

# Computer Graphics 2019

## 5. Geometry and Transform

Hongxin Zhang

State Key Lab of CAD&CG, Zhejiang University

2019-10-16

# Today outline

- Triangle rasterization
  - Basic vector algebra ~ geometry
- Antialiasing revisit
- Clipping
- Transforms (I)



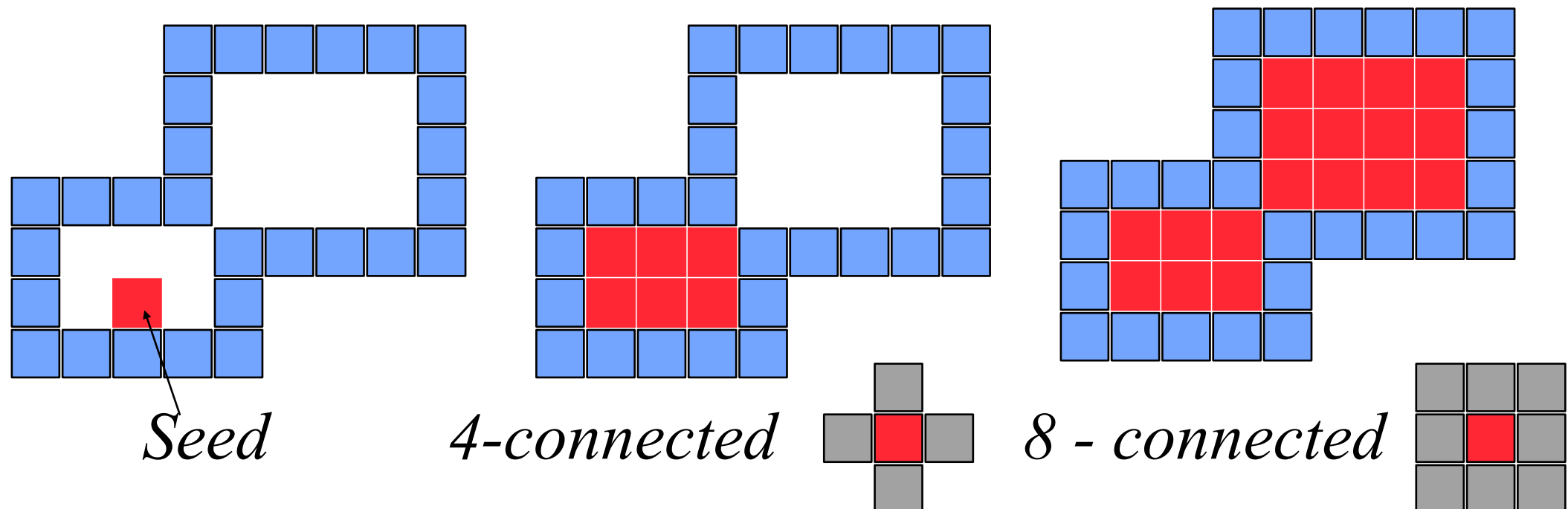
# Previous lesson

- Primitive attributes
- Rasterization and scan line algorithm
  - line,
  - general polygon



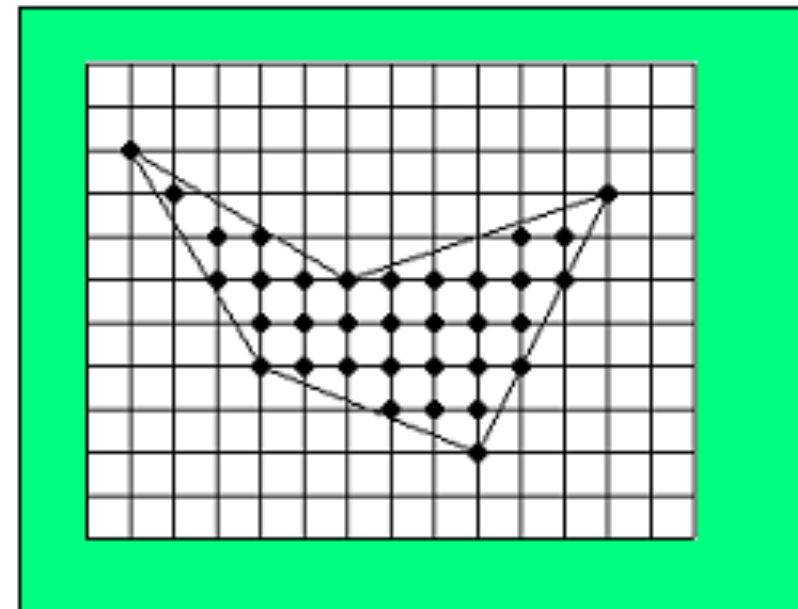
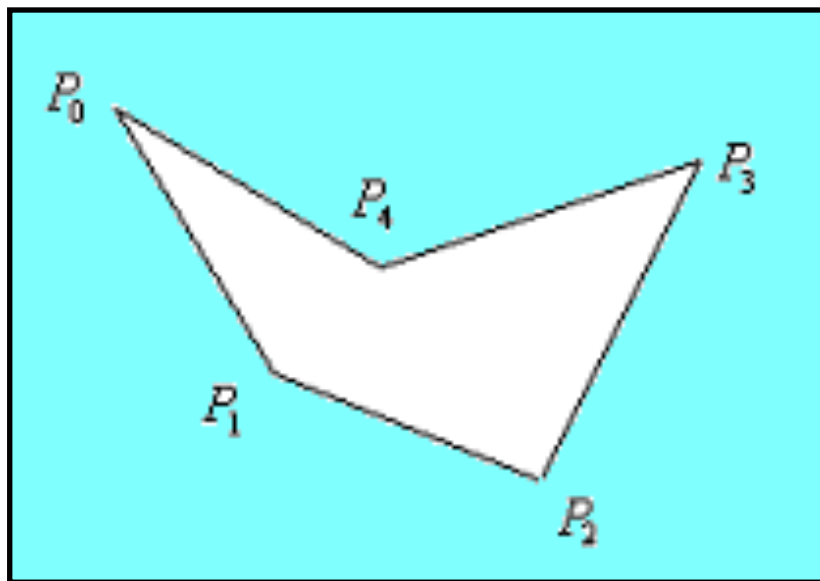
# Seed Fill Algorithms

- Assumes that at least one pixel interior to the polygon is known
- It is a recursive algorithm
- Useful in interactive paint packages



# Polygon filling

- Polygon representation



By vertex

By lattice

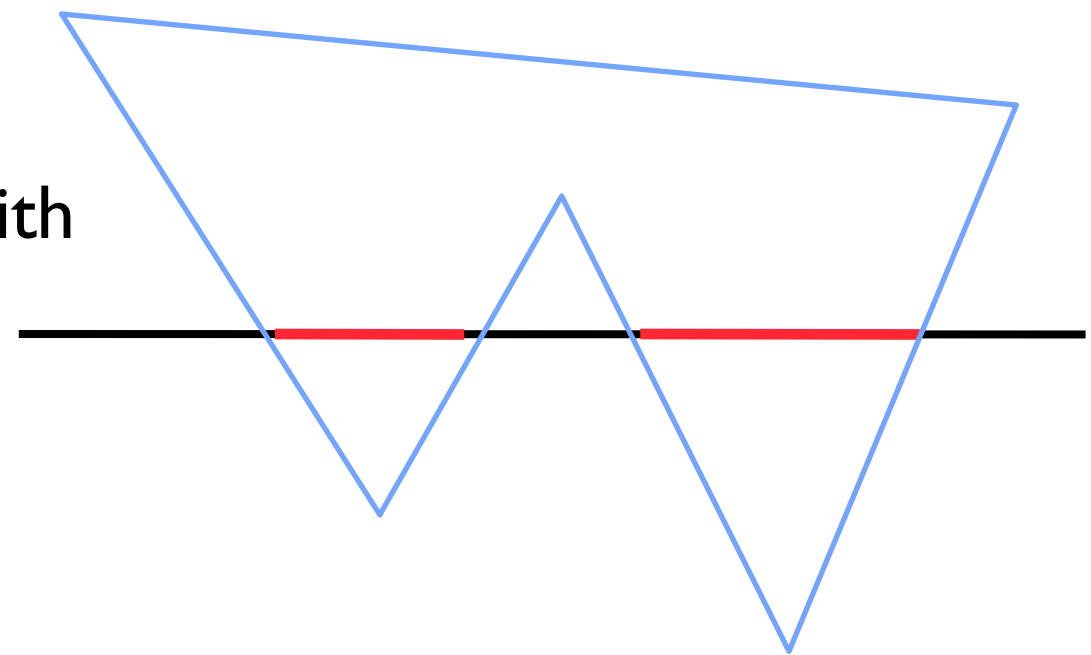
- Polygon filling:

- vertex representation vs lattice representation

# Scan Line Method

- Proceeding from left to right the intersections are paired and intervening pixels are set to the specified intensity
- Algorithm
  - Find the intersections of the scan line with all the edges in the polygon
  - Sort the intersections by increasing X-coordinates
  - Fill the pixels between pair of intersections

From top to down



<http://www.cecs.csulb.edu/~pnguyen/cecs449/lectures/fillalgorithm.pdf>

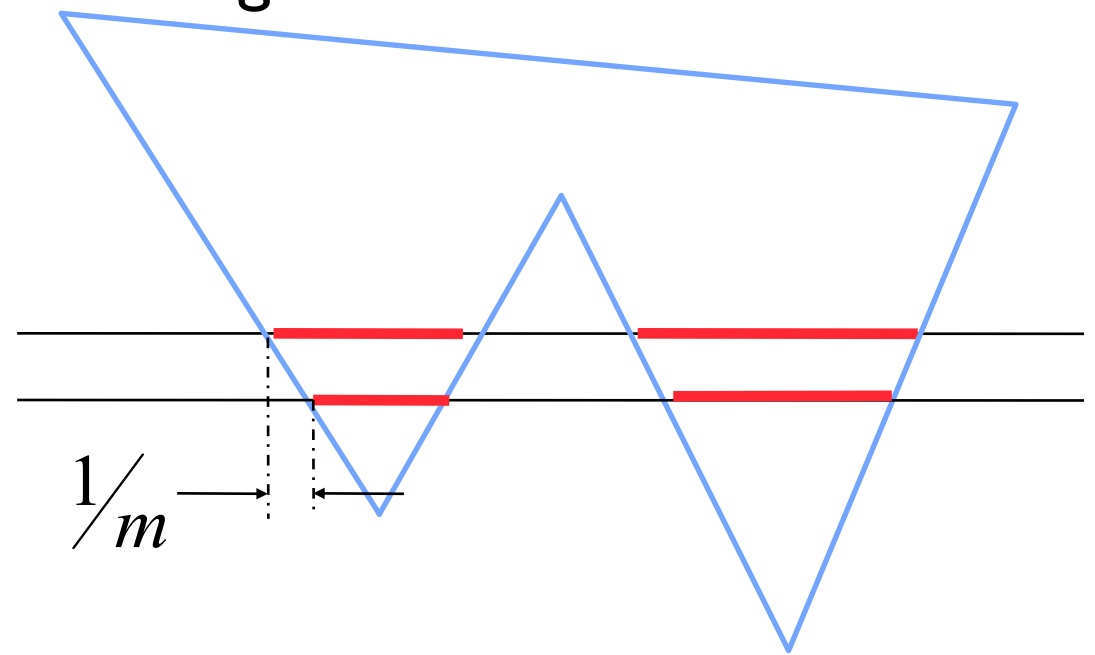


# Efficiency Issues in Scan Line Method

- Intersections could be found using edge coherence

the X-intersection value  $x_{i+1}$  of the lower scan line can be computed from the X-intersection value  $x_i$  of the preceding scan line as

$$x_{i+1} = x_i + \frac{1}{m}$$



- List of active edges could be maintained to increase efficiency

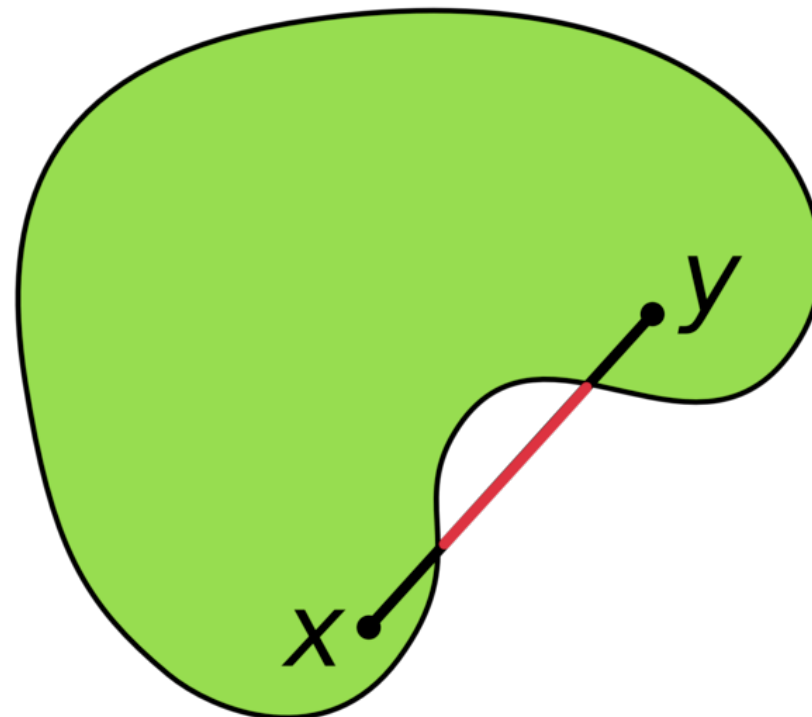
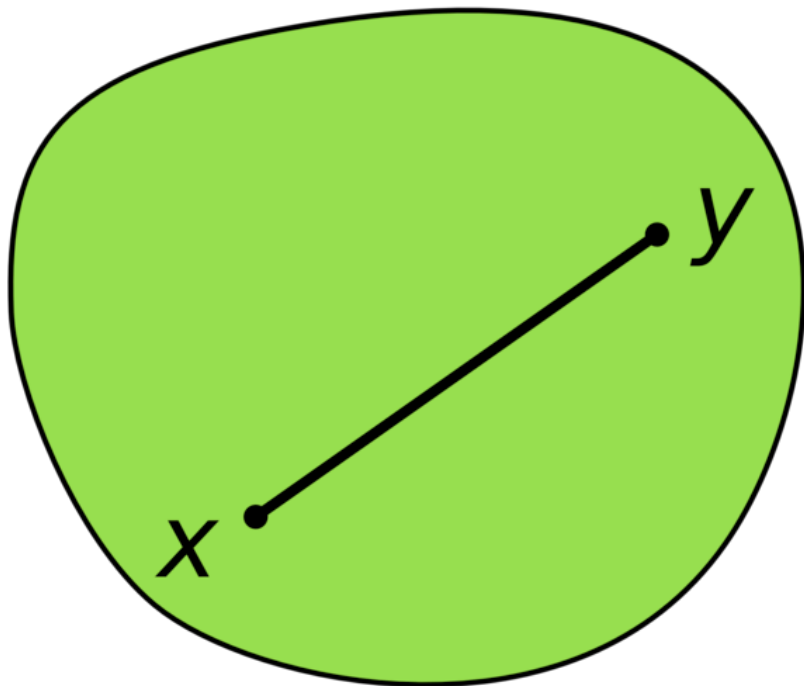
# Advantages of Scan Line method

- The algorithm is efficient
- Each pixel is visited only once
- Shading algorithms could be easily integrated with this method to obtain shaded area
- Efficiency could be further improved if polygons are **convex**,
- much better if they are **only triangles**





# Convex?

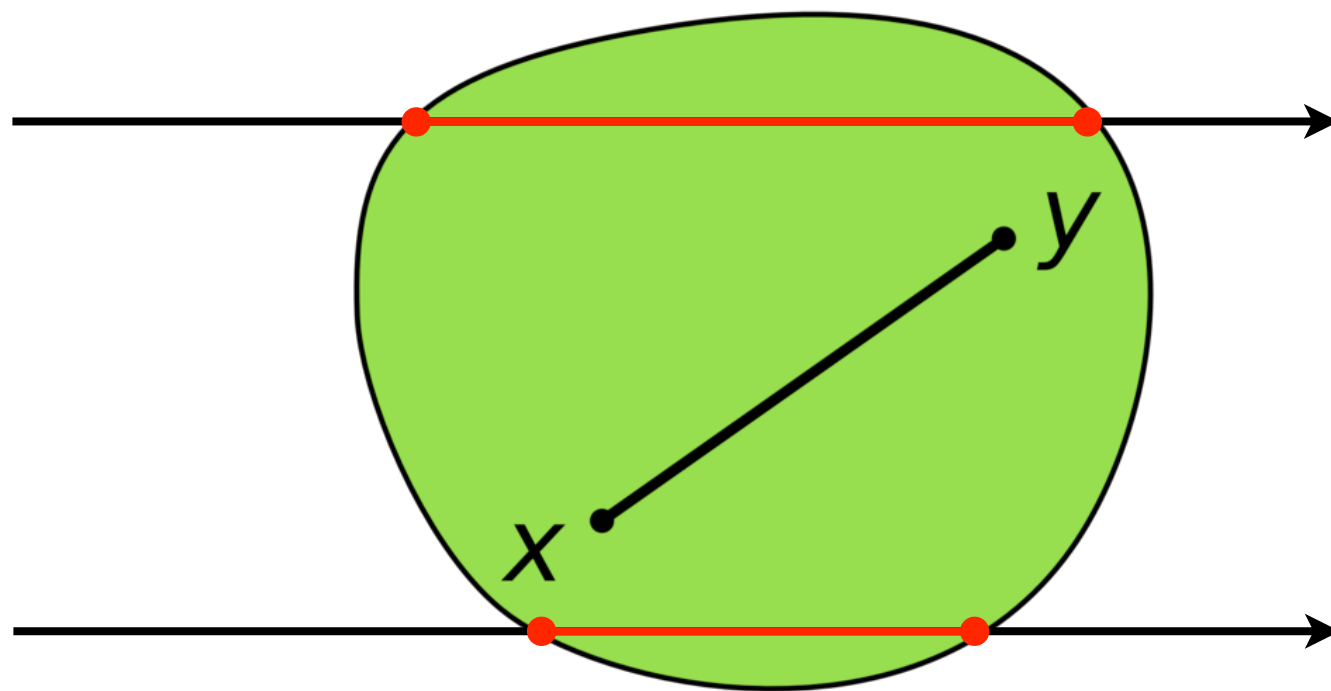


A set  $C$  in  $S$  is said to be **convex** if, for all  $x$  and  $y$  in  $C$  and all  $t$  in the interval  $[0, 1]$ , the point

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

is in  $C$ .

# Convex polygon rasterization



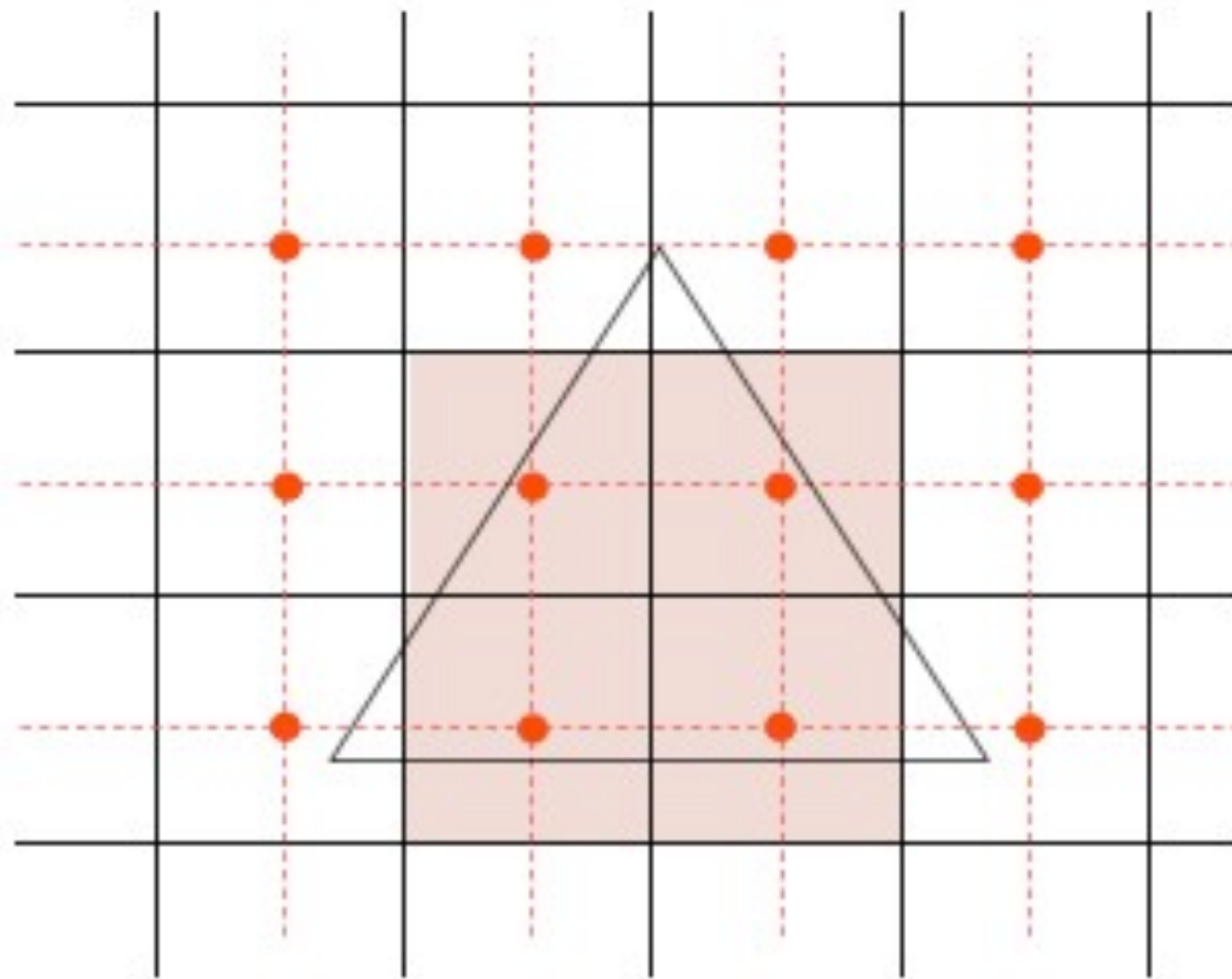
One in and one out

# Triangle Rasterization



# Triangle Rasterization?

Output fragment if pixel center is **inside** the triangle



# Triangle Rasterization

```
rasterize( vert v[3] )
```

```
{
```

```
    bbox b; bound3(v,b);
```

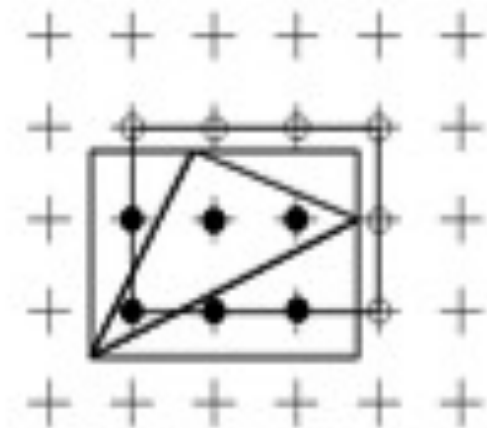
```
    for( int y=b.ymin; y<b.ymax, y++ )
```

```
        for( int x=b.xmin; x<b.xmax, x++ )
```

```
            if( inside3(v,x,y) )
```

```
                fragment(x,y);
```

```
}
```



**GPUs contain triangle rasterization hardware**  
**Can output billions of fragments per second**

# Compute Bounding Box

```
bound3( vert v[3], bbox& b )
```

```
{
```

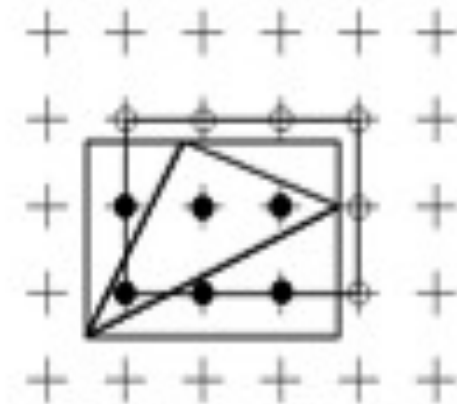
```
    b.xmin = ceil(min(v[0].x, v[1].x, v[2].x));
```

```
    b.xmax = ceil(max(v[0].x, v[1].x, v[2].x));
```

```
    b.ymin = ceil(min(v[0].y, v[1].y, v[2].y));
```

```
    b.ymax = ceil(max(v[0].y, v[1].y, v[2].y));
```

```
}
```



**Calculate tight bound around the triangle**  
**Round coordinates upward (ceil) to the nearest integer**



# Point Inside Triangle Test

```
rasterize( vert v[3] )
```

```
{
```

```
  bbox b; bound3(v, b);
```

```
  line l0, l1, l2;
```

```
  makeline(&v[0], &v[1], &l2);
```

```
  makeline(&v[1], &v[2], &l0);
```

```
  makeline(&v[2], &v[0], &l1);
```

```
  for( y=b.ymin; y<b.ymax, y++ ) {
```

```
    for( x=b.xmin; x<b.xmax, x++ ) {
```

```
      e0 = l0.A * x + l0.B * y + l0.C;
```

```
      e1 = l1.A * x + l1.B * y + l1.C;
```

```
      e2 = l2.A * x + l2.B * y + l2.C;
```

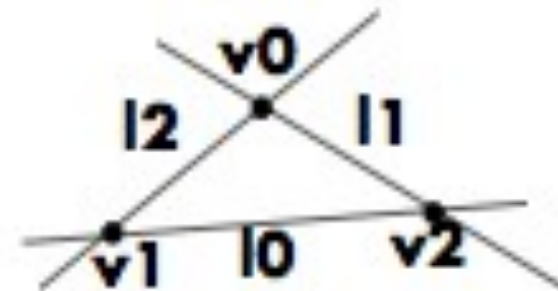
```
      if( e0<=0 && e1<=0 && e2<=0 )
```

```
        fragment(x, y);
```

```
    }
```

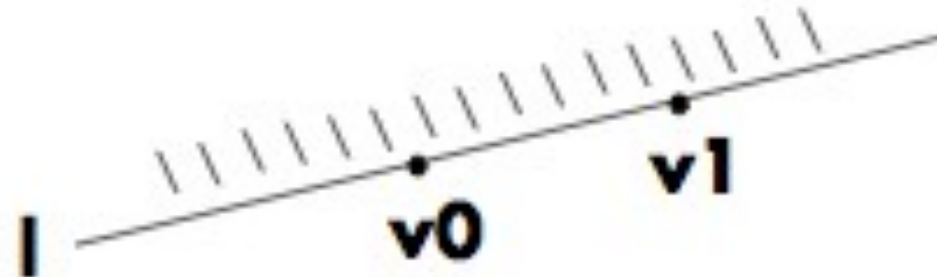
```
  }
```

```
}
```



# Line equation

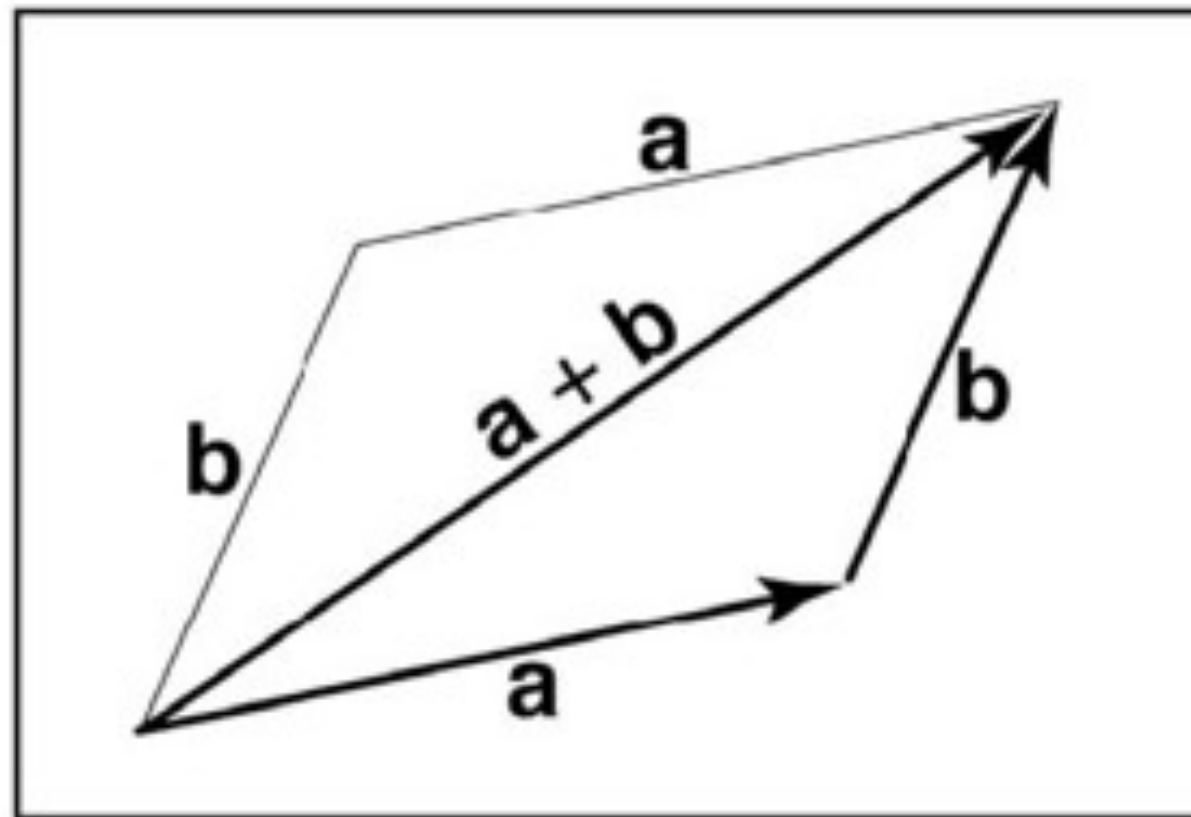
**Inside on the left for CCW polygons**



```
makeline( vert& v0, vert& v1, line& l )
{
    l.a = v1.y - v0.y;
    l.b = v0.x - v1.x;
    l.c = -(l.a * v0.x + l.b * v0.y);
}
```

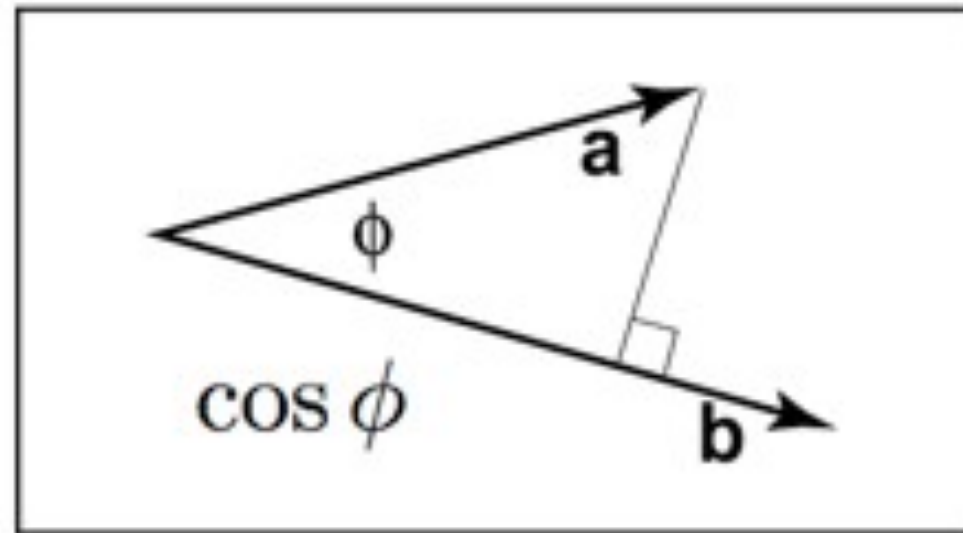


# The Parallelogram Rule



**Vector addition define for any number of dimensions**

# Dot Product

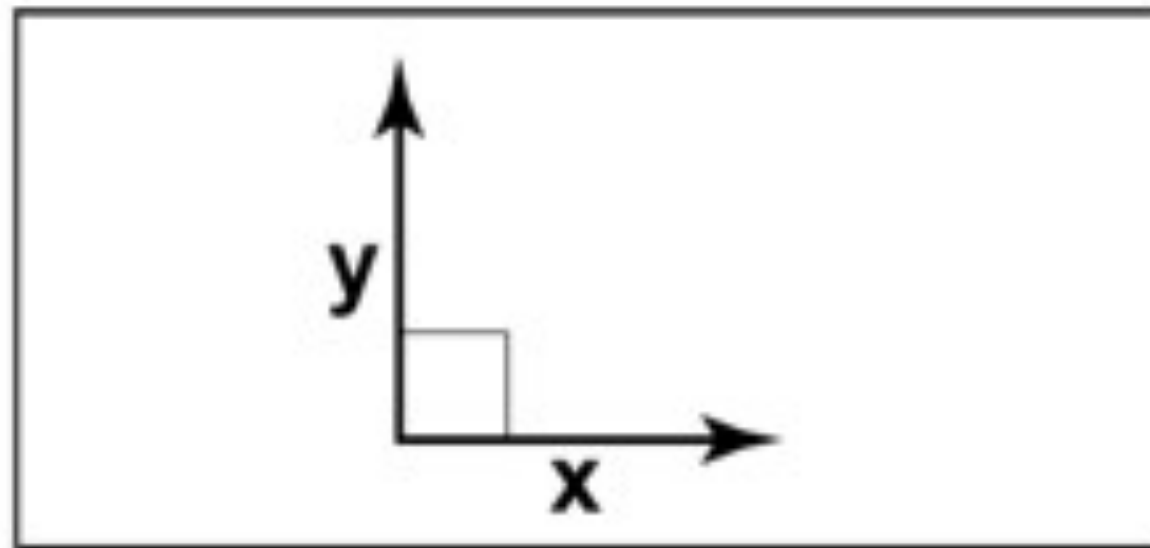


$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \phi$$

**The projection of  $\mathbf{a}$  onto  $\mathbf{b}$**

**N. B. the projection is 0 if  $\mathbf{a}$  is perpendicular to  $\mathbf{b}$**

# Orthonormal Vectors

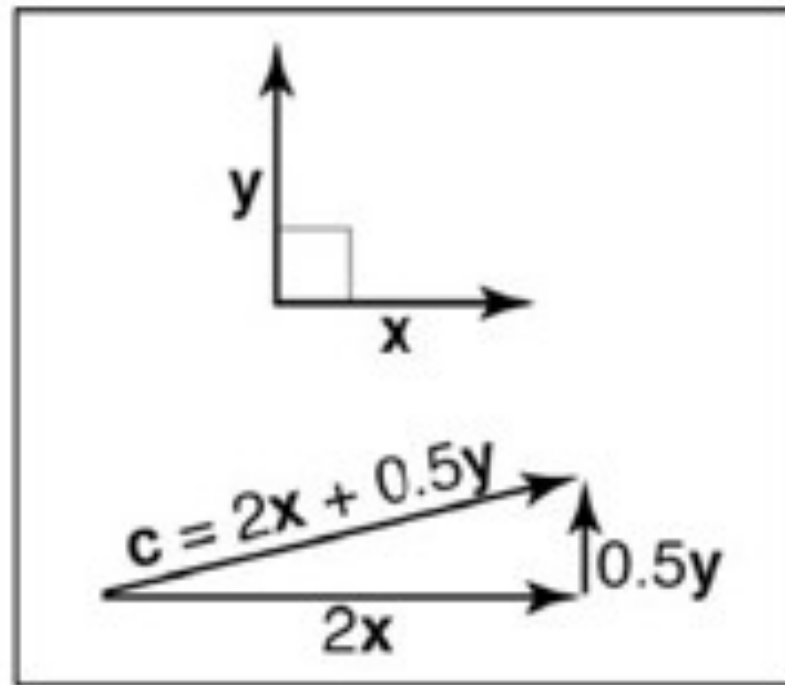


**Perpendicular**      $x \cdot y = 0$

**Unit length**      $x \cdot x = 1$

$$y \cdot y = 1$$

# Coordinates and Vectors



$$\mathbf{c} = \alpha \mathbf{x} + \beta \mathbf{y}$$

$$\alpha = \mathbf{x} \cdot \mathbf{c} = \alpha \mathbf{x} \cdot \mathbf{x} + \beta \mathbf{x} \cdot \mathbf{y}$$

$$\beta = \mathbf{y} \cdot \mathbf{c} = \alpha \mathbf{y} \cdot \mathbf{x} + \beta \mathbf{y} \cdot \mathbf{y}$$

# Dot product between two vectors

$$\mathbf{a} = x_a \mathbf{x} + y_a \mathbf{y}$$

$$\mathbf{b} = x_b \mathbf{x} + y_b \mathbf{y}$$

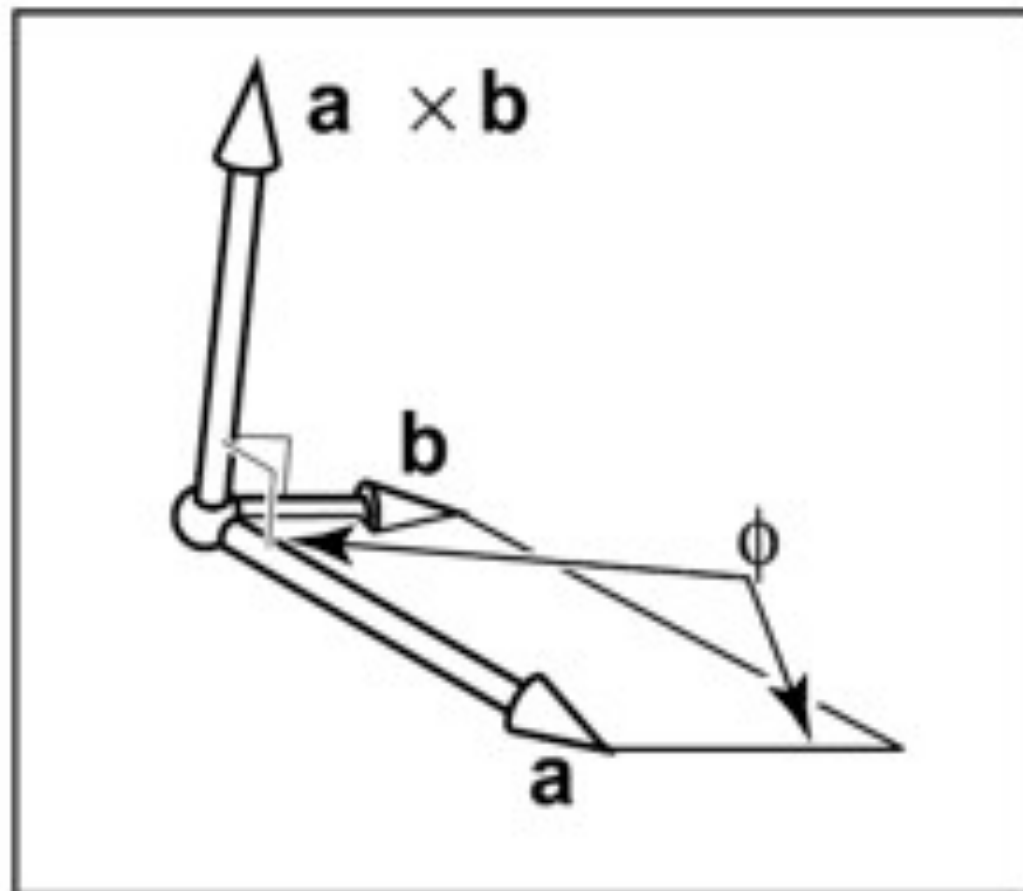
$$\mathbf{a} \cdot \mathbf{b} = x_a x_b + y_a y_b$$

$$\mathbf{a} \cdot \mathbf{a} = x_a^2 + y_a^2 = |\mathbf{a}|^2$$

$$|\mathbf{a}| = \sqrt{x_a^2 + y_a^2} = \sqrt{\mathbf{a} \cdot \mathbf{a}}$$



# Cross Product



$$\mathbf{c} = \mathbf{a} \times \mathbf{b}$$

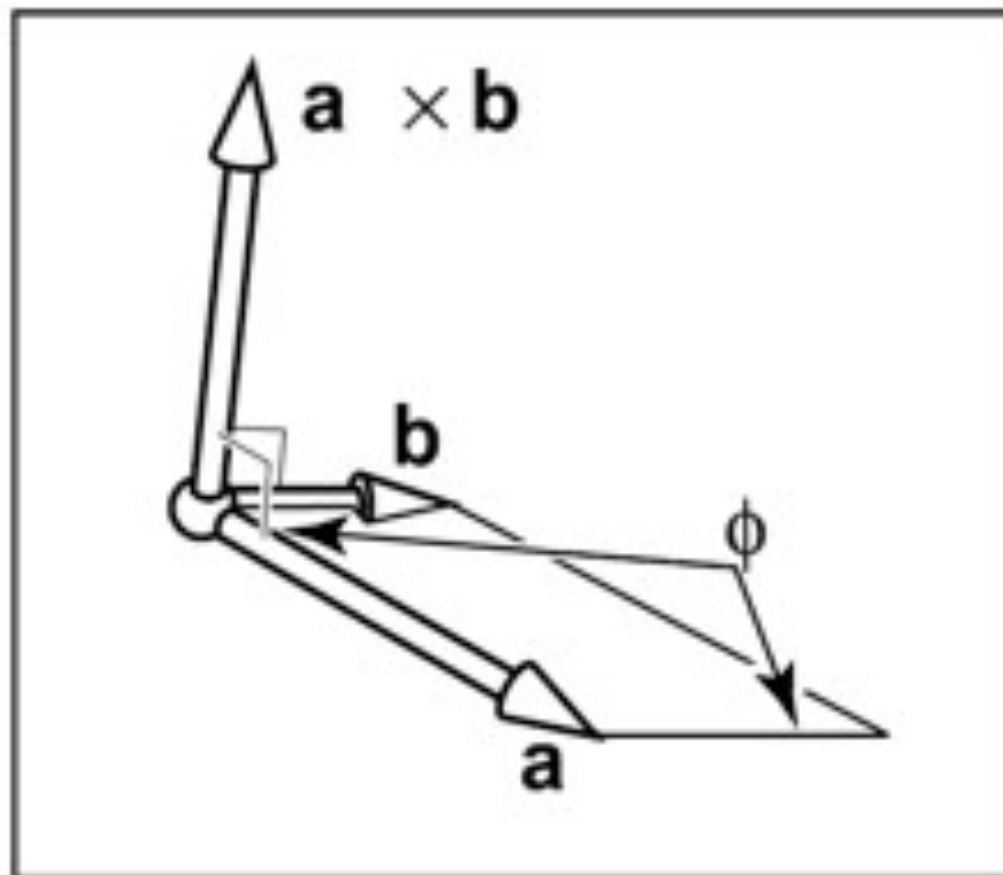
$$x_c = y_a z_b - z_a y_b$$

$$y_c = z_a x_b - x_a z_b$$

$$z_c = x_a y_b - y_a x_b$$

**c perpendicular to both a and b**  
**|c| is equal to the area of quadrilateral a b**

# Cross Product



**Right-Hand Rule**

$$\mathbf{x} \times \mathbf{y} = \mathbf{z}$$

$$\mathbf{y} \times \mathbf{z} = \mathbf{x}$$

$$\mathbf{z} \times \mathbf{x} = \mathbf{y}$$

$$\mathbf{x} \times \mathbf{x} = 0$$

$$\mathbf{y} \times \mathbf{y} = 0$$

$$\mathbf{z} \times \mathbf{z} = 0$$

# 2~3D

```
typedef float float2[2];  
typedef float float3[3];
```

```
float2 p2;  
float3 p3;
```

```
glVertex2fv( p2 );  
glVertex3fv( p3 );
```





# Vector operations

**Vectors:  $u, v, w$**

**$\langle \text{Vector} \rangle = \langle \text{Scalar} \rangle * \langle \text{Vector} \rangle$**

$$v = \alpha w$$

**$\langle \text{Vector} \rangle = \langle \text{Vector} \rangle + \langle \text{Vector} \rangle$**

$$u = v + w$$

**Implementation of parallelogram rule**



# Point operations

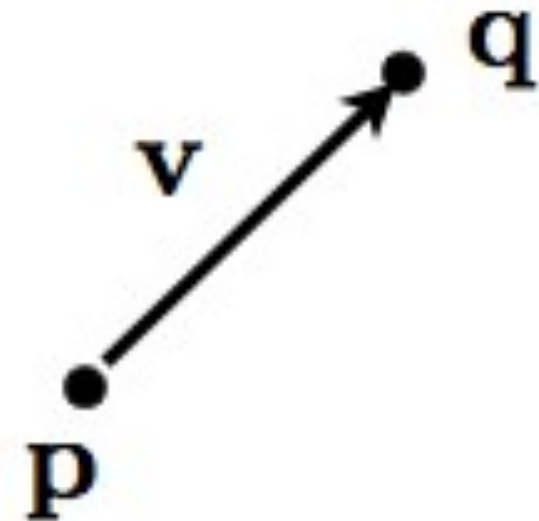
**Points:**  $p, q, r$

$$\langle \text{Point} \rangle = \langle \text{Point} \rangle + \langle \text{Vector} \rangle$$

$$q = p + v$$

$$\langle \text{Vector} \rangle = \langle \text{Point} \rangle - \langle \text{Point} \rangle$$

$$v = q - p$$



**A point is an origin and a vector displacement**

# illegal operations

**<Point> = <Scalar> \* <Point>**

$$p = \alpha q$$

**<Point> = <Point> + <Point>**

$$p = q + r$$

**<Vector> = <Point> + <Vector>**

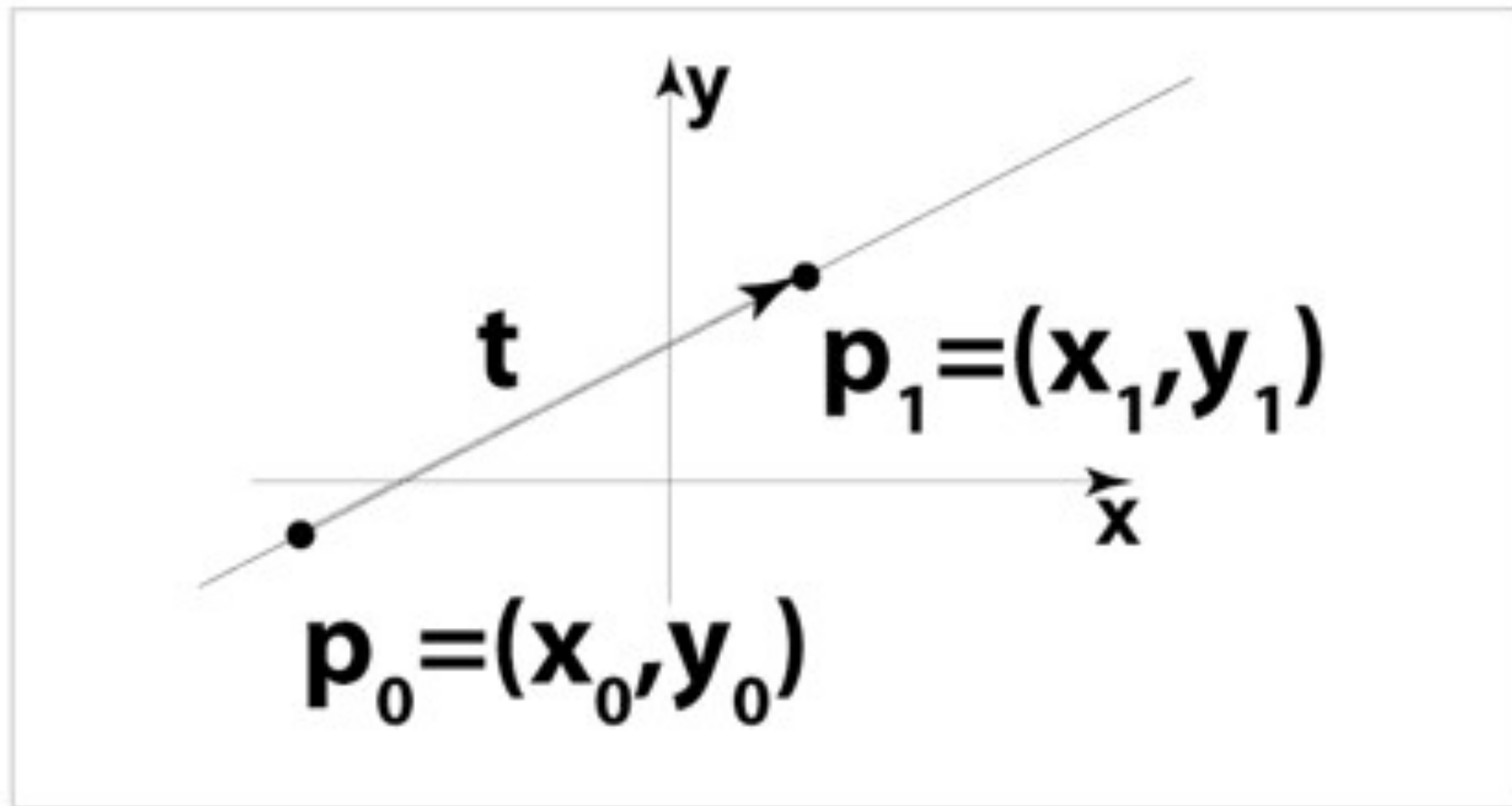
$$v = p + w$$

**<Point> = <Point> - <Point>**

$$p = q - r$$

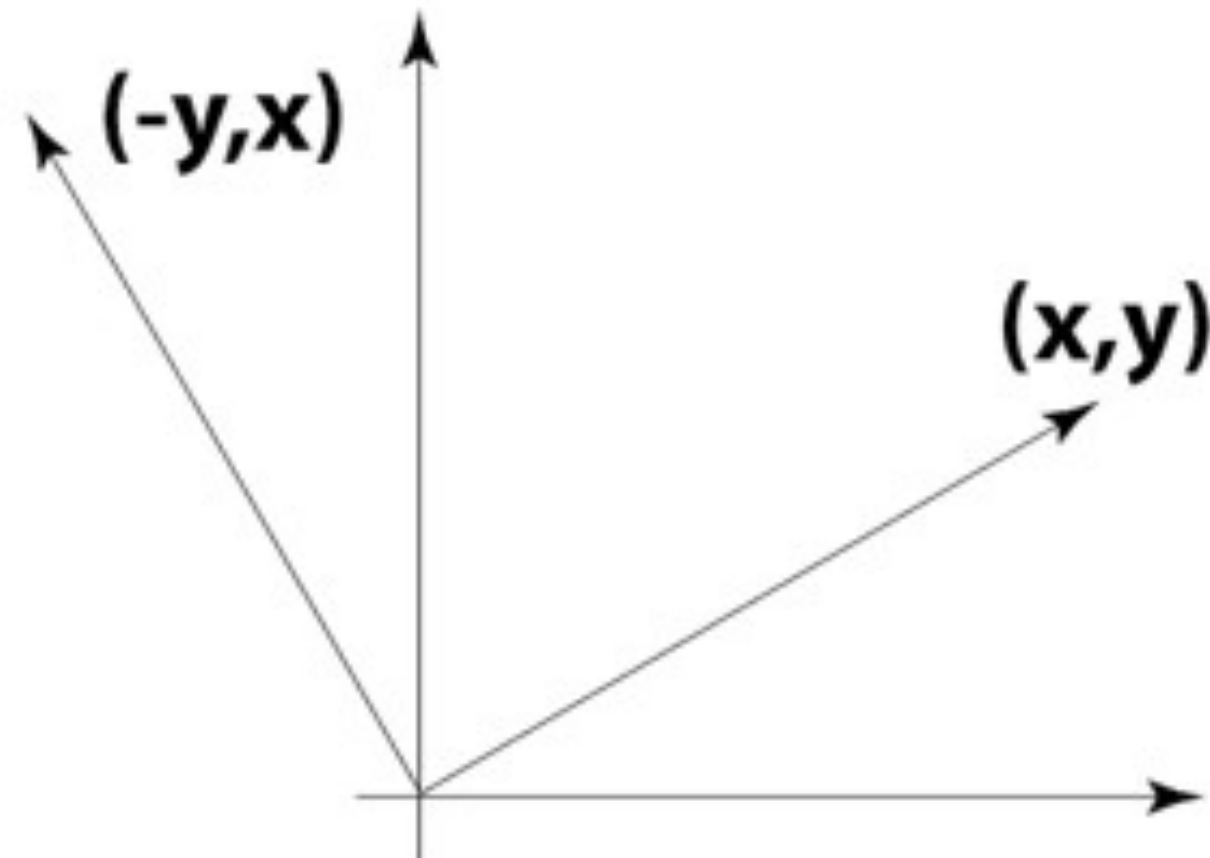


# Directed line



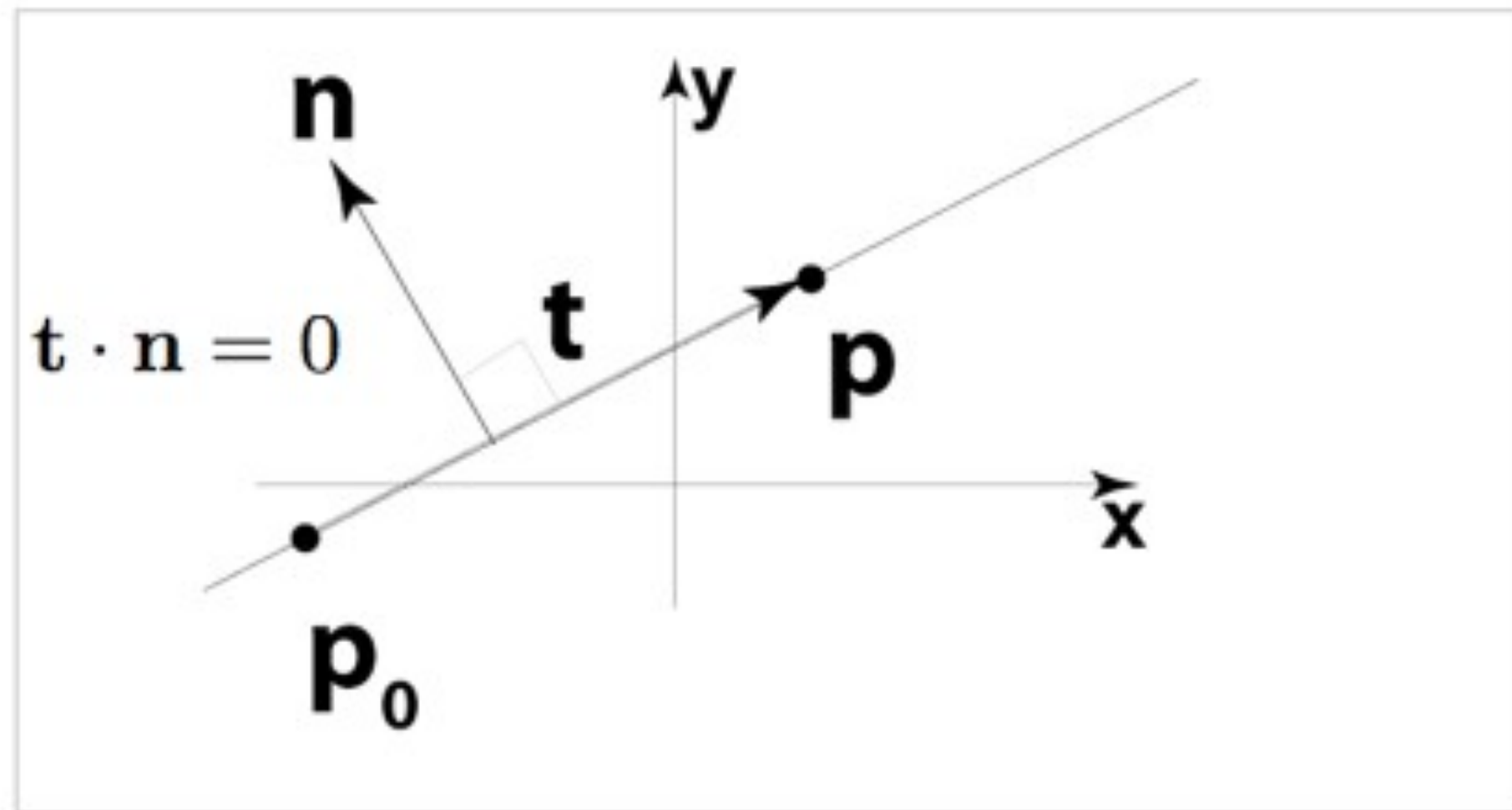
$$t = p_1 - p_0 = (x_1 - x_0, y_1 - y_0)$$

# Perpendicular vector in 2D



$$\text{Perp}((x, y)) = (-y, x)$$

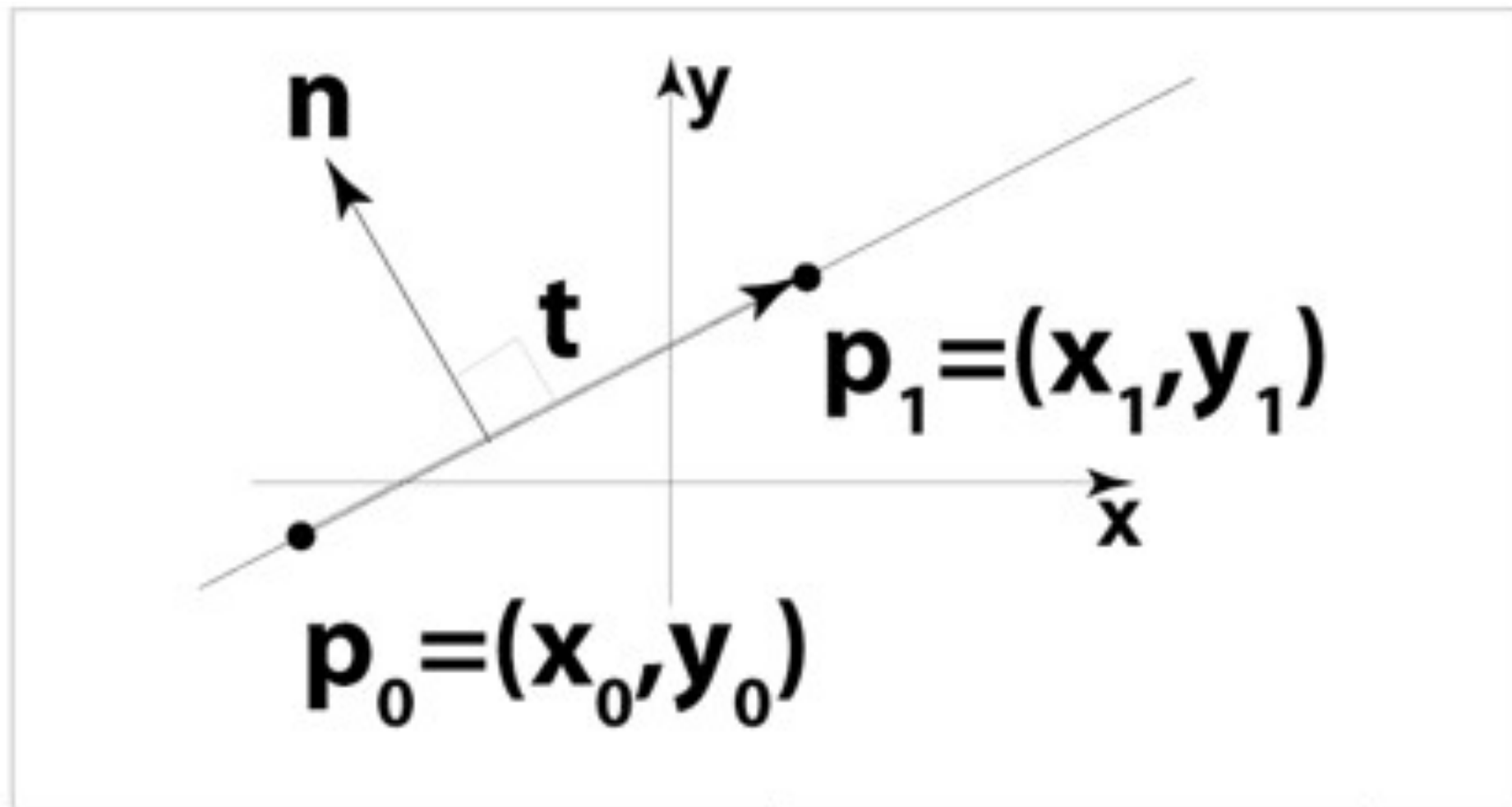
# Line equation



$$(\mathbf{p} - \mathbf{p}_0) \cdot \mathbf{n} = 0$$

**This equation must be true for all point  $\mathbf{p}$  on the line**

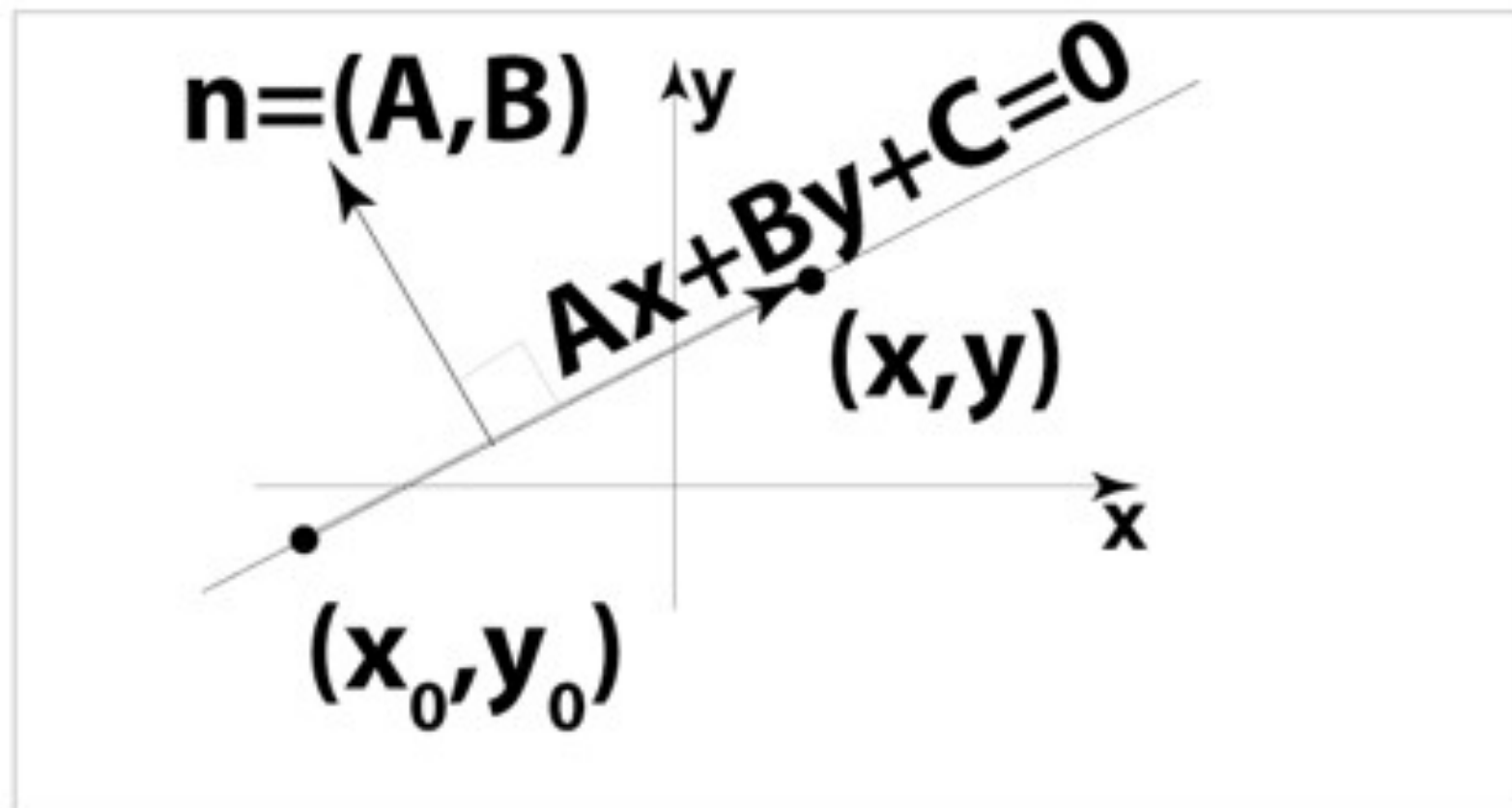
# Normal to the line



$$\mathbf{t} = \mathbf{p}_1 - \mathbf{p}_0 = (x_1 - x_0, y_1 - y_0)$$

$$\mathbf{n} = \text{Perp}(\mathbf{t}) = (y_0 - y_1, x_1 - x_0)$$

# Line equation

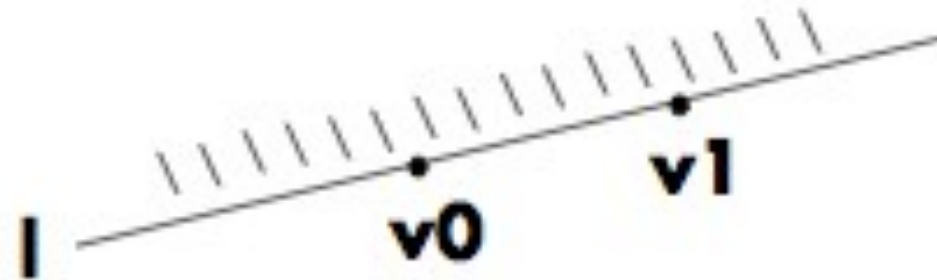


$$\begin{aligned} A &= y_1 - y_0 \\ B &= x_0 - x_1 \\ C &= x_0 y_1 - y_0 x_1 \end{aligned}$$



# Line equation

**Inside on the left for CCW polygons**



```
makeline( vert& v0, vert& v1, line& l )
{
    l.a = v1.y - v0.y;
    l.b = v0.x - v1.x;
    l.c = -(l.a * v0.x + l.b * v0.y);
}
```

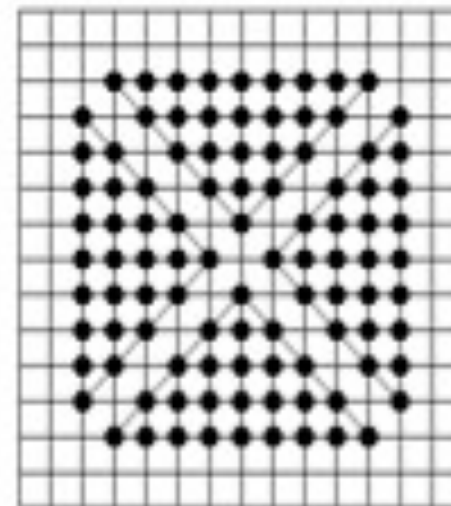
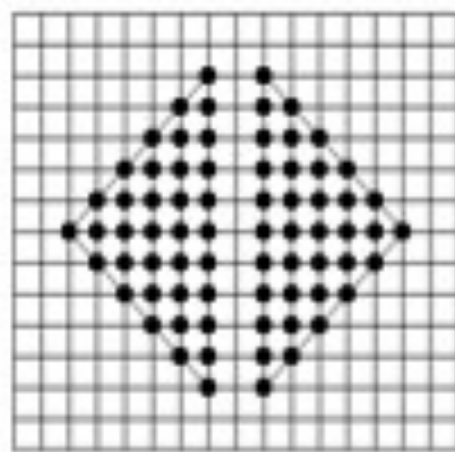
# Singularities

**Singularities: Edges that touch pixels ( $e == 0$ )**

**Causes two fragments to be generated**

- **Wasted effort**

- **Problems with transparency (later lecture)**



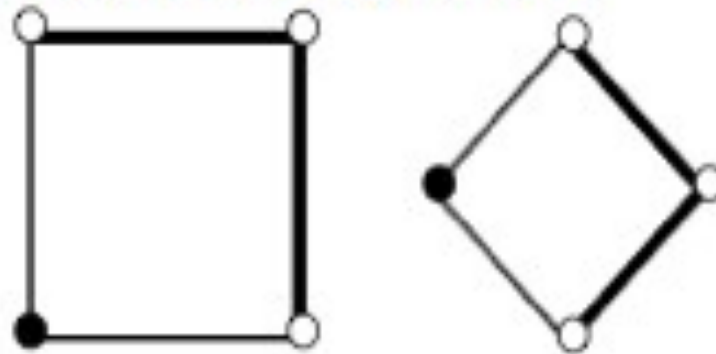
**Not including singularities ( $e < 0$ ) causes gaps**

# Handling singularity

**Create shadowed edges (thick lines)**

**Don't draw pixels on shadowed edges**

**Solid drawn; hollow not drawn**



```
int shadow( value a, value b ) {  
    return (a>0) || (a==0 && b > 0);  
}  
int inside( value e, value a, value b ) {  
    return (e == 0) ? !shadow(a,b) : (e < 0);  
}
```

# Antialiasing



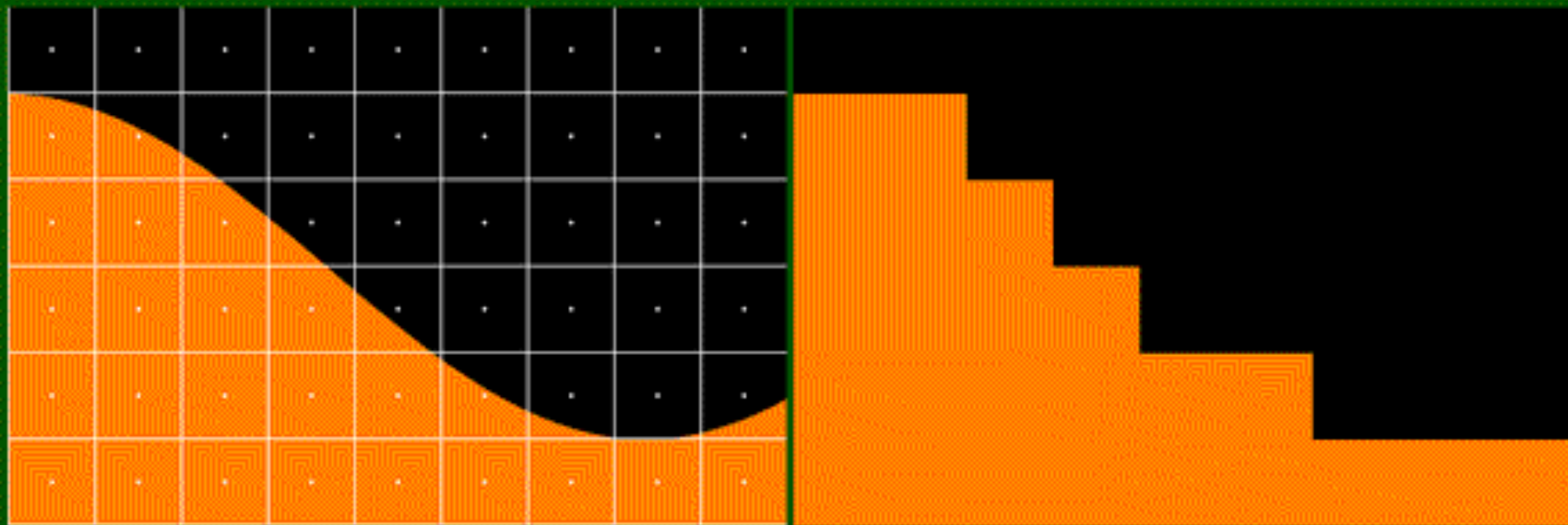
# Aliasing

- Aliasing is caused due to the discrete nature of the display device
- Rasterizing primitives is like sampling a continuous signal by a finite set of values (point sampling)
- Information is lost if the rate of sampling is not sufficient. This sampling error is called ***aliasing***.
- Effects of aliasing are
  - Jagged edges
  - Incorrectly rendered fine details
  - Small objects might miss





# Aliasing(examples)

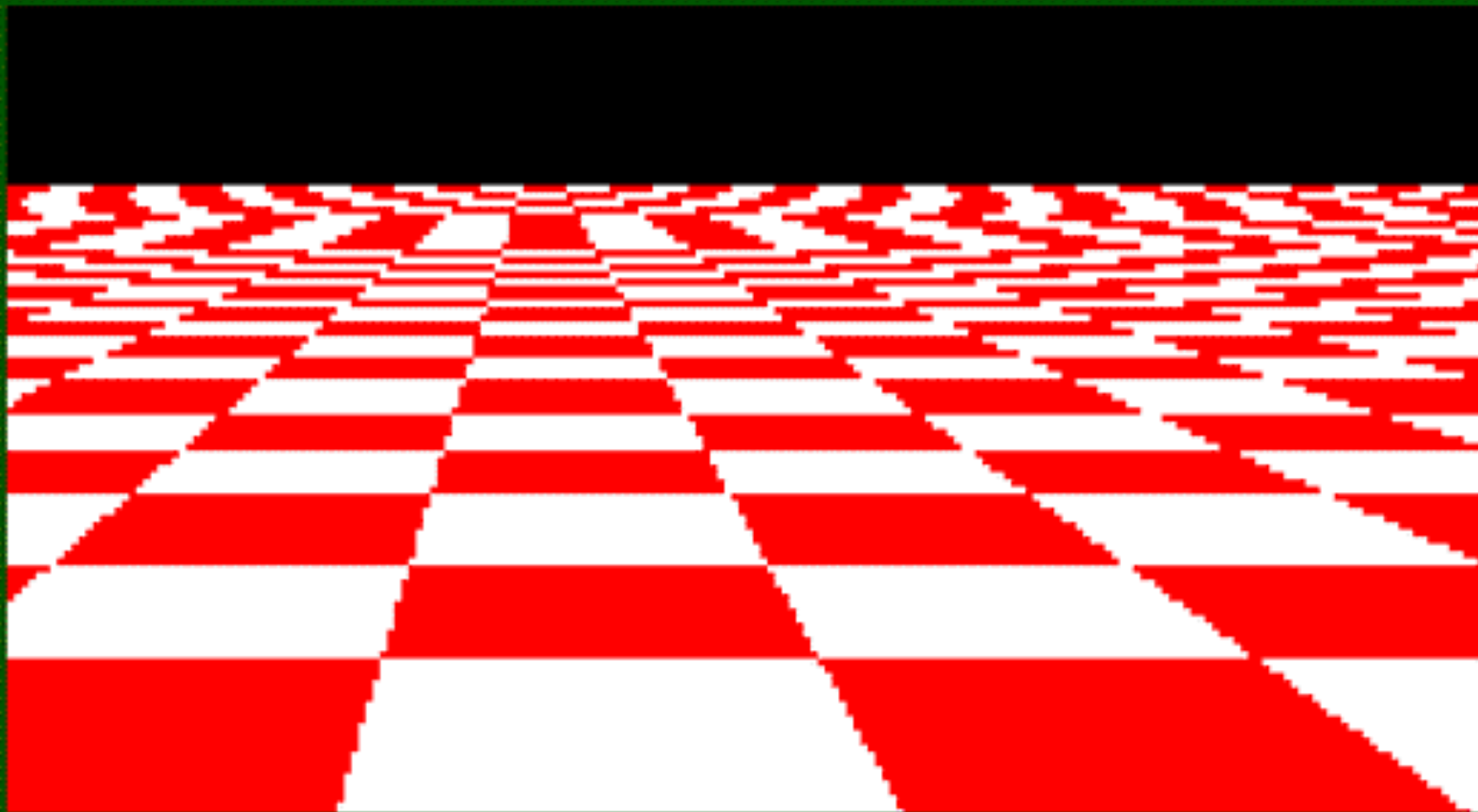


*Original*

*Rendered*

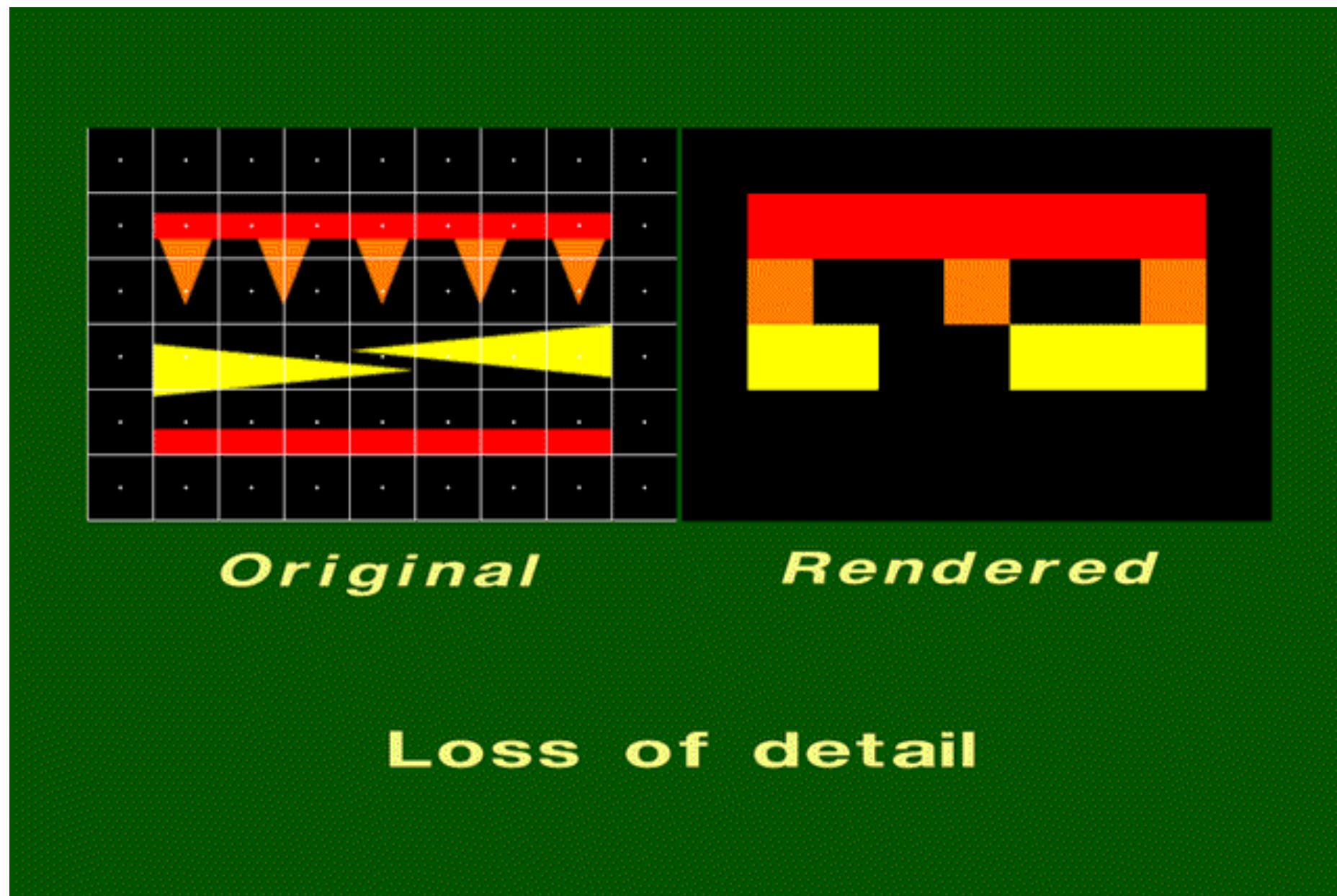
**Jagged profiles**

# Aliasing(examples)



**Disintegrating textures**

# Aliasing(examples)





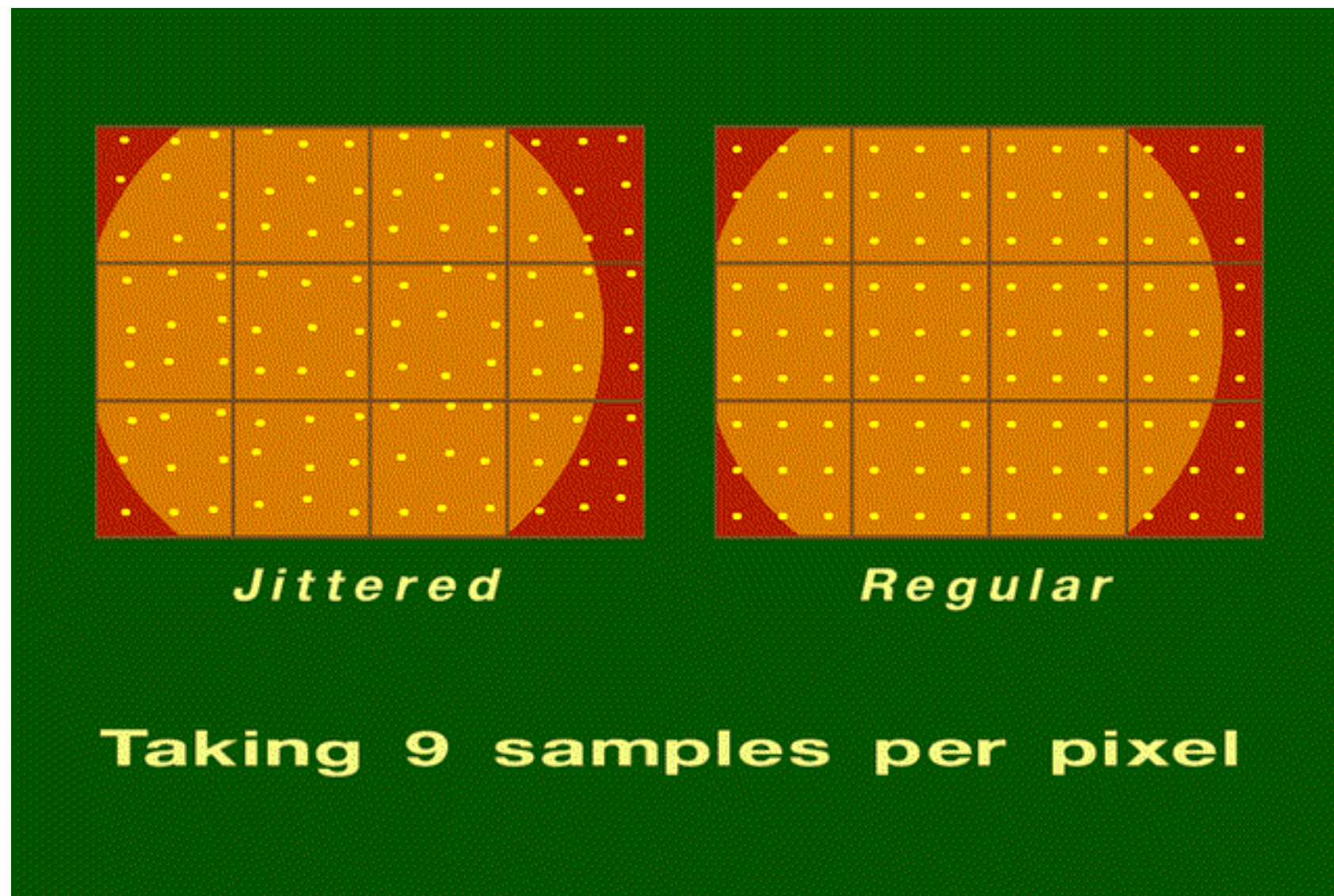
# Antialiasing

- Application of techniques to reduce/eliminate aliasing artifacts
- Some of the methods are
  - increasing sampling rate by increasing the resolution. Display memory requirements increases four times if the resolution is doubled
  - averaging methods (post processing). Intensity of a pixel is set as the weighted average of its own intensity and the intensity of the surrounding pixels
  - Area sampling, more popular



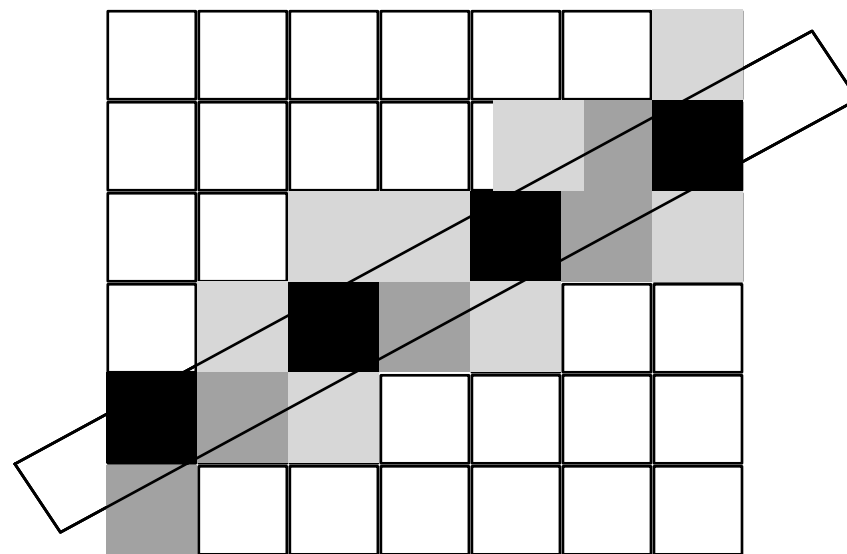
# Antialiasing (postfiltering)

**How should one supersample?**



# Area Sampling

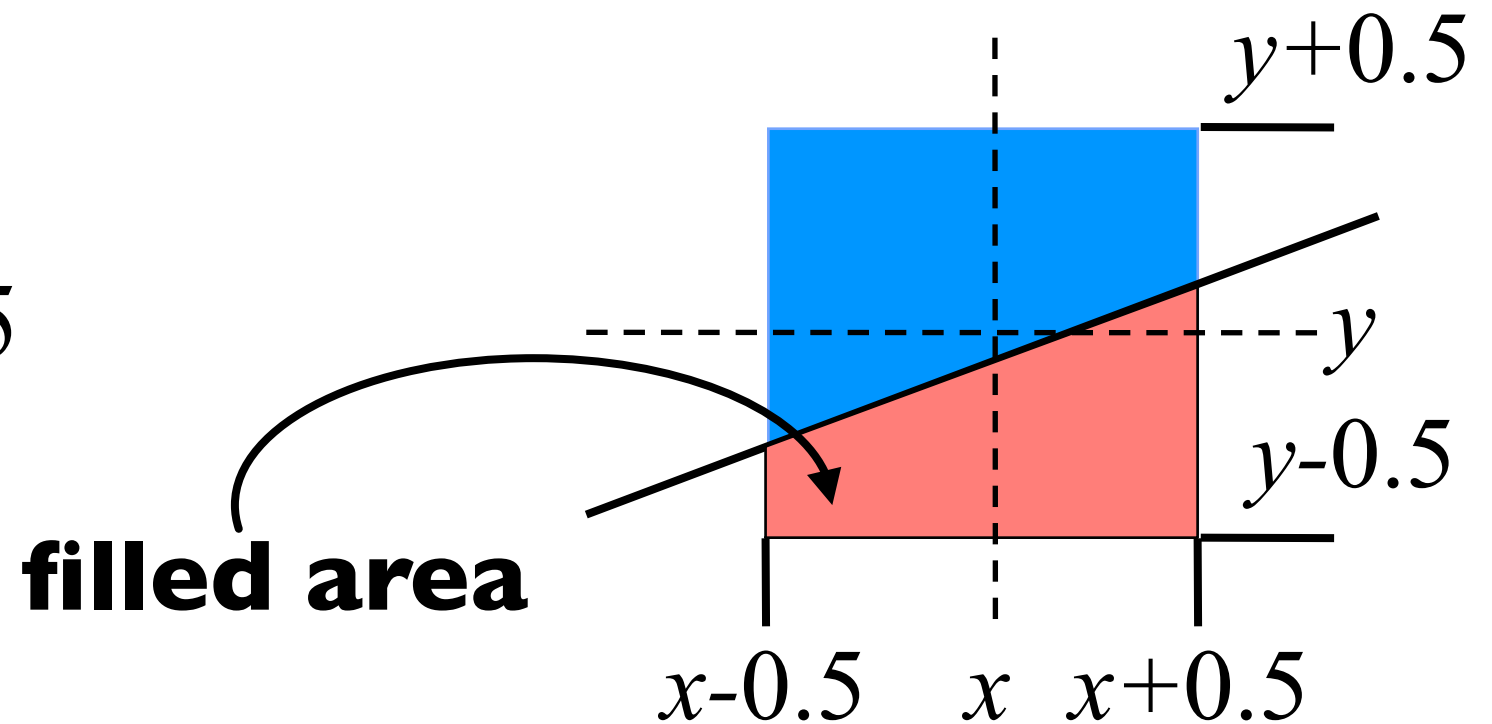
- A scan converted primitive occupies finite area on the screen
- Intensity of the boundary pixels is adjusted depending on the percent of the pixel area covered by the primitive. This is called **weighted area sampling**



# Area Sampling

- Methods to estimate percent of pixel covered by the primitive
  - subdivide pixel into sub-pixels and determine how many sub-pixels are inside the boundary
  - Incremental line algorithm can be extended, with area calculated as

$$Area = m \times x - y + c + 0.5$$



# Clipping



# Clipping

- Clipping of primitives is done usually before scan converting the primitives
- Reasons being
  - scan conversion needs to deal only with the clipped version of the primitive, which might be much smaller than its unclipped version
  - Primitives are usually defined in the real world, and their mapping from the real to the integer domain of the display might result in the overflowing of the integer values resulting in unnecessary artifacts



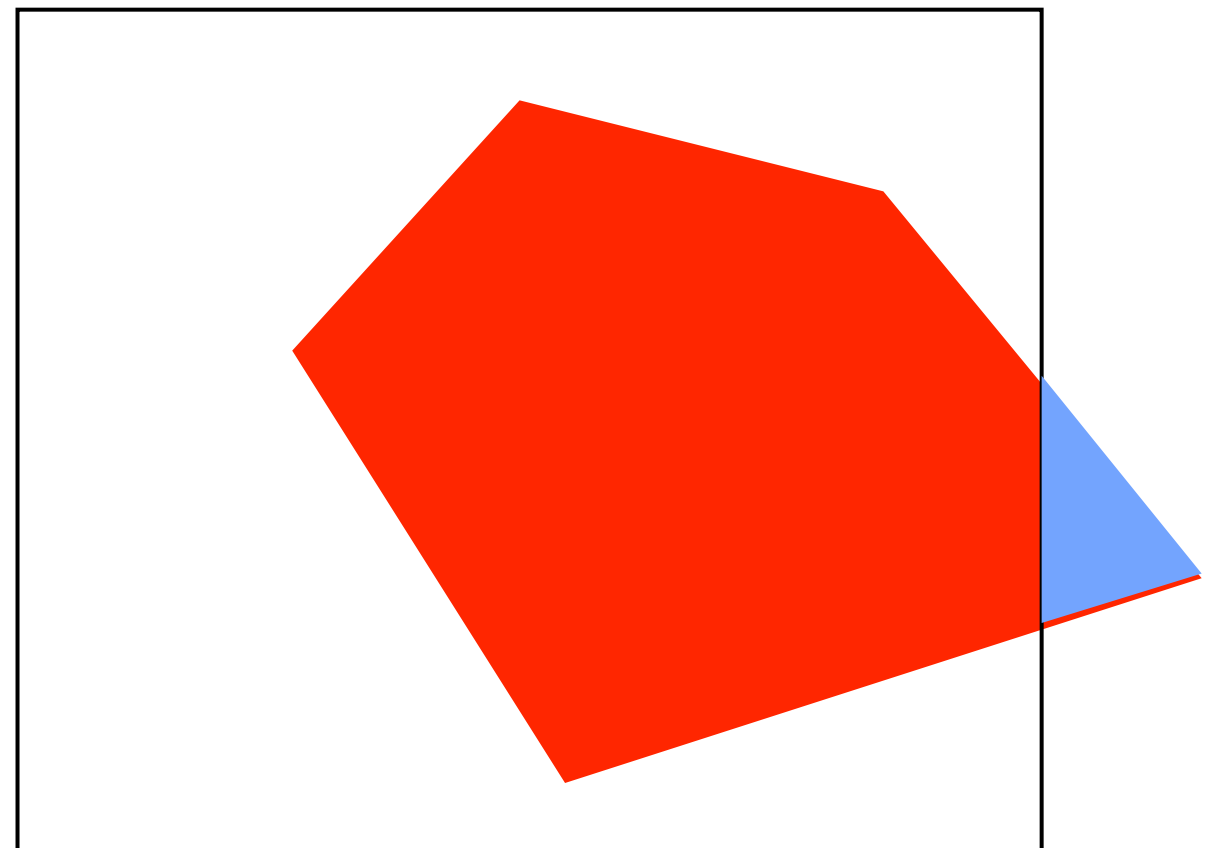
# Clipping

- Why Clipping?
- How Clipping?
  - Lines
  - Polygons
- Note: Content from chapter 4.
  - Lots of stuff about rendering systems and mathematics in that chapter.



# Definition

- Clipping – Removal of content that is not going to be displayed
  - Behind camera
  - Too close
  - Too far
  - Off sides of the screen





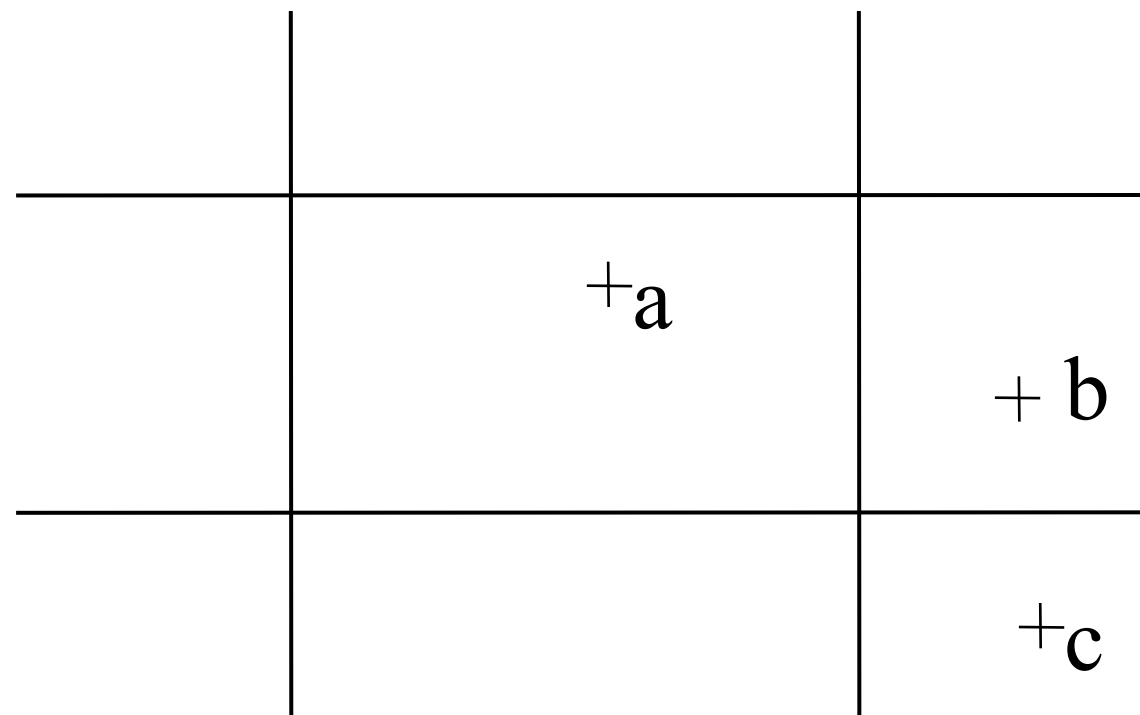
# How would we clip?

- Points?
- Lines?
- Polygons?
- Other objects?



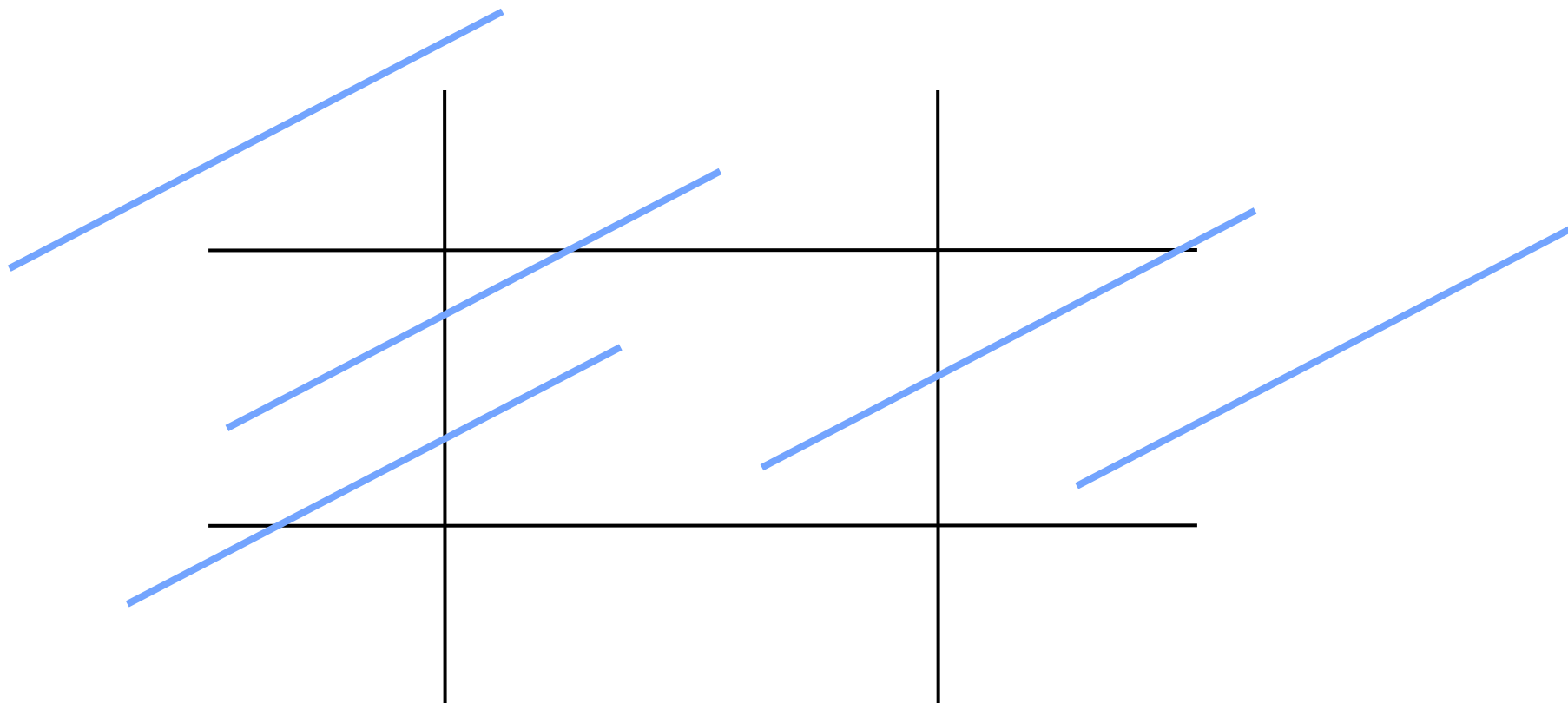
# We'll start in 2D

- Assume a 2D upright rectangle we are clipping against
  - Common in windowing systems
  - Points are trivial
    - $x \geq \min_x$  and  $x \leq \max_x$  and  $y \geq \min_y$  and  $y \leq \max_y$



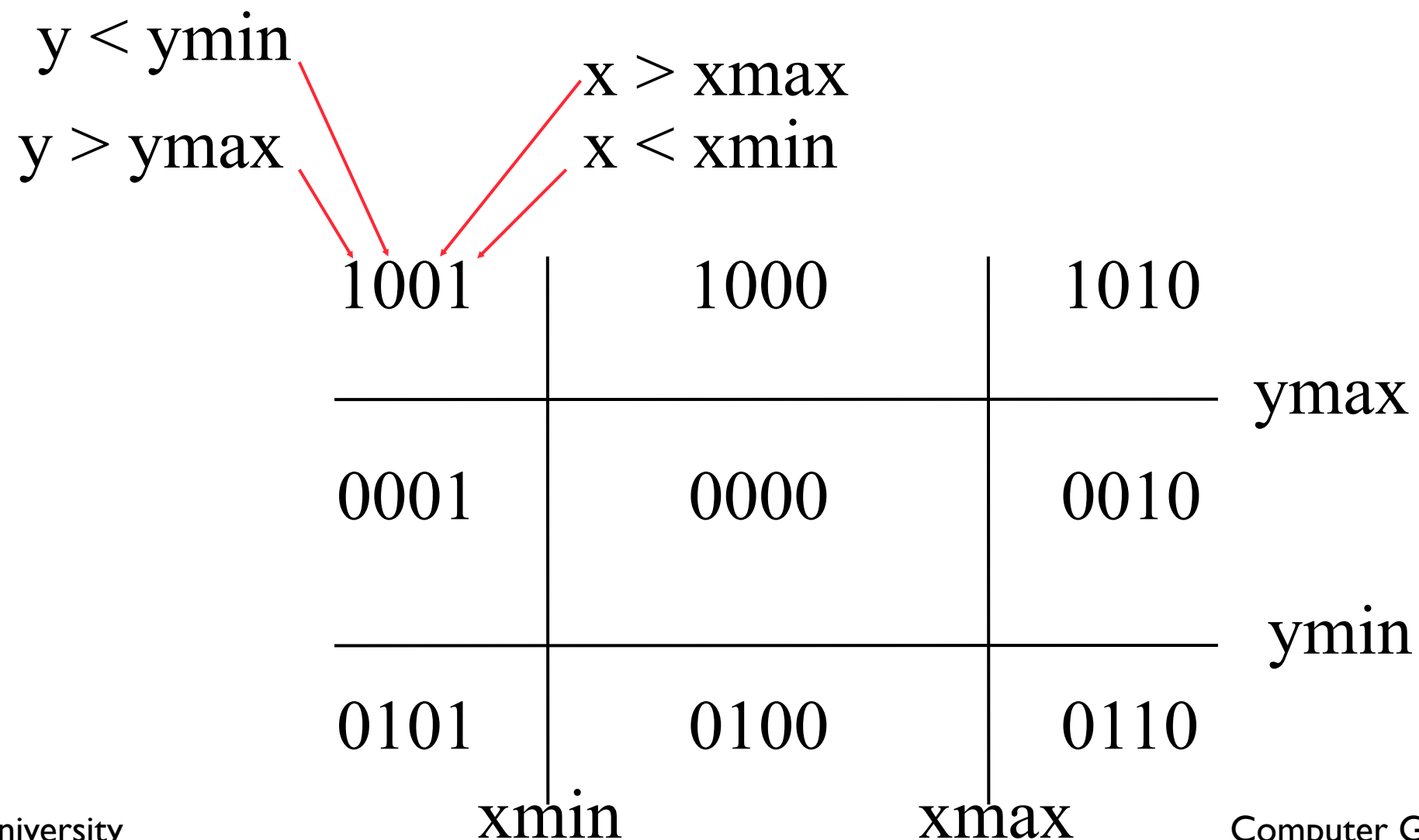
# Line Segments

- What can happen when a line segment is clipped?



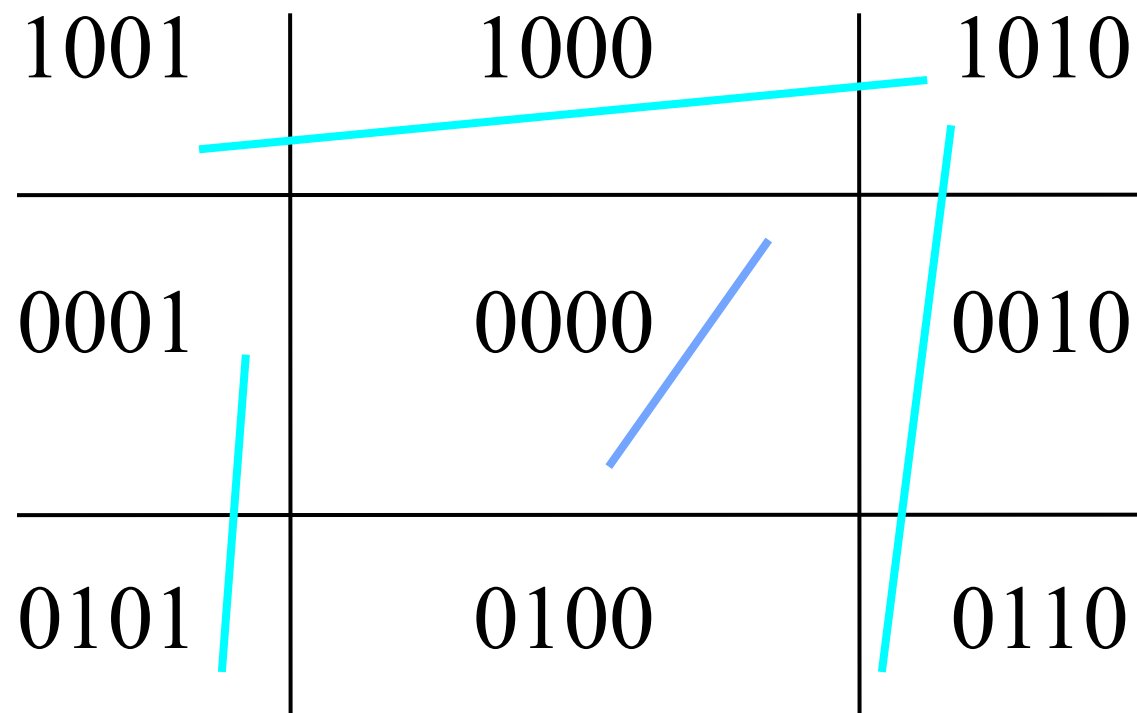
# Cohen-Sutherland Line Clipping

- We'll assign the ends of a line “outcodes”, 4 bit values that indicate if they are inside or outside the clip area.



# Outcode cases

- We'll call the two endpoint outcodes  $o_1$  and  $o_2$ .
  - If  $o_1 = o_2 = 0$ , both endpoints are inside.
  - else if  $(o_1 \& o_2) \neq 0$ , both ends points are on the same side, the edge is discarded.



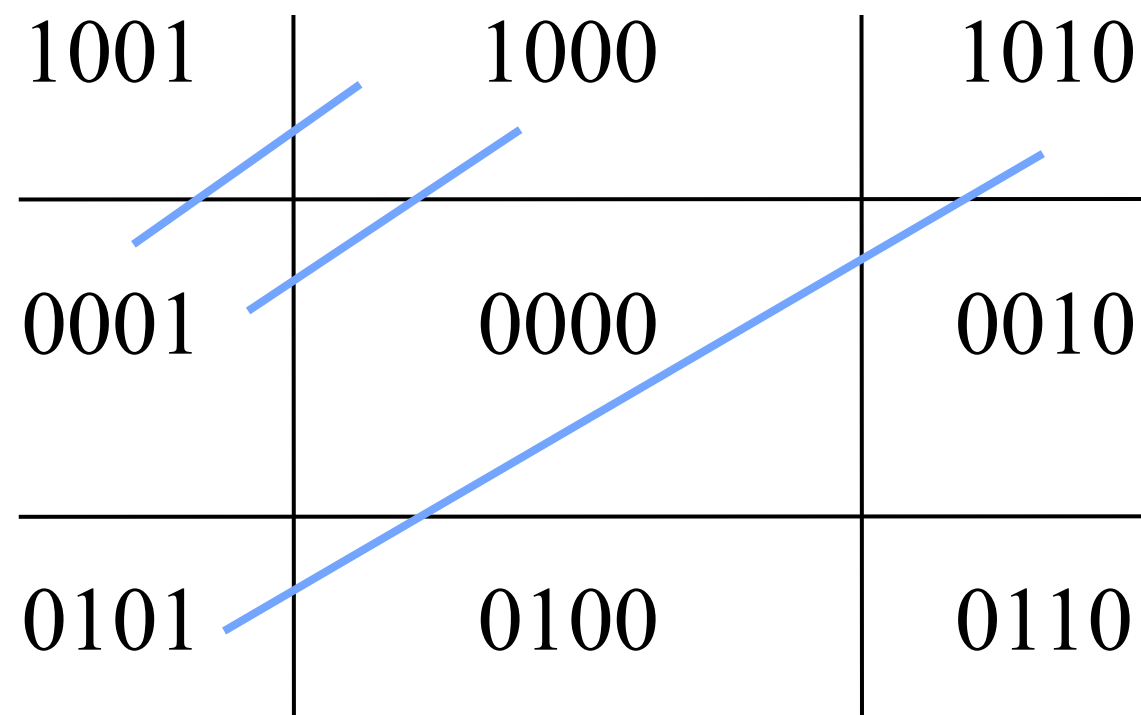
# More cases

- else if  $(o_1 \neq 0)$  and  $(o_2 = 0)$ , (or vice versa), one end is inside, other is outside.
  - Clip and recompute *one that's outside* until inside.
  - Clip edges with bits set...
  - May require two clip computations

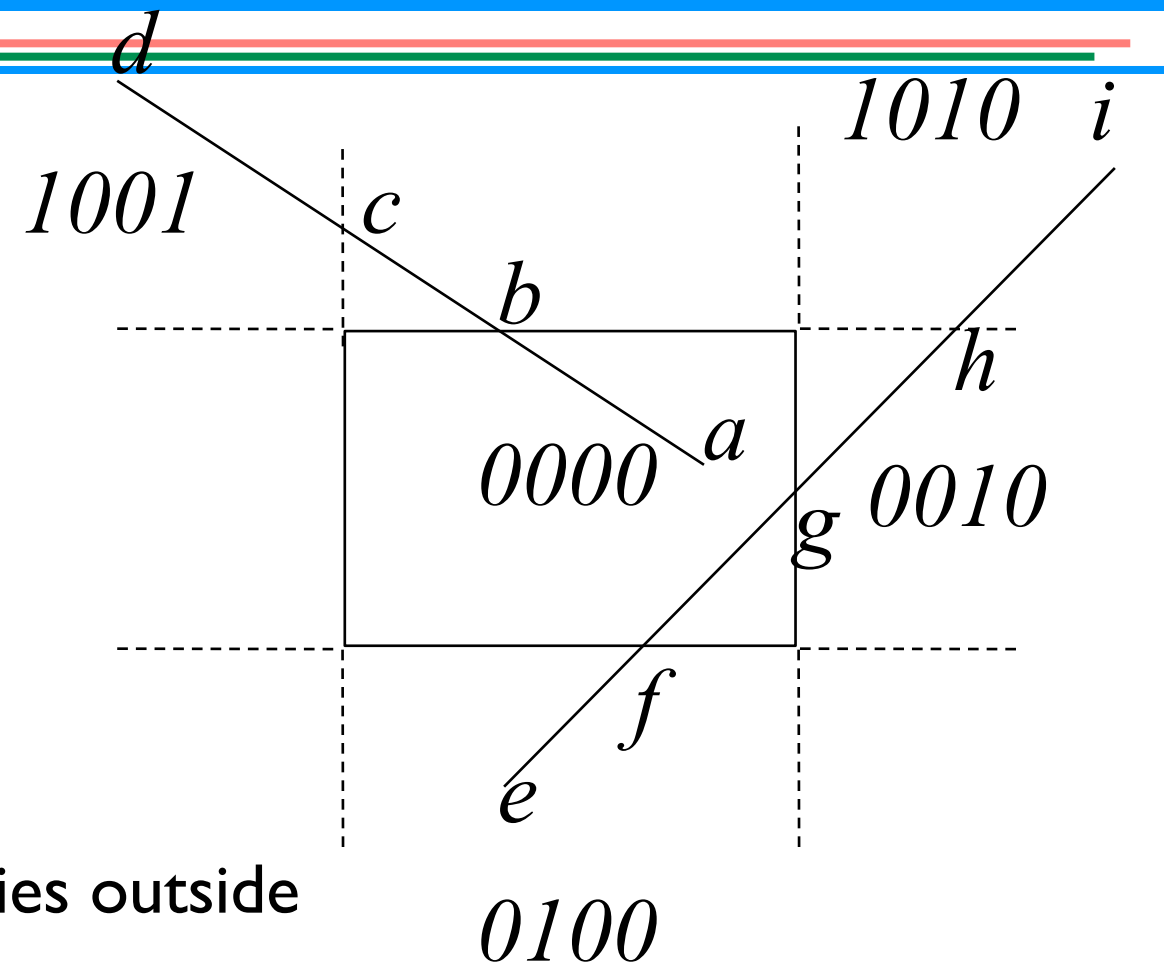
1001	1000	1010
0001	0000	0010
0101	0100	0110

# Last case...

- else if  $(o1 \ \& \ o2) = 0$ , end points are on different sides.
  - Clip and recompute.
  - May have some inside part or may not...
  - May require up to 4 clips!



# Cohen-Sutherland Line-Clipping Algorithm



- To do the clipping find the end point that lies outside
- Test the outcode to find the edge that is crossed and determine the corresponding intersection point
- Replace the outside end-point by intersection-point
- Repeat the above steps for the new line



```
typedef int OutCode;
```

```
const int INSIDE = 0; // 0000
const int LEFT = 1; // 0001
const int RIGHT = 2; // 0010
const int BOTTOM = 4; // 0100
const int TOP = 8; // 1000
```

```
// Compute the bit code for a point (x, y) using the clip rectangle
// bounded diagonally by (xmin, ymin), and (xmax, ymax)
```

```
// ASSUME THAT xmax, xmin, ymax and ymin are global constants.
```

```
OutCode ComputeOutCode(double x, double y)
```

```
{
    OutCode code;

    code = INSIDE; // initialised as being inside of [[clip window]]

    if (x < xmin) // to the left of clip window
        code |= LEFT;
    else if (x > xmax) // to the right of clip window
        code |= RIGHT;
    if (y < ymin) // below the clip window
        code |= BOTTOM;
    else if (y > ymax) // above the clip window
        code |= TOP;

    return code;
}
```

```
// Cohen-Sutherland clipping algorithm clips a line from
// P0 = (x0, y0) to P1 = (x1, y1) against a rectangle with
// diagonal from (xmin, ymin) to (xmax, ymax).
```

```
void CohenSutherlandLineClipAndDraw(double x0, double y0, double x1, double y1)
```

```
{
    // compute outcodes for P0, P1, and whatever point lies outside the clip rectangle
    OutCode outcode0 = ComputeOutCode(x0, y0);
    OutCode outcode1 = ComputeOutCode(x1, y1);
    bool accept = false;

    while (true) {
        if (!(outcode0 | outcode1)) { // Bitwise OR is 0. Trivially accept and get out of loop
            accept = true;
        }
    }
}
```

```

// Cohen–Sutherland clipping algorithm clips a line from
// P0 = (x0, y0) to P1 = (x1, y1) against a rectangle with
// diagonal from (xmin, ymin) to (xmax, ymax).
void CohenSutherlandLineClipAndDraw(double x0, double y0, double x1, double y1)
{
    // compute outcodes for P0, P1, and whatever point lies outside the clip rectangle
    OutCode outcode0 = ComputeOutCode(x0, y0);
    OutCode outcode1 = ComputeOutCode(x1, y1);
    bool accept = false;

    while (true) {
        if (!(outcode0 | outcode1)) { // Bitwise OR is 0. Trivially accept and get out of loop
            accept = true;
            break;
        } else if (outcode0 & outcode1) { // Bitwise AND is not 0. Trivially reject and get out of loop
            break;
        } else {
            // failed both tests, so calculate the line segment to clip
            // from an outside point to an intersection with clip edge
            double x, y;

            // At least one endpoint is outside the clip rectangle; pick it.
            OutCode outcodeOut = outcode0 ? outcode0 : outcode1;

            // Now find the intersection point;
            // use formulas  $y = y_0 + \text{slope} * (x - x_0)$ ,  $x = x_0 + (1 / \text{slope}) * (y - y_0)$ 
            if (outcodeOut & TOP) { // point is above the clip rectangle
                x = x0 + (x1 - x0) * (ymax - y0) / (y1 - y0);
                y = ymax;
            } else if (outcodeOut & BOTTOM) { // point is below the clip rectangle
                x = x0 + (x1 - x0) * (ymin - y0) / (y1 - y0);
                y = ymin;
            } else if (outcodeOut & RIGHT) { // point is to the right of clip rectangle
                y = y0 + (y1 - y0) * (xmax - x0) / (x1 - x0);
                x = xmax;
            } else if (outcodeOut & LEFT) { // point is to the left of clip rectangle
                y = y0 + (y1 - y0) * (xmin - x0) / (x1 - x0);
                x = xmin;
            }

            // Now we move outside point to intersection point to clip
            // and get ready for next pass.
            if (outcodeOut == outcode0) {
                x0 = x;
                y0 = y;
            }
        }
    }
}

```

```

        break;
    } else {
        // failed both tests, so calculate the line segment to clip
        // from an outside point to an intersection with clip edge
        double x, y;

        // At least one endpoint is outside the clip rectangle; pick it.
        OutCode outcodeOut = outcode0 ? outcode0 : outcode1;

        // Now find the intersection point;
        // use formulas  $y = y_0 + \text{slope} * (x - x_0)$ ,  $x = x_0 + (1 / \text{slope}) * (y - y_0)$ 
        if (outcodeOut & TOP) { // point is above the clip rectangle
            x = x0 + (x1 - x0) * (ymax - y0) / (y1 - y0);
            y = ymax;
        } else if (outcodeOut & BOTTOM) { // point is below the clip rectangle
            x = x0 + (x1 - x0) * (ymin - y0) / (y1 - y0);
            y = ymin;
        } else if (outcodeOut & RIGHT) { // point is to the right of clip rectangle
            y = y0 + (y1 - y0) * (xmax - x0) / (x1 - x0);
            x = xmax;
        } else if (outcodeOut & LEFT) { // point is to the left of clip rectangle
            y = y0 + (y1 - y0) * (xmin - x0) / (x1 - x0);
            x = xmin;
        }

        // Now we move outside point to intersection point to clip
        // and get ready for next pass.
        if (outcodeOut == outcode0) {
            x0 = x;
            y0 = y;
            outcode0 = ComputeOutCode(x0, y0);
        } else {
            x1 = x;
            y1 = y;
            outcode1 = ComputeOutCode(x1, y1);
        }
    }
}

if (accept) {
    // Following functions are left for implementation by user based on
    // their platform (OpenGL/graphics.h etc.)
    DrawRectangle(xmin, ymin, xmax, ymax);
    LineSegment(x0, y0, x1, y1);
}
}

```

# Liang–Barsky algorithm

---

Consider first the usual parametric form of a straight line:

$$x = x_0 + u(x_1 - x_0) = x_0 + u\Delta x$$

$$y = y_0 + u(y_1 - y_0) = y_0 + u\Delta y$$

A point is in the clip window, if

$$x_{\min} \leq x_0 + u\Delta x \leq x_{\max}$$

and

$$y_{\min} \leq y_0 + u\Delta y \leq y_{\max},$$

which can be expressed as the 4 inequalities

$$up_k \leq q_k, \quad k = 1, 2, 3, 4,$$

where

$$p_1 = -\Delta x, q_1 = x_0 - x_{\min} \text{ (left)}$$

$$p_2 = \Delta x, q_2 = x_{\max} - x_0 \text{ (right)}$$

$$p_3 = -\Delta y, q_3 = y_0 - y_{\min} \text{ (bottom)}$$

$$p_4 = \Delta y, q_4 = y_{\max} - y_0 \text{ (top)}$$



Brian Barsky

# Liang–Barsky algorithm

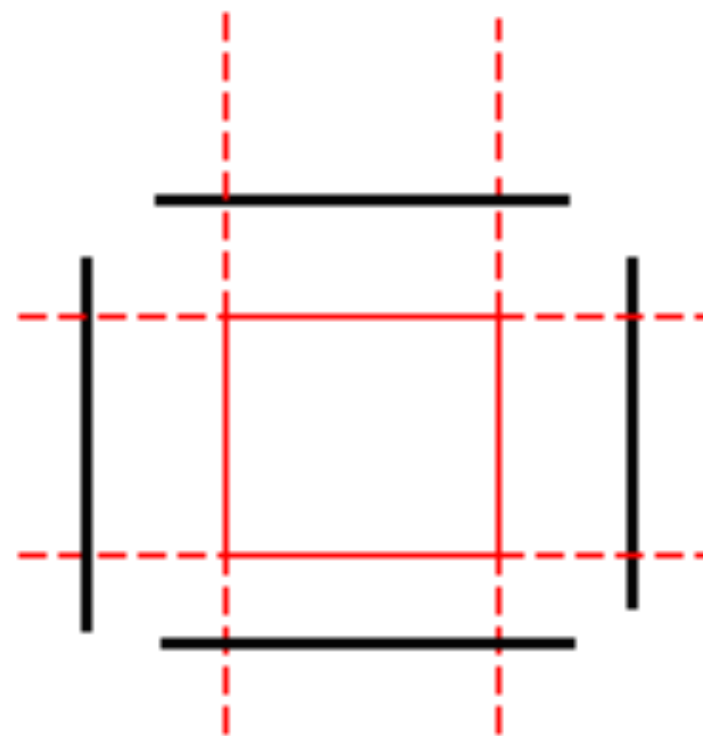
---

To compute the final line segment:

1. A line parallel to a clipping window edge has  $p_k = 0$  for that boundary.
2. If for that  $k$ ,  $q_k < 0$ , the line is completely outside and can be eliminated.
3. When  $p_k < 0$  the line proceeds outside to inside the clip window and when  $p_k > 0$ , the line proceeds inside to outside.
4. For nonzero  $p_k$ ,  $u = \frac{q_k}{p_k}$  gives the intersection point.
5. For each line, calculate  $u_1$  and  $u_2$ . For  $u_1$ , look at boundaries for which  $p_k < 0$  (i.e. outside to inside). Take  $u_1$  to be the largest among  $\left\{0, \frac{q_k}{p_k}\right\}$ . For  $u_2$ , look at boundaries for which  $p_k > 0$  (i.e. inside to outside). Take  $u_2$  to be the minimum of  $\left\{1, \frac{q_k}{p_k}\right\}$ . If  $u_1 > u_2$ , the line is outside and therefore rejected.

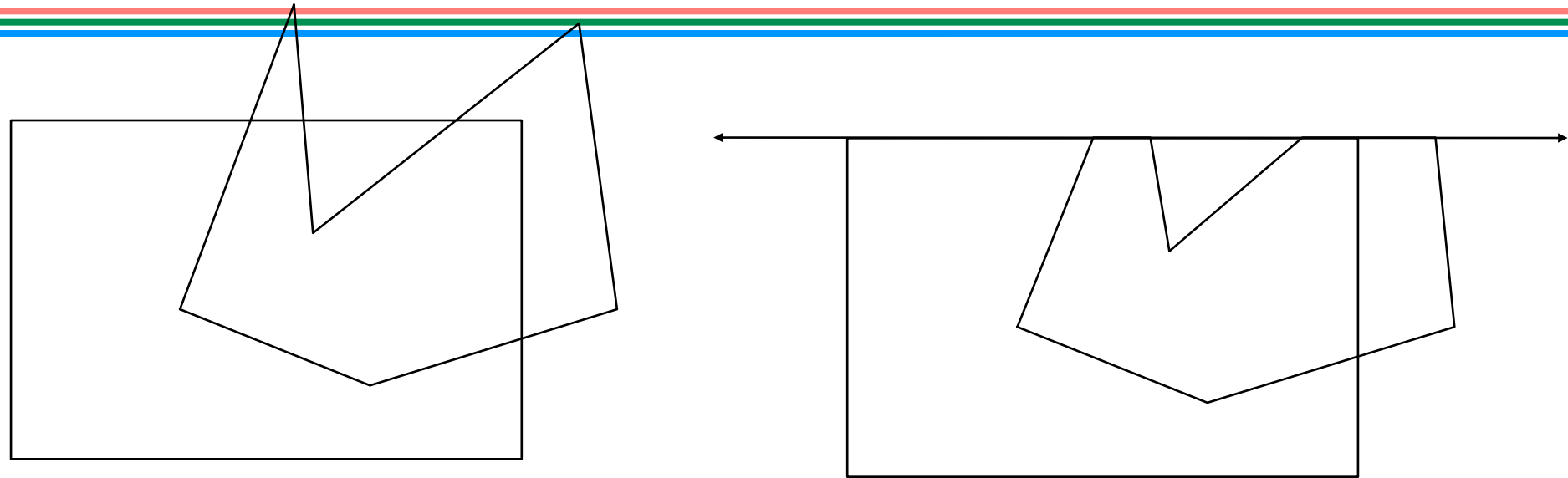
# Liang–Barsky algorithm

- 1、初始化线段交点的参数： $u_1=0$ ， $u_2=1$ ；
- 2、计算出各个裁剪边界的p、q值；
- 3、根据p、q来判断：是舍弃线段还是改变交点的参数。
  - (1) 当 $p<0$ 时，参数r用于更新 $u_1$ ； $(u_1=\max\{u_1, \dots, r_k\})$
  - (2) 当 $p>0$ 时，参数r用于更新 $u_2$ 。 $(u_2=\min\{u_2, \dots, r_k\})$
  - (3) 如果更新了 $u_1$ 或 $u_2$ 后，使 $u_1>u_2$ ，则舍弃该线段。
  - (4) 当 $p=0$ 且 $q<0$ 时，因为线段平行于边界并且位于边界之外，则舍弃该线段。
- 4、p、q的四个值经判断后，如果该线段未被舍弃，则裁剪线段的端点坐标由参数 $u_1$ 和 $u_2$ 的值决定。



$p=0$ 且 $q<0$ 的情况

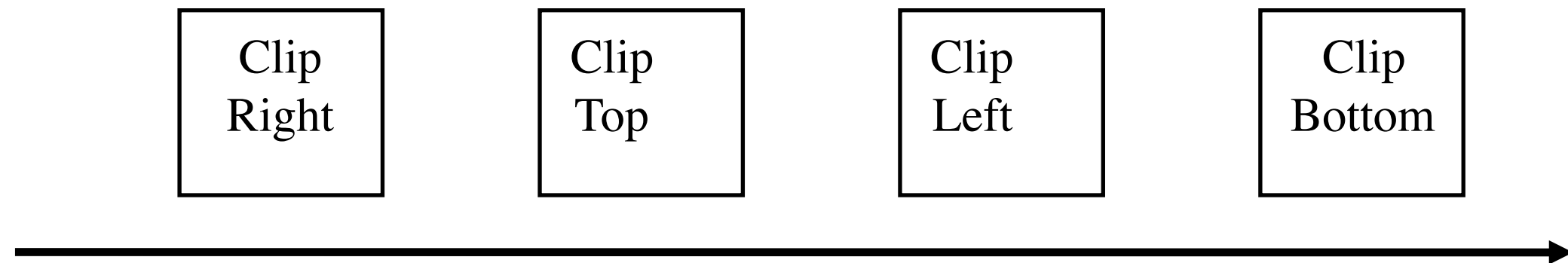
# Sutherland-Hodgeman Polygon-Clipping Algorithm



- Polygons can be clipped against each edge of the window one edge at a time. Window/edge intersections, if any, are easy to find since the X or Y coordinates are already known.
- Vertices which are kept after clipping against one window edge are saved for clipping against the remaining edges.



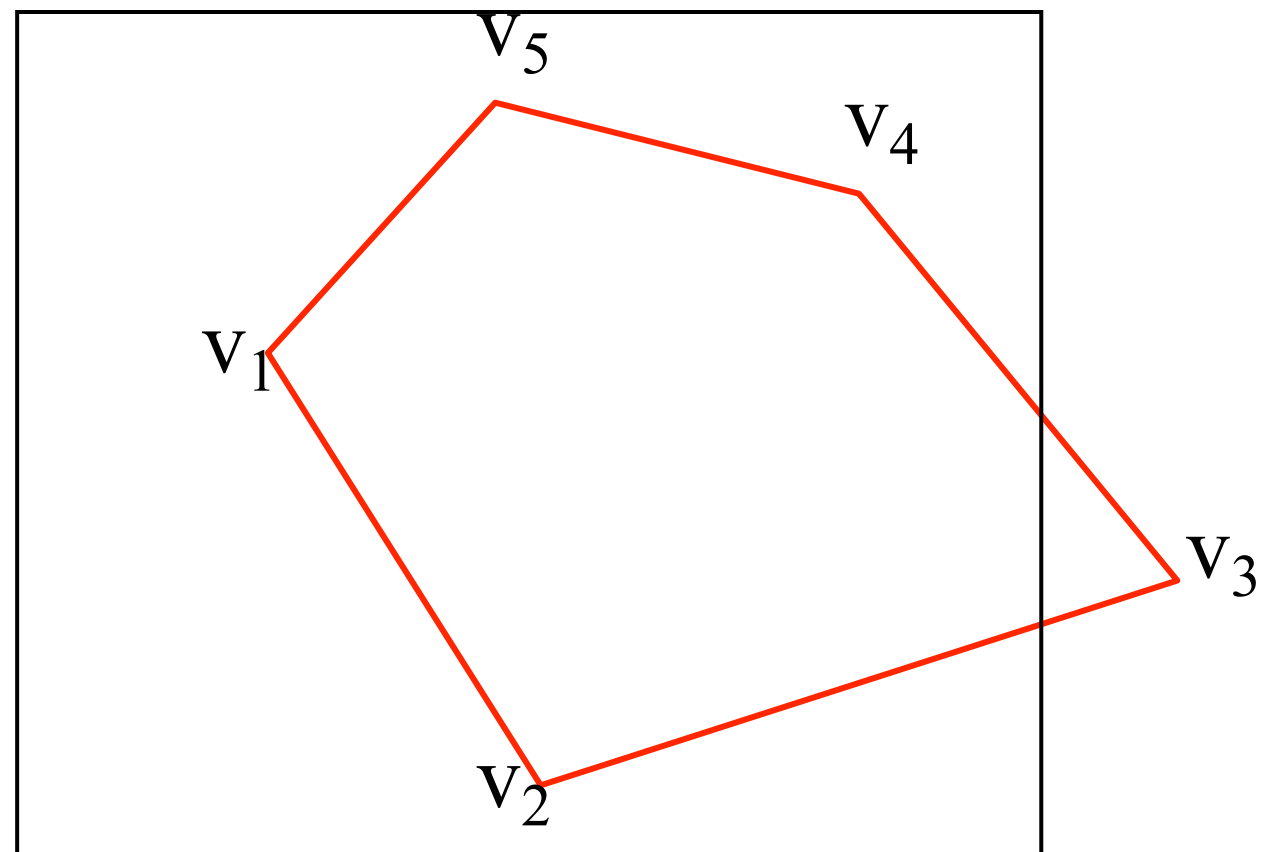
# Pipelined Polygon Clipping



- Because polygon clipping does not depend on any other polygons, it is possible to arrange the clipping stages in a **pipeline**. the input polygon is clipped against one edge and any points that are kept are passed on as input to the next *stage* of the pipeline.
- This way four polygons can be at different *stages* of the clipping process simultaneously. This is often implemented in hardware.

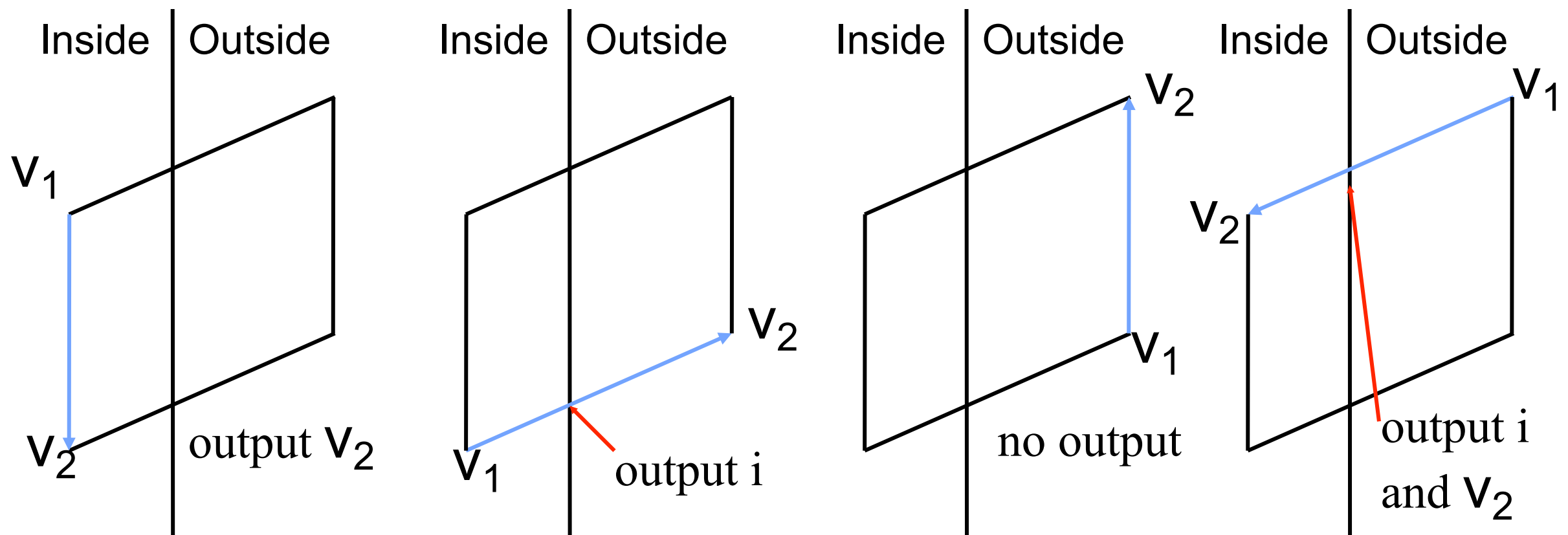
# Sutherland-Hodgeman Polygon Clipping Algorithm

- Polygon clipping is similar to line clipping except we have to keep track of inside/outside relationships
  - Consider a polygon as a list of vertices
  - Note that clipping can increase the number of vertices!
  - Typically clip one edge at a time...



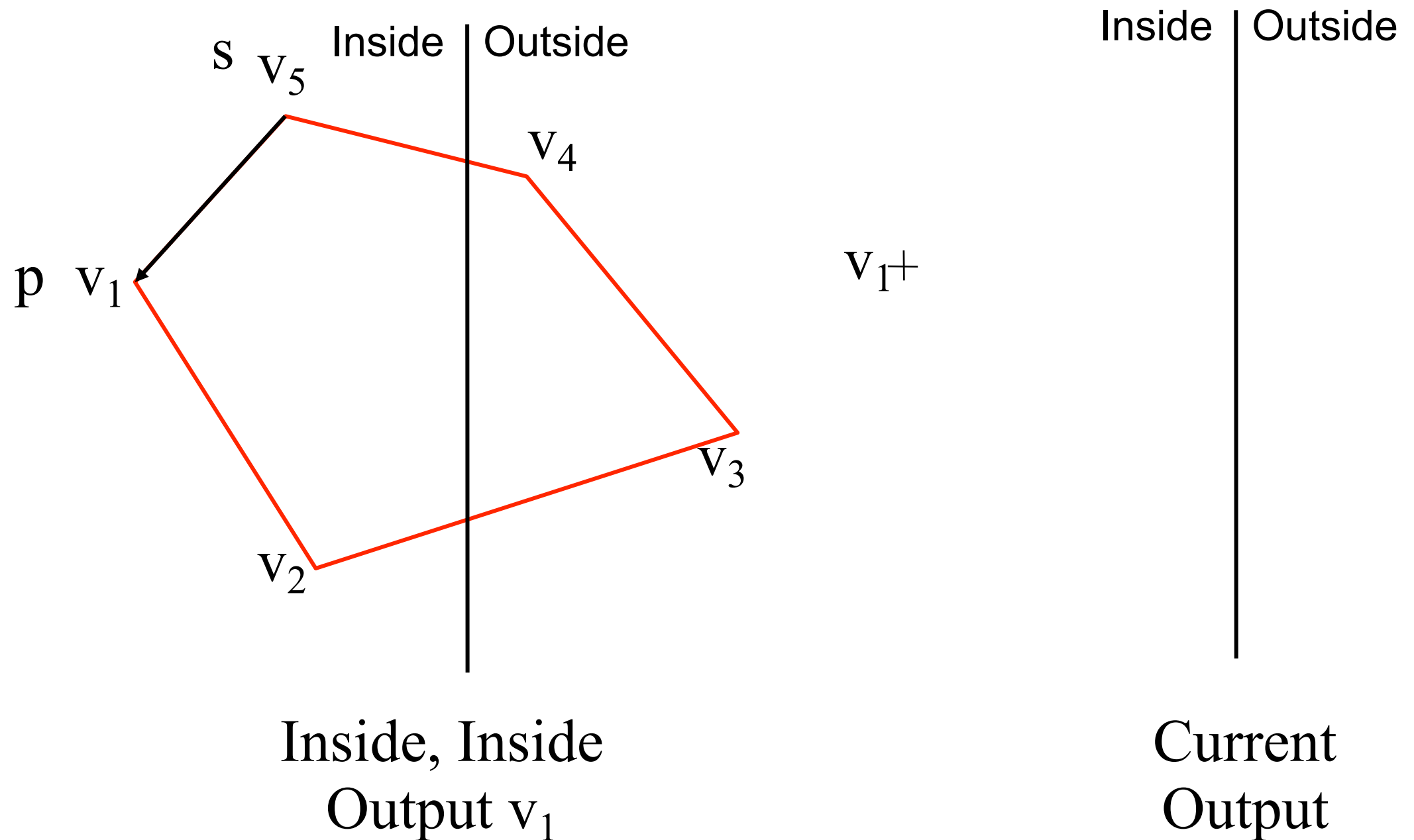
# Sutherland-Hodgeman algorithm

- Present the vertices in pairs
  - $(v_n, v_1), (v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)$
  - For each pair, what are the possibilities?
  - Consider  $v_1, v_2$

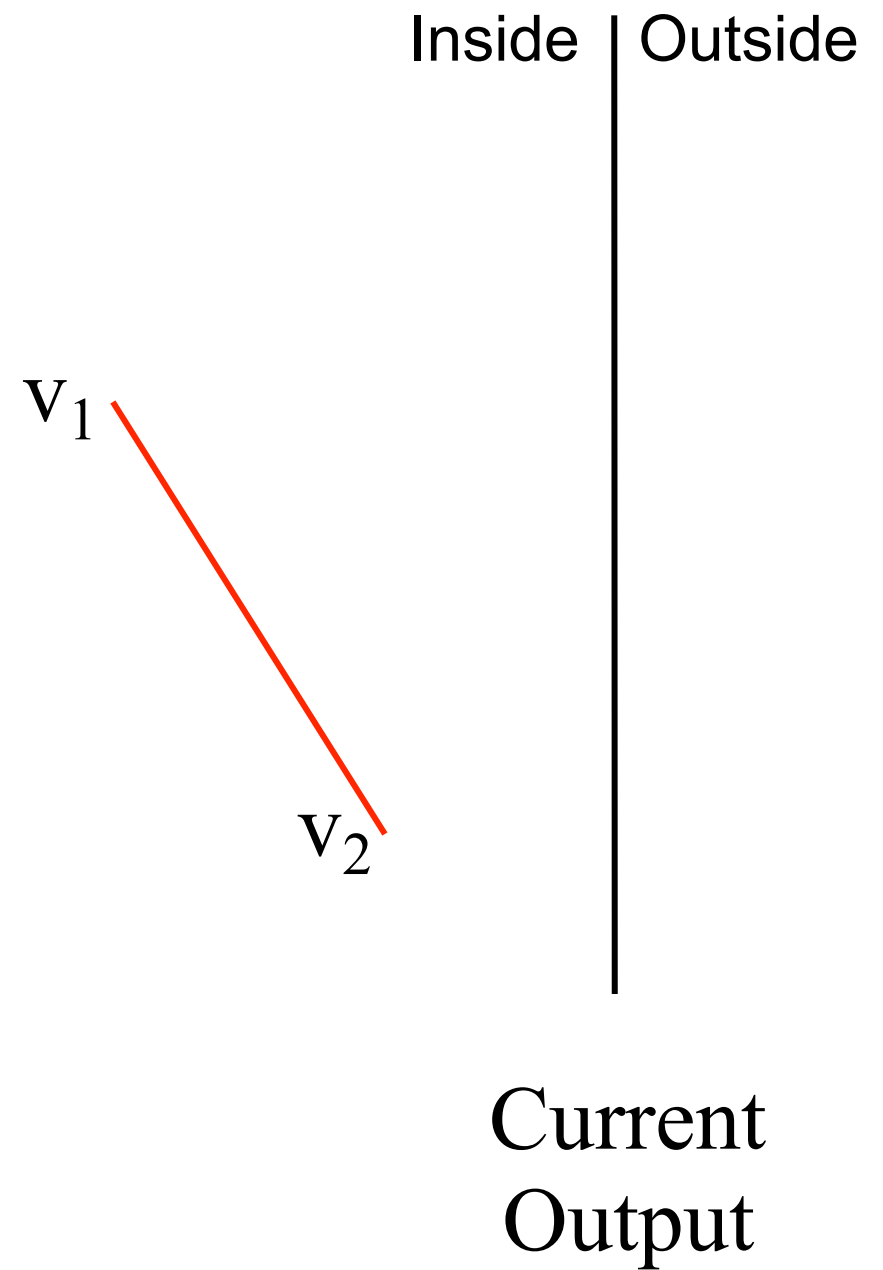
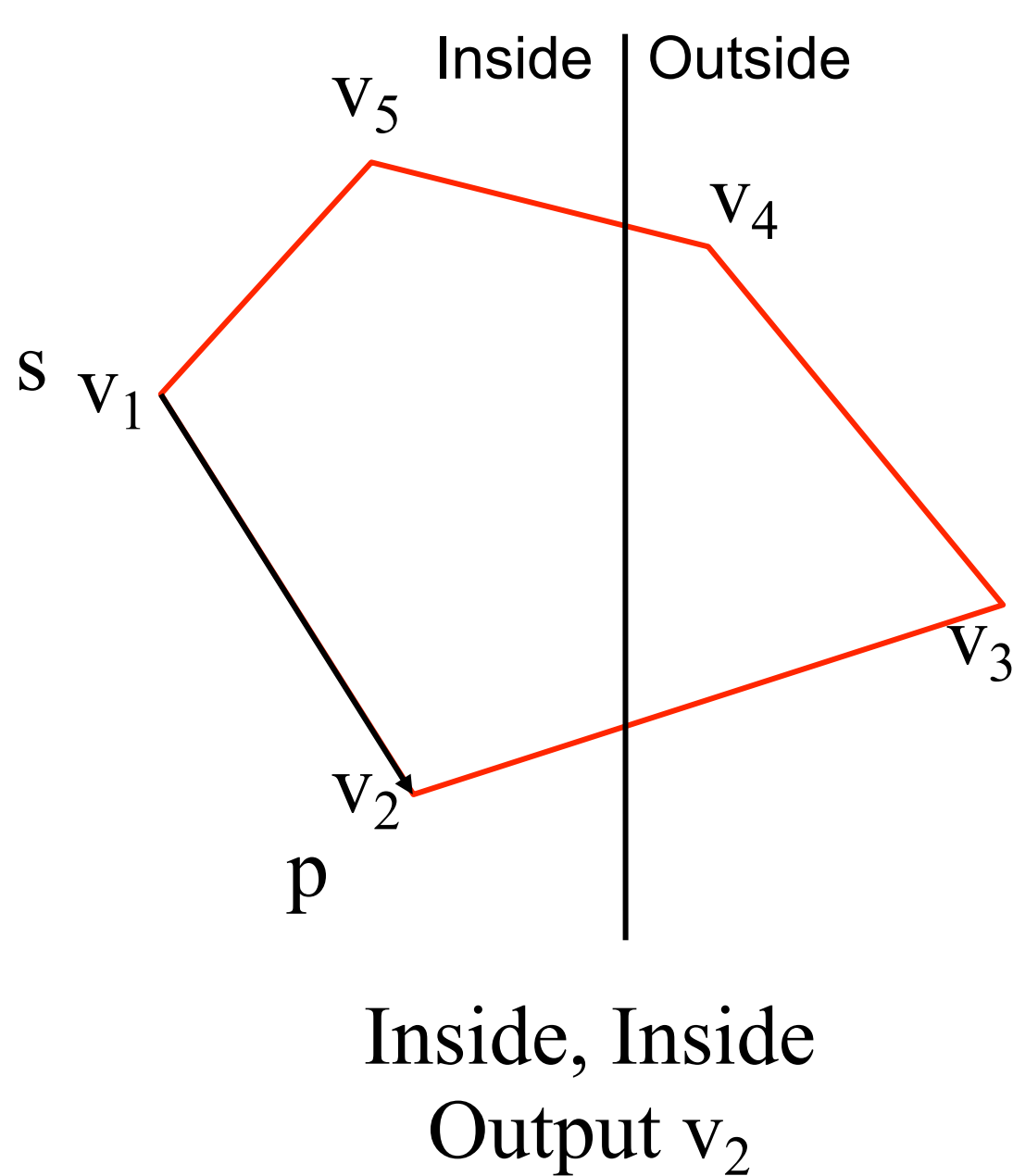


# Example

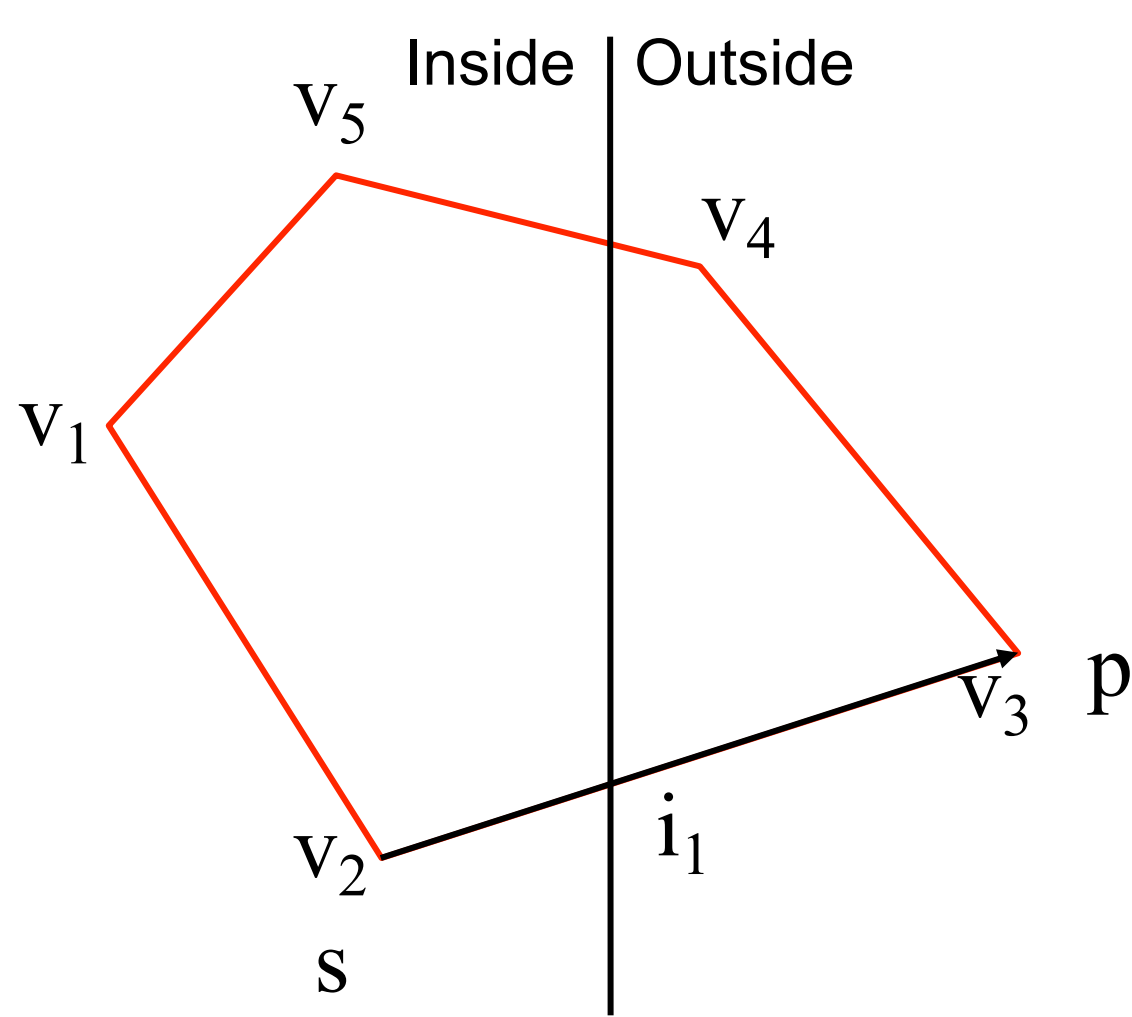
$v_5, v_1$



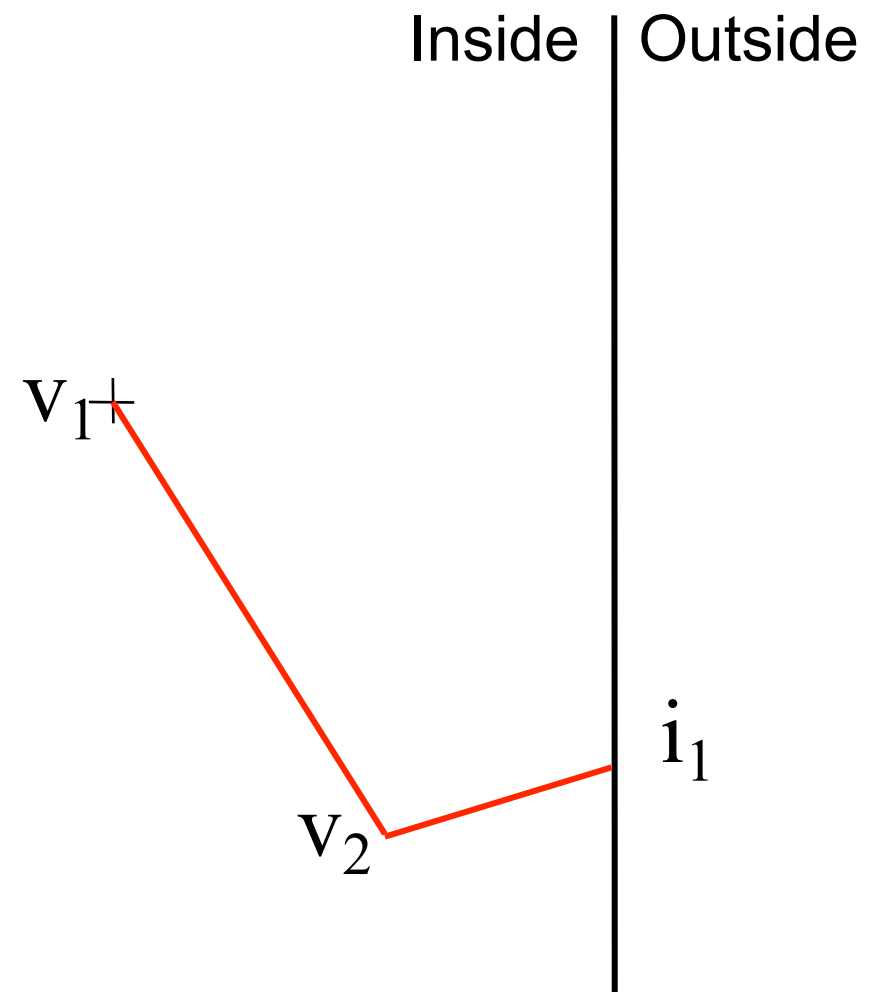
# $V_1, V_2$



# $V_2, V_3$

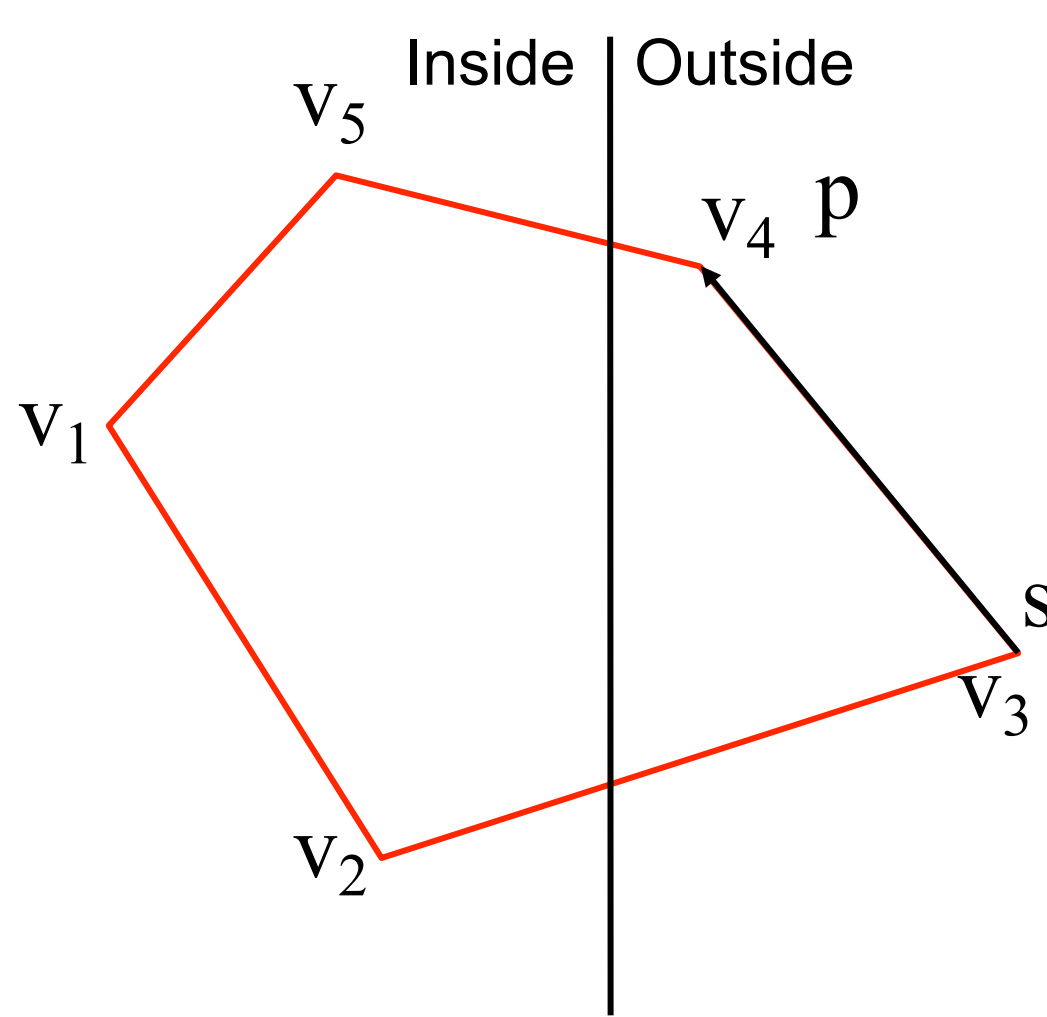


Inside, Outside  
Output  $i_1$

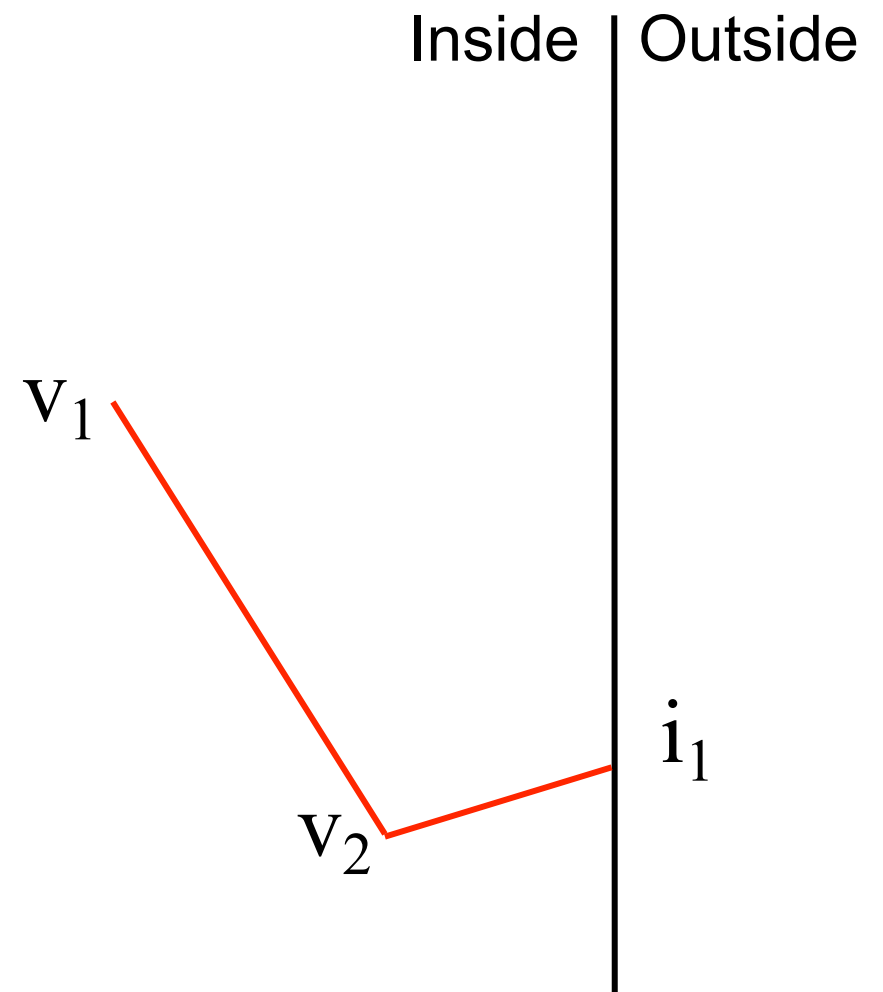


Current  
Output

# $V_3, V_4$



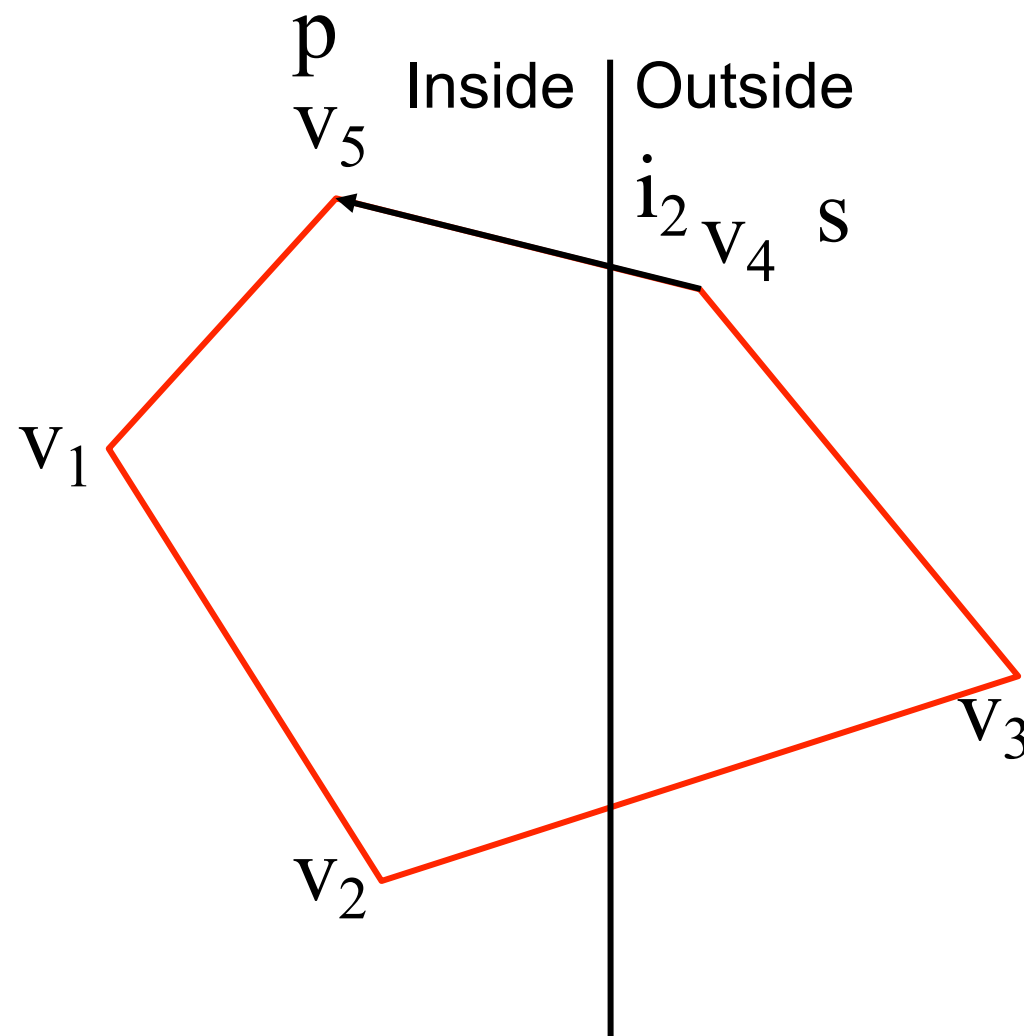
Outside, Outside  
No output



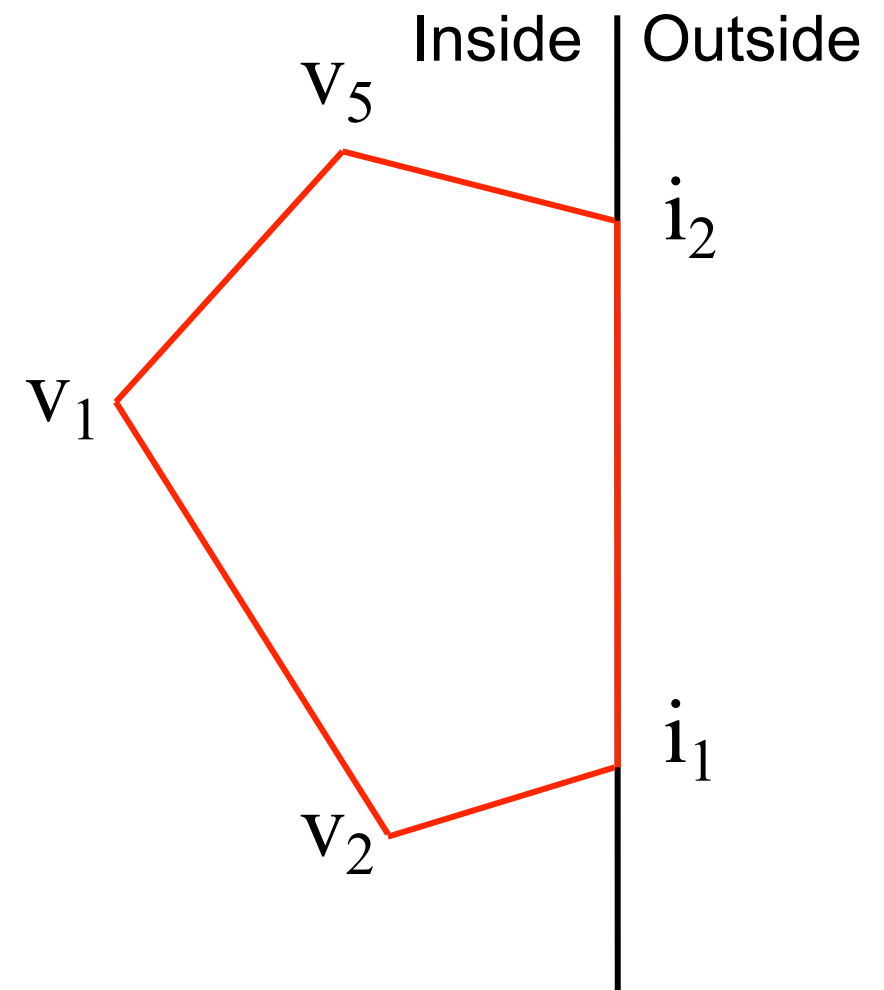
Current  
Output



# $v_4, v_5$ – last edge...



Outside, Inside  
Output  $i_2, v_5$



Current  
Output

# Transforms



# Transformations

- Procedures to compute new positions of objects
- Used to modify objects or to transform (map) from one co-ordinate system to another co-ordinate system

**As all objects are eventually represented using points, it is enough to know how to transform points.**



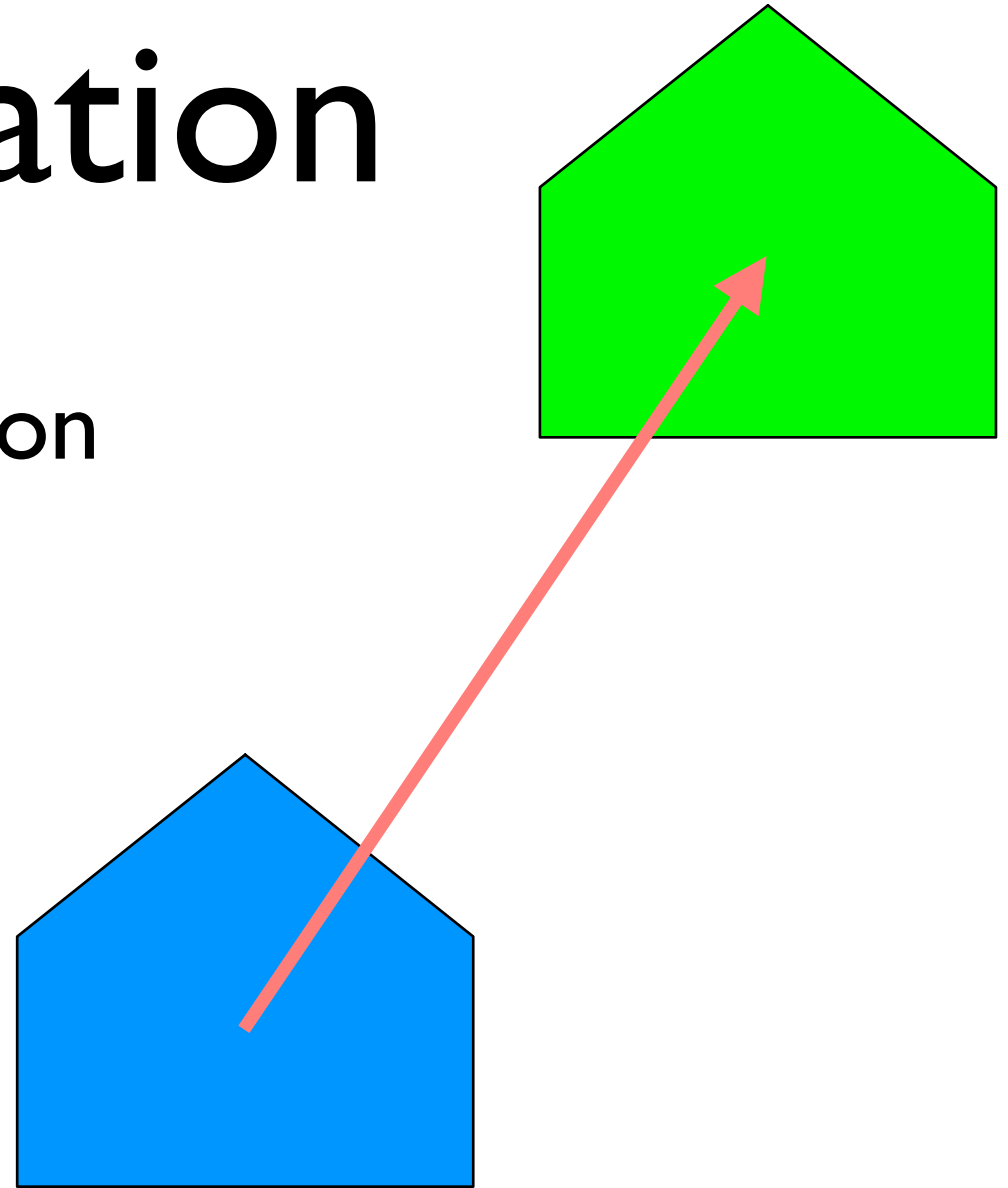
# Translation

- Is a Rigid Body Transformation

$$x \Rightarrow x + T_x$$

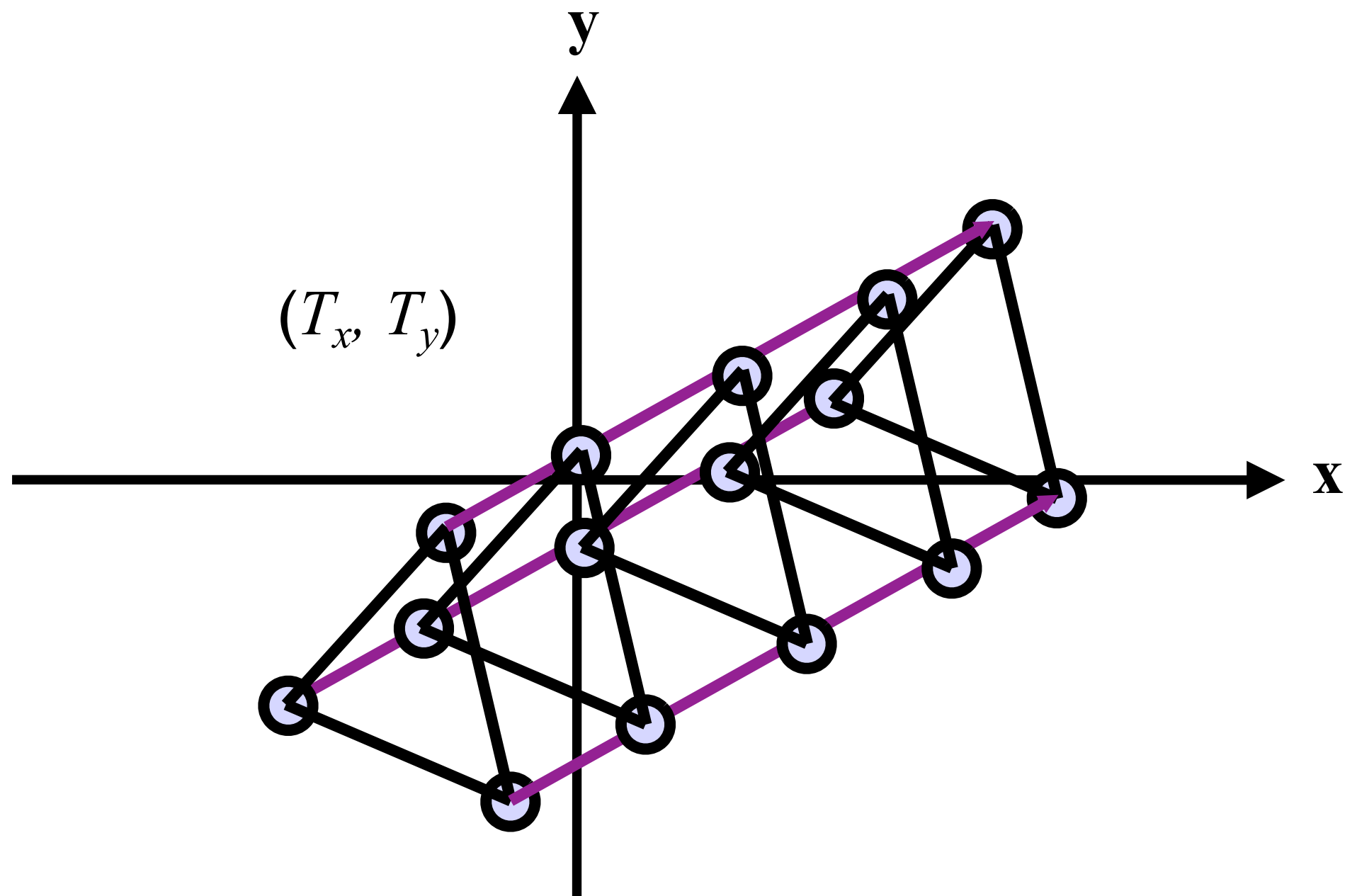
$$y \Rightarrow y + T_y$$

$$z \Rightarrow z + T_z$$



- Translation vector  $(T_x, T_y, T_z)$  or **shift vector**

# Translating an Object



# Scaling

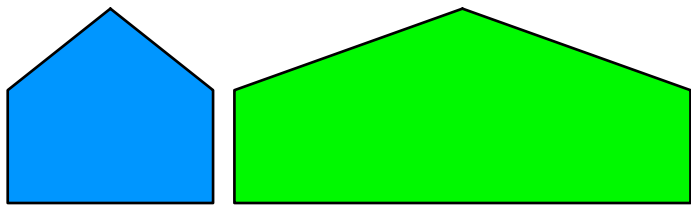
- Changing the size of an object

$$x \Rightarrow x * S_x$$

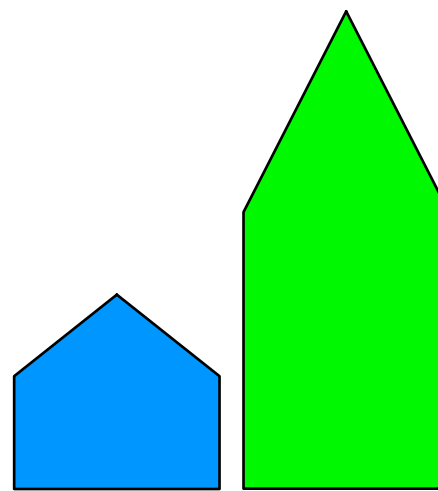
$$y \Rightarrow y * S_y$$

$$z \Rightarrow z * S_z$$

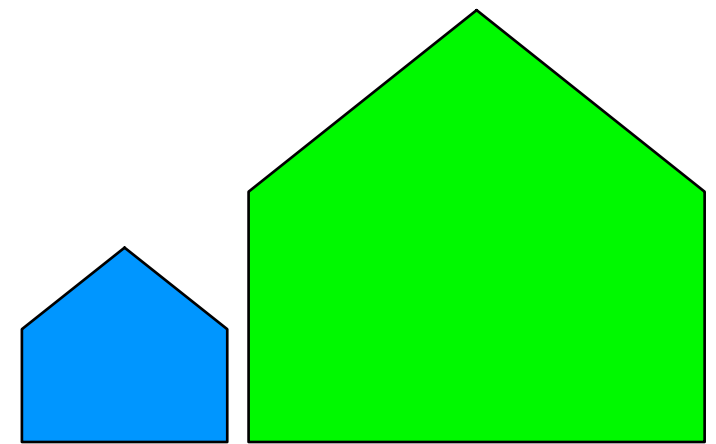
- Scale factor ( $S_x, S_y, S_z$ )



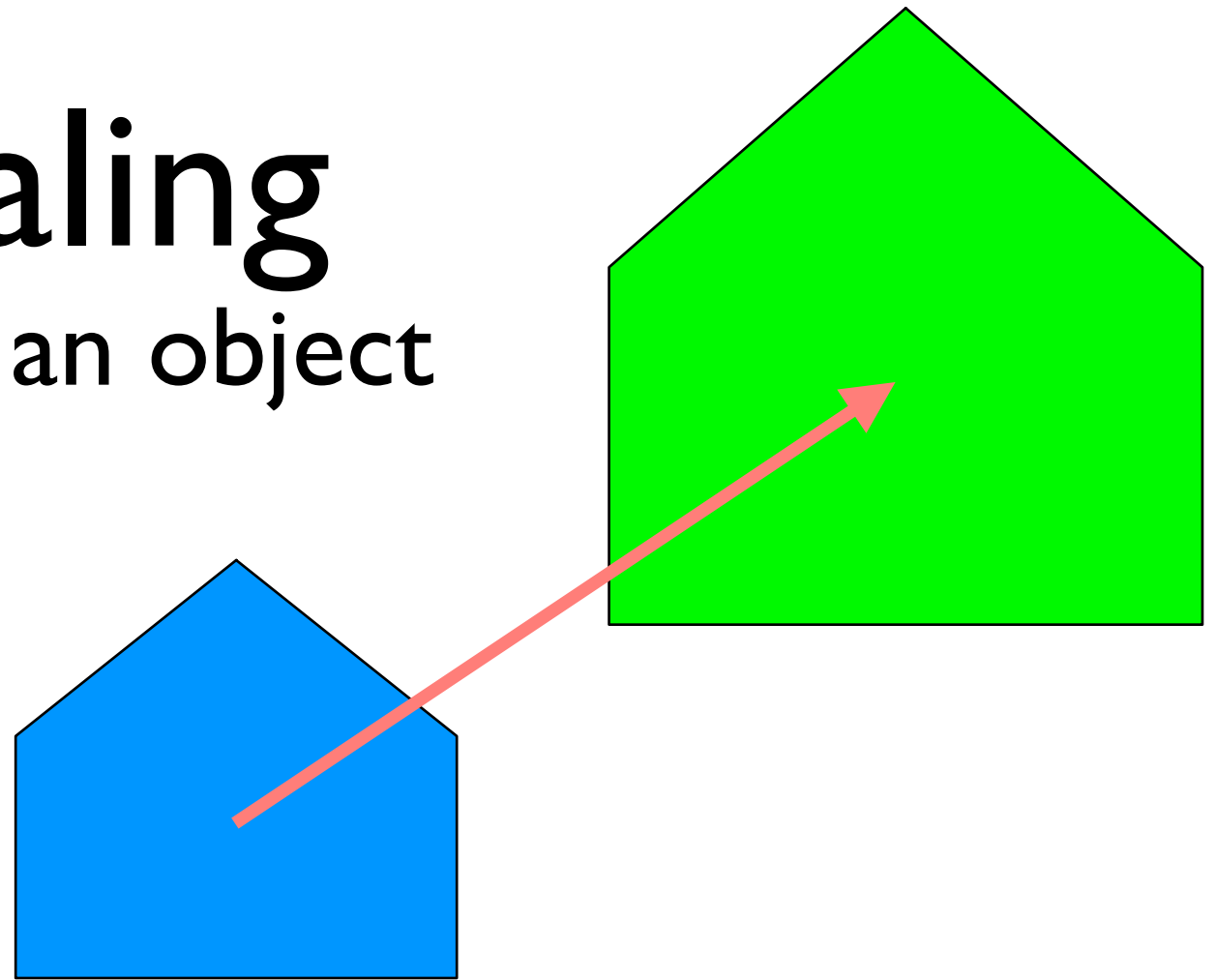
$$S_y = 1$$



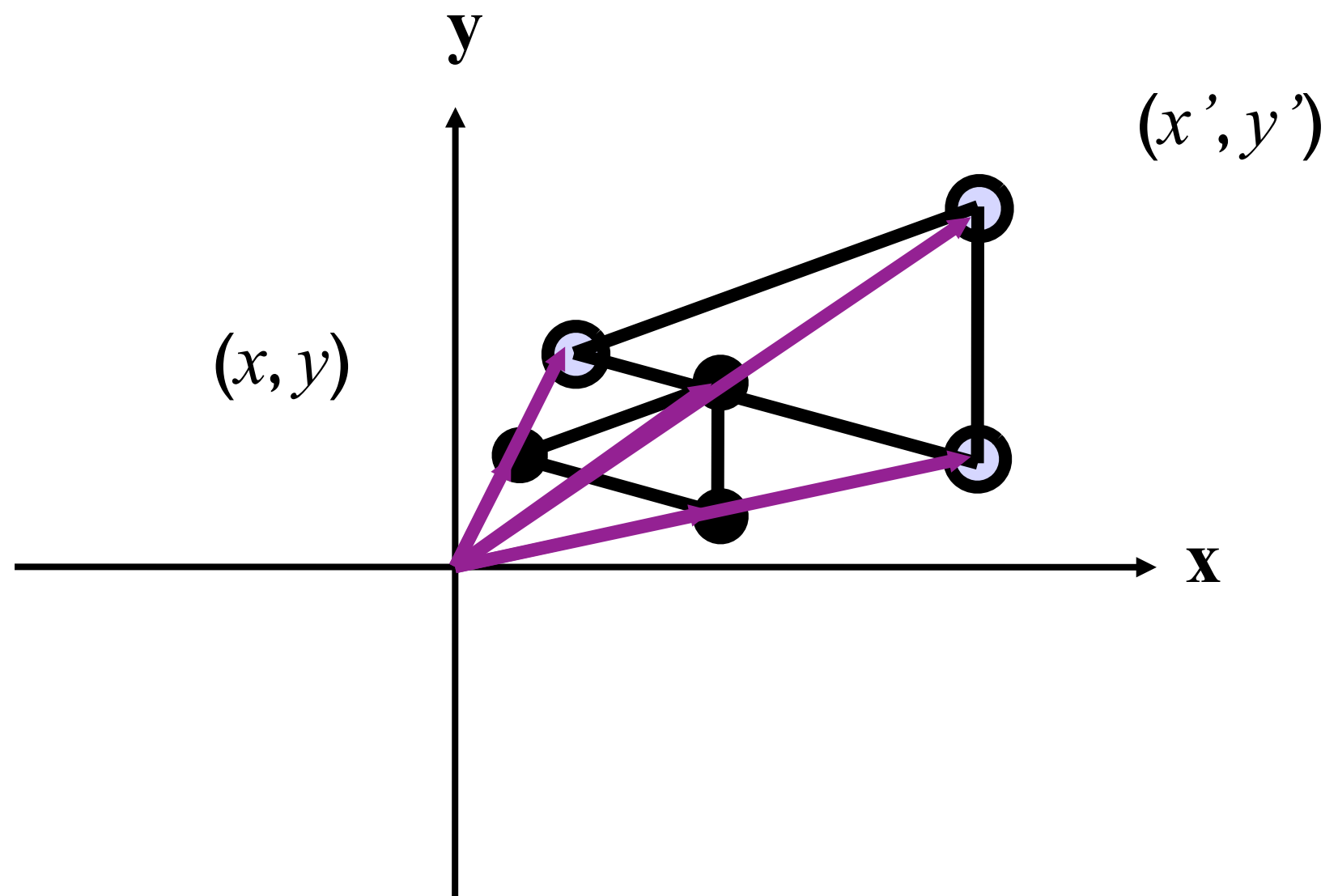
$$S_x = 1$$



$$S_x = S_y$$

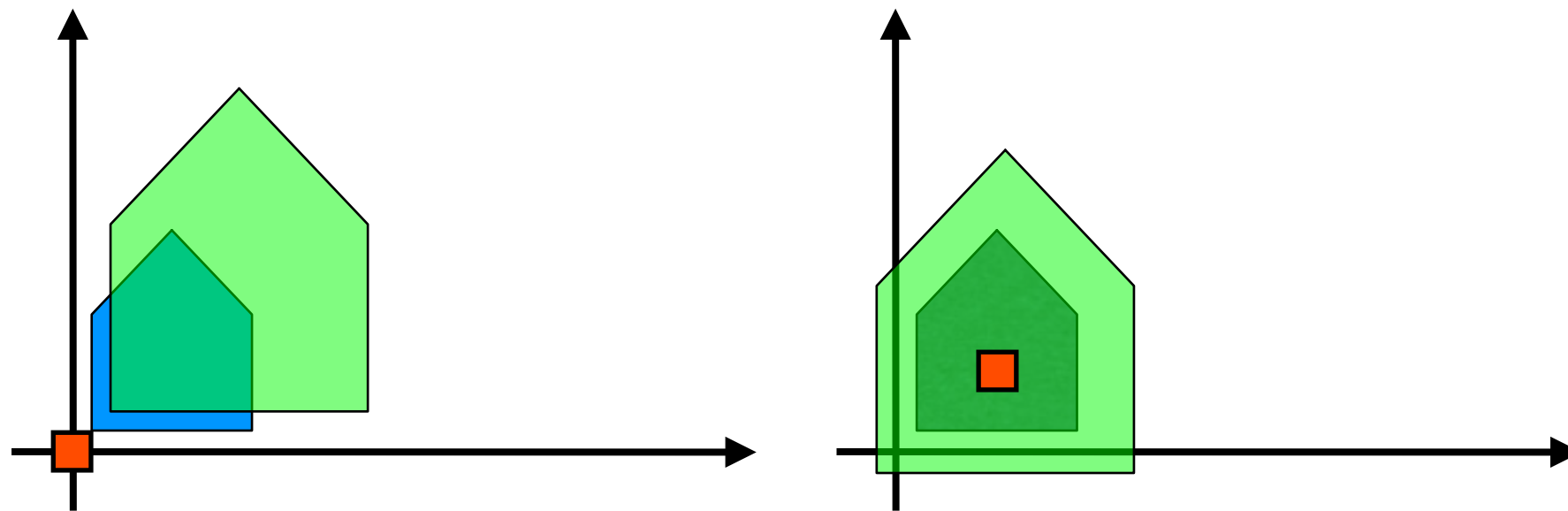


# Scaling an Object



# Scaling (contd.)

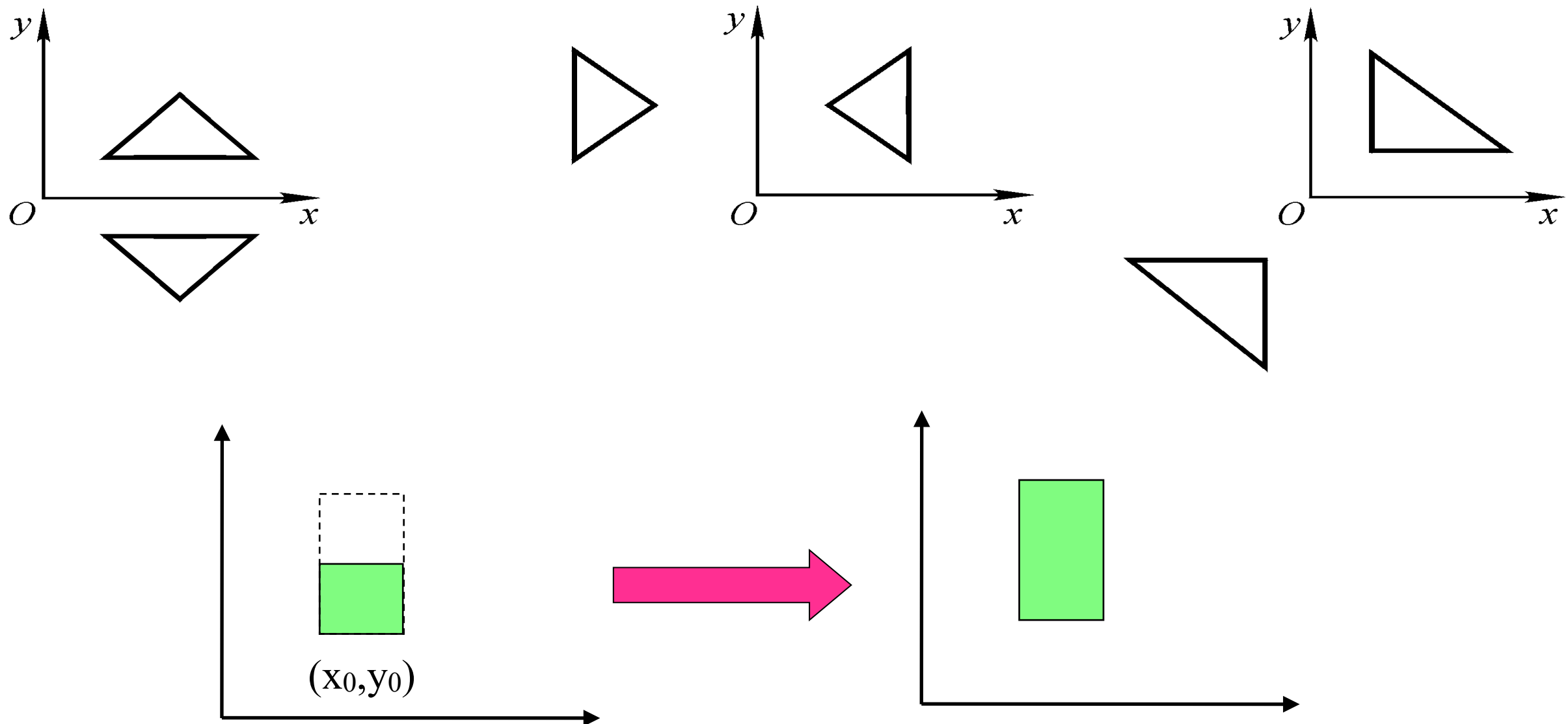
Scaling is always with respect to the origin. The origin does not move.



Scaling wrt a reference point can be achieved as a **composite transformation**



# Scaling an Object



***Question 8 : How ?***

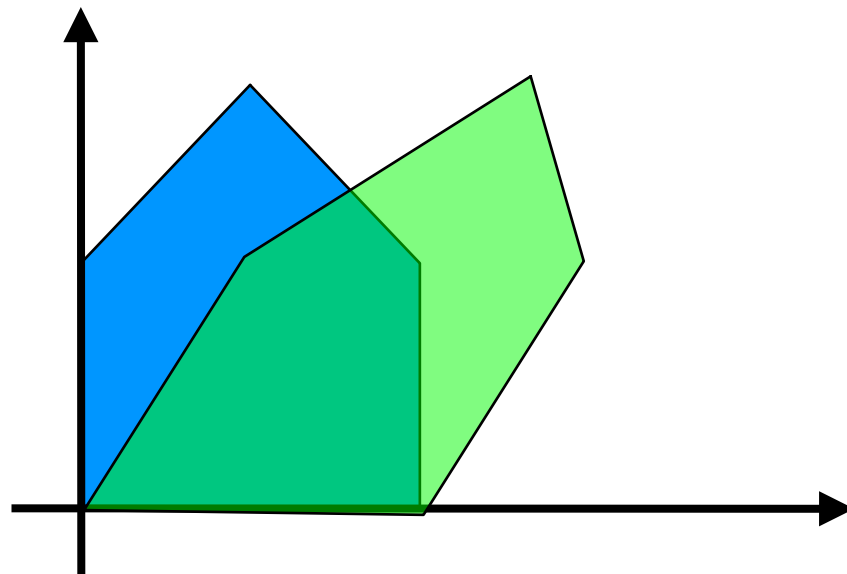
# Shearing

- Produces shape distortions
- Shearing in x-direction

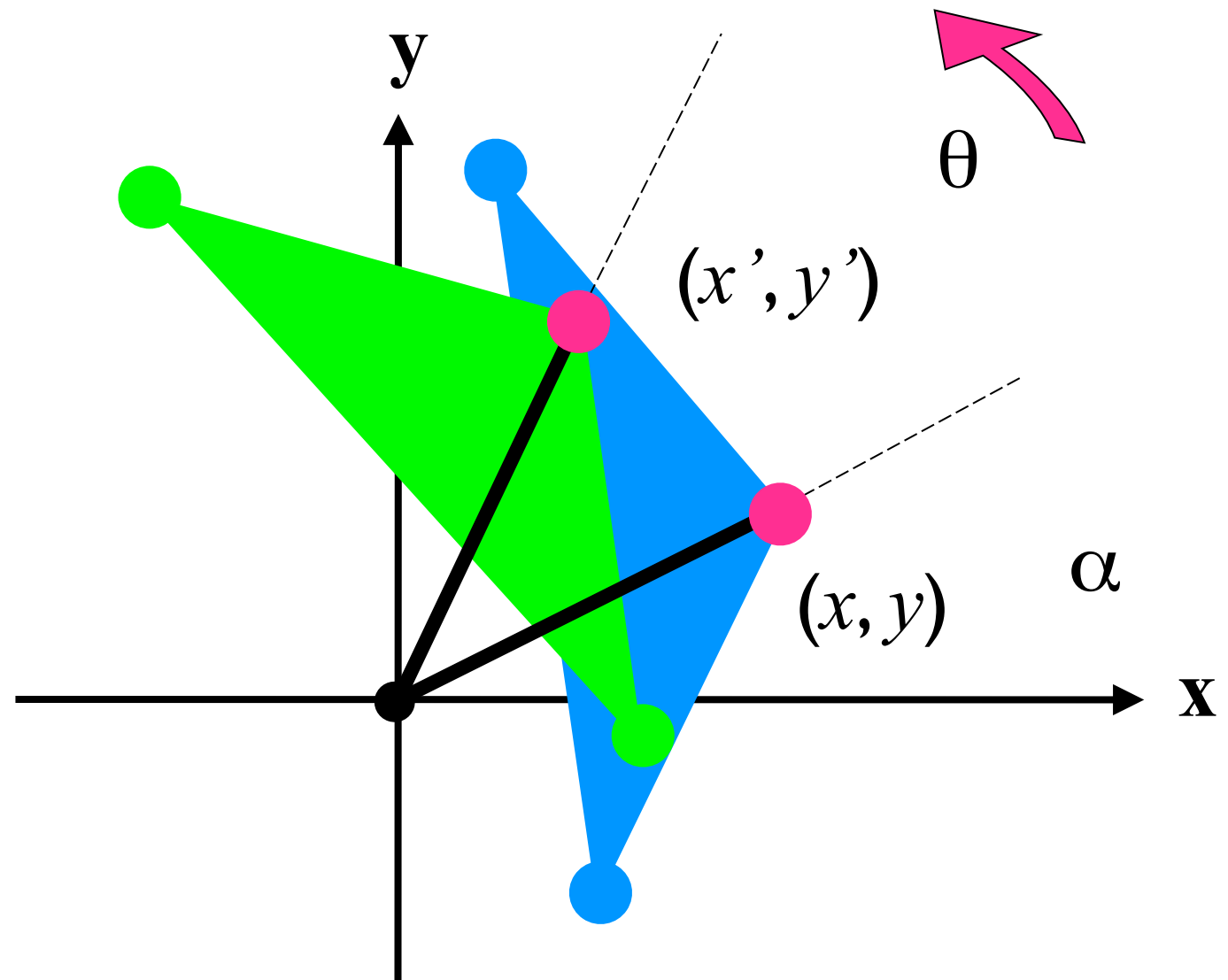
$$x \Rightarrow x + a * y$$

$$y \Rightarrow y$$

$$z \Rightarrow z$$



# Rotation



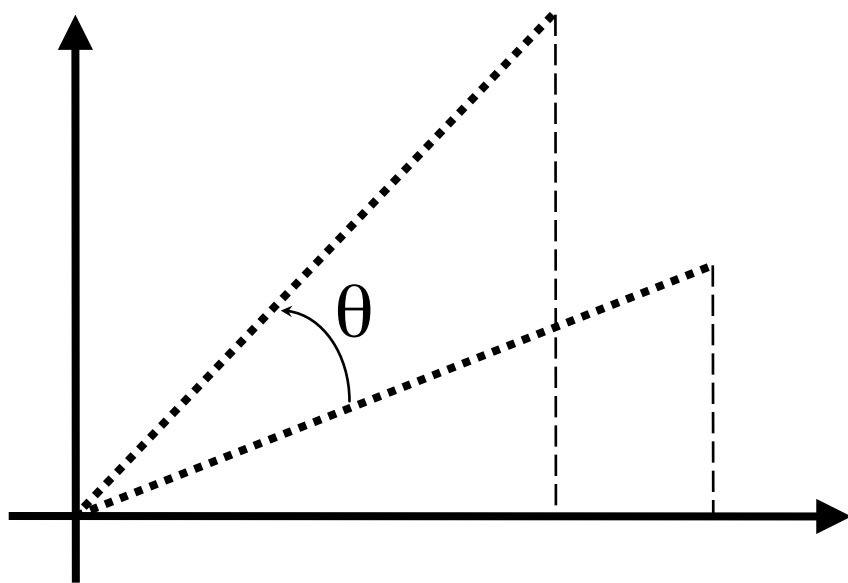
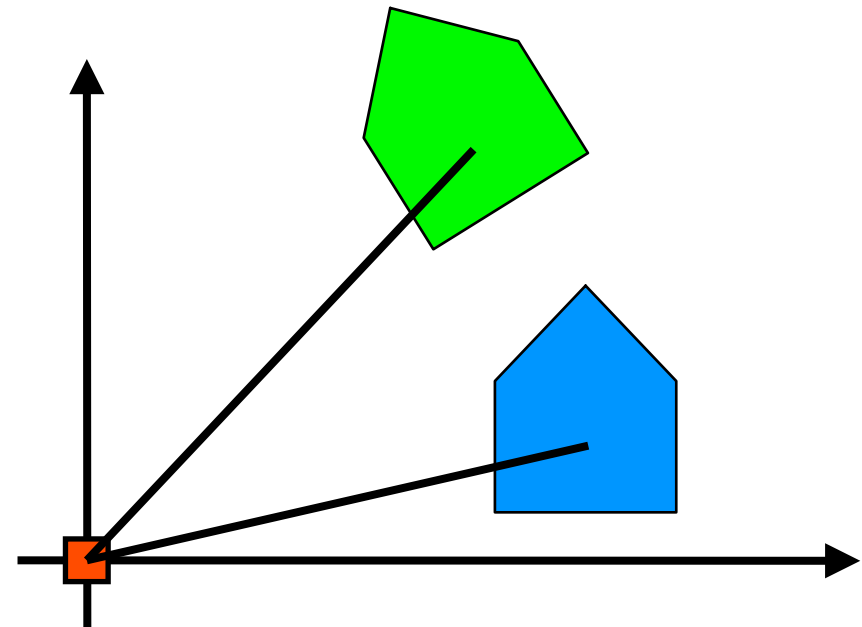
# Rotation

- Is a Rigid Body Transformation

$$x' = x \cdot \cos(\theta) - y \cdot \sin(\theta)$$

$$y' = x \cdot \sin(\theta) + y \cdot \cos(\theta)$$

$$z' = z$$



$$\begin{aligned} x' &= r \cdot \cos(\alpha + \theta) \\ &= r \cos \alpha \cos \theta - r \sin \alpha \sin \theta \\ &= x \cos \theta - y \sin \theta \end{aligned}$$

# Rotation (contd.)

- Rotation also is wrt to a reference -
  - A Reference Line in 3D
  - A Reference Point in 2D
- Define 2D rotation about arbitrary point

## *Rotate around $(x_r, y_r)$*

$$newx = x - x_r$$

$$newy = y - y_r$$

$$newx' = newx \cos\theta - newy \sin\theta$$

$$newy' = newy \cos\theta + newx \sin\theta$$

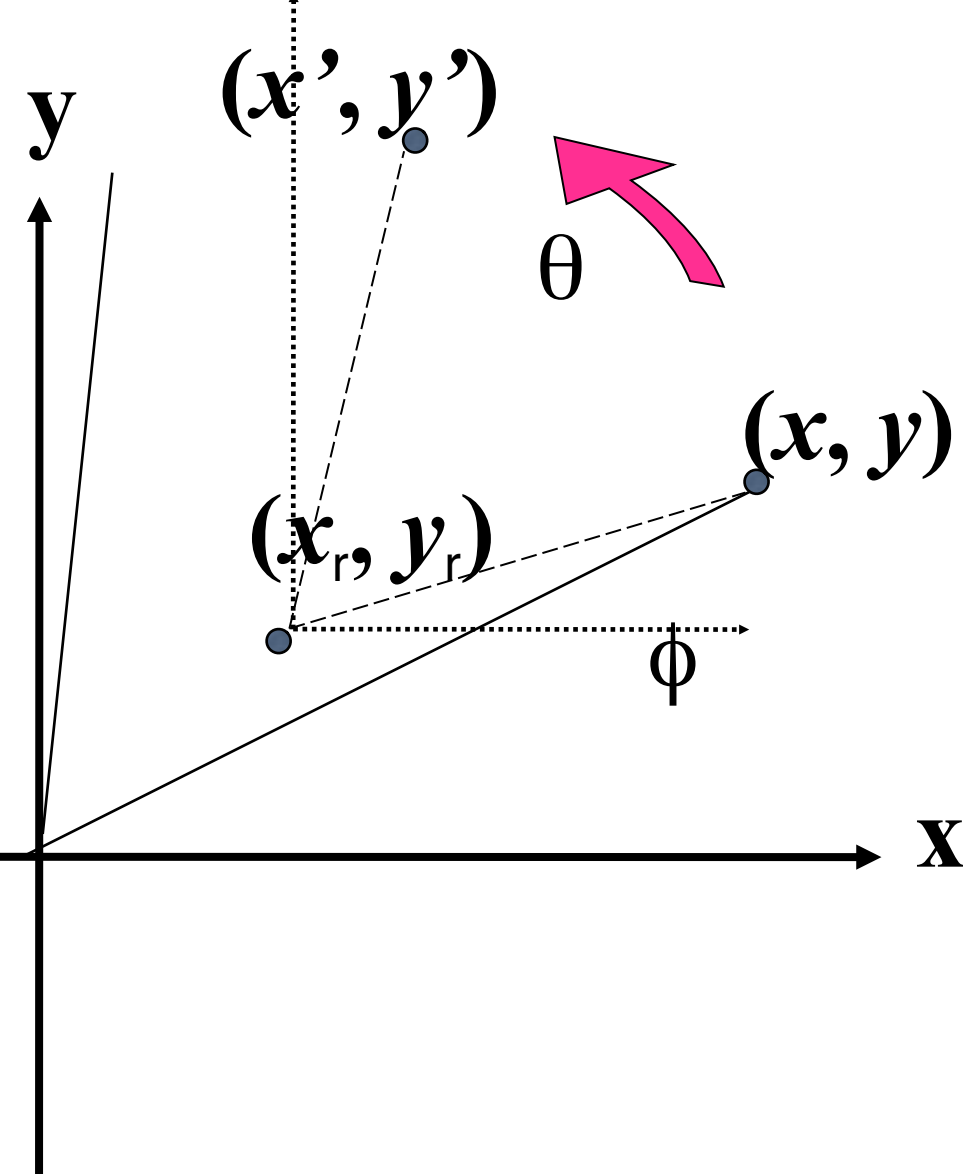
$$x' = newx' + x_r$$

$$y' = newy' + y_r$$



$$x' = x_r + (x - x_r) \cos\theta - (y - y_r) \sin\theta$$

$$y' = y_r + (y - y_r) \cos\theta + (x - x_r) \sin\theta$$



# General Linear Transformation

$$\begin{aligned}x &= a*x + b*y + c*z \\y &= d*x + e*y + f*z \\z &= g*x + h*y + i*z\end{aligned} \quad \text{or} \quad \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

- Which of the following can be represented in this form?
  - Translation
  - Scaling
  - Rotation