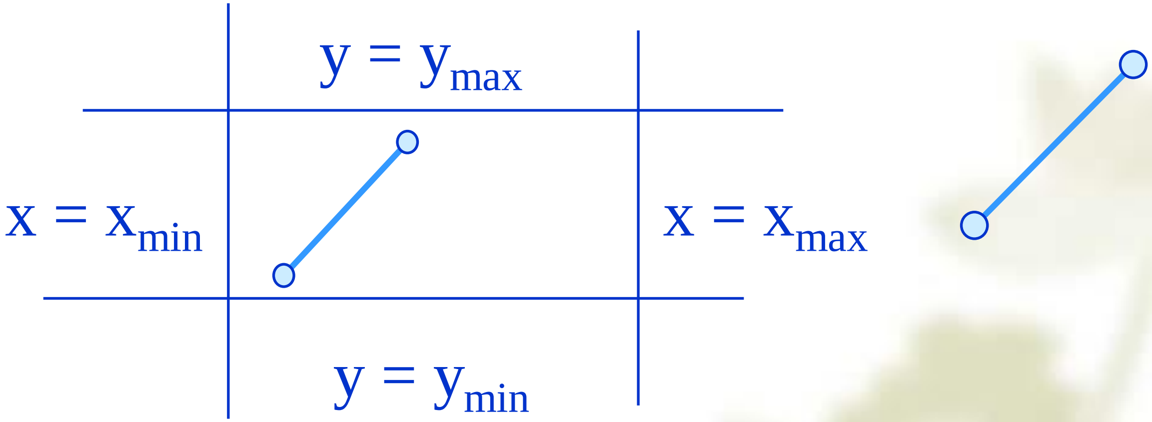
1.1 Cohen-Sutherland线段裁剪算法

Two Steps：

1. 判断线段是否需要裁剪
2. 裁剪线段

The Cases

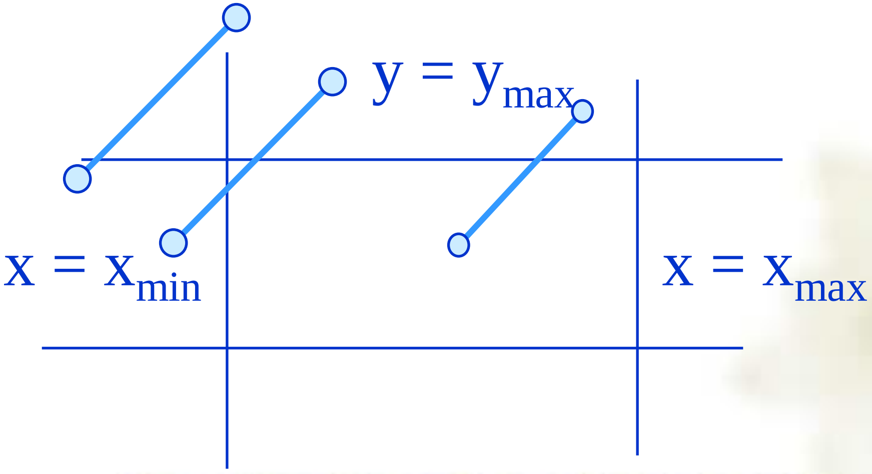
- Case 1: both endpoints of line segment inside all four lines
  - Draw (accept) line segment as is



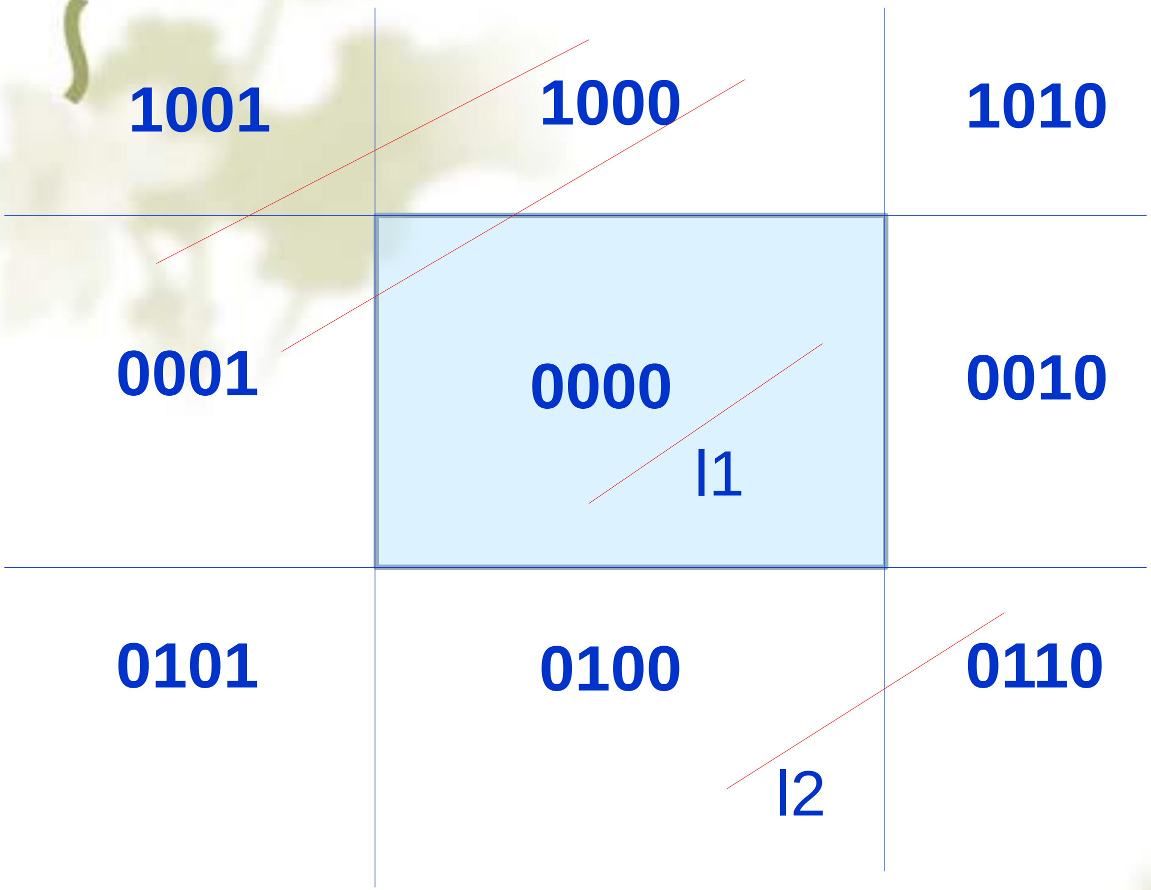
- Case 2: both endpoints outside all lines and on same side of a line
  - Discard (reject) the line segment

The Cases

- Case 3: One endpoint inside, one outside
  - Must do at least one intersection
- Case 4: Both outside
  - May have part inside
  - Must do at least one intersection



Region Outcodes：



XXXX X=0/1

↓↓↓↓

TBRL

for examples：

l1: code1 = 0000; code2 = 0000

l2: code1 = 0100; code2 = 0110

Three cases to decide:

1.线段完全保留;

if (code1==0&&code2==0)

or: (code1|code2)==0

2.线段完全放弃;

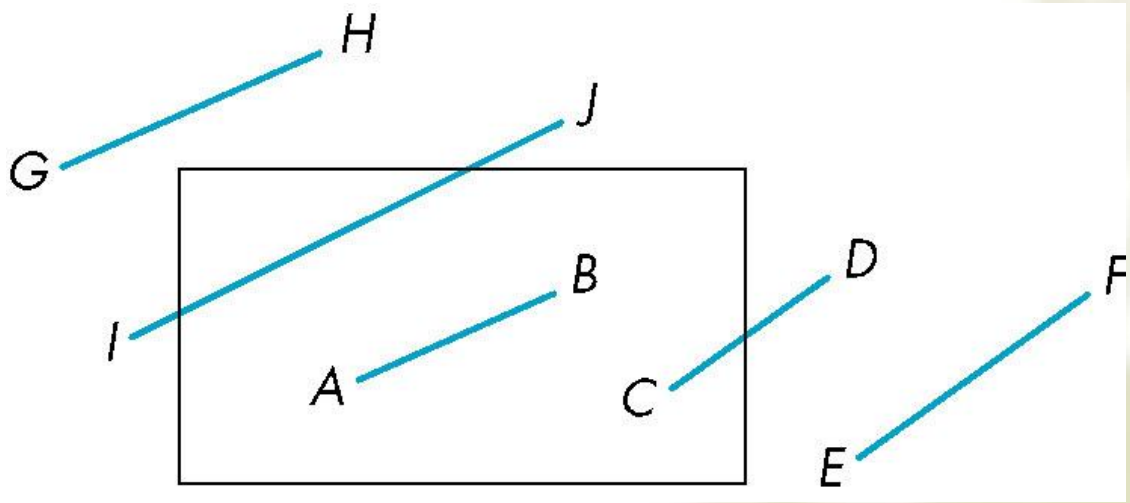
if ((code1&code2)<>0)

3.裁剪线段；

else computing intersection

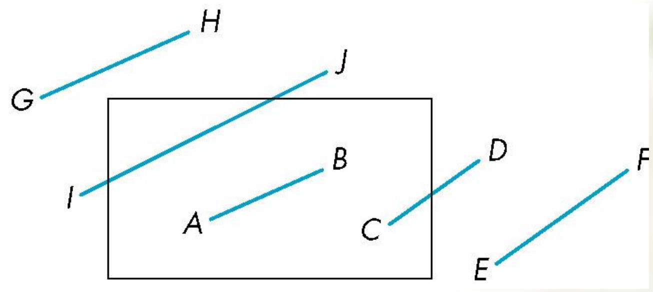
Using Outcodes

- Consider the 5 cases below
- AB: outcode(A) = outcode(B) = 0
  - **Accept** line segment



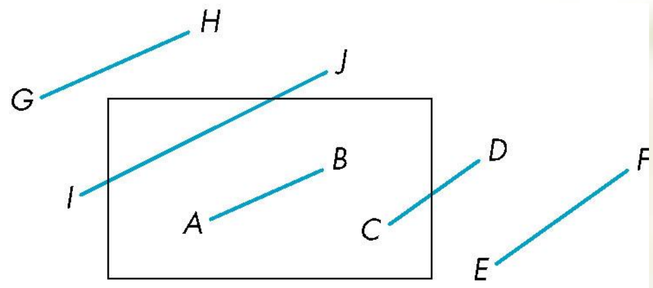
Using Outcodes

- CD: outcode (C) = 0, outcode(D) $\neq$ 0
  - Compute intersection
  - Location of 1 in outcode(D) determines which edge to intersect with
  - Note if there were a segment from A to a point in a region with 2 ones in outcode, we might have to do two interesections



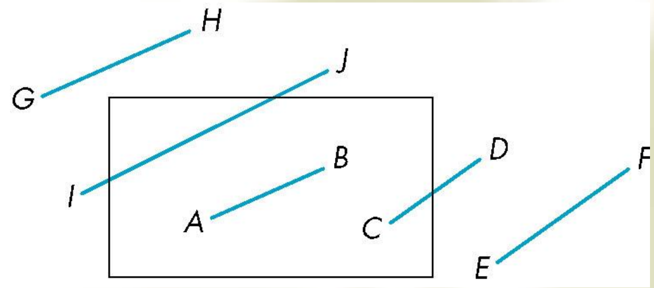
Using Outcodes

- EF: outcode(E) logically ANDed with outcode(F) (bitwise) $\neq 0$
  - Both outcodes have a 1 bit in the same place
  - Line segment is outside of corresponding side of clipping window
  - **reject**



Using Outcodes

- GH and IJ: same outcodes, neither zero but logical AND yields zero
- Shorten line segment by intersecting with one of sides of window
- Compute outcode of intersection (new endpoint of shortened line segment)
- **Reexecute** algorithm

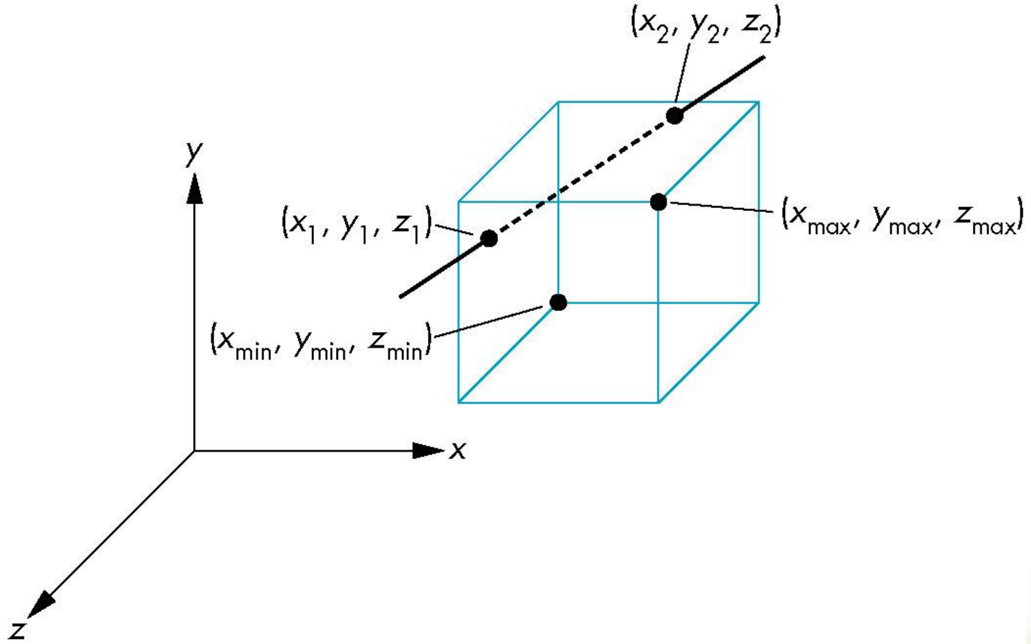


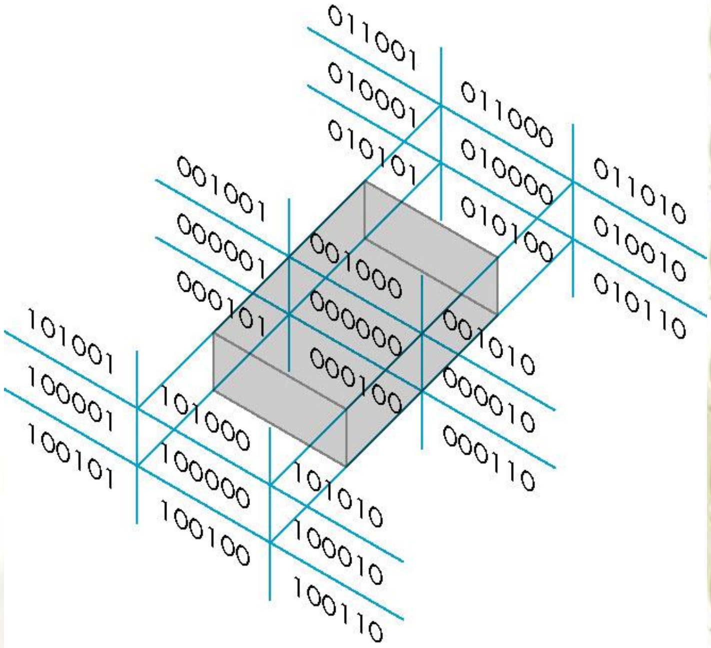
Efficiency

- In many applications, the clipping window is small relative to the size of the entire data base
  - Most line segments are outside one or more side of the window and can be eliminated based on their outcodes
- Inefficiency when code has to be reexecuted for line segments that must be shortened in more than one step

Cohen Sutherland in 3D

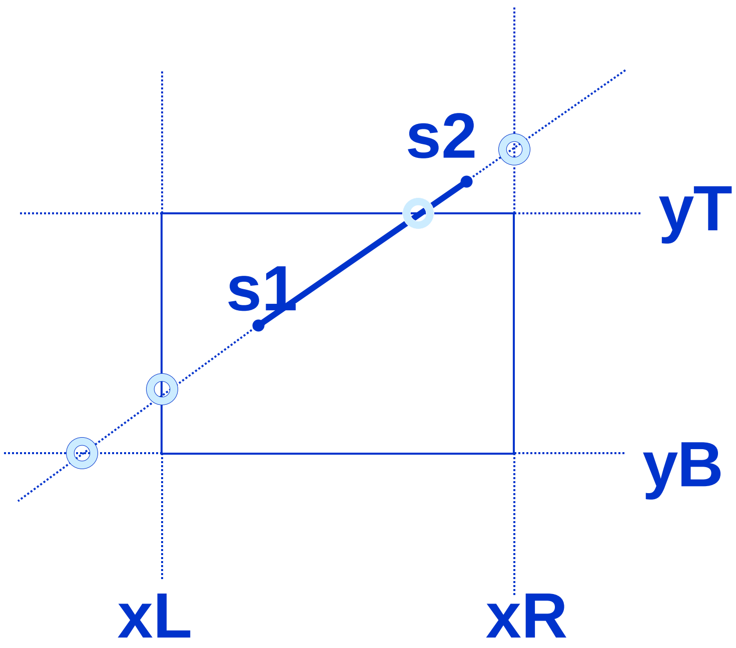
- Use 6-bit outcodes
- When needed, clip line segment against planes





1.2 梁友栋-Barsky线段裁剪算法

线段的参数方程: $\begin{cases} x = x1 + t*(x2 - x1) \\ y = y1 + t*(y2 - y1) \end{cases}$ $0 <= t <= 1$



始边:靠近s1的窗边界线
终边:靠近s2的窗边界线
(由t值决定)

(s1,s2)的始边: xL，yB
终边: xR，yT

对任一线段, 有:

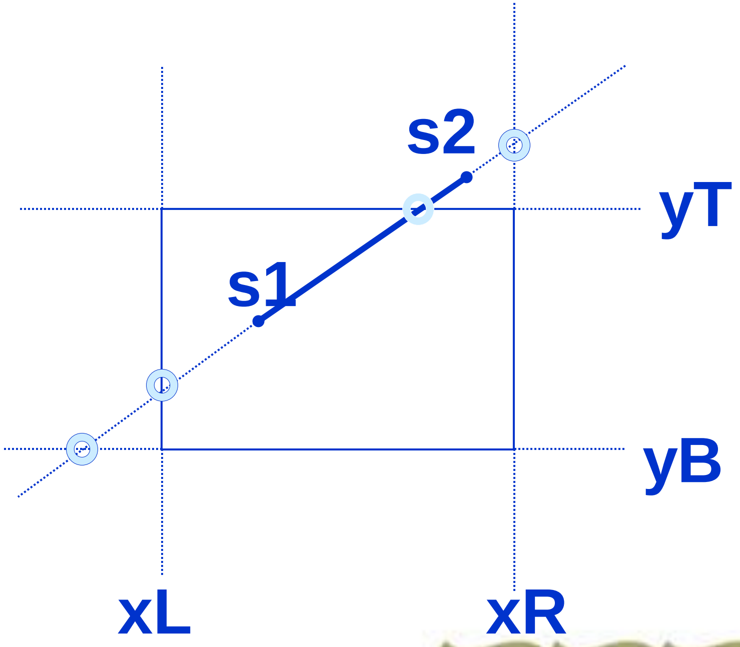
当: x2 - x1 >= 0 , xL为始边, xR为终边;

y2 - y1 >= 0 , yB为始边, yT为终边;

当: x2 - x1 < 0 , xR为始边, xL为终边;

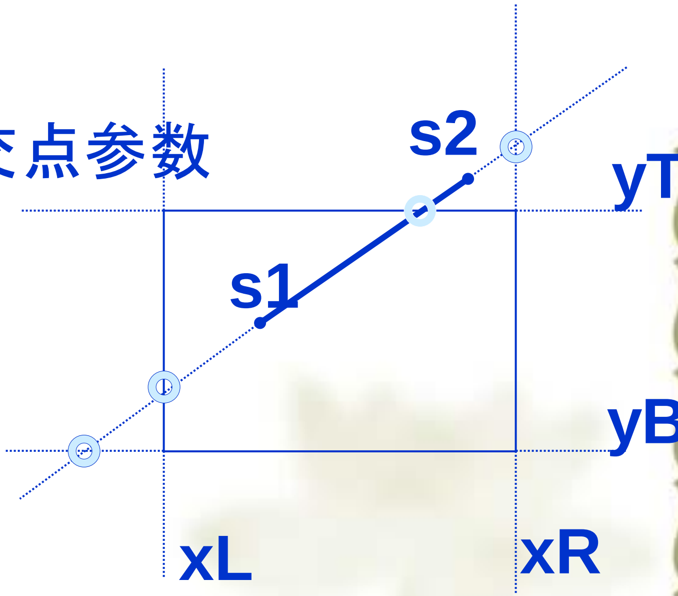
y2 - y1 < 0 , yT为始边, yB为终边.

(1)



设: t_1', t_1''为s1,s2与两个始边的交点参数
则: $t_1 = \max\{t_1', t_1'', 0\}$
t_1为最靠近s2的裁剪点参数

同理, 设: t_2', t_2''为s1,s2与两个终边的交点参数
则: $t_2 = \min\{t_2', t_2'', 1\}$
t_2为最靠近s1的裁剪点参数



当 $t_1<t_2$ 时, $\begin{cases} x = (x2 - x1)*t + x1 \\ y = (y2 - y1)*t + y1 \end{cases}$ $\quad t_1 < t < t_2$
为可见的直线段

当 $t_1>t_2$ 时, 直线不可见

求t_1', t_1''， t_2', t_2'':

$$xL <= (x2 - x1)*t + x1 <= xR$$

$$yB <= (y2 - y1)*t + y1 <= yT$$

简化为: $t*p_k <= q_k$ k=1,2,3,4

其中: $p1 = -(x2 - x1)$ $q1 = x1 - xL$

$p2 = (x2 - x1)$ $q2 = xR - x1$

$p3 = -(y2 - y1)$ $q3 = y1 - yB$

$p4 = (y2 - y1)$ $q4 = yT - y1$

与四个窗边界线的交点参数为: $t_k = q_k / p_k$,但不需做除法

由(1)式得: 若 $p_k < 0$, 则 t_k为始边之交点参数

若 $p_k > 0$, 则 t_k为终边之交点参数

若 $p_k = 0$, 当$q_k<0$, 则线段完全不可见

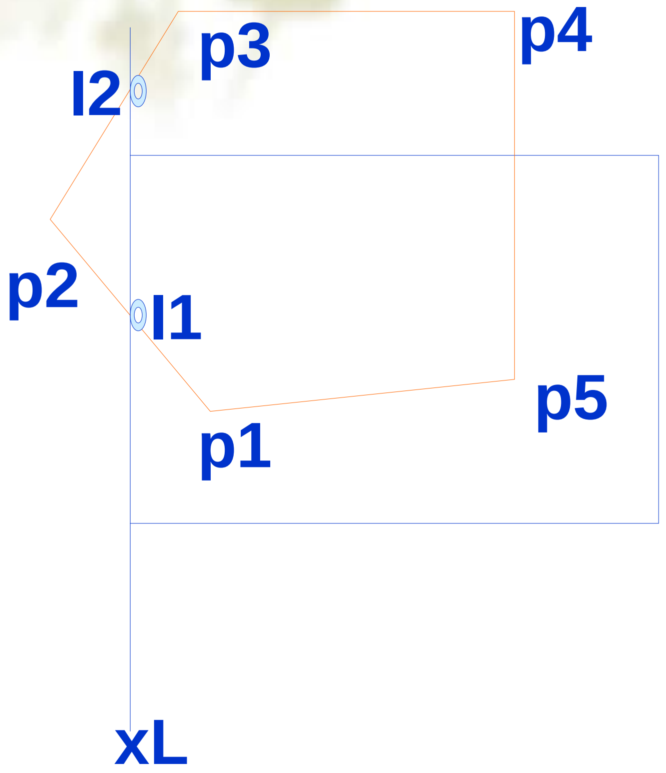
Advantages

- Can accept/reject as easily as with Cohen-Sutherland
- Using values of *t*, we do not have to use algorithm recursively as with C-S
- Extends to 3D

1.3 Polygon Fill-Area Clipping

Sutherland-Hodgman Polygon Clipping:

思想: 用四个窗边界线依次裁剪多边形的所有边

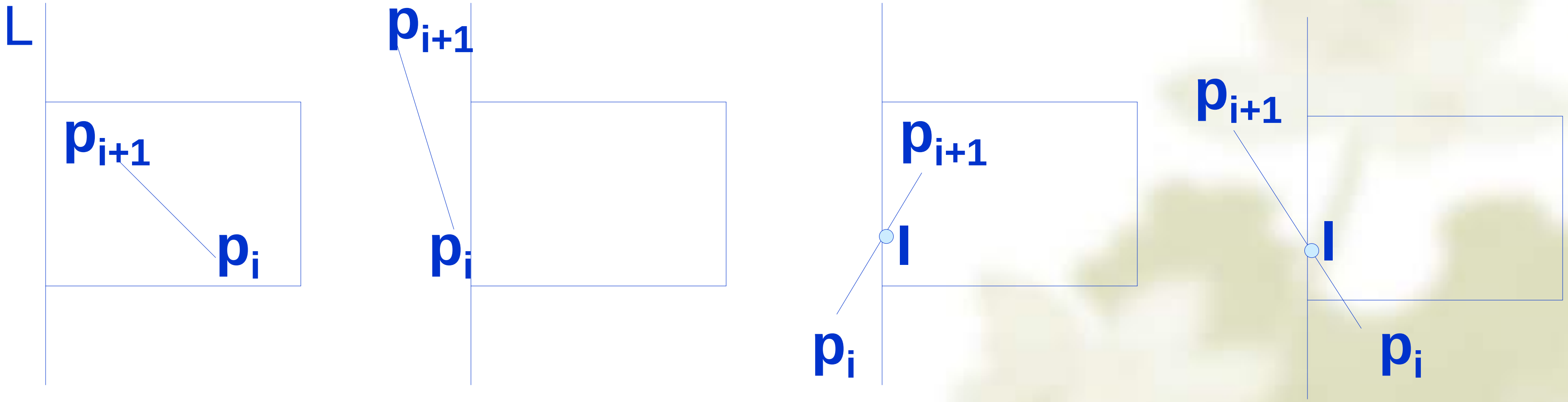



```
for(窗边界线:1 to 4) {
    for(多边形的边界线:1 to n)
        求输出顶点;
    得到一个裁剪后的顶点序列;
}
得到最终裁剪后的顶点序列;
```

判断多边形的边界线p_i,p_{i+1}与窗边界线L的相交情况:

1. 当p_i,p_{i+1}在L的同一侧, 则它们的可见性相同, 且无交点;
2. 当p_i,p_{i+1}在L的两侧, 则它们的可见性不相同, 且有交点;

四种情况:



输出p_{i+1}　**不输出顶点**　**输出I和p_{i+1}**　**输出I**

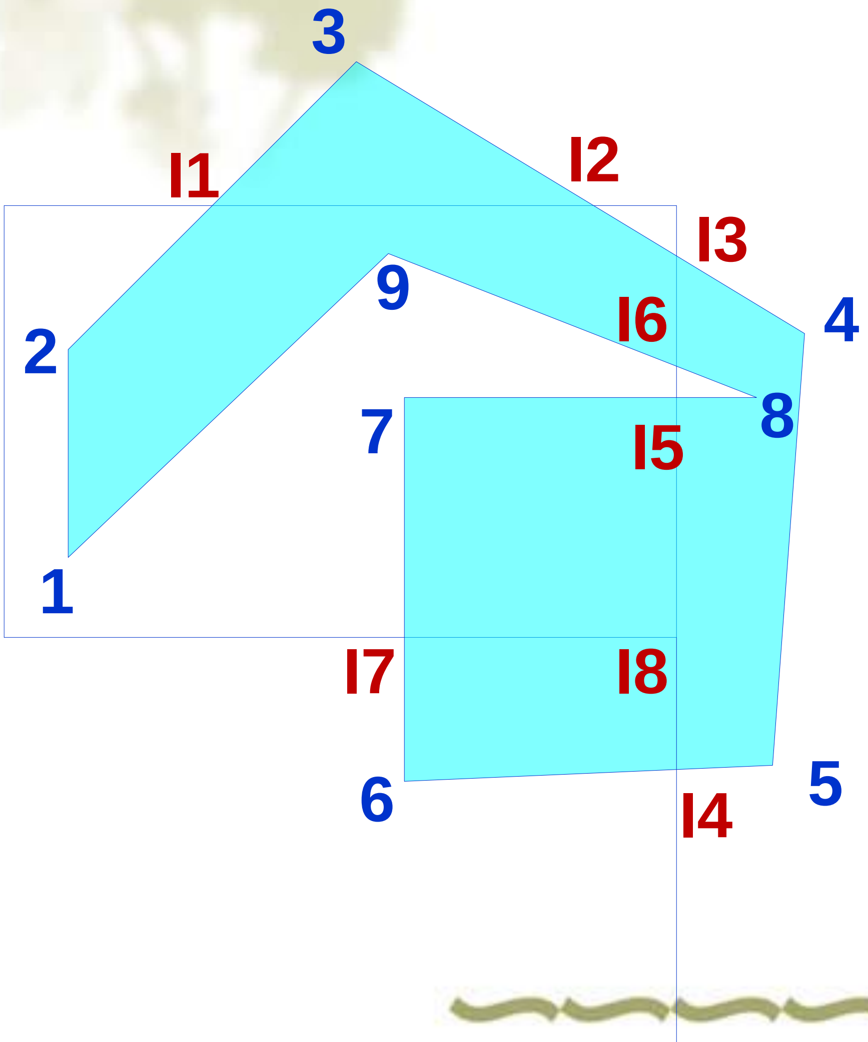
例: 多边形1,2,3,4,5,6,7,8,9

裁剪xL后:1,2,3,4,5,6,7,8,9

裁剪yT后:1,2,I1,I2,4,5,6,7,8,9

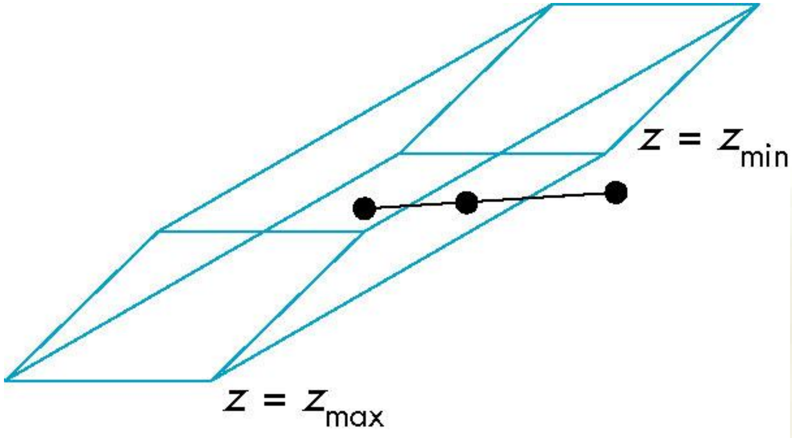
裁剪xR后:1,2,I1,I2,I3,I4,6,7,I5,I6,9

裁剪yB后:1,2, I1,I2,I3,I8,I7,7,I5,I6,9



Clipping and Normalization

- General clipping in 3D requires intersection of line segments against arbitrary plane
- Example: oblique view

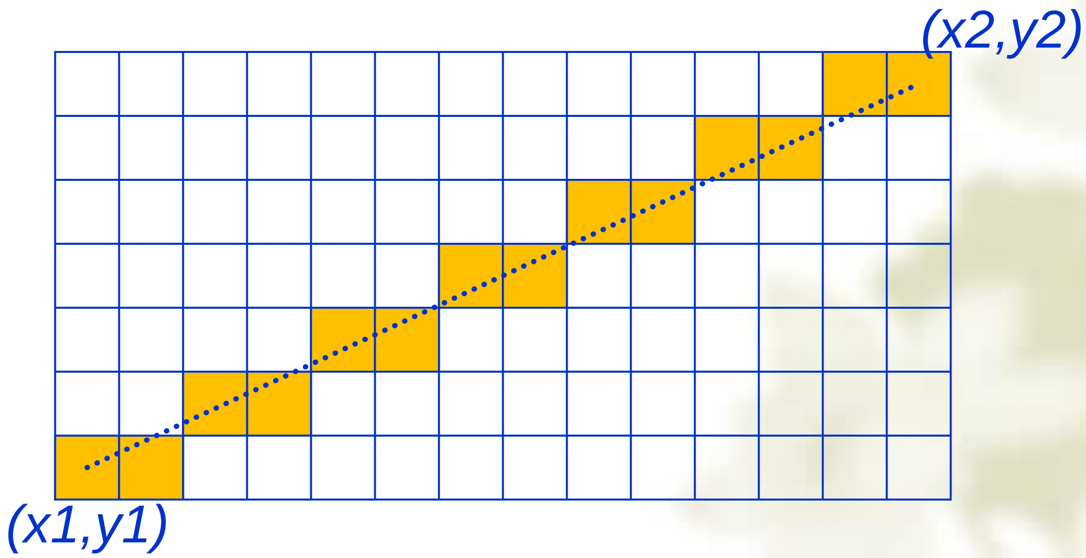


2. Rasterization: Scan Conversion

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

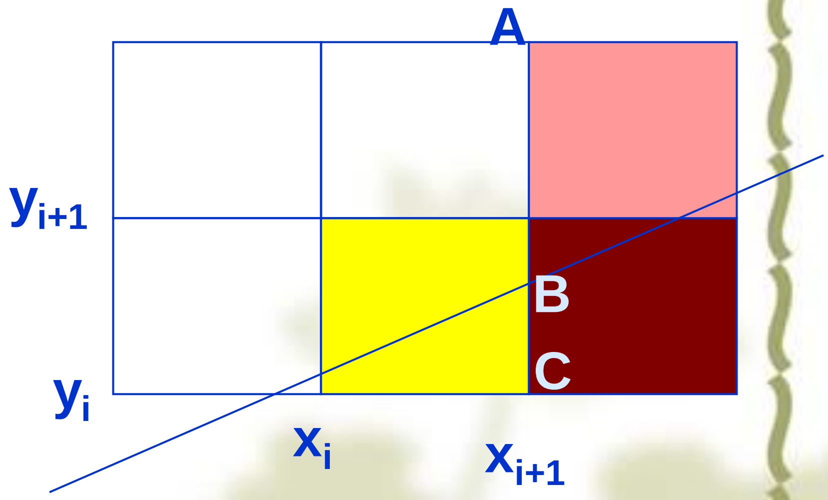
Drawing edges

- Let ($x1, y1$) and ($x2, y2$) be the screen coordinates of the two end points of a line segment(edge), which $x1 \leq x2$
- Step through the x-coordinate from $x1$ to $x2$. Set the pixels closest to the line.



2.1 Scan Conversion for Line -- DDA Algorithm

DDA Algorithm (Digital Differential Analyzer)



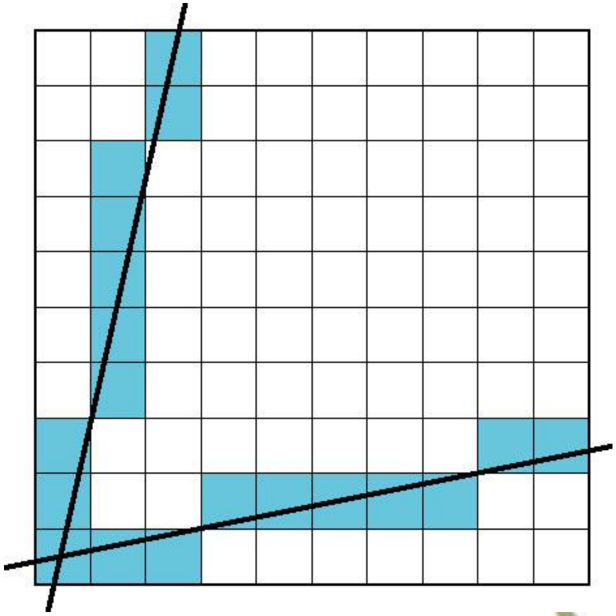
Line equation: $y = m*x+b$

$$m = dy / dx = (y_{i+1} - y_i) / (x_{i+1} - x_i)$$

$$y_{i+1} = y_i + m * (x_{i+1} - x_i) \qquad \text{-----(1)}$$

Get the next pixel (x_{i+1}, y_{i+1}) from the former pixel (x_i, y_i)

here: $|x_{i+1} - x_i| \leq 1$, $|y_{i+1} - y_i| \leq 1$



The vertices of the line segment are (x_1,y_1) and (x_2,y_2) $x_1 \neq x_2$

while, **$|m| \leq 1$** (more increment on x axis than that on y axis)

if $x_1 < x_2$, then $x_{i+1} = x_i + 1$, $y_{i+1} = y_i + m$

if $x_1 > x_2$, then $x_{i+1} = x_i - 1$, $y_{i+1} = y_i - m$

while, **$|m| > 1$** (more increment on y axis than that on x axis)

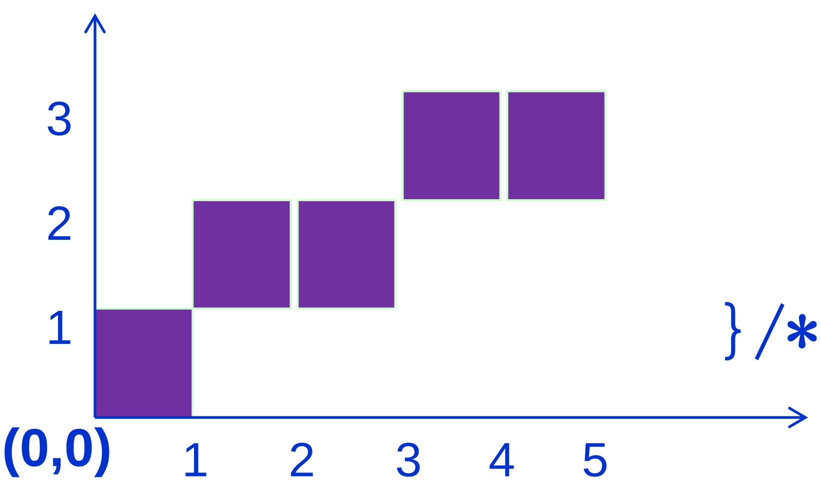
if $y_1 < y_2$, then $y_{i+1} = y_i + 1$, $x_{i+1} = x_i + 1/m$

if $y_1 > y_2$, then $y_{i+1} = y_i - 1$, $x_{i+1} = x_i - 1/m$

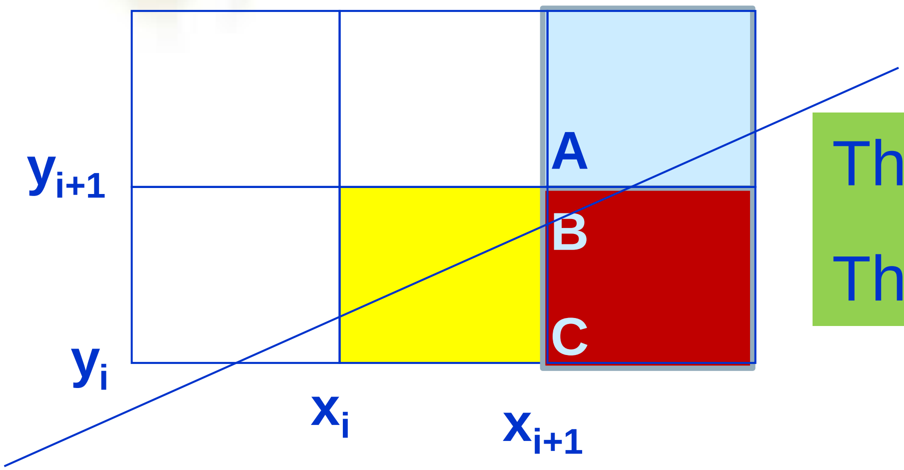
```
line(int x1, int y1, int x2, int y2)
  {     int length, i;
        float  dx,  dy,  x,  y;
        length = abs(x2-x1);
        if(abs(y2-y1) > length)   //m>1
            length = abs(y2 - y1);
        dx =(float)(x2 - x1)/length;
        dy =(float)(y2 - y1)/length;
         x = x1 +0.5*sign(dx);
         y = y1 +0.5*sign(dy);
        for(i=1; i <= length; i++) {
           setpixel((int)x,(int)y);
           x = x + dx;
           y = y + dy;
        }/* for */
  }/* line */
```

e.g. (x1,y1) = (0,0)

(x2,y2) = (5,3)



2.2 Scan Conversion for Line -- Bresenham Algorithm



The current pixel is yellow.

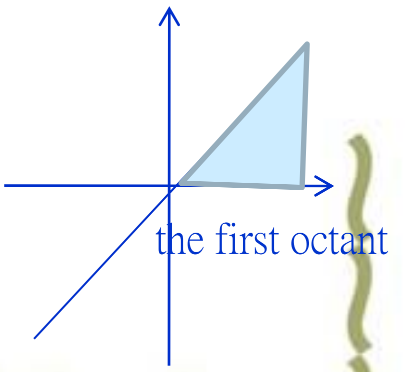
The next pixel is red or cyan?

To decide $y_{i+1}= y_i+1$ or $y_{i+1}= y_i$

according to the distance of AB and the distance of BC

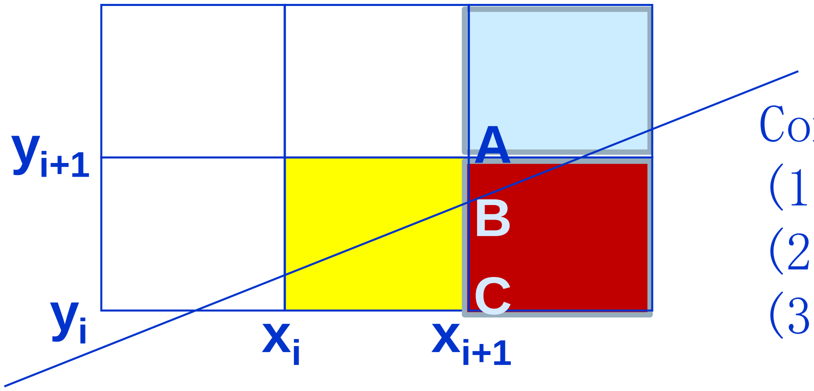
Now, we discuss the first octant: $0<|m|\le 1$, and $x_2>x_1$

The distances of d_1 and d_2 separately are



$$d_1 = y - y_i = - y_i + m*(x_i + 1) + b$$

$$\begin{aligned} d_2 &= (y_i + 1) - y \\ &= (y_i + 1) - (m*(x_i + 1) + b) \\ &= (y_i + 1) - m*(x_i + 1) - b \end{aligned}$$



d1 = BC

d2 = AB

Compare the distances of d_1 and d_2:

(1). if $d_1 - d_2 > 0$, we get (x_i+1, y_i+1) or cyan

(2). if $d_1 - d_2 < 0$, we get (x_i+1, y_i) or red

(3). if $d_1 - d_2 = 0$, we get (x_i+1, y_i), any

We have $d_1 - d_2 = 2*m*(x_i + 1) - 2*y_i + 2*b - 1$

Now, we concern the sign of $d_1 - d_2$

set $\Delta x = x2-x1 > 0$

So, $p_i=\Delta x*(d_1-d_2)=\Delta x(2*(\Delta y/\Delta x)*(x_i+1)-2*y_i+2b-1)$

$=2*\Delta y*x_i-2*\Delta x*y_i+\Delta x*(2b-1)+2*\Delta y$ ------(1)

And,

$p_{i+1}=2*(\Delta y*x_{i+1}-\Delta x*y_{i+1})+\Delta x*(2b-1)+2*\Delta y$

$p_{i+1} - p_i=2*(\Delta y(x_{i+1}-x_i)-\Delta x(y_{i+1}-y_i))$

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

$\therefore p_{i+1} = p_i + 2*\Delta y - 2*\Delta x*(y_{i+1}-y_i)$ ------(2)

Initial value:

by(1), we have: $p_i=2*\Delta y*x_i-2*\Delta x*y_i+\Delta x*(2b-1)+2*\Delta y$

$p_1=2*\Delta y*x_1-2*\Delta x*y_1+\Delta x*(2b-1)+2*\Delta y$

as well, $\because\ y_1 = \Delta y/\Delta x * x_1 + b$

$\therefore p_1=2*\Delta y*x_1-2*\Delta y*x_1-2*\Delta x*b+\Delta x*(2b-1)+2*\Delta y$

$=2*\Delta y-\Delta x$

by(1), we know, the sign of p_i is the same of d_1-d_2

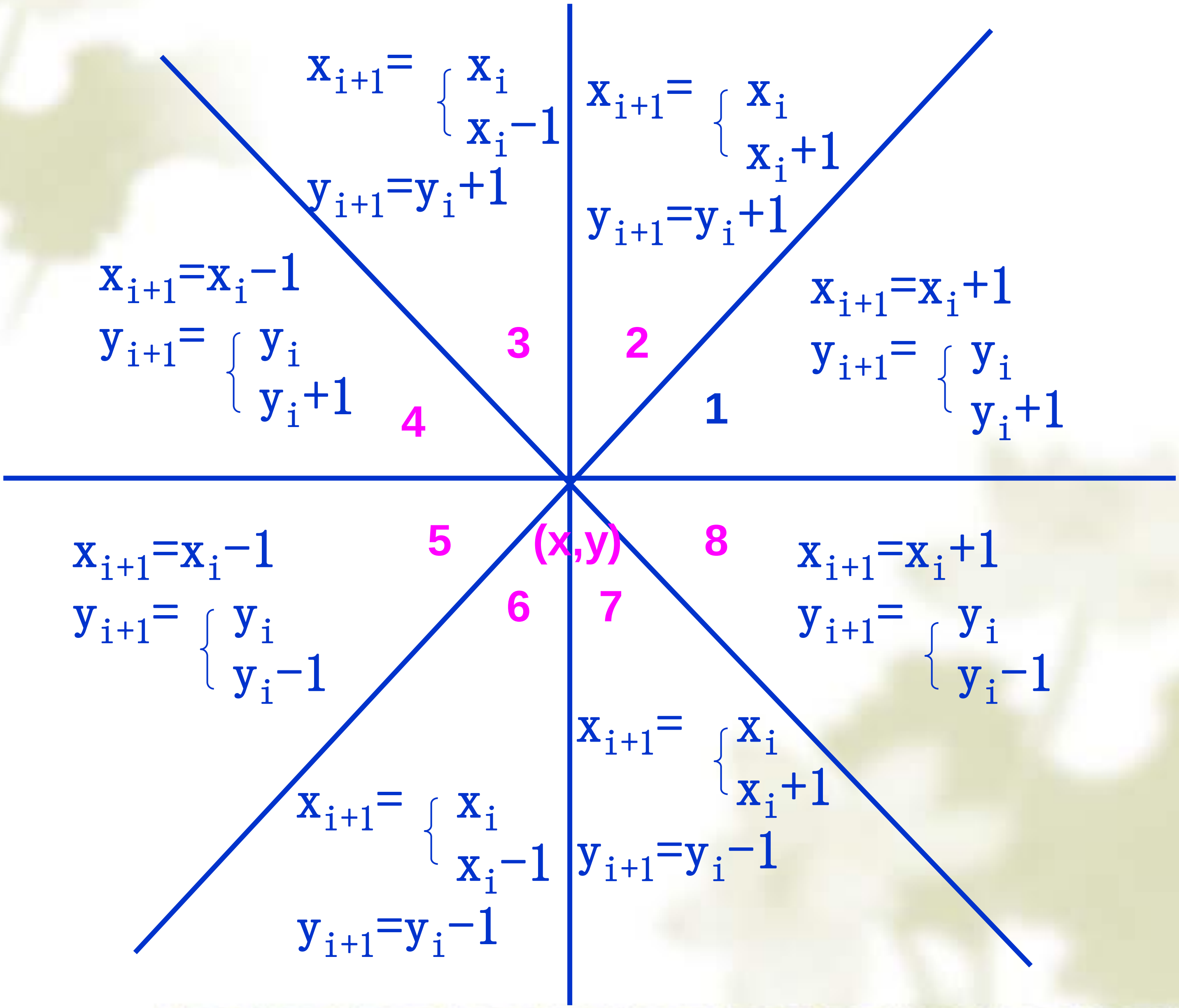
if $p_i>0$, or $d_1-d_2>0$, $y_{i+1}=y_i+1$ it is cyan

if $p_i\le 0$, or $d_1-d_2\le 0$, $y_{i+1}=yi$ it is red

So, we have:

$$
\begin{cases}
p_1 = 2 * \Delta y - \Delta x \\
x_{i+1} = x_i + 1 \\
y_{i+1} = \begin{cases} y_i + 1 & \text{if } p_i>0 \\ y_i & \text{if } p_i\le 0 \end{cases} \\
p_{i+1} = \begin{cases} p_i + 2 * (\Delta y - \Delta x) & \text{if } p_i>0 \\ p_i + 2 * \Delta y & \text{if } p_i\le 0 \end{cases}
\end{cases}
$$

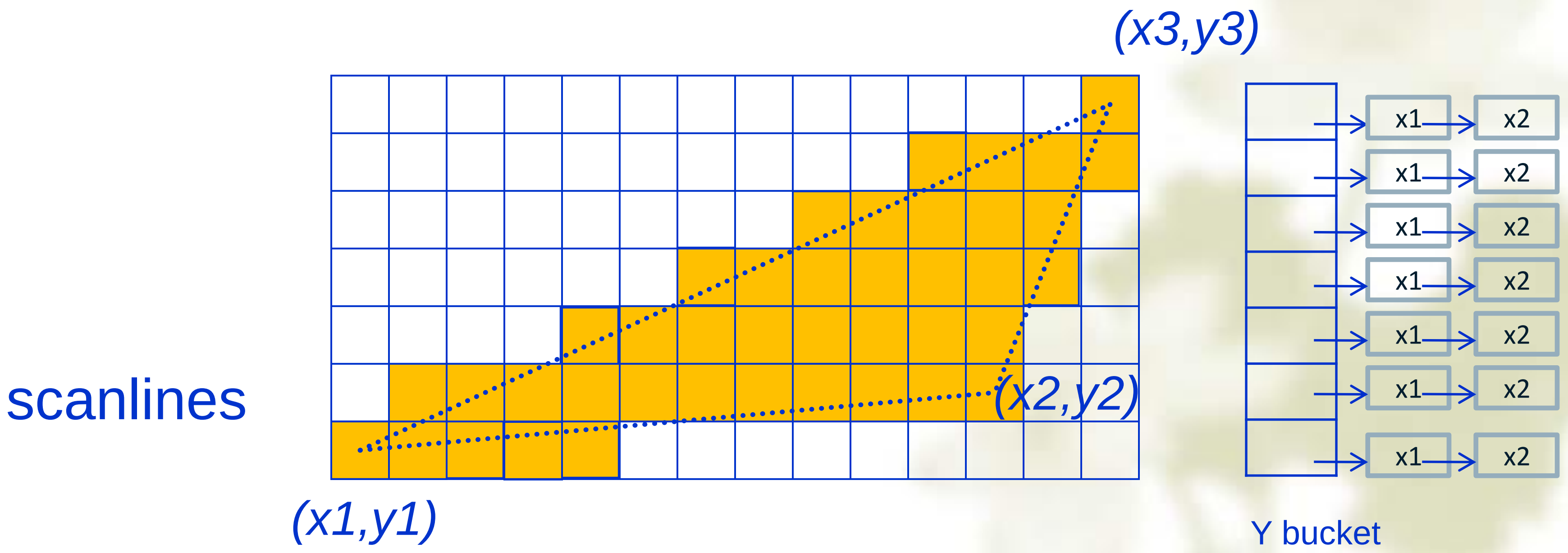
For any direction of the line with the first pixel(x,y):



```
Bresenham_line(int x1 ,int y1, int x2 , int y2)
{   int dx,dy,s1,s2, temp, interchange=0, p, i;
    float x,y;
    dx = abs(x2 - x1);    dy = abs(y2 - y1);
    s1 = sign(x2 - x1);   s2 = sign(y2 - y1);    //direction
    x = x1 + 0.5*s1;      y = y1 + 0.5*s2;
    if(dy > dx){                                 //decide m value
       temp = dx; dx = dy; dy = temp;            //dx changes fast
       interchange = 1;}                         //in 2,3,6,7 octant
    p = 2 * dy - dx;                             //initial value
    for(i=1;i<=dx; i++) {
       setpixel((int)x,(int)y);
       if(p>0){
          if(interchange)
              x = x + s1;                         /*xi as yi */
          else
              y = y + s2;
          p = p - 2 * dx;                    //pi+1=pi +2*(Δy -Δx)
       }
       if(interchange)                      //if pi<=0, yi no change
          y = y + s2;                       //yi as xi
       else
          x = x + s1;
       p = p + 2 * dy;
    }/*for*/
}/* Bresenham_line */
```

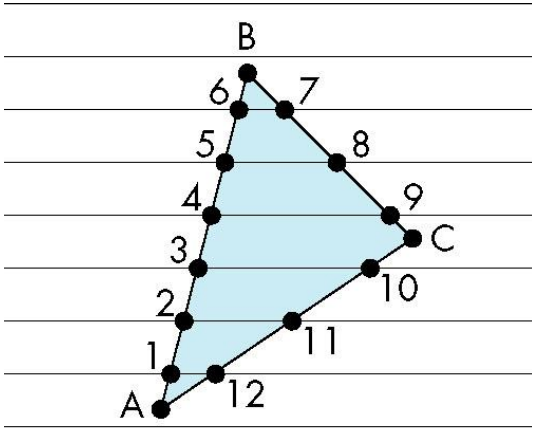
2.3 Scan Conversion for Polygon

- First find out the scanlines that intercept with the polygon. Calculate the overlapping segments
- For each scanline from top to bottom, set the pixels in the segment

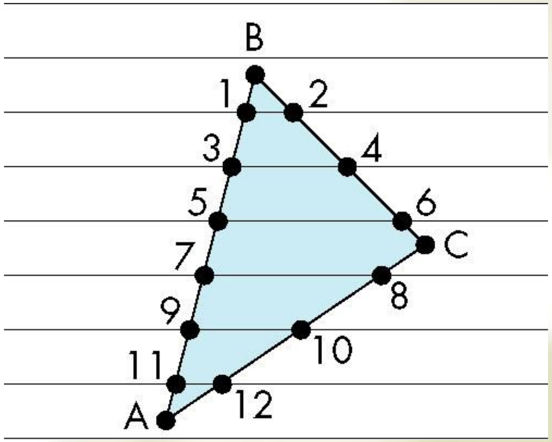


Scan Line Fill

- Can also fill by maintaining a data structure of all intersections of polygons with scan lines
  - Sort by scan line
  - Fill each span



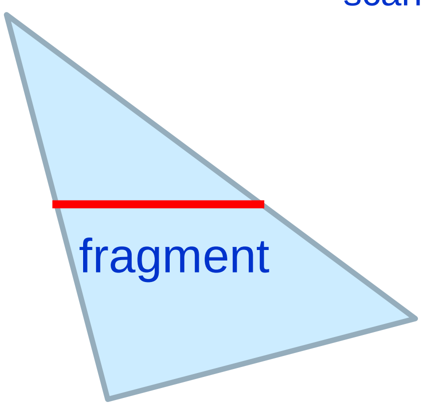
vertex order generated by vertex list



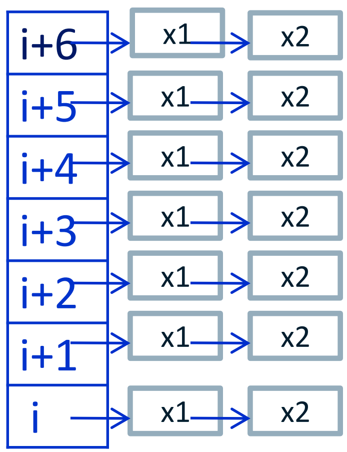
desired order

Fragments

- If there are exactly two edges that meet a scan line of pixels, we need to determine the color of all pixels between the two edge pixels on the scan line
- These pixels are called *a fragment*



scanline array



Y bucket

Two steps:

1.求交：计算扫描线(a scan-line)与多边形三个边(a polygon boundary)的交点

2.排序：把所有交点按x坐标递增顺序来排序

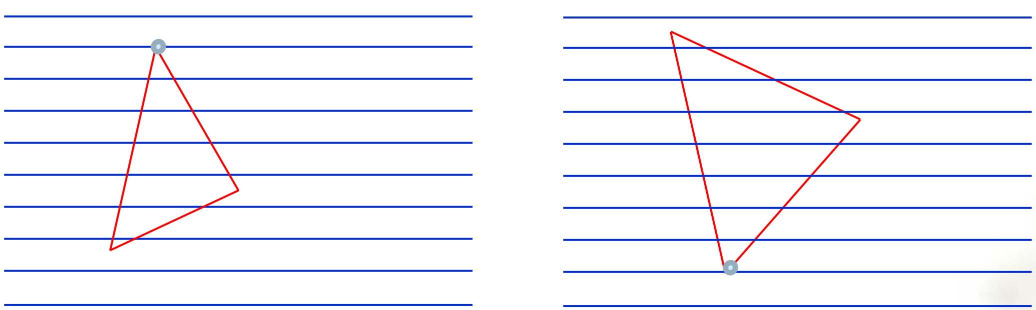
A special case:

if 扫描线与多边形的顶点相交

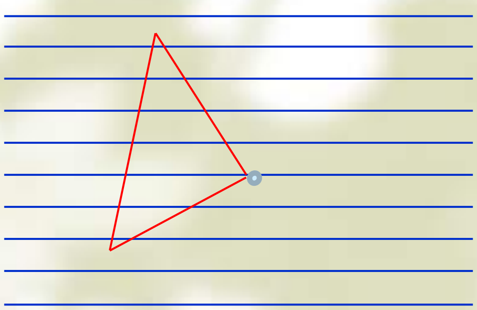
how to computing the intersection

There are two cases:

1. if 顶点是极值点，即顶点的两条相邻边位于一边
   then 交点算二个
2. if 顶点的两条相邻边分别位于扫描线的两边
   then 交点算一个



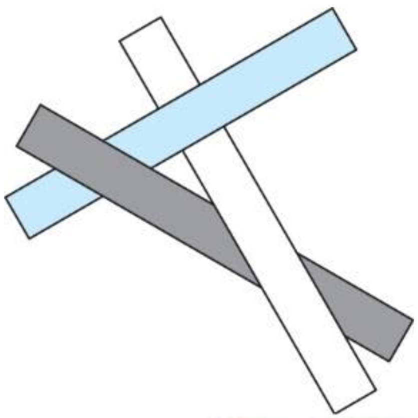
case 1

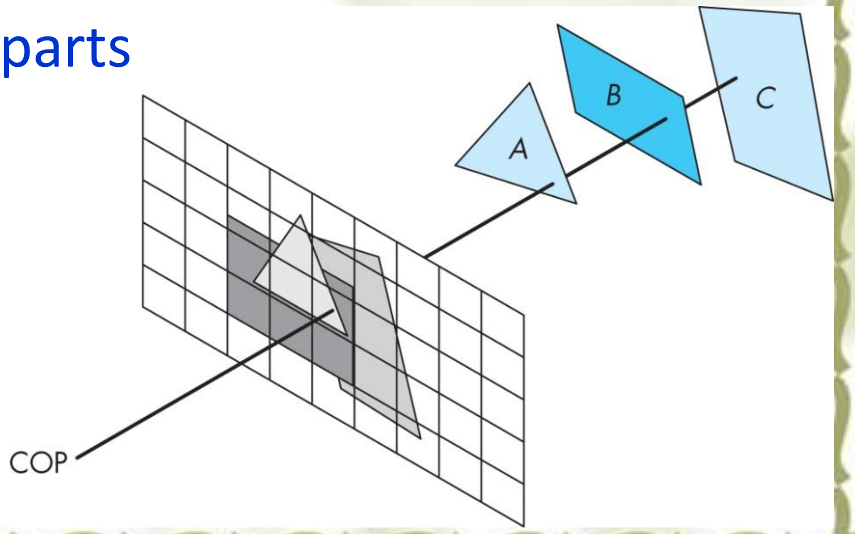


case 2

3. Hidden-Surface Removal

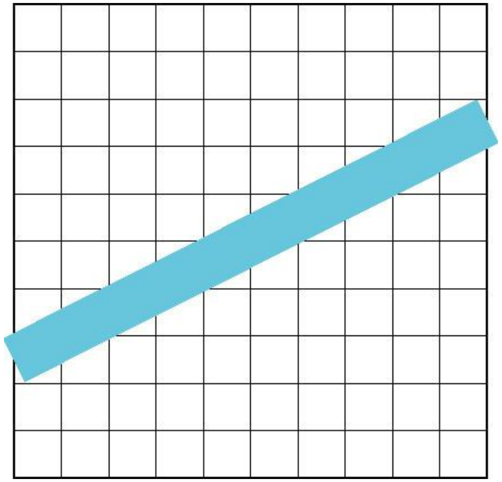
- The z-Buffer Algorithm
  - Depth Buffer same as Frame Buffer
  - saving the distance of the closest intersection
- The Painter's Algorithm
  - Rendering from back to front of the polygons
  - depth sort for all polygons
  - dividing the polygon into two parts

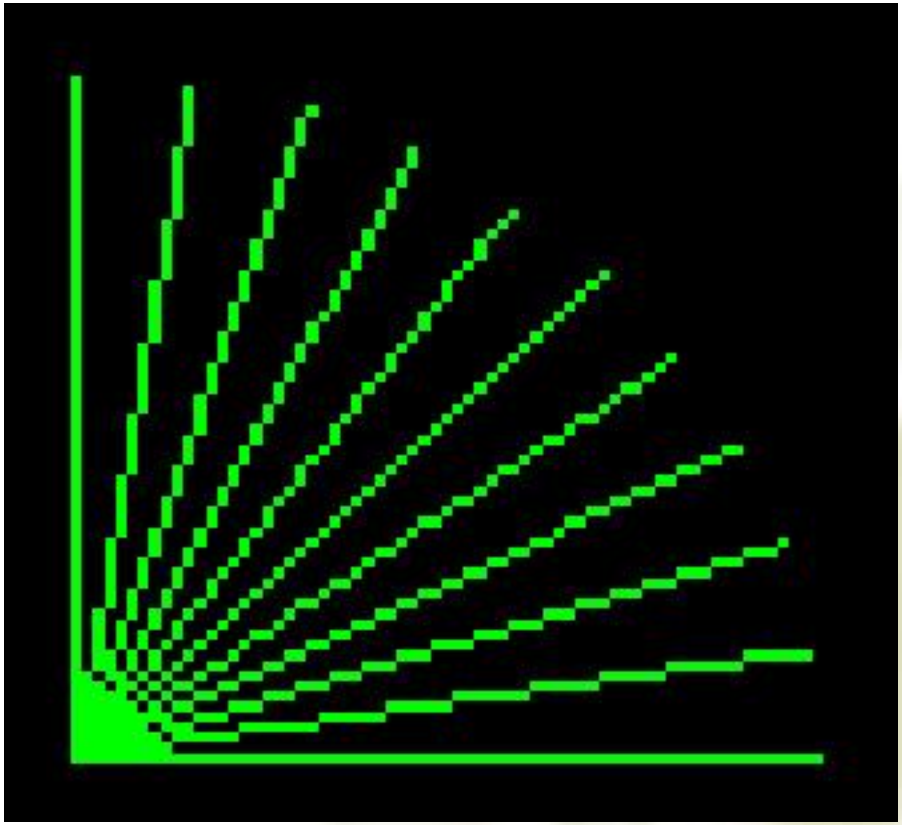




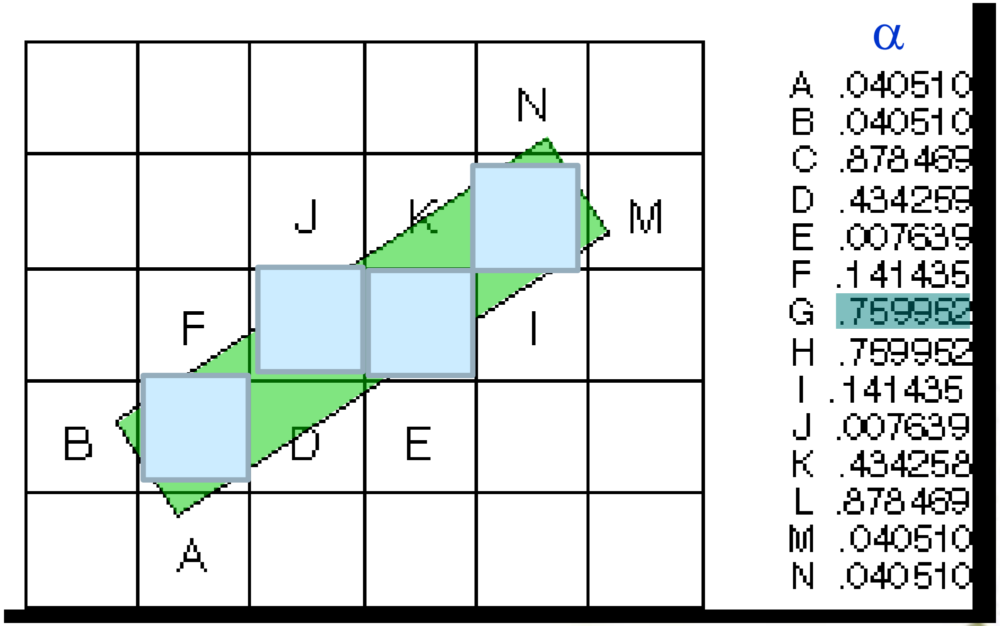
4. Antianliasing

- The ***jaggies*** appeared in the drawings of lines is called *aliasing*. It is due to the approximation of a continuous line with discrete pixels.





Antialiasing computing



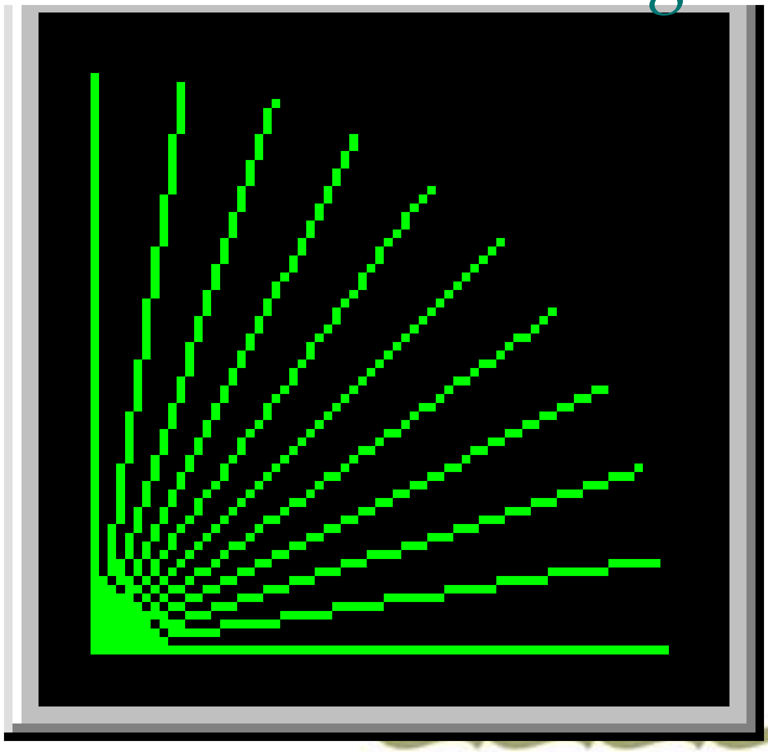
- Conceive that a line is 1 pixel wide which covers certain pixel squares.
- A simple drawing method sets the pixels in C, G, H and L to the current color.
- An *antialiasing* method first computes the overlapping proportions of the lines and the nearby pixel squares. Suppose that the overlapping proportion of a certain pixel is α, the new *pixel_color* is then set to

$$\alpha \times current\_color + (1- \alpha) \times existing\_color$$

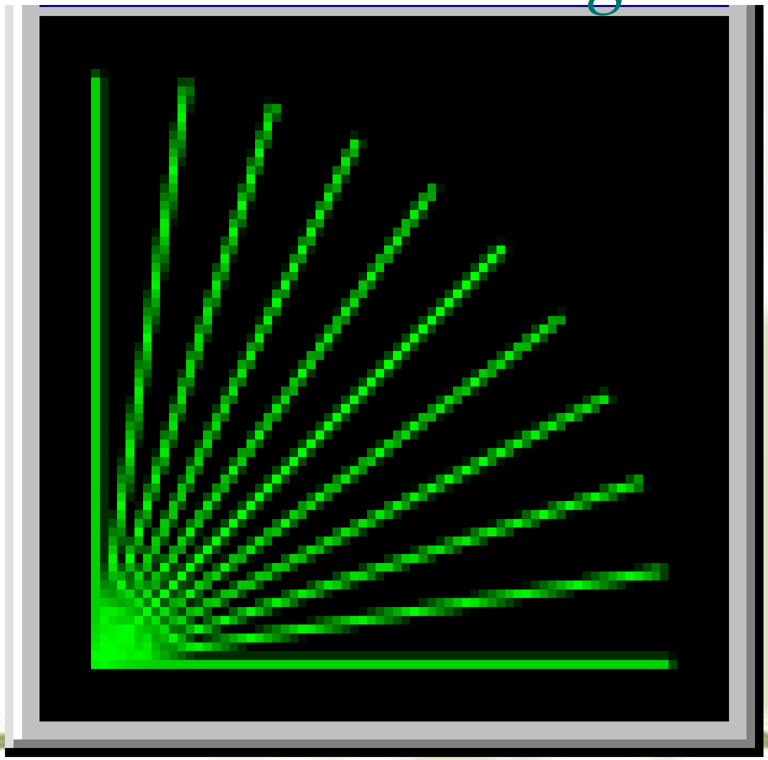
Line Antianliasing

- To turn on anitaliasing (draw slower)
  glEnable(GL_LINE_SMOOTH)
- To turn off antialiasing (draw faster)
  glDisable(GL_LINE_SMOOTH)

Without antialiasing



With antialiasing



Polygon Antianliasing

Jaggies may occur along edges and can be smoothen in the same way as in the drawing of lines

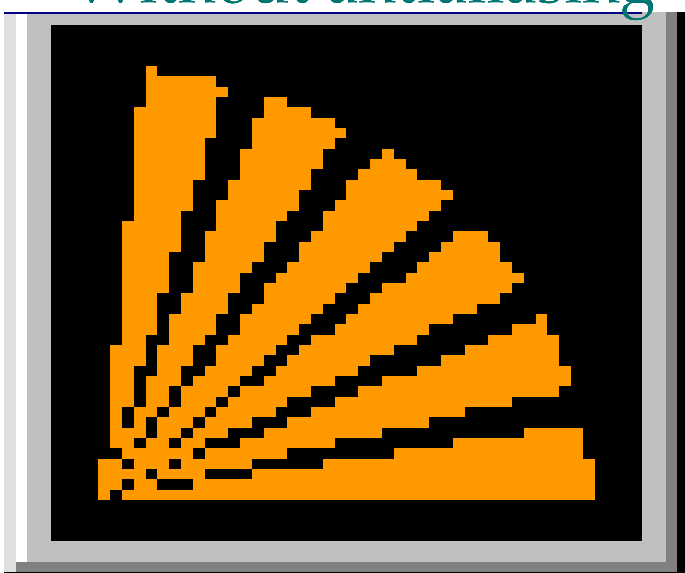
- To turn on anitaliasing (draw much slower)

  glEnable(GL_POLYGON_SMOOTH)

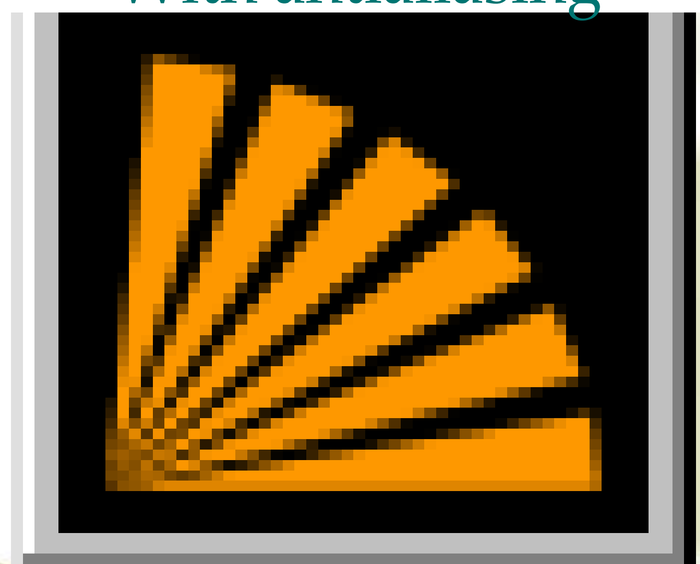
- To turn off antialiasing (draw much faster)

  glDisable(GL_POLYGON_SMOOTH)

Without antialiasing

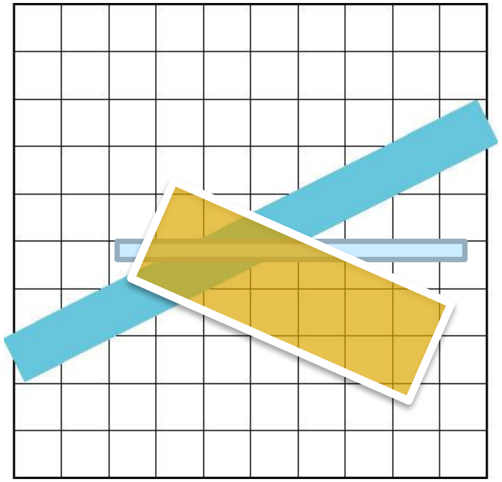


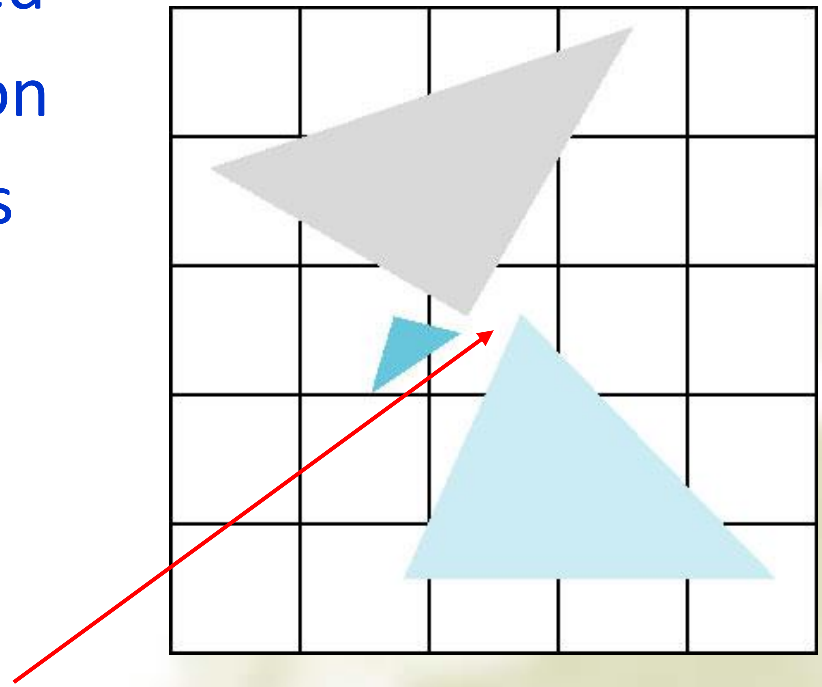
With antialiasing



Polygon Aliasing

- Aliasing problems can be serious for polygons
  - Small polygons neglected
  - Color of pixel depends on colors of multi-polygons



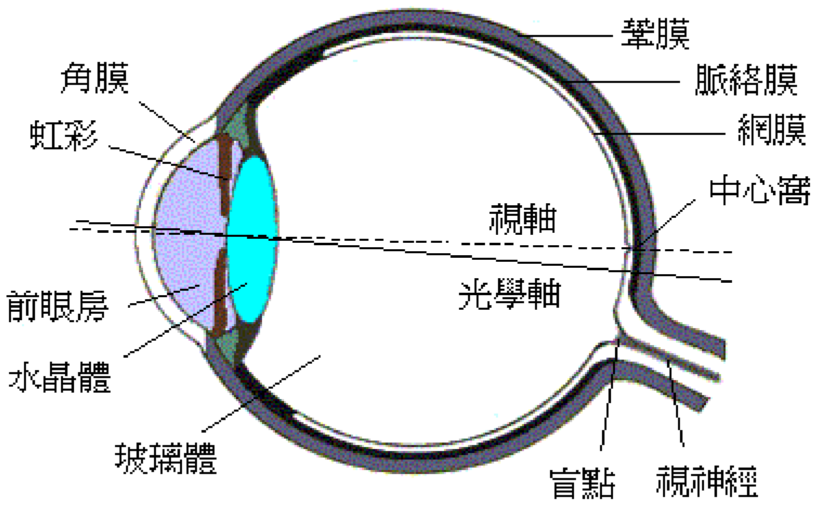


All three polygons should contribute to color

5. Color Model

--The Human Visual System

- Our eyes have two main kinds of cells involved in vision
  - Rods -- sense luminance or brightness
  - Cones -- sense chroma or color



Three kinds of photosensitive chemicals in cones
Generally these are sensitive to red, green, and blue light wavelengths

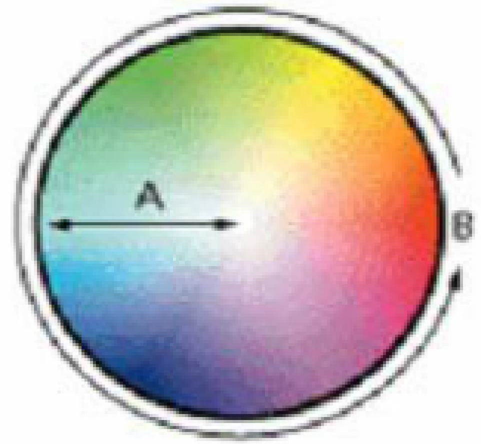
从人的主观感觉角度，颜色包含三个要素：

(1) 色调（hue）：色调反映颜色的类别，如红色、绿色、蓝色等。色调大致对应光谱分布中的主波长。

or:色度chroma

(2) 饱和度（Saturation）

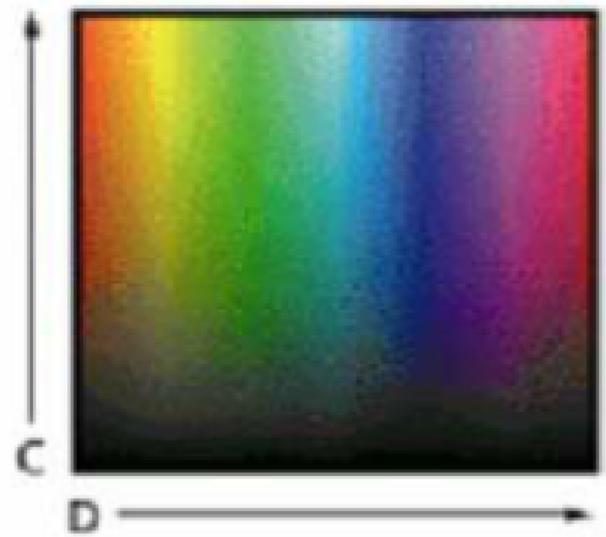
饱和度是指彩色光所呈现颜色的深浅或纯洁程度。对于同一色调的彩色光，其饱和度越高，颜色就越深，或越纯；而饱和度越小，颜色就越浅，或纯度越低。高饱和度的彩色光可因掺入白光而降低纯度或变浅，变成低饱和度的色光。100%饱和度的色光就代表完全没有混入白光的纯色光。



(3) 明亮度（luminance）

明亮度是光作用于人眼时引起的明亮程度的感觉。一般来说，彩色光能量大则显得亮，反之则暗。

大量试验表明，人的眼睛能分辨128种不同的色调，10－30种不同的饱和度，而对亮度非常敏感。人眼大约可以分辨35万种颜色。



Color Model

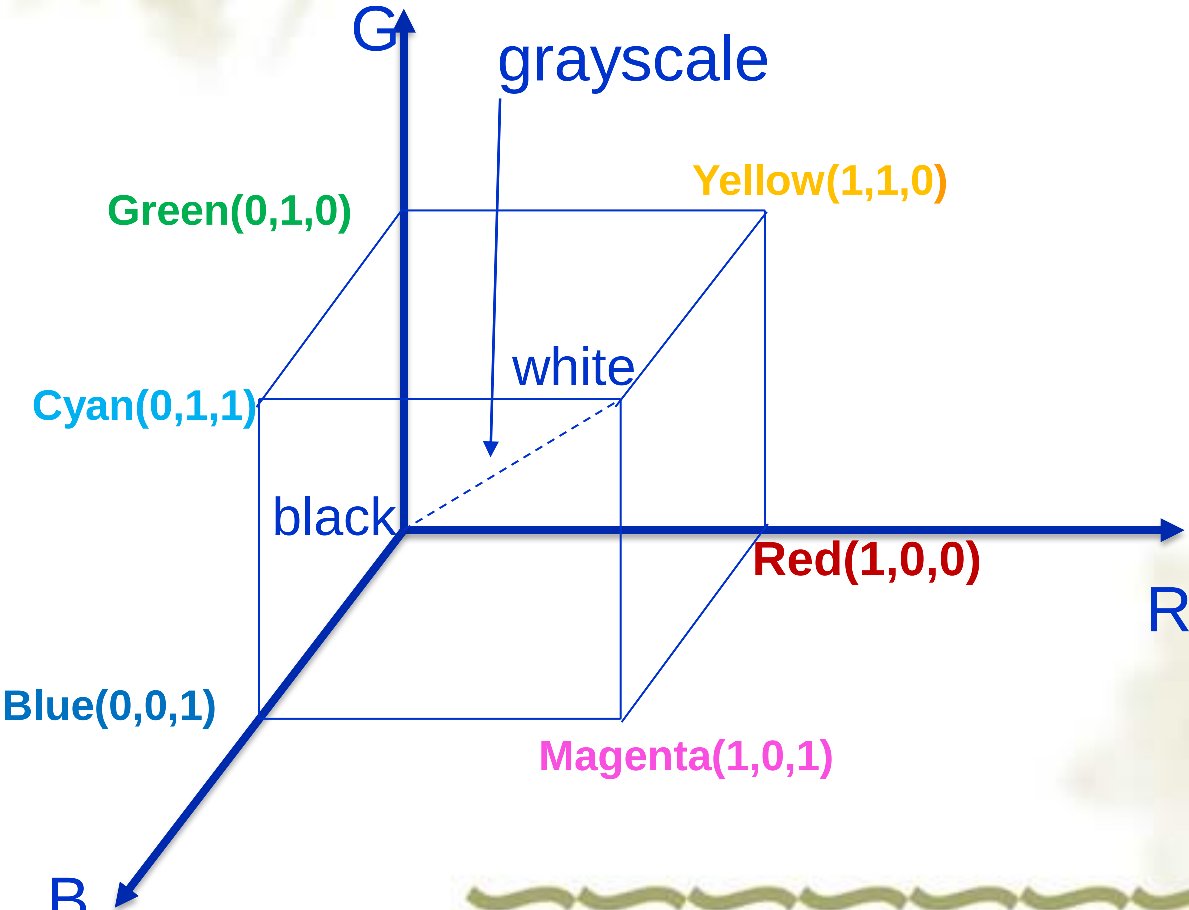
Color model:

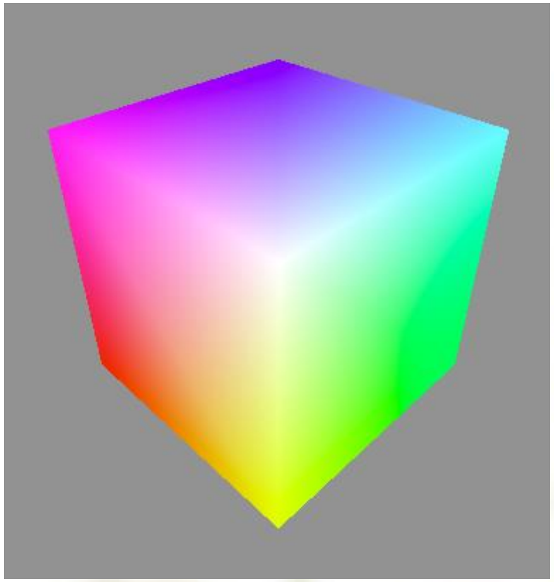
a geometric presentation of the color space with real-valued numbers in [0,1]

- RGB – cube for screen
- CMYK – cube for printer
- HSV – cone for artist
- HLS – double cone for artist on painting

The RGB and CMY Color Model

Colors are specified by naming their red, green, and blue components





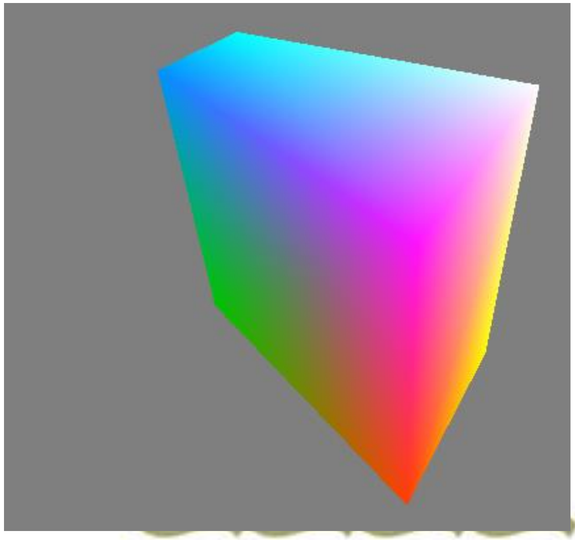
pure primaries at the vertices

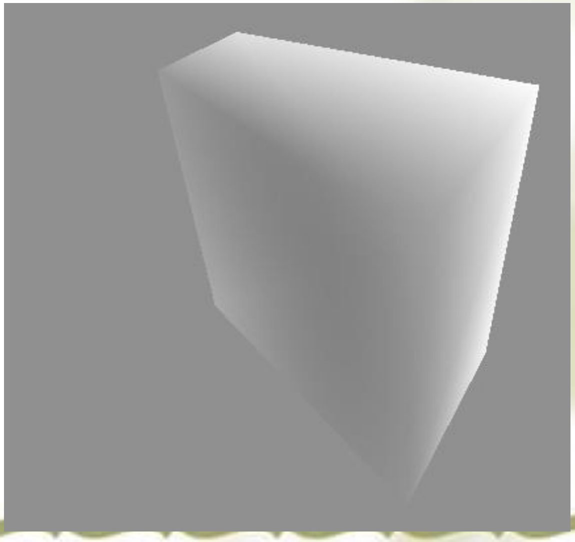
Color and Luminance

- It can be important to think of the luminance of a color as well as the chroma (RGB values) of the color
- Luminance can be approximated by 0.30*red + 0.59*green + 0.11*blue
- If a viewer has color deficiencies, he can usually distinguish color by luminance

Luminance Example

- The RGB cube clipped by a plane $0.3r+0.59g+0.11b+t=0$, shown in color and grayscale
- The grayscale conversion is computer-based, not visually-based; is it right?





Other Color Models: HSV

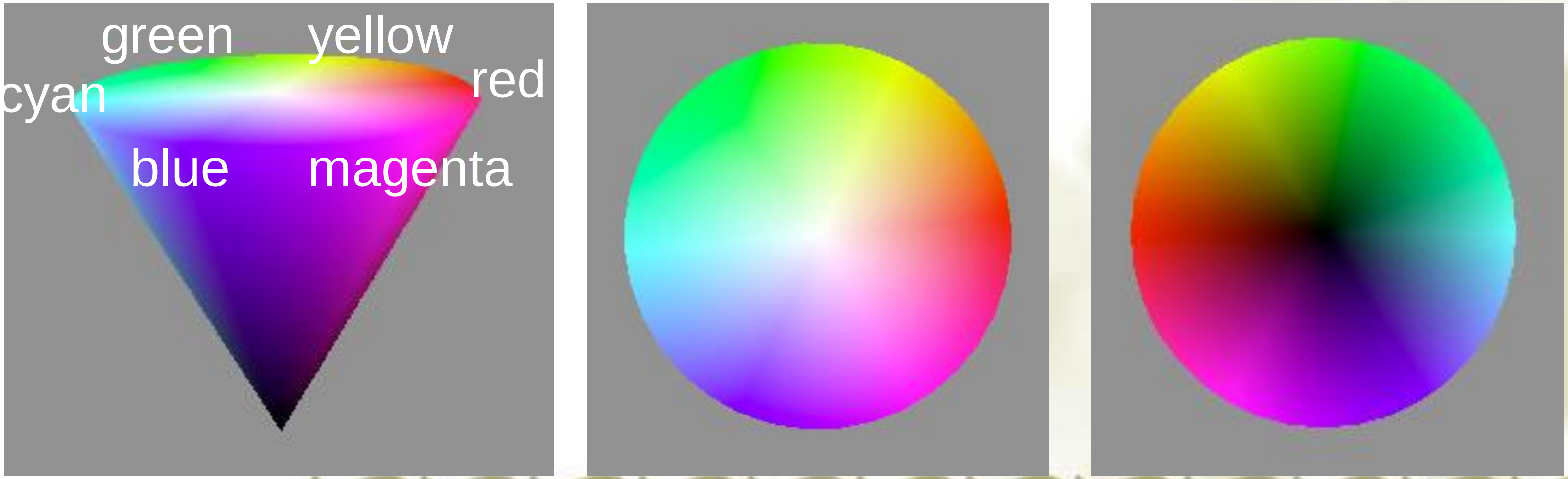
- Hue - Saturation - Value : cone model

  Hue = angle

  Saturation = radius

  Value = height

- Views from side, top, and bottom



Other Color Models: HLS

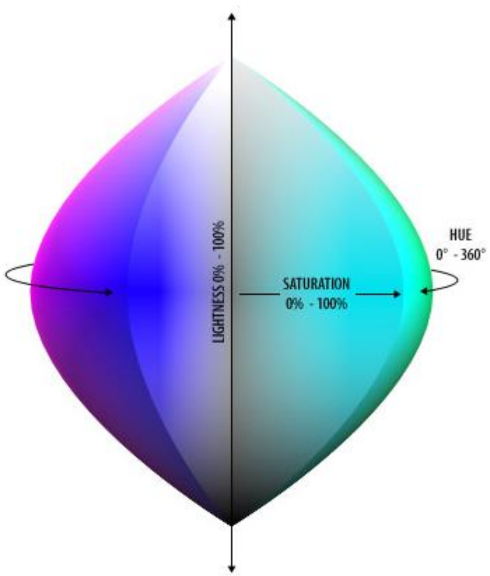
- Hue - Lightness – Saturation: dual cone model

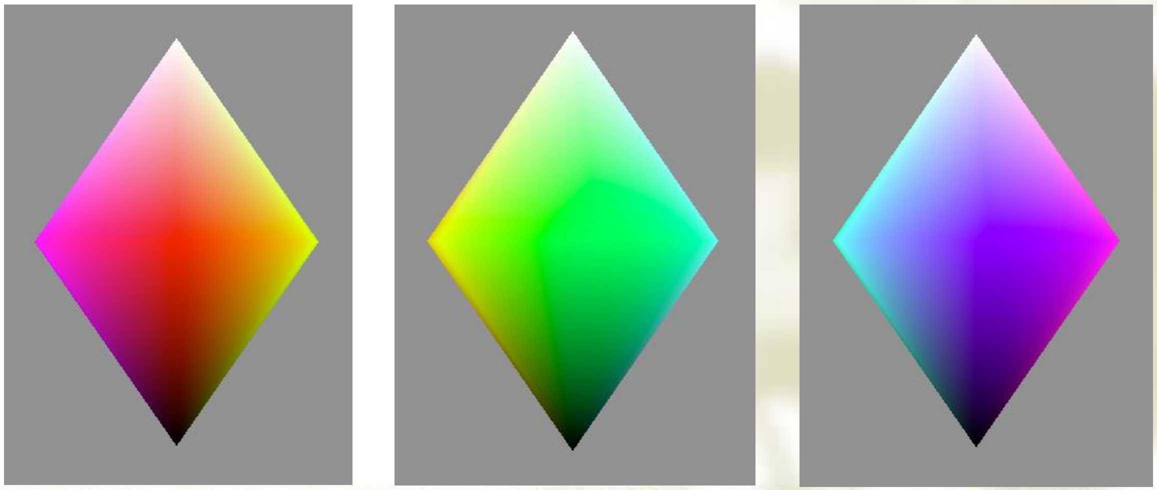
  Hue = angle

  Lightness = height

  Saturation = radius

- Views from red, green, and blue sides





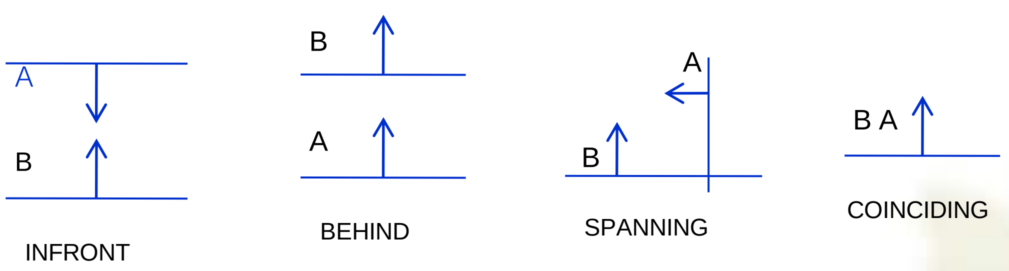
Emissive vs Transmissive Colors

- Emissive colors are from screen
- Transmissive colors are from inks
- RGB emissive model (left) -additive colors
- CMYK transmissive model (right) -subtractive colors

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

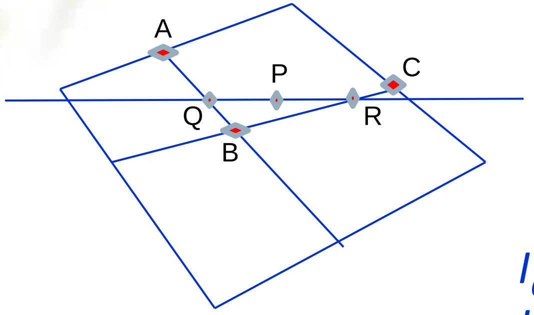
作业8

1. 如何判断空间上两个三角形的位置关系？位置关系包括：三角形A在三角形B的前方；三角形A在三角形B的后方；两个三角形相交且有交线或交点；两个三角形共面。4种情况



作业8

2. How to compute the interpolation illumination Ip2 by previous illumination Ip1 and increment.



$I_Q = (1-u) \times I_A + u \times I_B$ $0 \le u \le 1$, u = AQ/AB
$I_R = (1-w) \times I_B + w \times I_C$ $0 \le w \le 1$, w = BR/BC
$I_P = (1-t) \times I_Q + t \times I_R$ $0 \le t \le 1$, t = QP/QR

Increment Computing:

$I_{P2} = (1-t_2) \times I_Q + t_2 \times I_R$
$I_{P1} = (1-t_1) \times I_Q + t_1 \times I_R$