Computer Graphics 2016

5. Geometry and Transform

Hongxin Zhang
State Key Lab of CAD&CG, Zhejiang University

2016-10-24

1024
节日快乐！

Today outline

- Triangle rasterization

  - Basic vector algebra ~ geometry

- Antialiasing revisit
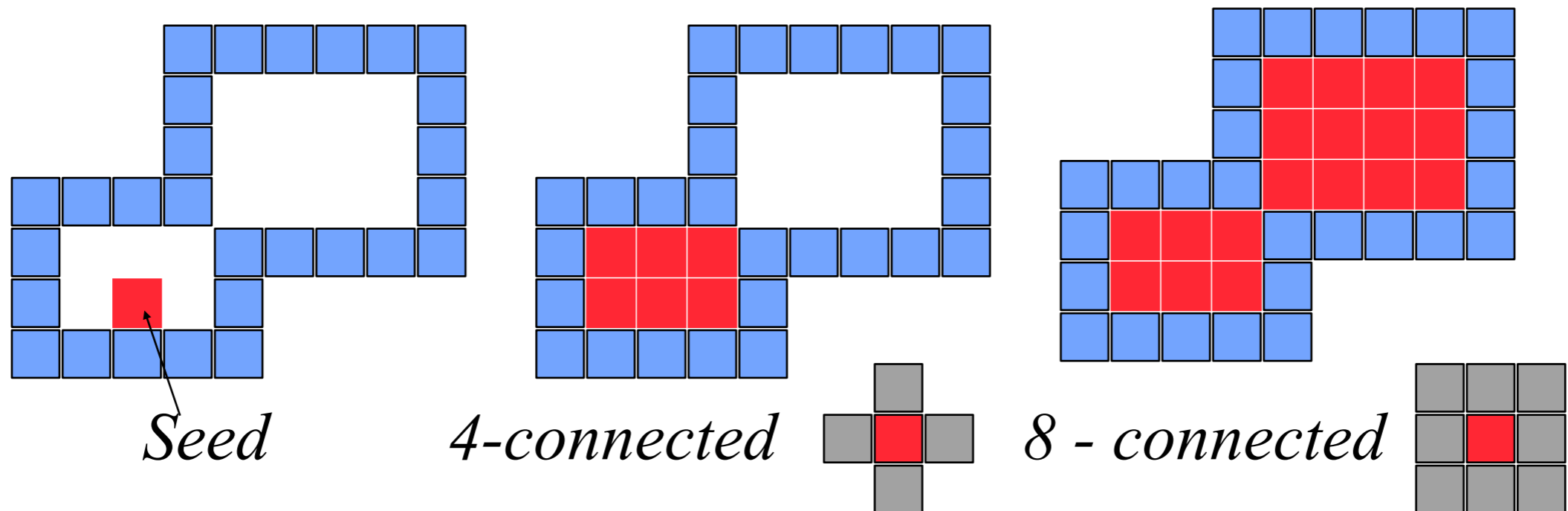
- Clipping

- Transforms (I)

Previous lesson

- Primitive attributes

- Rasterization and scan line algorithm

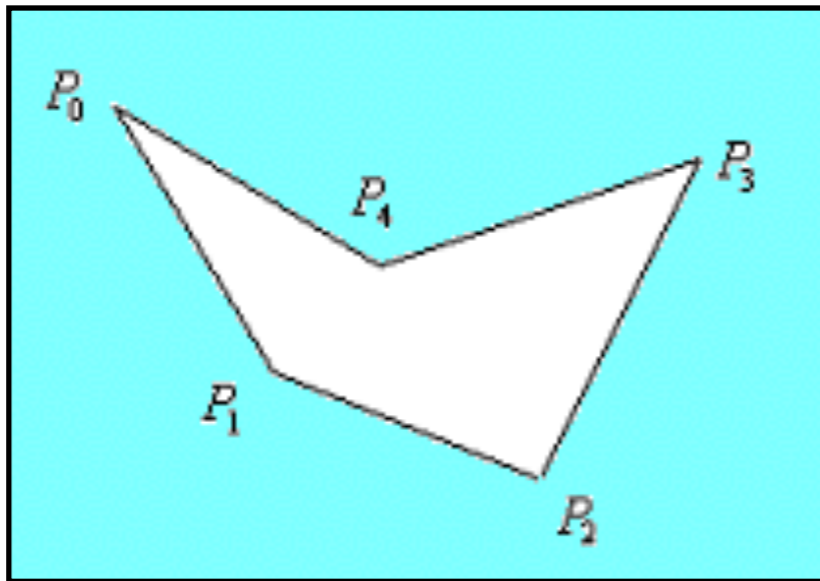 - line,

  - general polygon

Seed Fill Algorithms

- Assumes that at least one pixel interior to the polygon is known

- It is a recursive algorithm
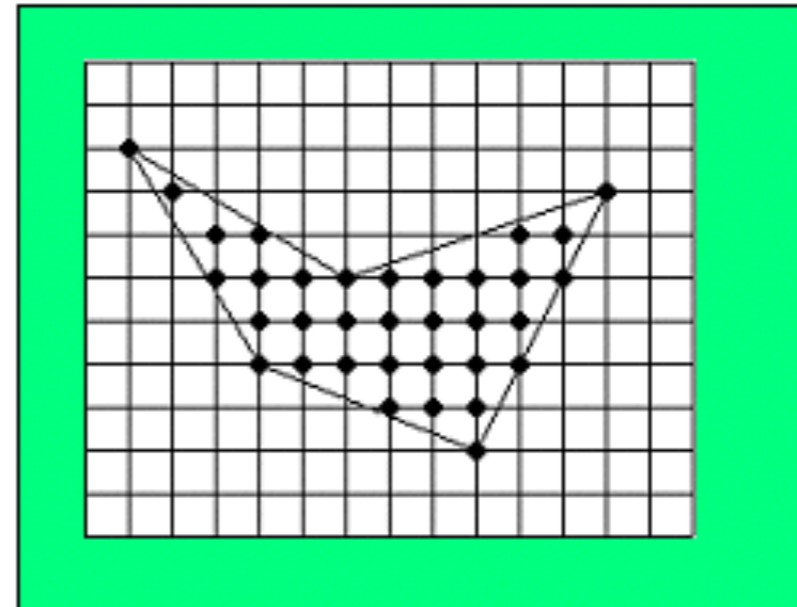
- Useful in interactive paint packages

Seed   *4-connected*   *8 - connected*

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
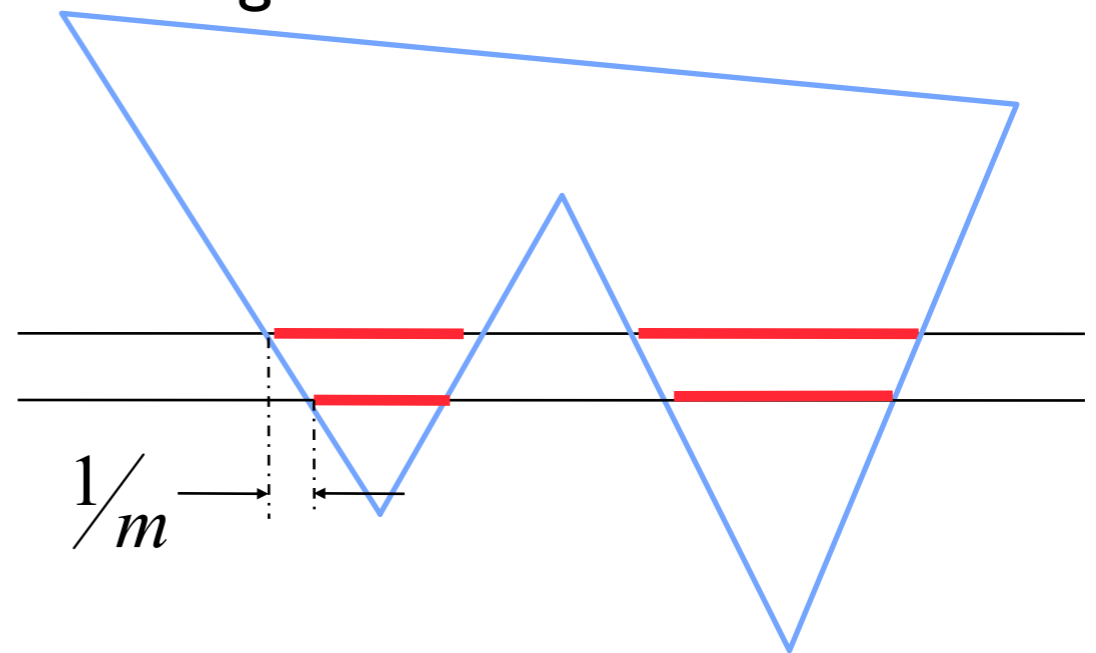
From top to down

http://www.cecs.csulb.edu/~pnguyen/cecs449/lectures/fillalgorithm.pdf
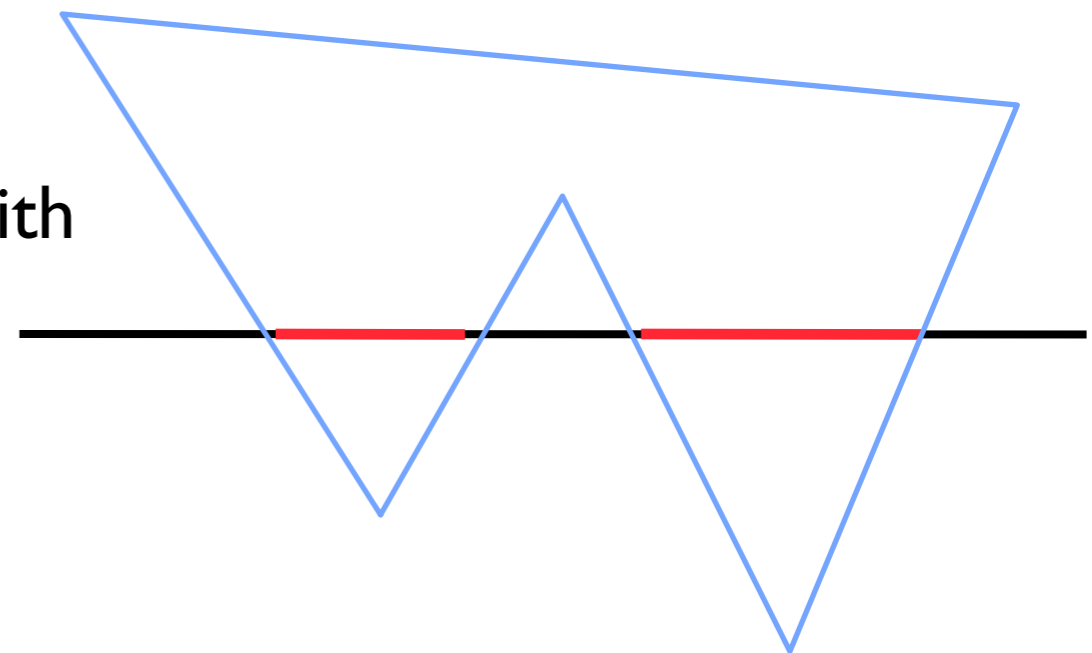
Efficiency Issues in Scan Line Method

- 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 proceeding scan line as

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

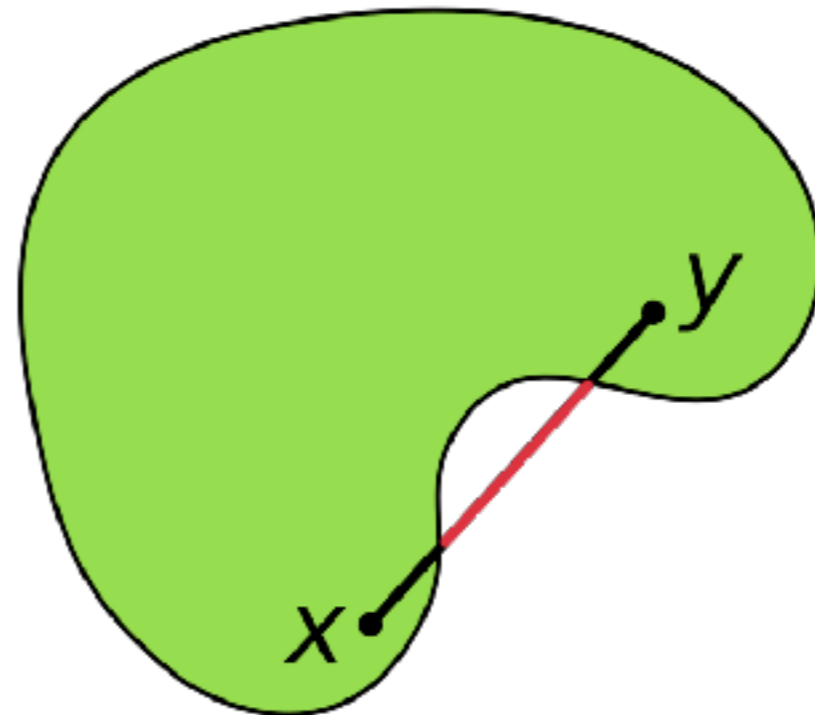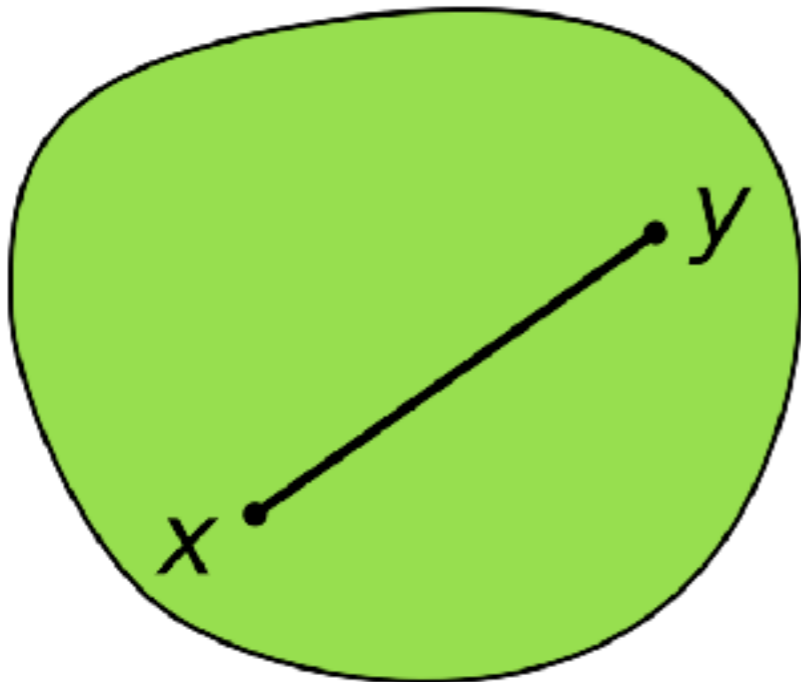- List of active edges could be maintained to increase efficiency

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**,

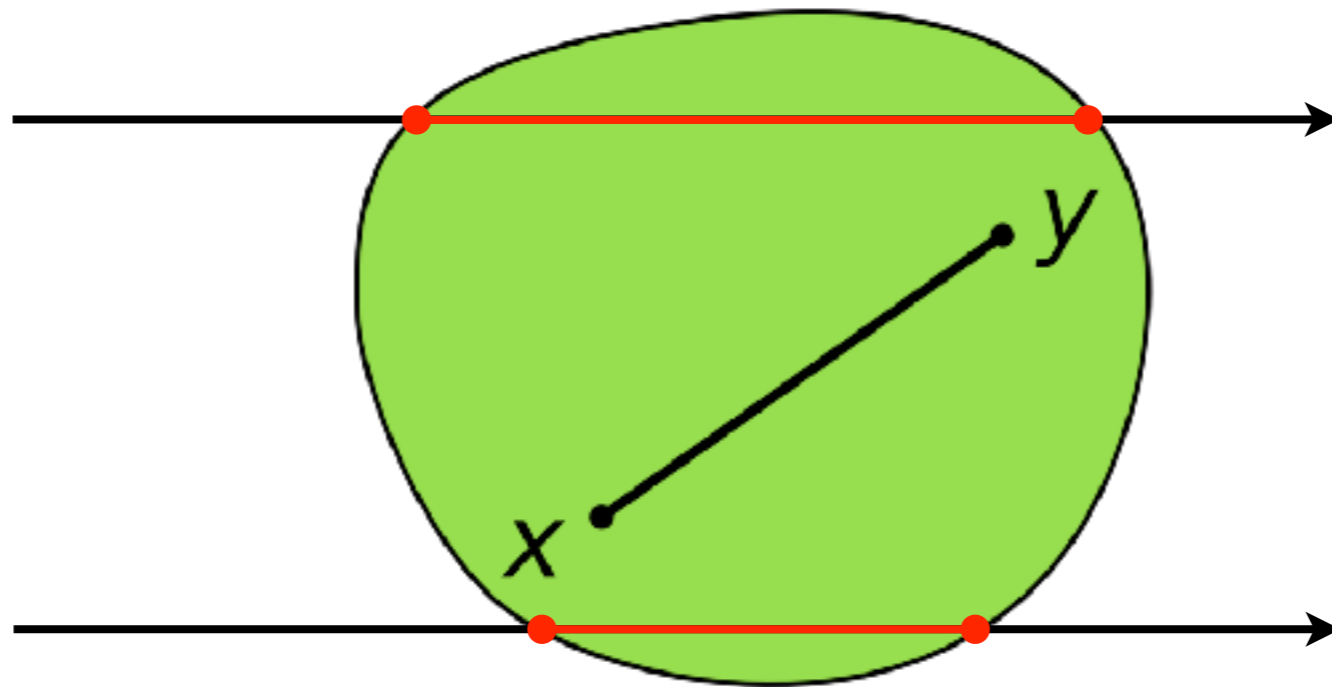- much better if they are **only triangles**

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
$$(1 - t)\,x + t\,y$$
is in C.

Convex polygon rasterization
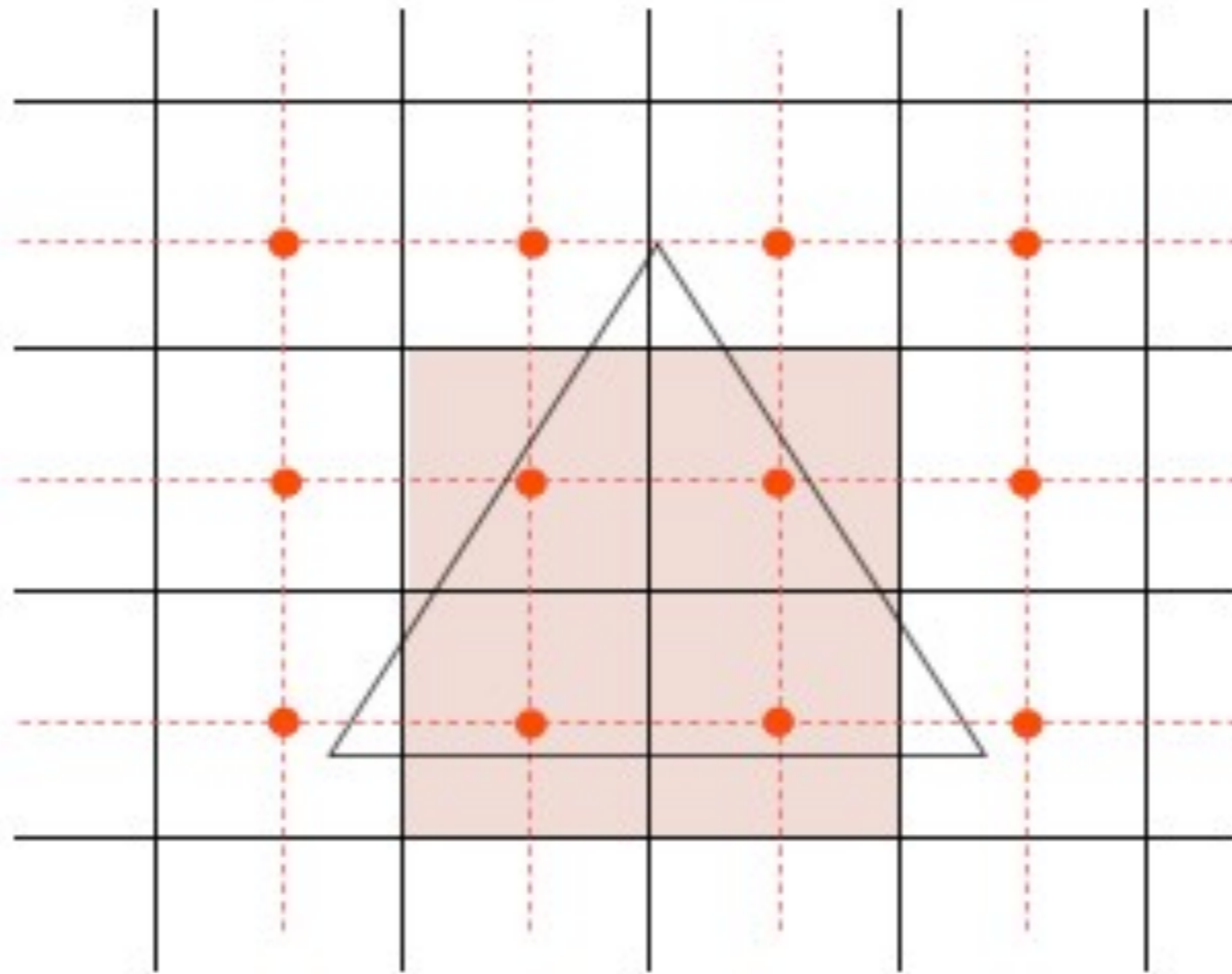

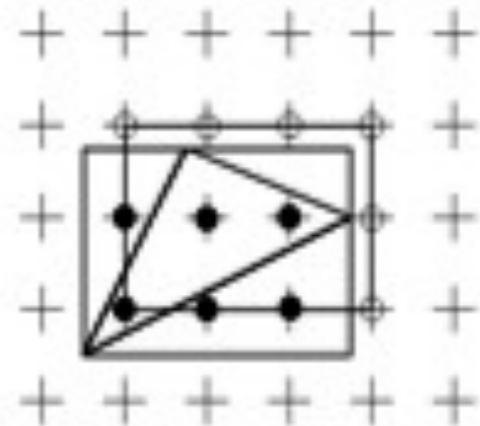One in and one out

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

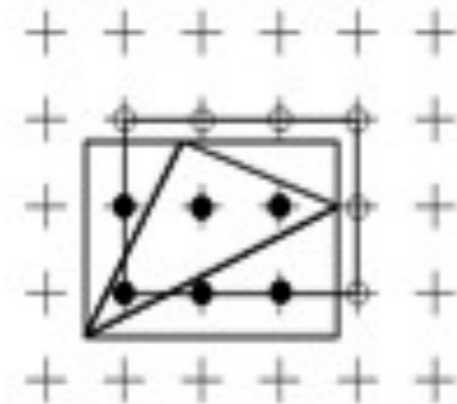# Compute Bounding Box

```
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 10, 11, 12;

    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);
        }
    }
}
```
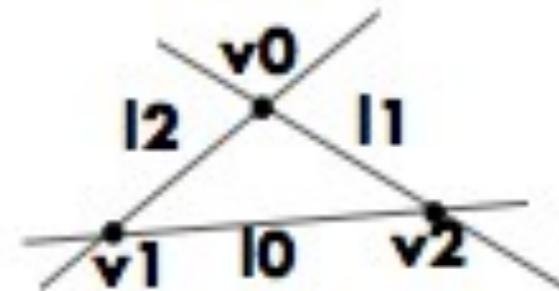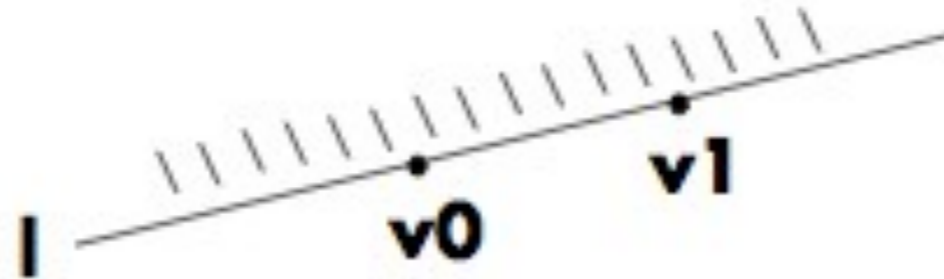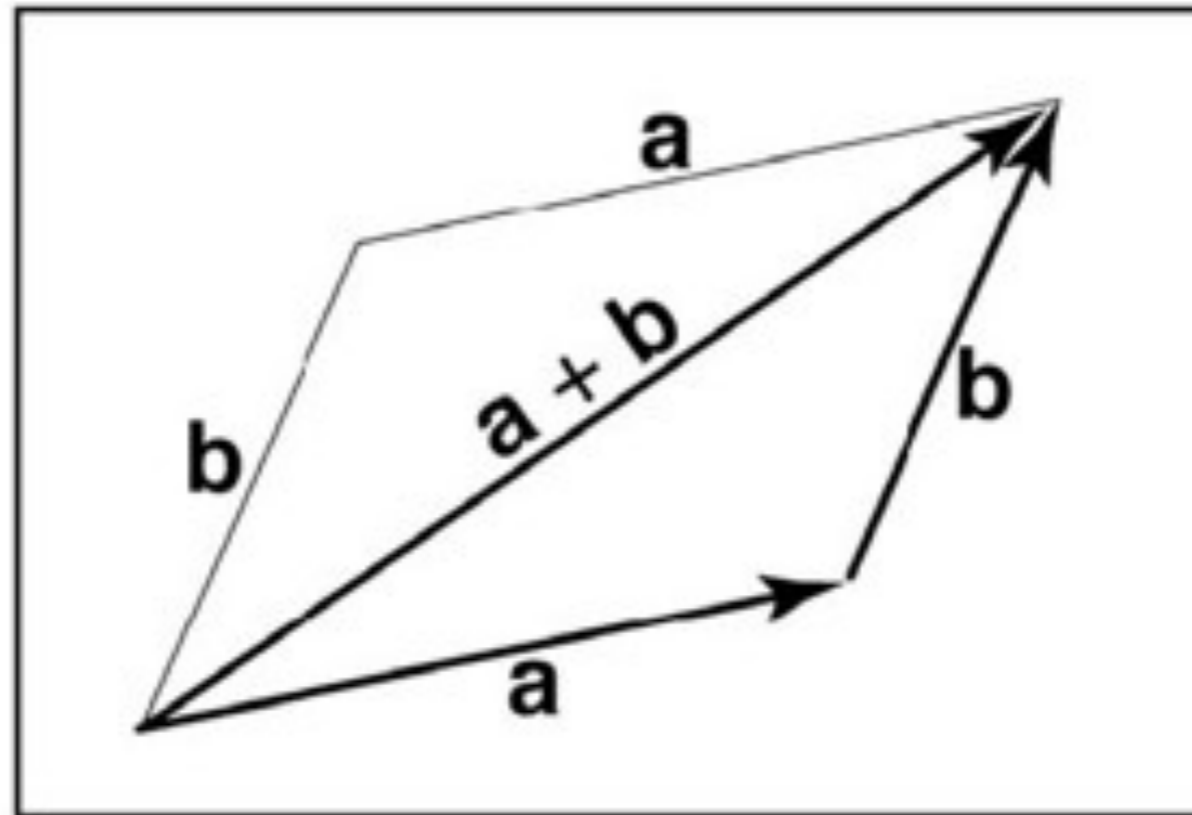
Line equation

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);
}
```
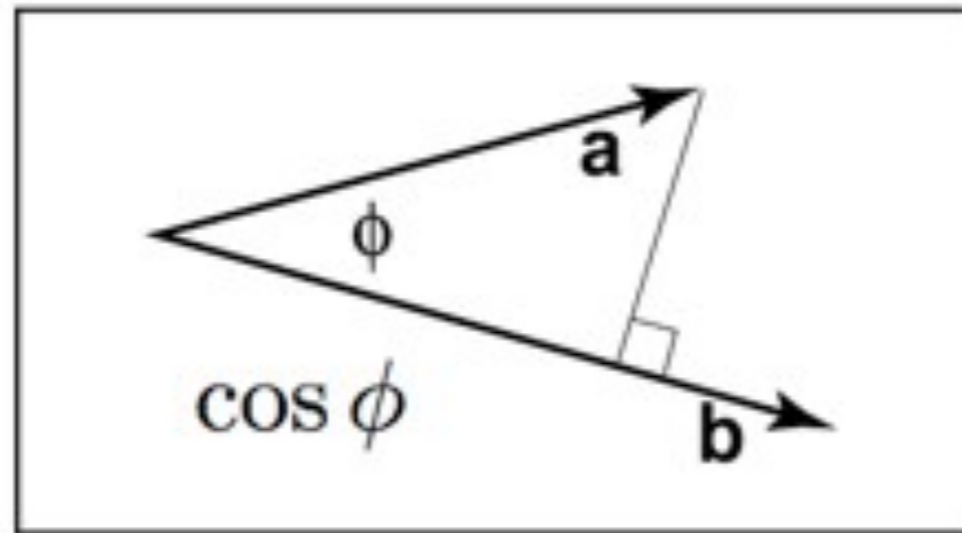
The Parallelogram Rule

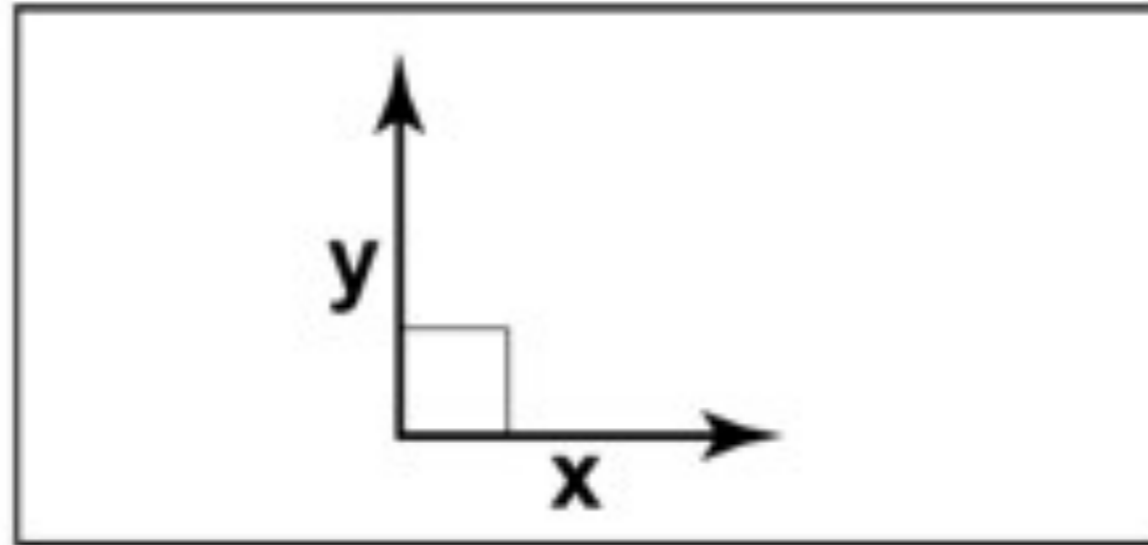Vector addition define for any number of dimensions

Dot Product

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

The projection of a onto b

N. B. the projection is 0 if a is perpendicular to b

Orthonormal Vectors

Perpendicular $\quad \mathbf{x} \cdot \mathbf{y} \;=\; 0$

Unit length $\qquad \mathbf{x} \cdot \mathbf{x} \;=\; 1$

$\qquad\qquad\qquad\;\; \mathbf{y} \cdot \mathbf{y} \;=\; 1$

Coordinates and Vectors

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

$$
\begin{aligned}
\alpha &= \mathbf{x} \cdot \mathbf{c} = \alpha \mathbf{x} \cdot \mathbf{x} + \beta \mathbf{x} \cdot \mathbf{y} \\
\beta &= \mathbf{y} \cdot \mathbf{c} = \alpha \mathbf{y} \cdot \mathbf{x} + \beta \mathbf{y} \cdot \mathbf{y}
\end{aligned}
$$
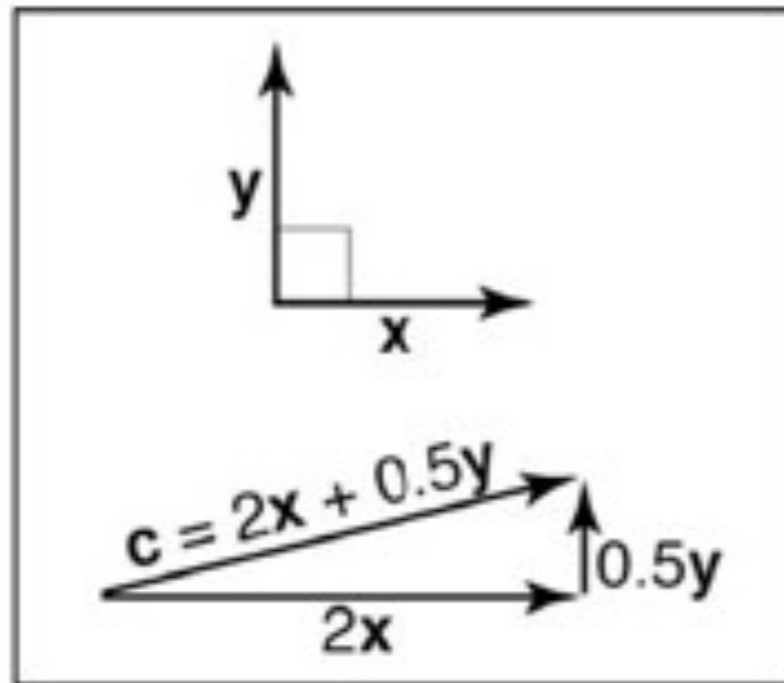
Dot product between two vectors

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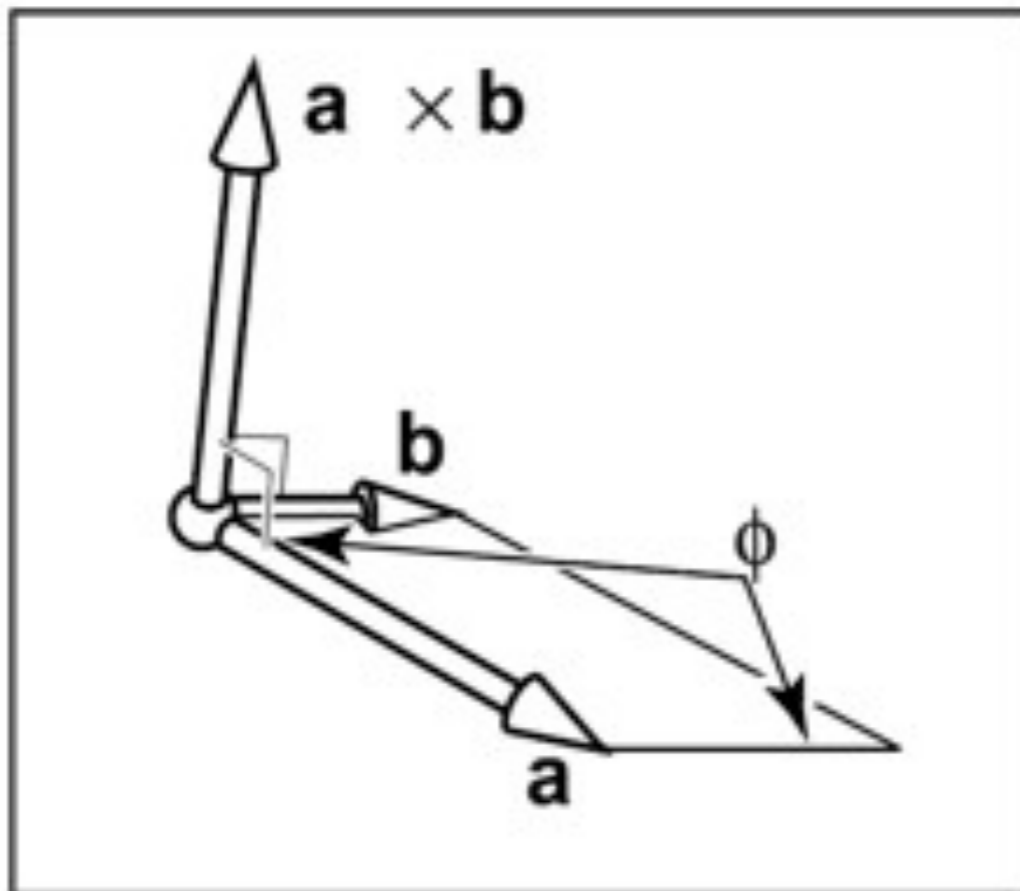
$$\mathbf{b} = x_b \mathbf{x} + y_b \mathbf{y}$$

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

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

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

Cross Product

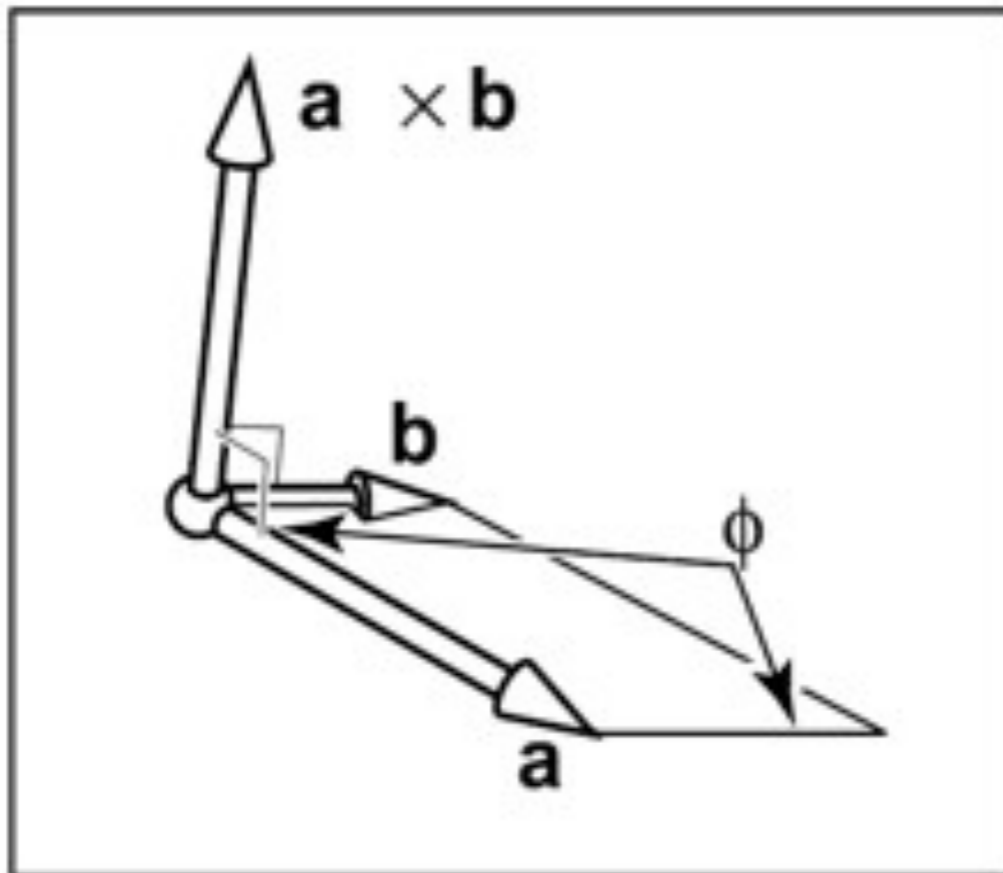$$\mathbf{c} = \mathbf{a} \times \mathbf{b}$$

$$
\begin{aligned}
x_c &= y_a z_b - z_a y_b \\
y_c &= z_a x_b - x_a z_b \\
z_c &= x_a y_b - z_a x_b
\end{aligned}
$$

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

`<Vector> = <Scalar> * <Vector>`

$$v = \alpha w$$

`<Vector> = <Vector> + <Vector>`

$$u = v + w$$

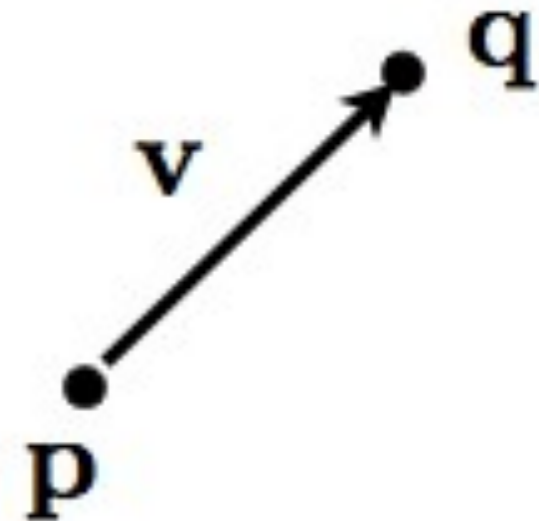**Implementation of parallelogram rule**

Point operations

Points: p, q, r

<Point> = <Point> + <Vector>

$$q = p + v$$

<Vector> = <Point> - <Point>

$$v = q - p$$

A point is an origin and a vector displacement

illegal operations

`<Point> = <Scalar> * <Point>`

$$\mathbf{p} = \alpha \mathbf{q}$$

`<Point> = <Point> + <Point>`

$$\mathbf{p} = \mathbf{q} + \mathbf{r}$$

`<Vector> = <Point> + <Vector>`

$$\mathbf{v} = \mathbf{p} + \mathbf{w}$$
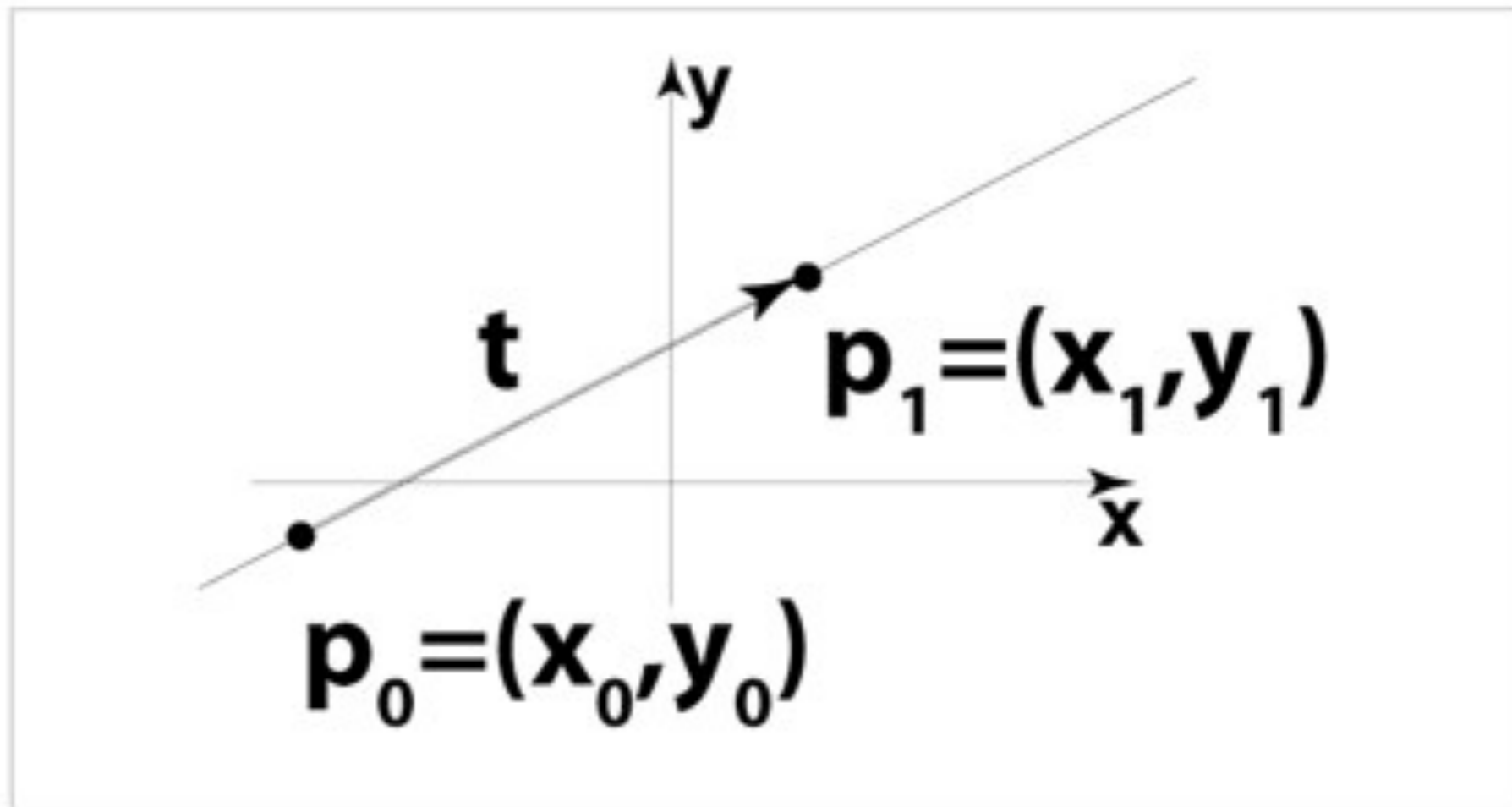
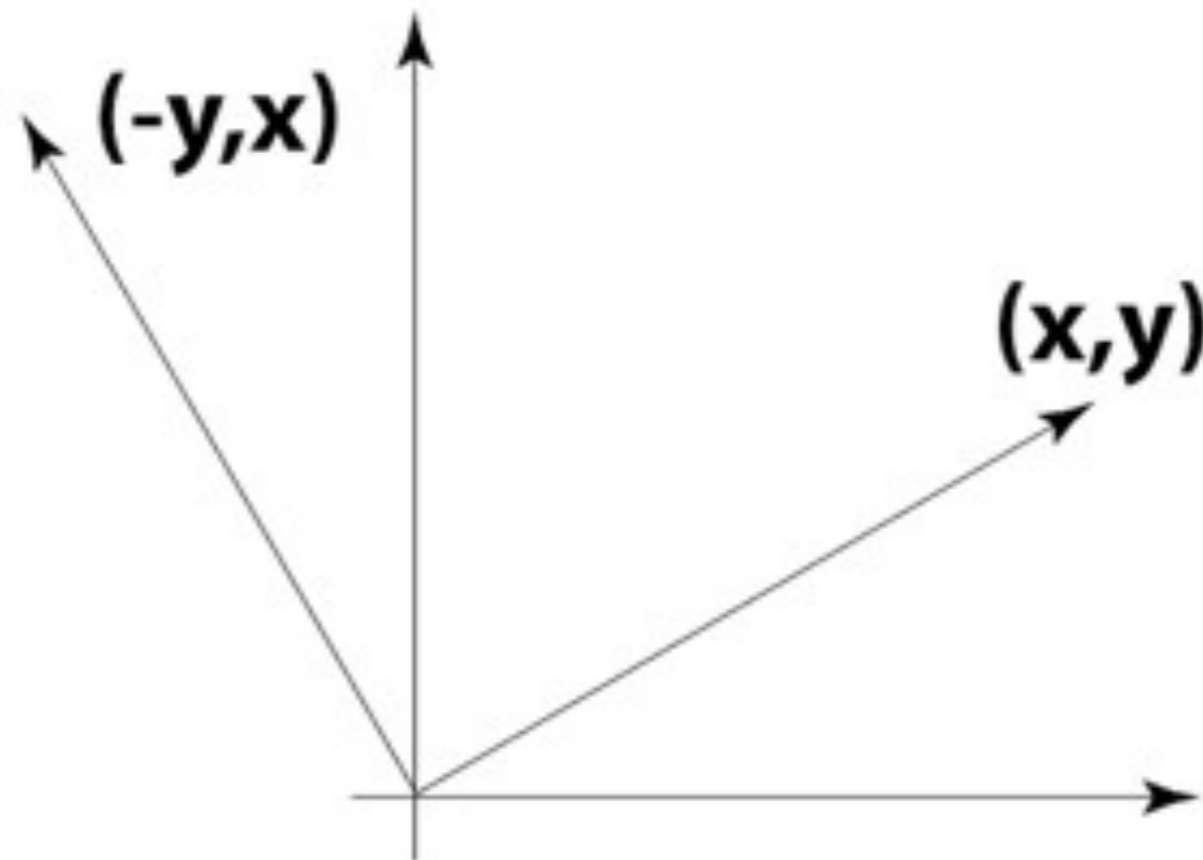`<Point> = <Point> - <Point>`

$$\mathbf{p} = \mathbf{q} - \mathbf{r}$$

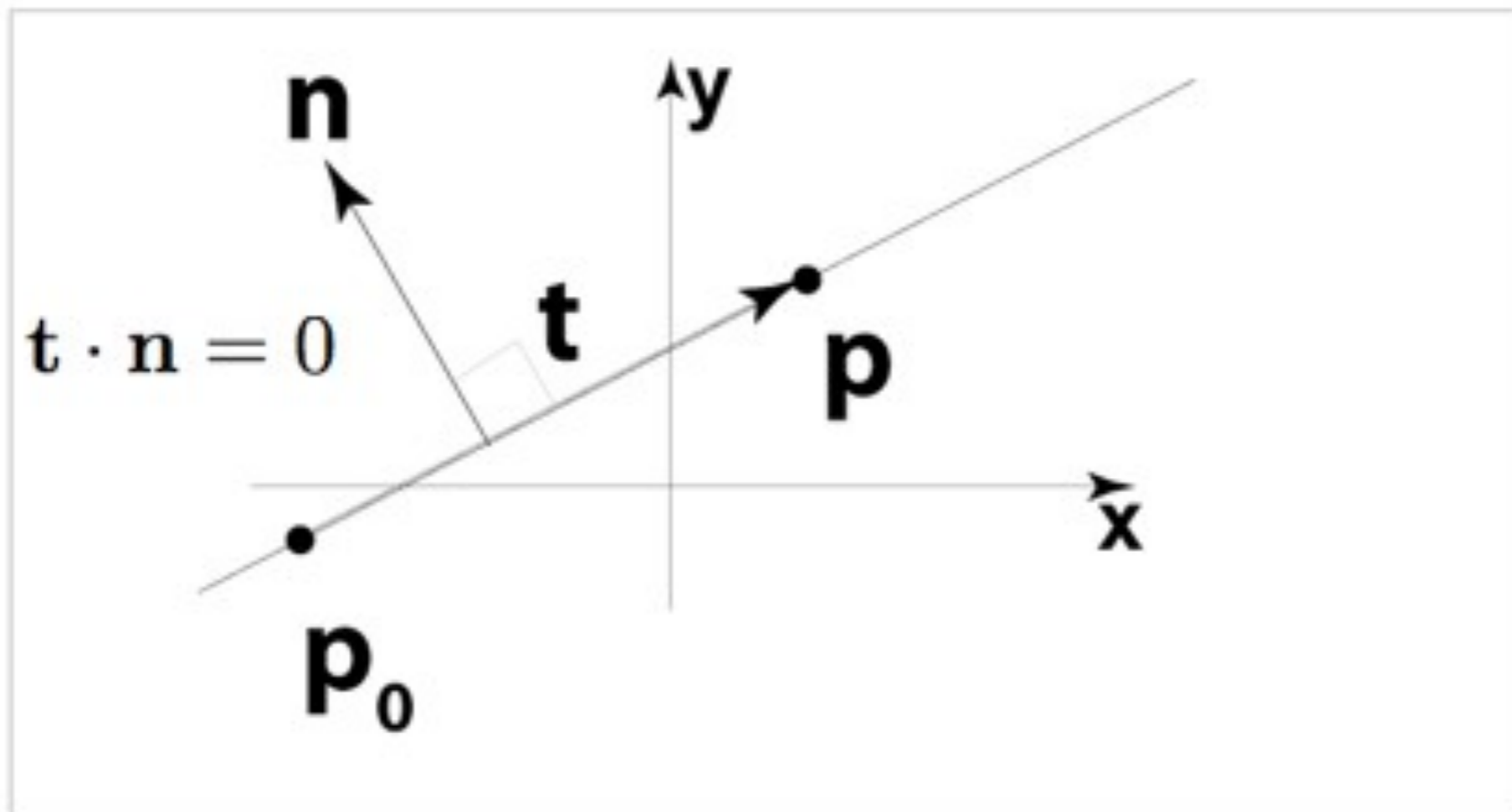# Directed line

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

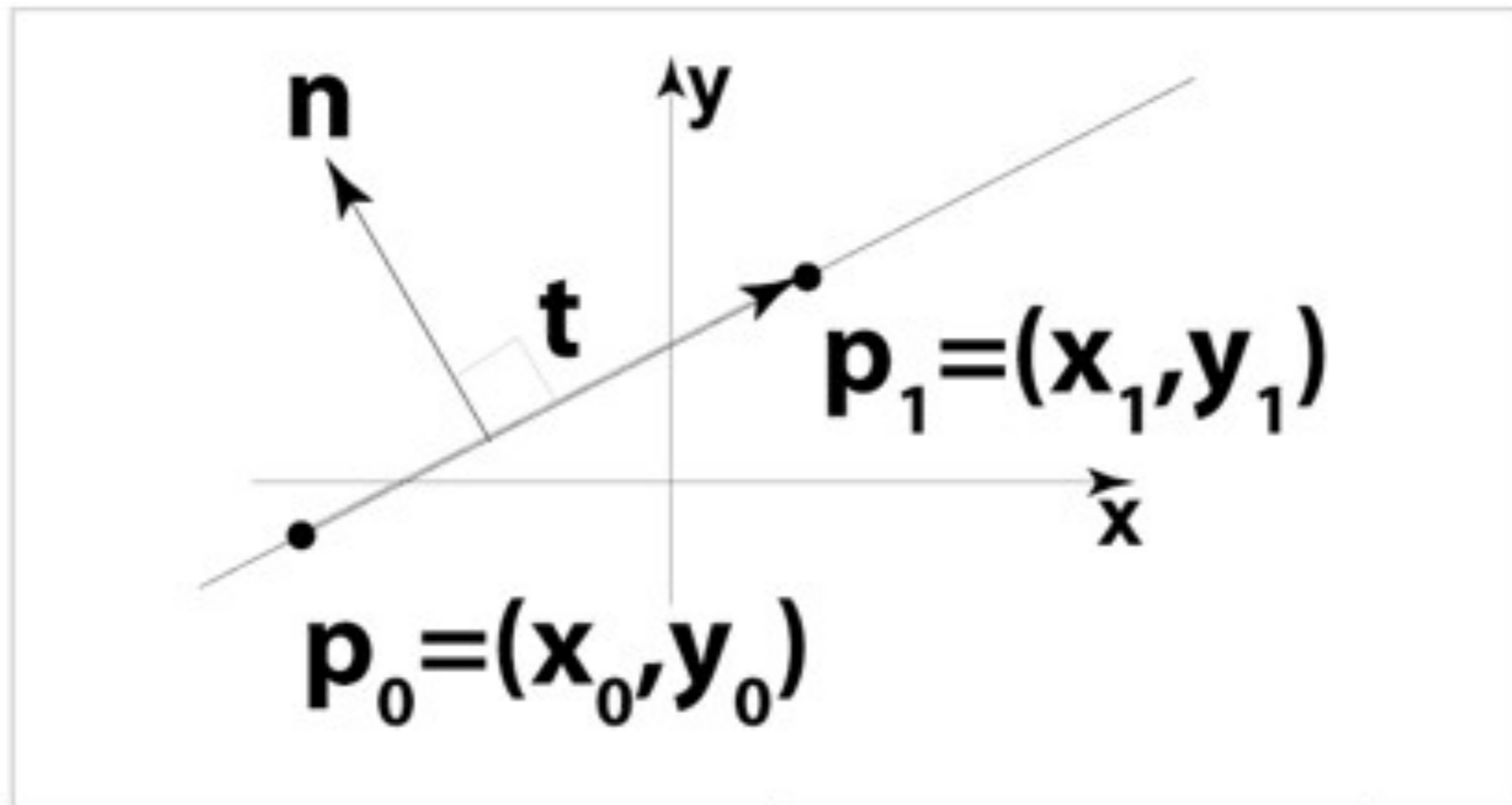# Perpendicular vector in 2D

$$Perp((x,y))=(-y,x)$$

Line equation

$$(p - p_0) \cdot n = 0$$

This equation must be true for all point p on the line
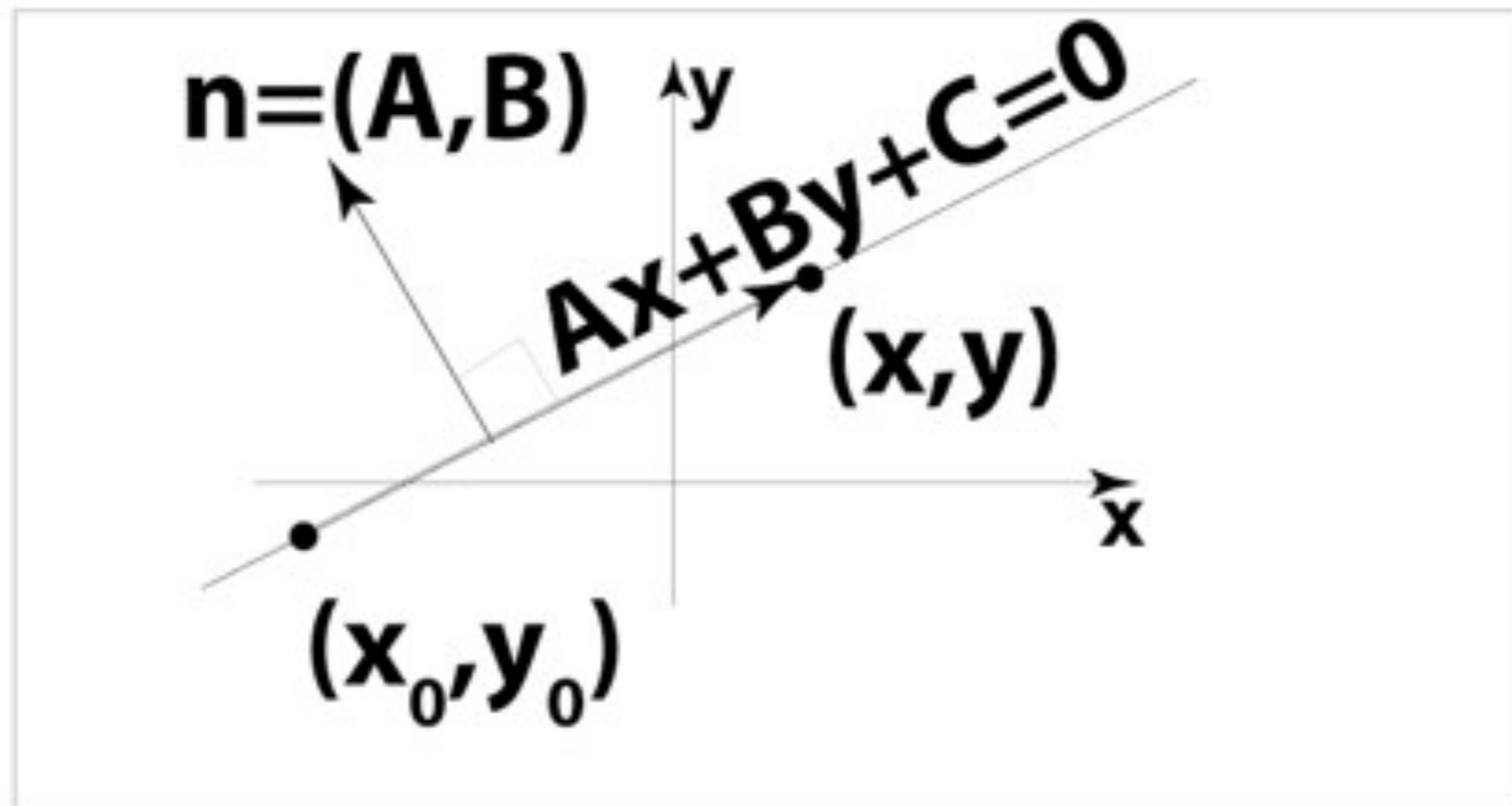
Normal to the line

$$\mathbf{t} = \mathbf{p}_1 - \mathbf{p}_0 = (x_1 - x_0, y_1 - y_0)$$
$$\mathbf{n} = \mathrm{Perp}(\mathbf{t}) = (y_0 - y_1, x_1 - x_0)$$
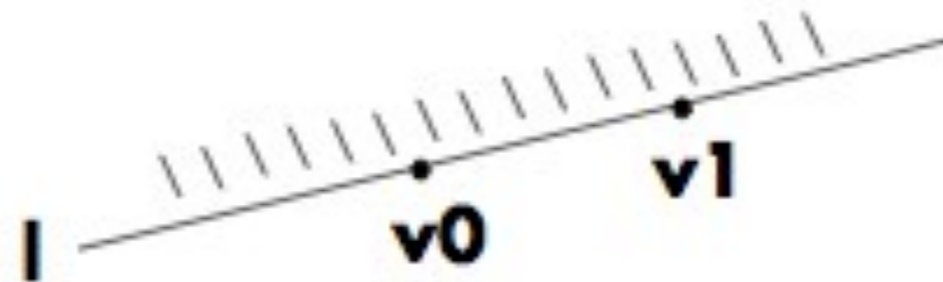
Line equation

$$A = y_1 - y_0$$
$$B = x_0 - x_1$$
$$C = x_0 y_1 - y_0 x_1$$

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);
}
```
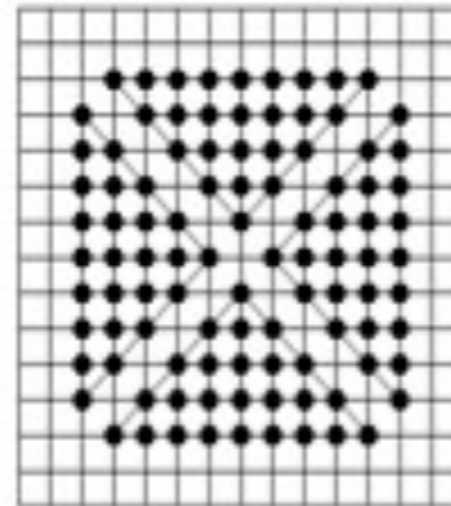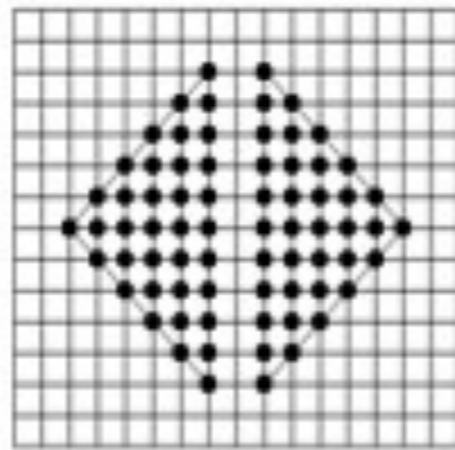
Singularities

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

Causes two fragments to be generated

- ■ **Wasted effort**
- ■ **Problems with transparency (later lecture)**

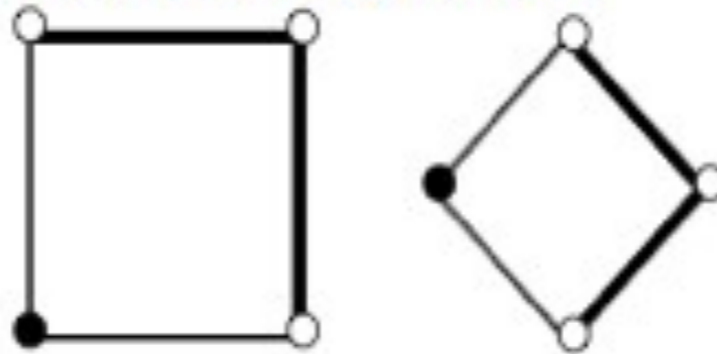**Not including singularities (e < 0) causes gaps**

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);
}
```
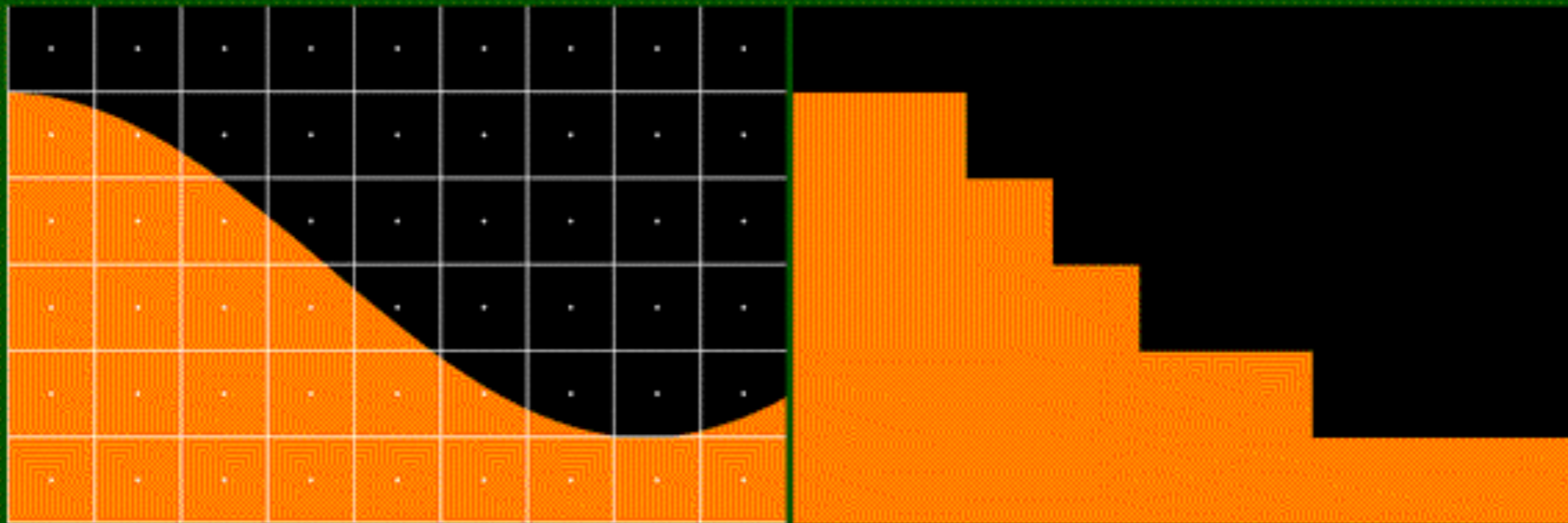
Antialiasing

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

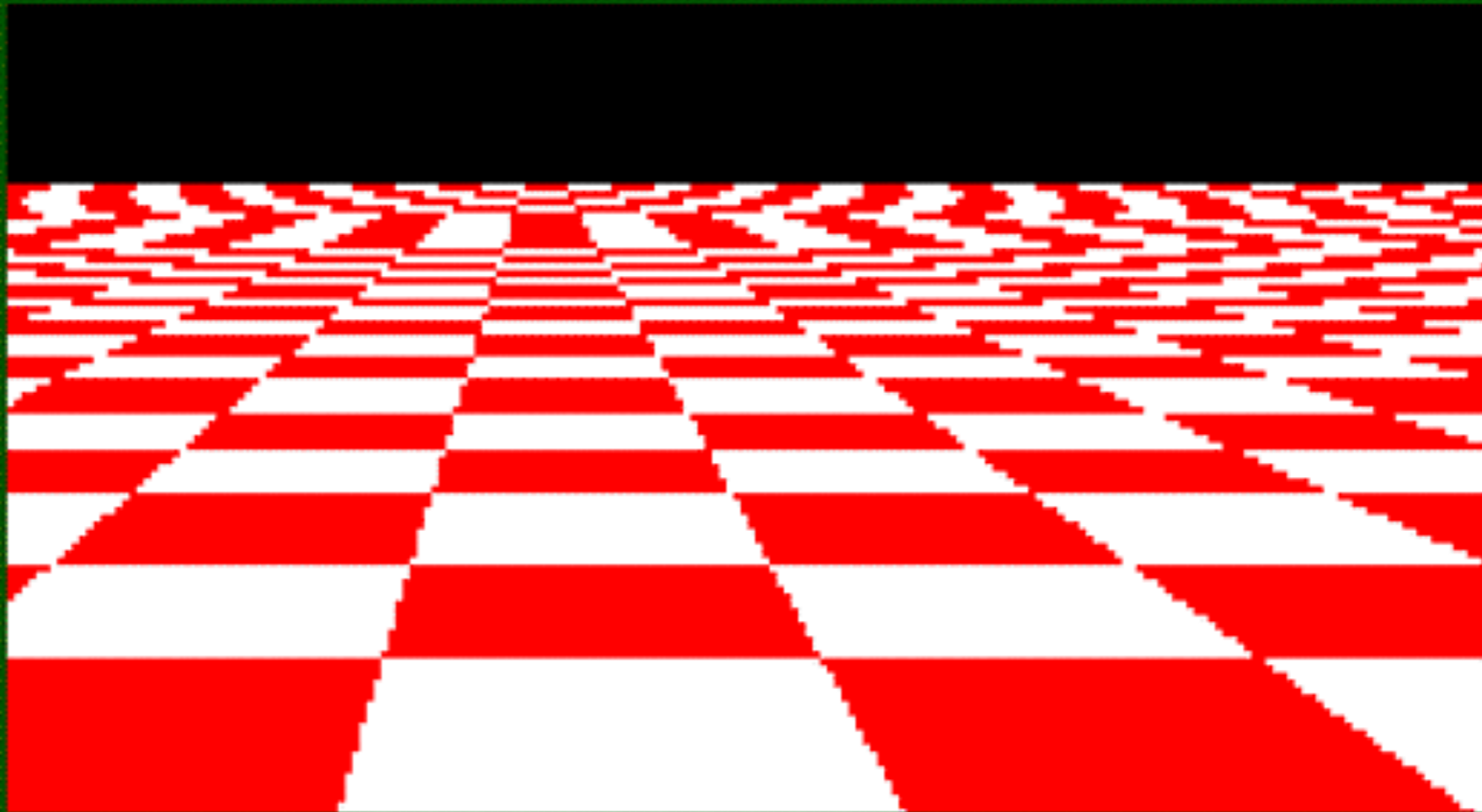 - Small objects might miss

Aliasing(examples)

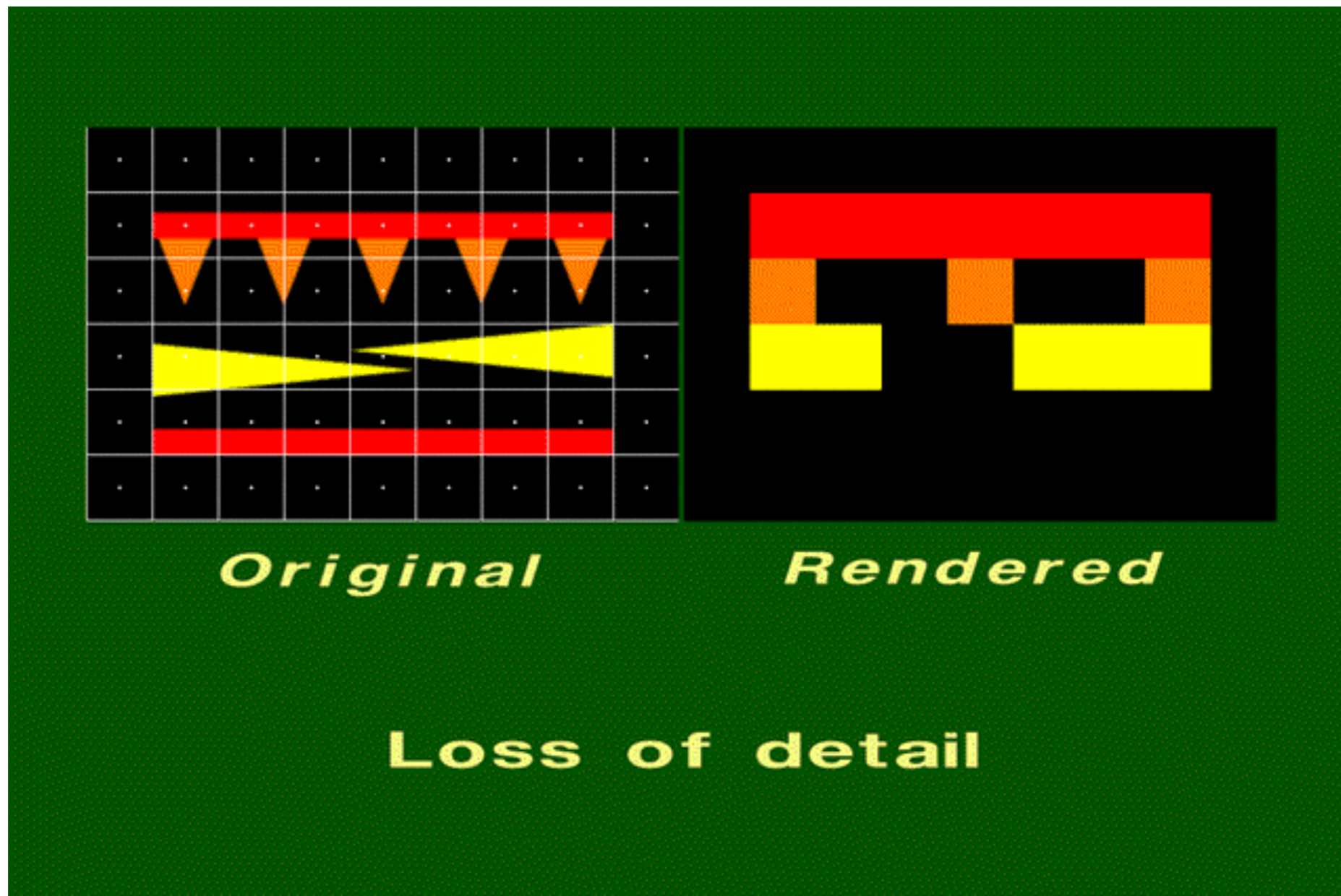

Original Rendered

Jagged profiles

Aliasing(examples)

Disintegrating textures

Aliasing(examples)

Original   Rendered

Loss of detail

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

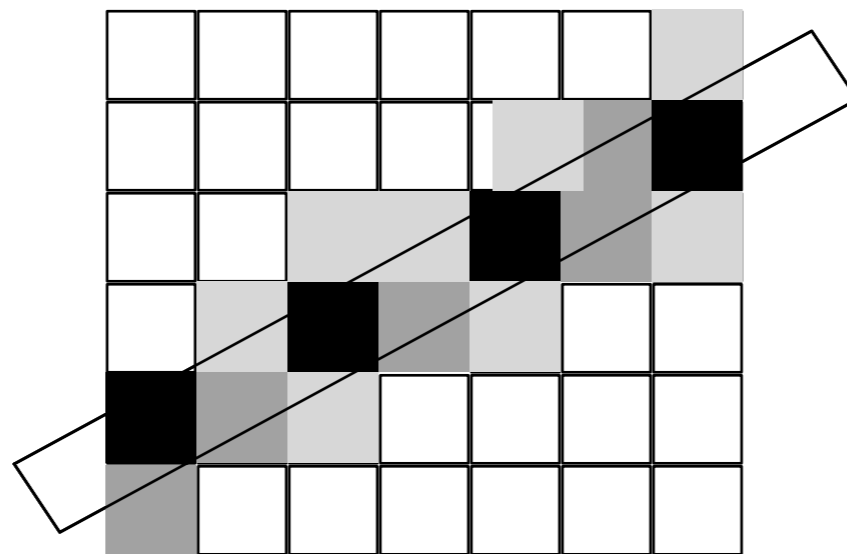# Antialiasing (postfiltering)

How should one supersample?

Jittered   Regular

Taking 9 samples per pixel
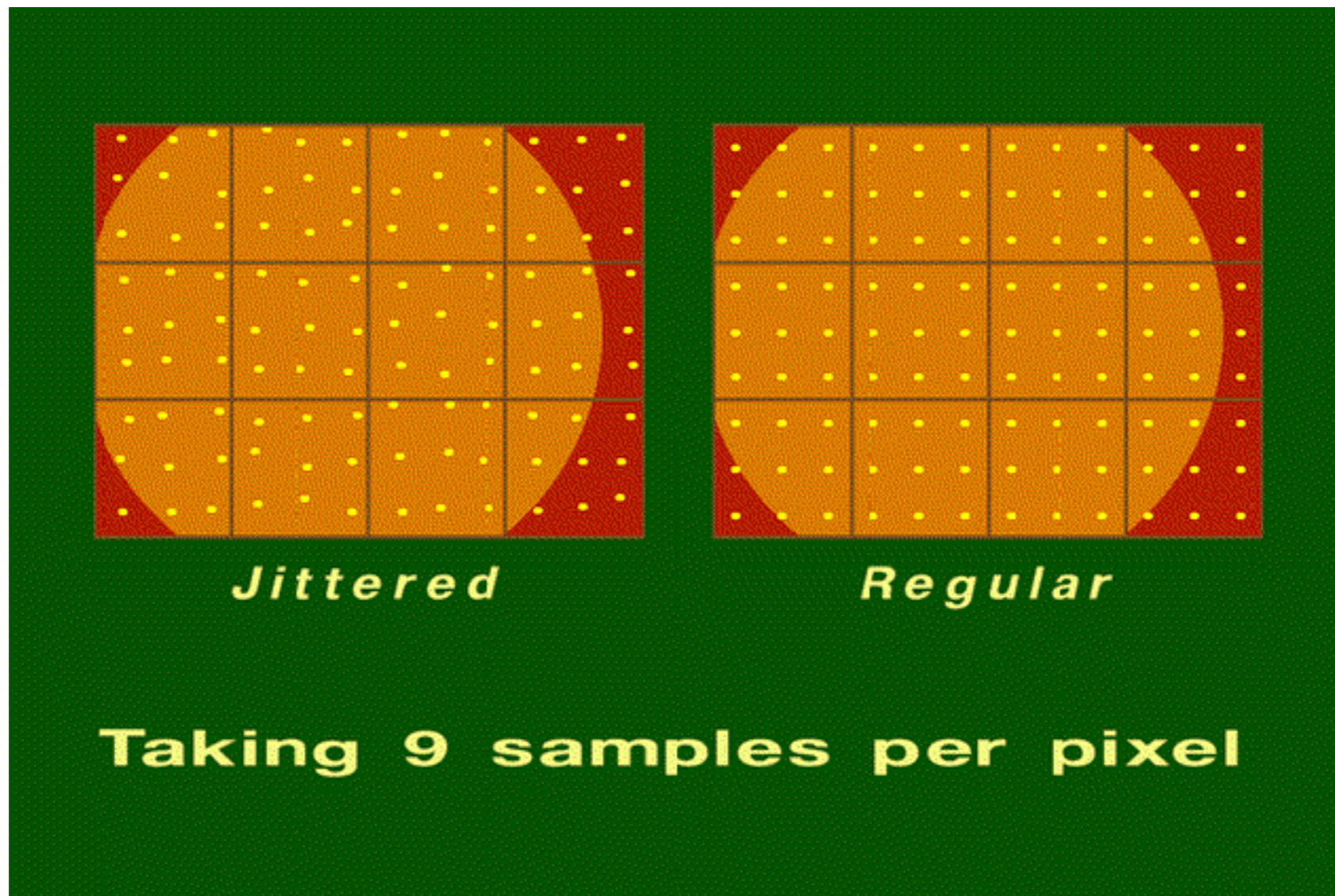
Area Sampling

- 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

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

$y+0.5$

y

$y\text{-}0.5$

filled area

$x\text{-}0.5$ x $x\text{+}0.5$
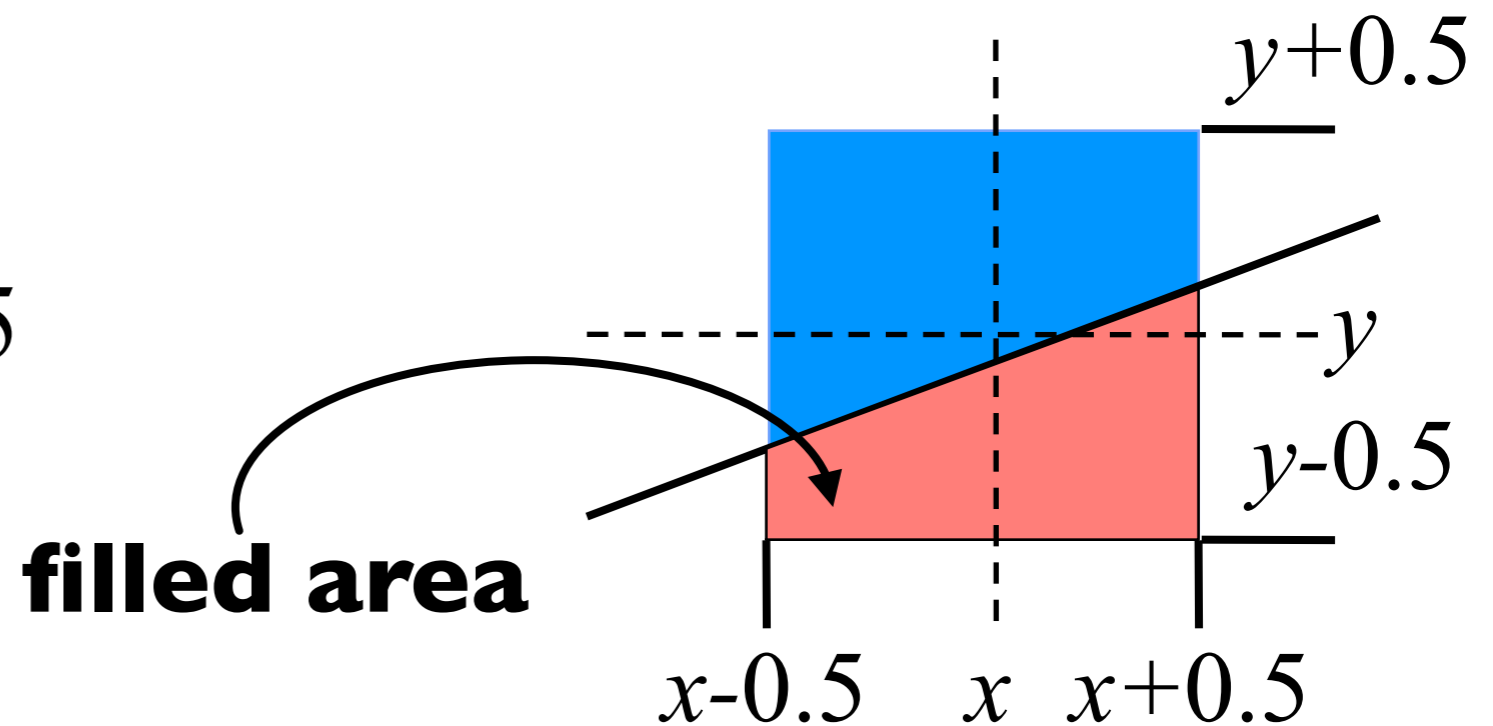
Clipping

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
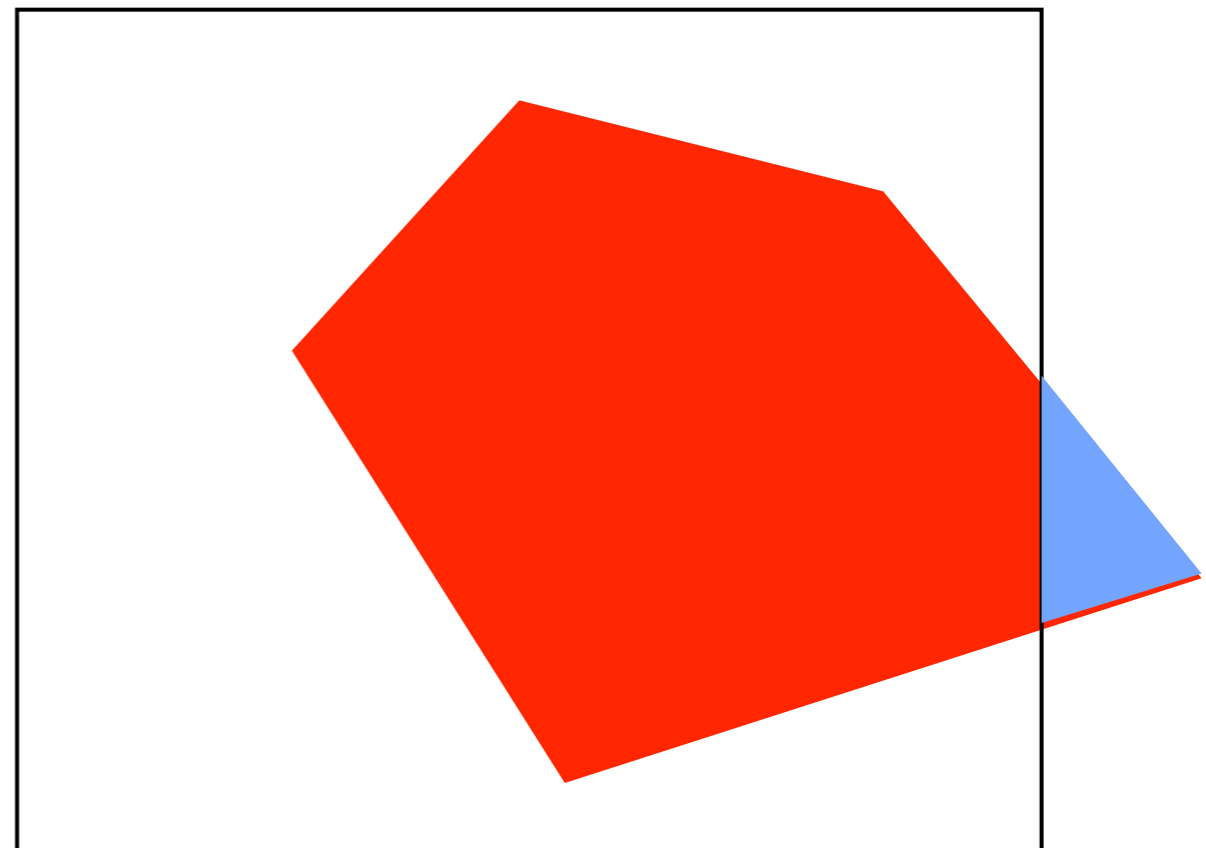
Clipping

- 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
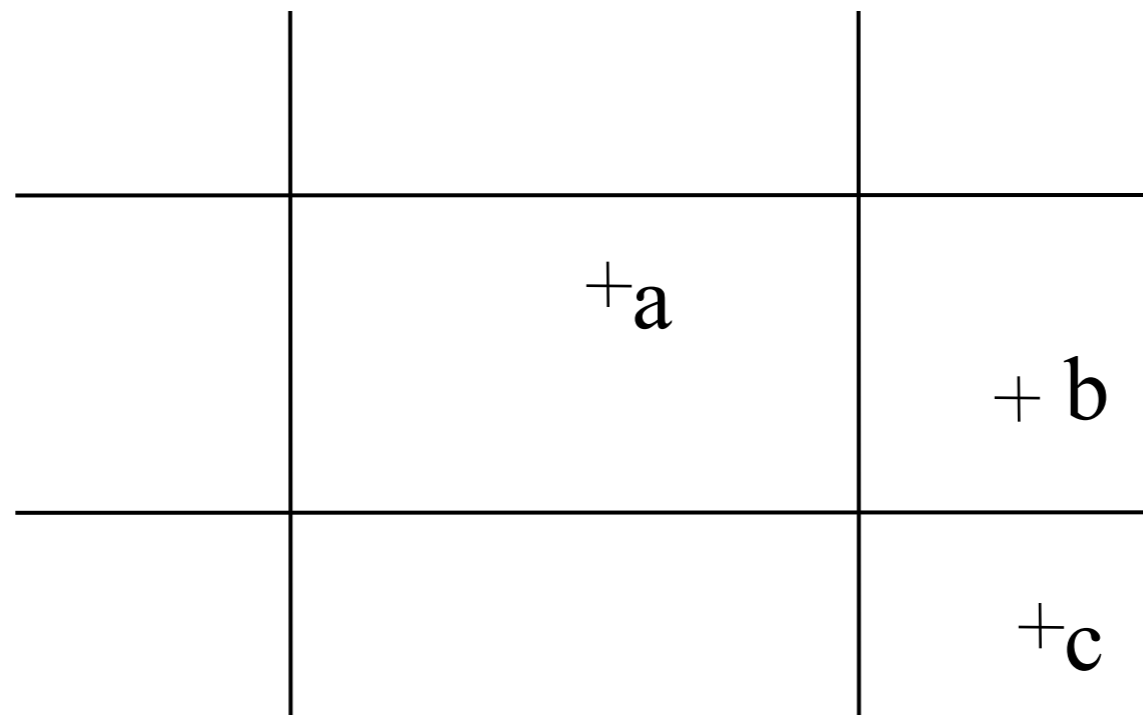
  —Too close

  —Too far

  —Off sides of the screen

How would we clip?
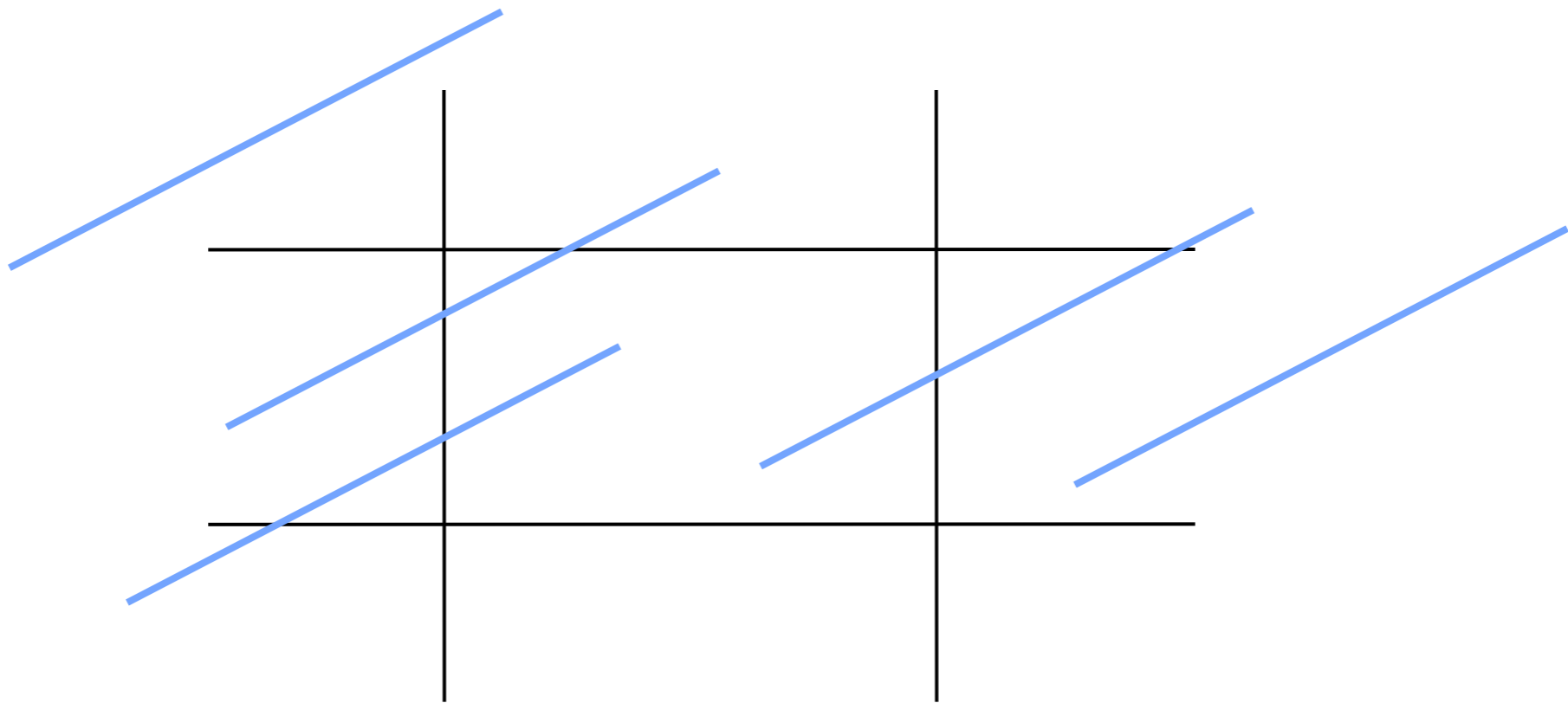
- Points?

- 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

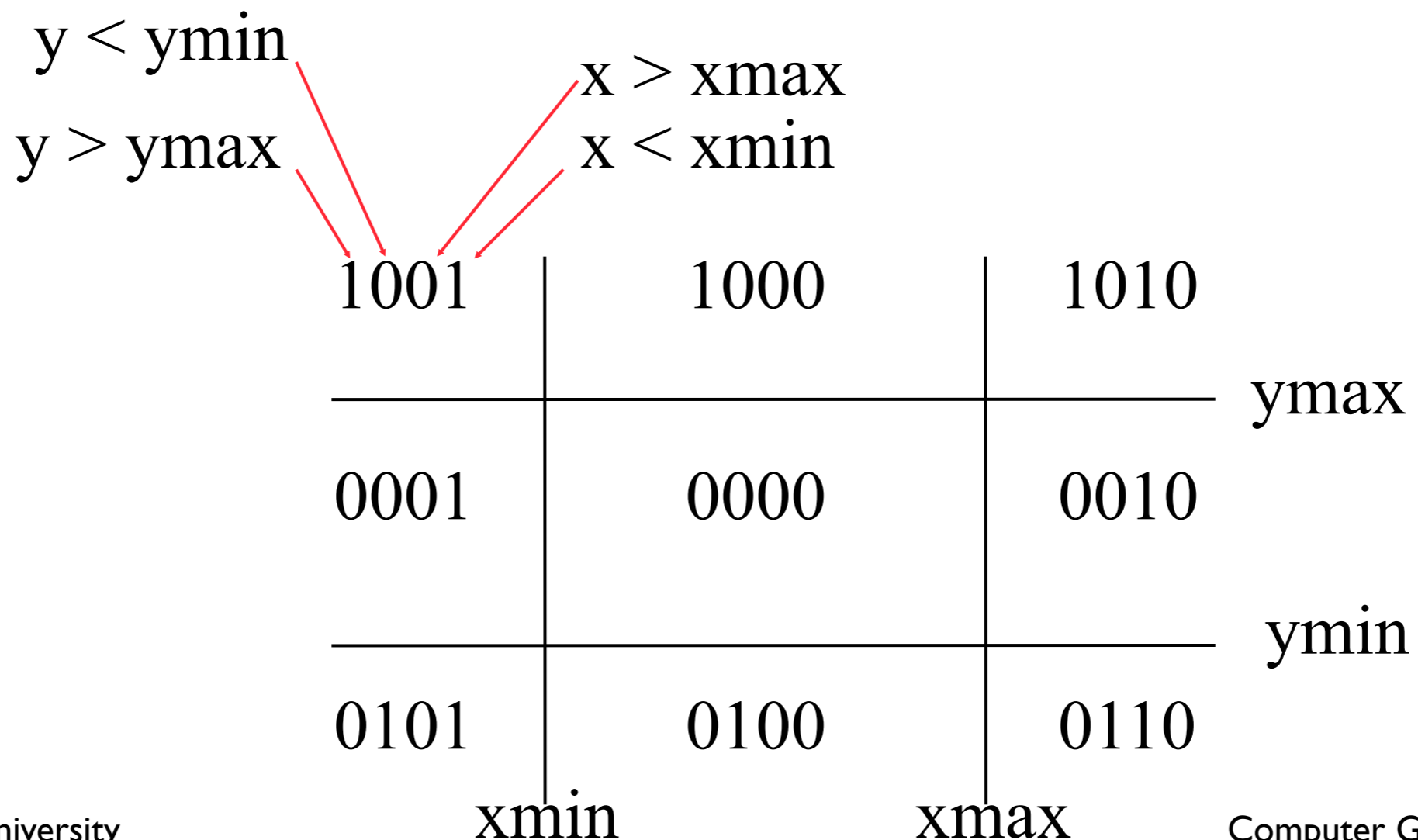   - \>= minx and <= maxx and >= miny and <= maxy

$+a$

$+\,b$

$+c$

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.

$y < ymin$

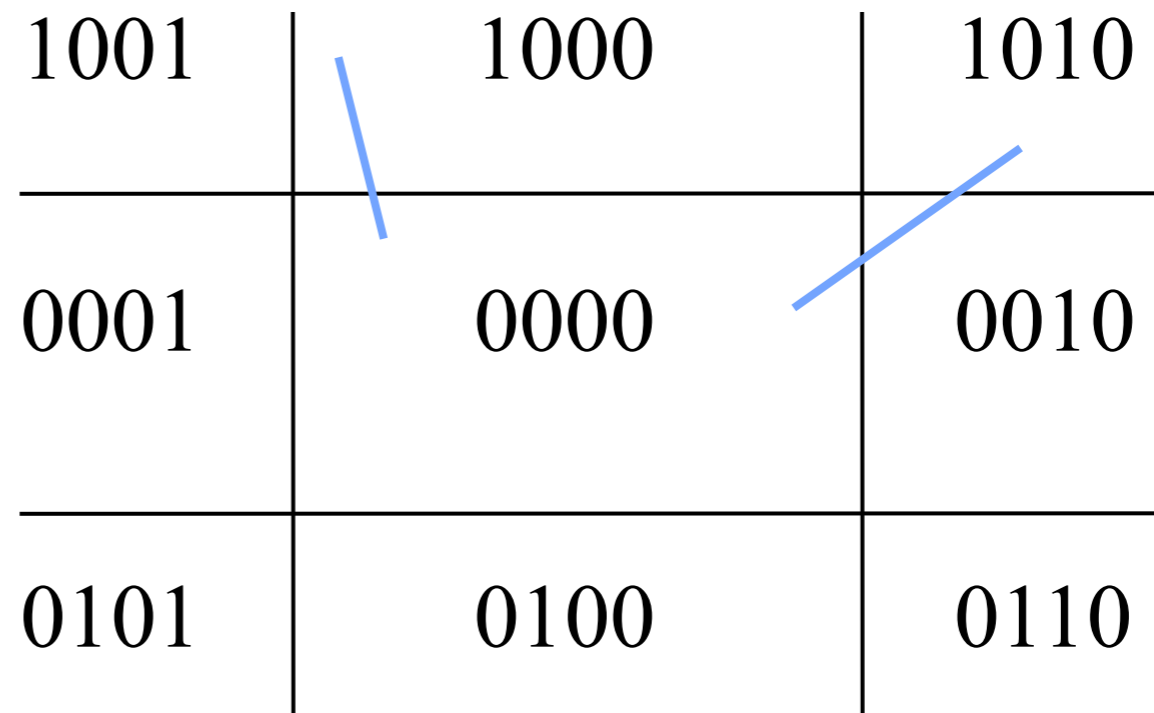$y > ymax$

$x > xmax$

$x < xmin$

1001	1000	1010
0001	0000	0010
0101	0100	0110

ymax

ymin

xmin   xmax

Outcode cases

- We'll call the two endpoint outcodes o_1 and o_2.

  — If $o_1 = o_2 = 0$, both endpoints are <u>inside</u>.

  — else if $(o_1 \ \& \ o_2) \ != 0$, both ends points are on the <u>same side</u>, the edge is discarded.

1001	1000	1010
0001	0000	0010
0101	0100	0110
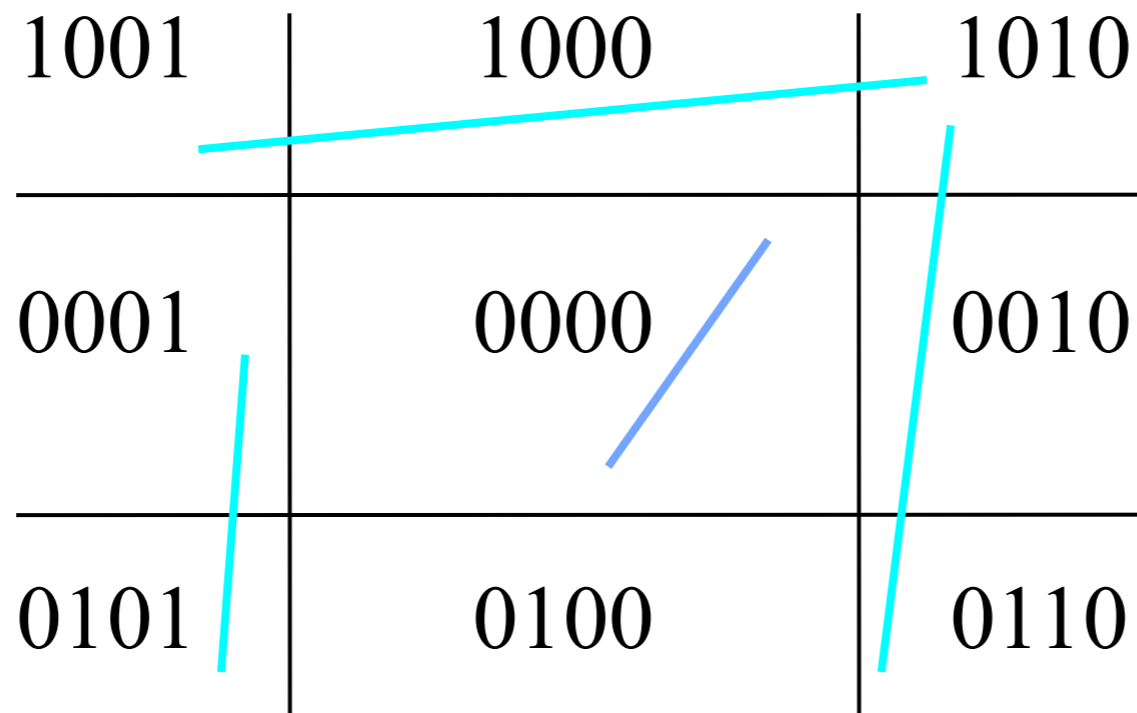
More cases

- else if (o_1 != 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!

1001	1000	1010
0001	0000	0010
0101	0100	0110
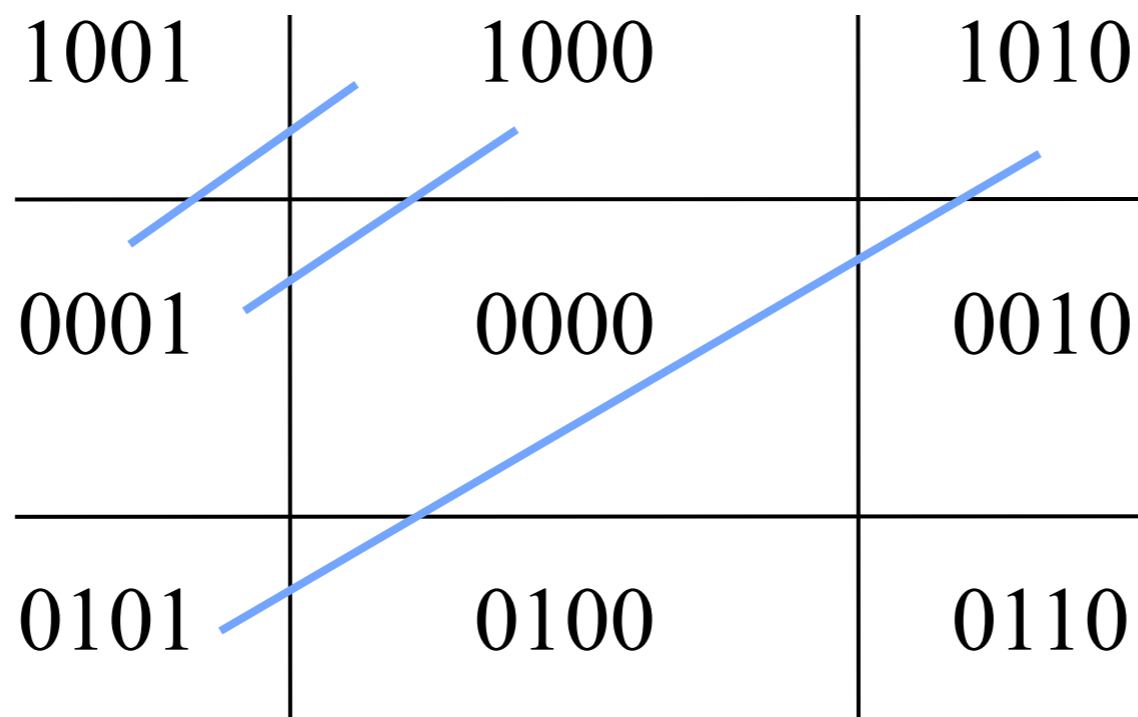
Cohen-Sutherland Line-Clipping Algorithm

d

1010 i

1001 *c*

b

h

0000 *a*

g *0010*

f

e

0100
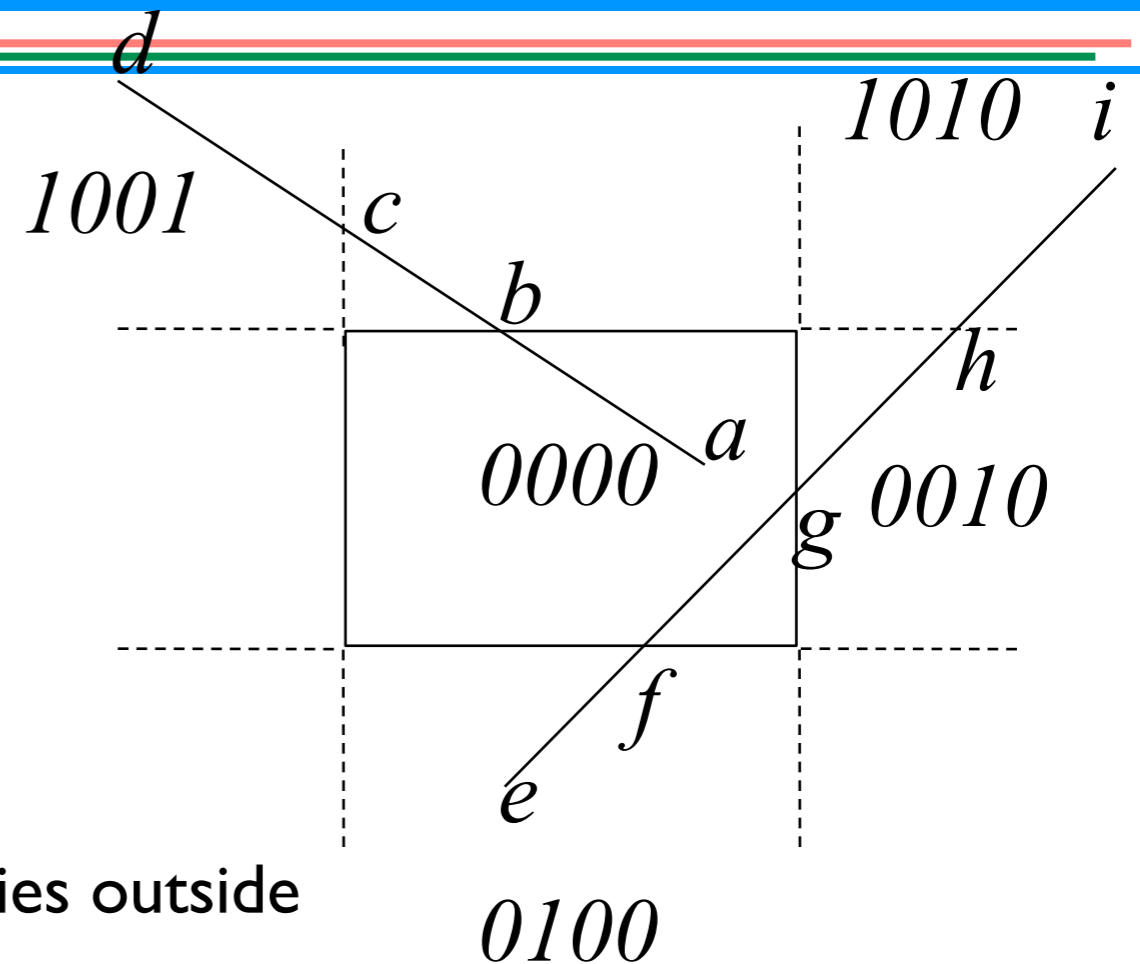
- To do the clipping find the end point that lies outside
- Test the outcode to find the edge that is crossed and determine the corresponding intersection point
- Replace the outside end-point by intersection-point
- Repeat the above steps for the new line

Sutherland-Hodgeman Polygon-Clipping Algorithm
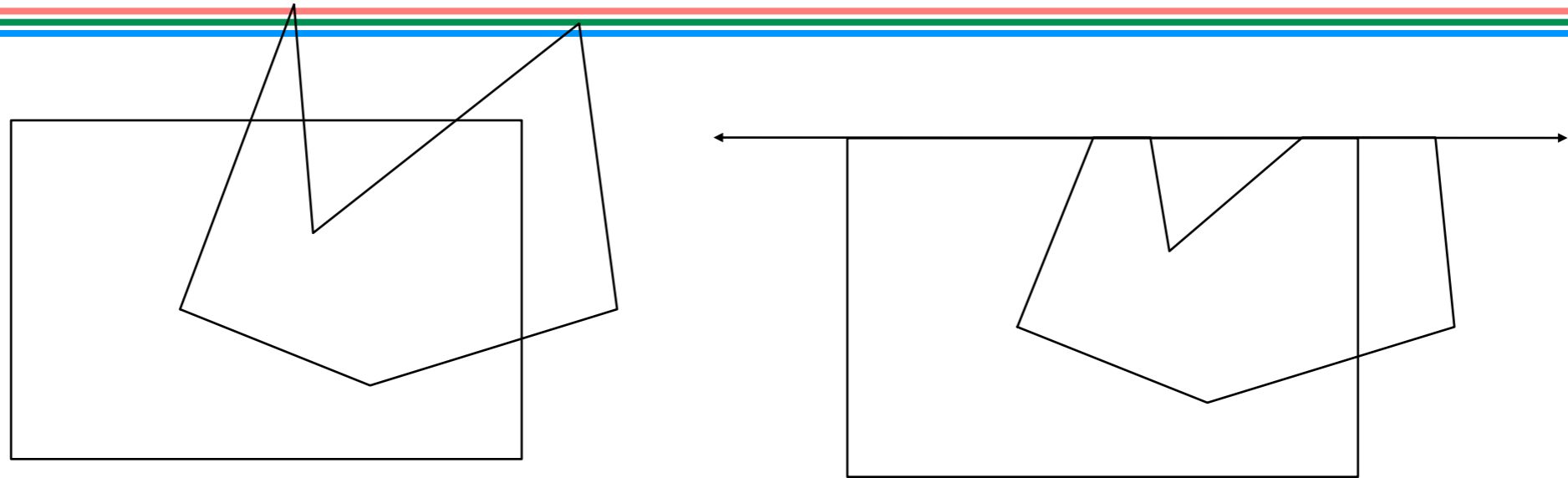
- Polygons can be clipped against each edge of the window one edge at a time. Window/edge intersections, if any, are easy to find since the X or Y coordinates are already known.

- Vertices which are kept after clipping against one window edge are saved for clipping against the remaining edges.

Pipelined Polygon Clipping

| Clip Right | → | Clip Top | → | Clip Left | → | Clip Bottom |
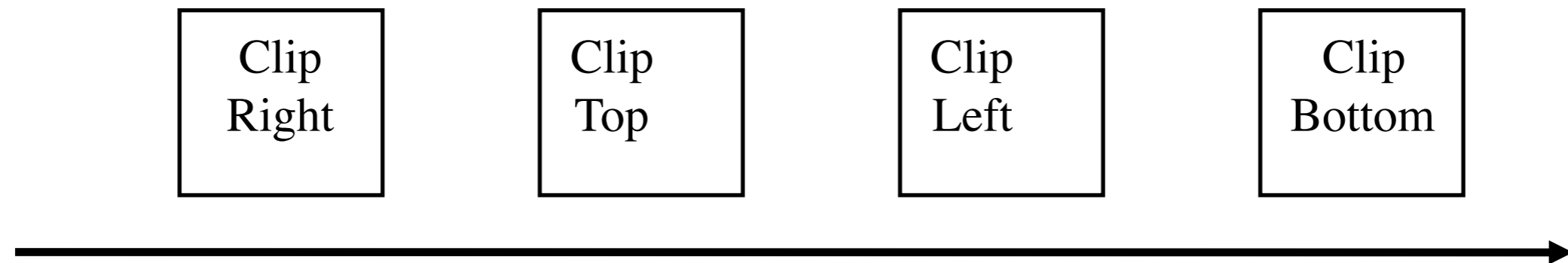
- Because polygon clipping does not depend on any other polygons, it is possible to arrange the clipping stages in a **pipeline**. the input polygon is clipped against one edge and any points that are kept are passed on as input to the next *stage* of the pipeline.

- 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…
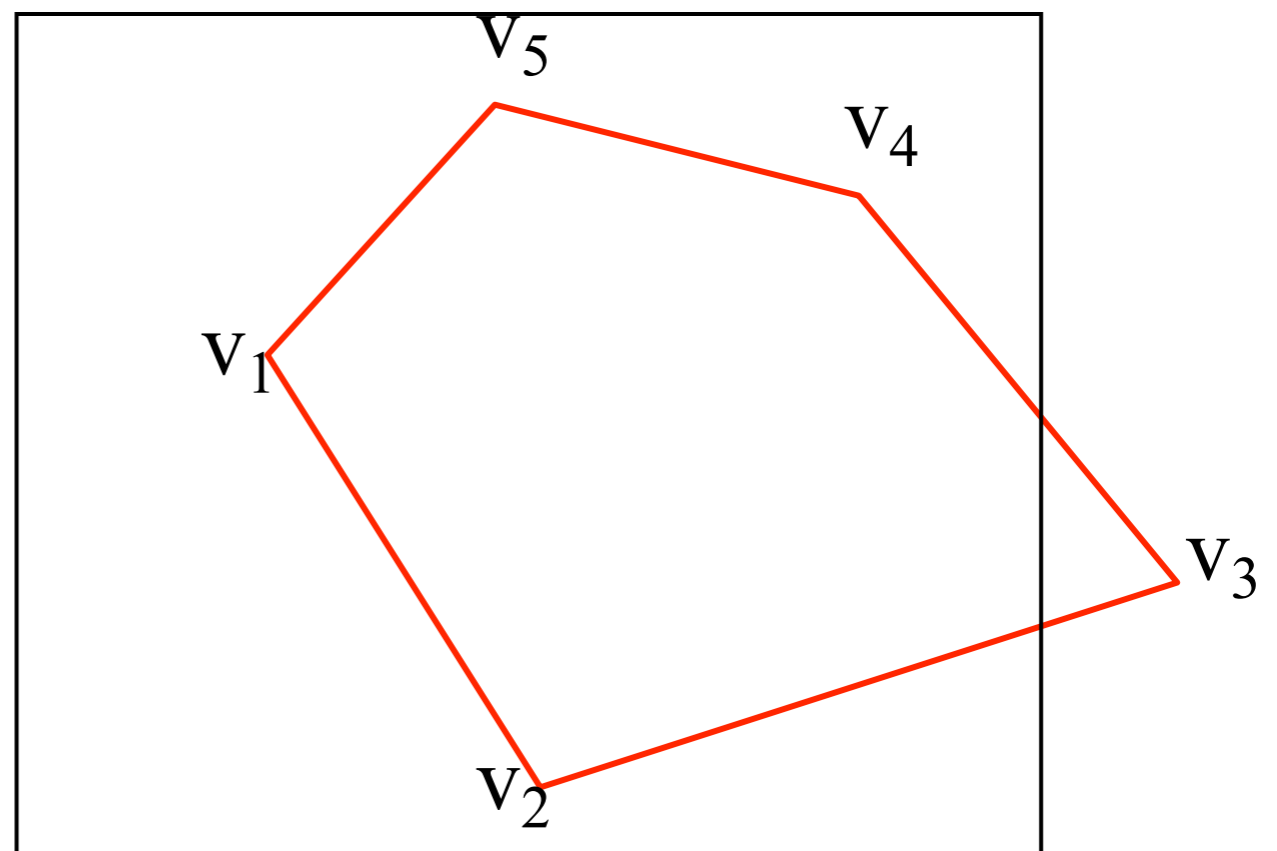
Sutherland-Hodgeman algorithm

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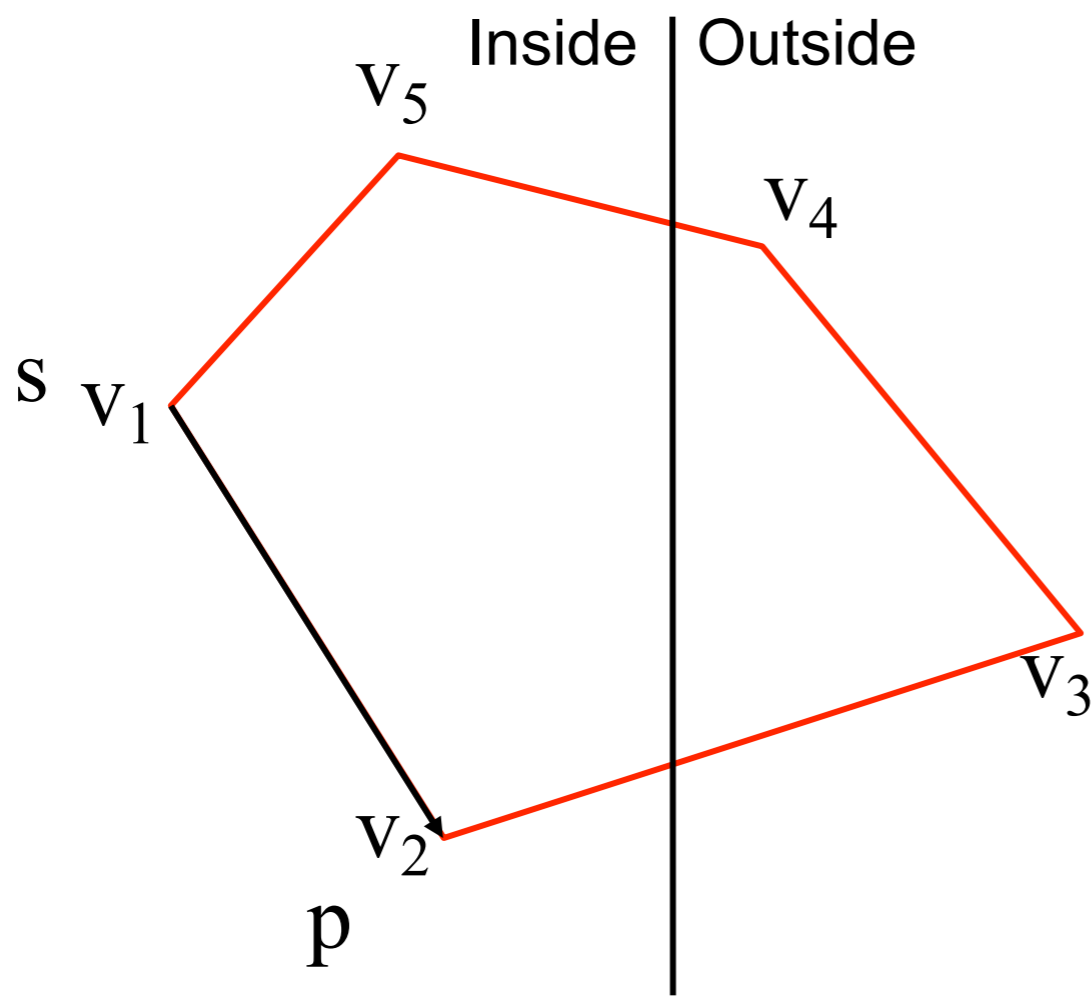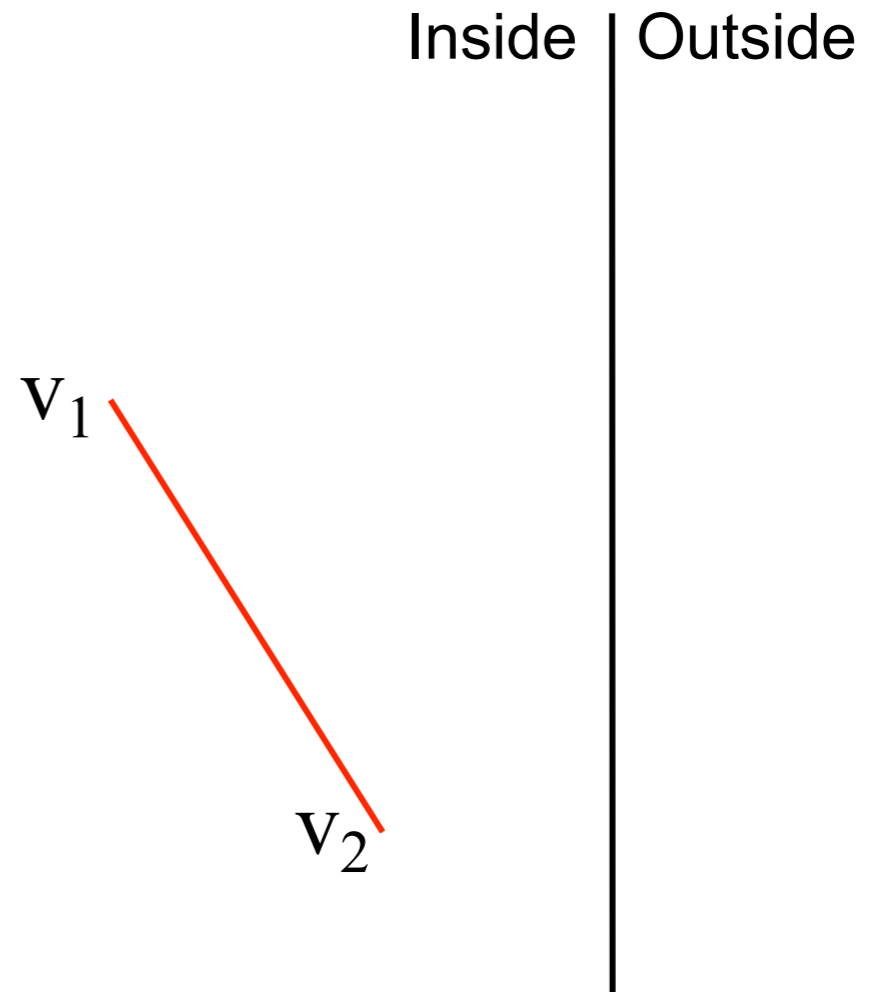 - For each pair, what are the possibilities?
  - Consider v_1, v_2

Inside | Outside — v_1, v_2 output v_2

Inside | Outside — v_1, v_2 output i

Inside | Outside — v_2, v_1 no output

Inside | Outside — v_1, v_2 output i and v_2

v_1, v_2

Inside | Outside

v_5

v_4

s v_1

v_3

v_2

p
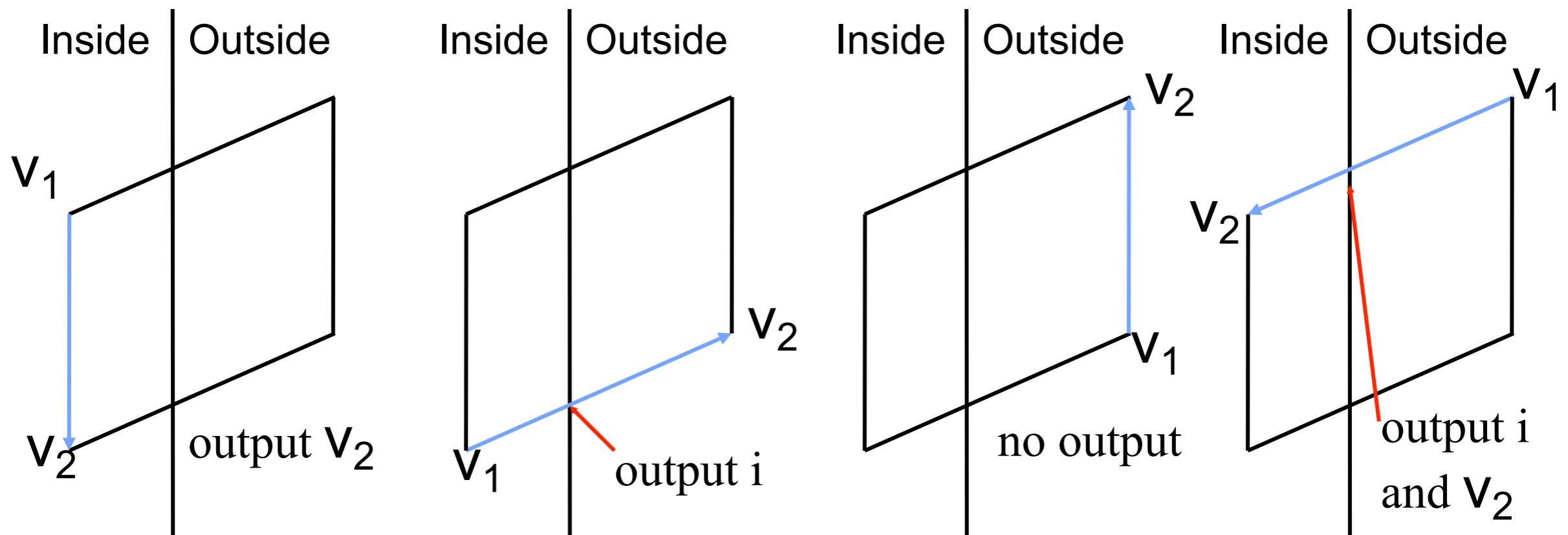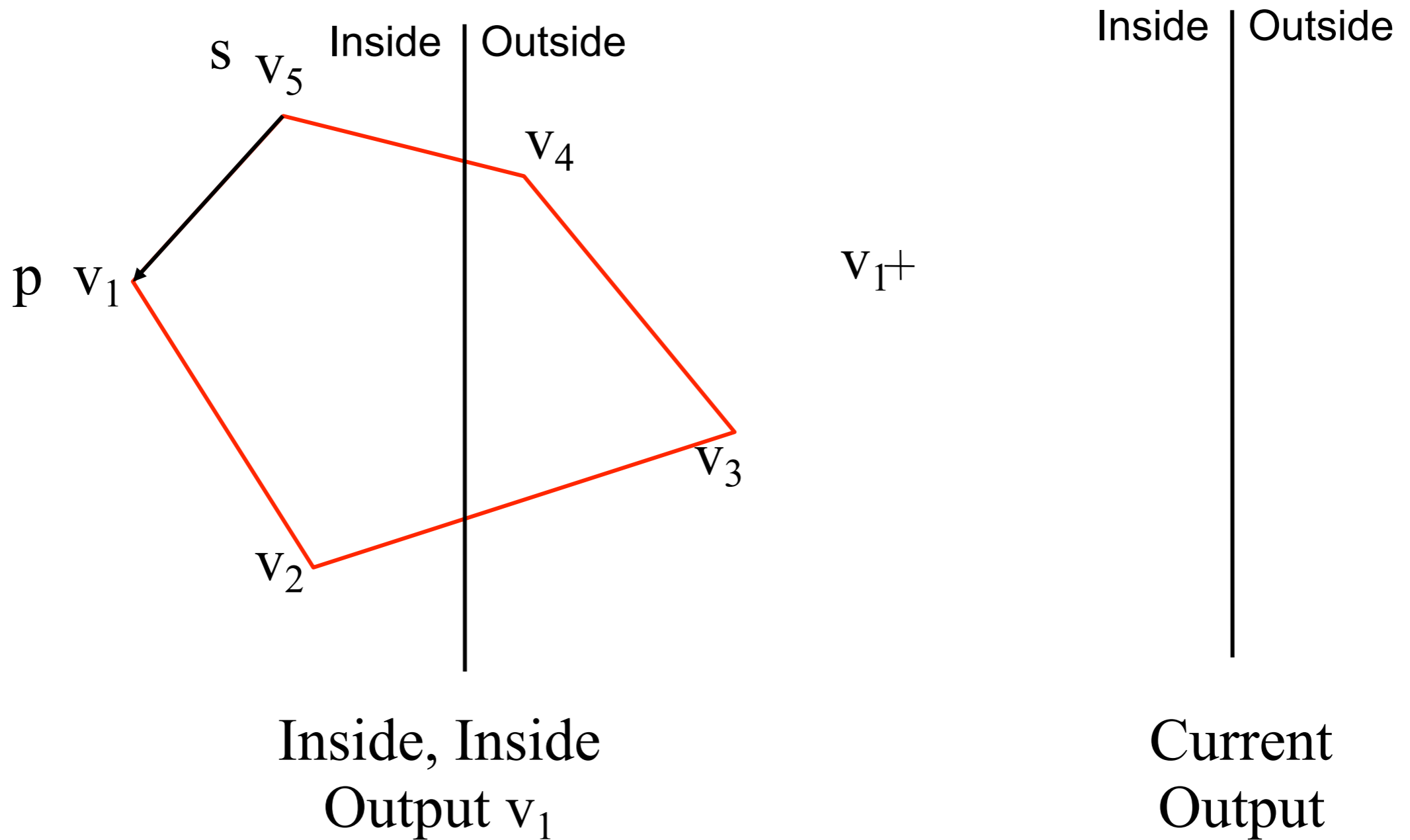
Inside, Inside
Output v_2

Inside | Outside

v_1

v_2

Current
Output

v_2, v_3

Inside | Outside

v_5

v_4

v_1

p

v_3

i_1

v_2

s
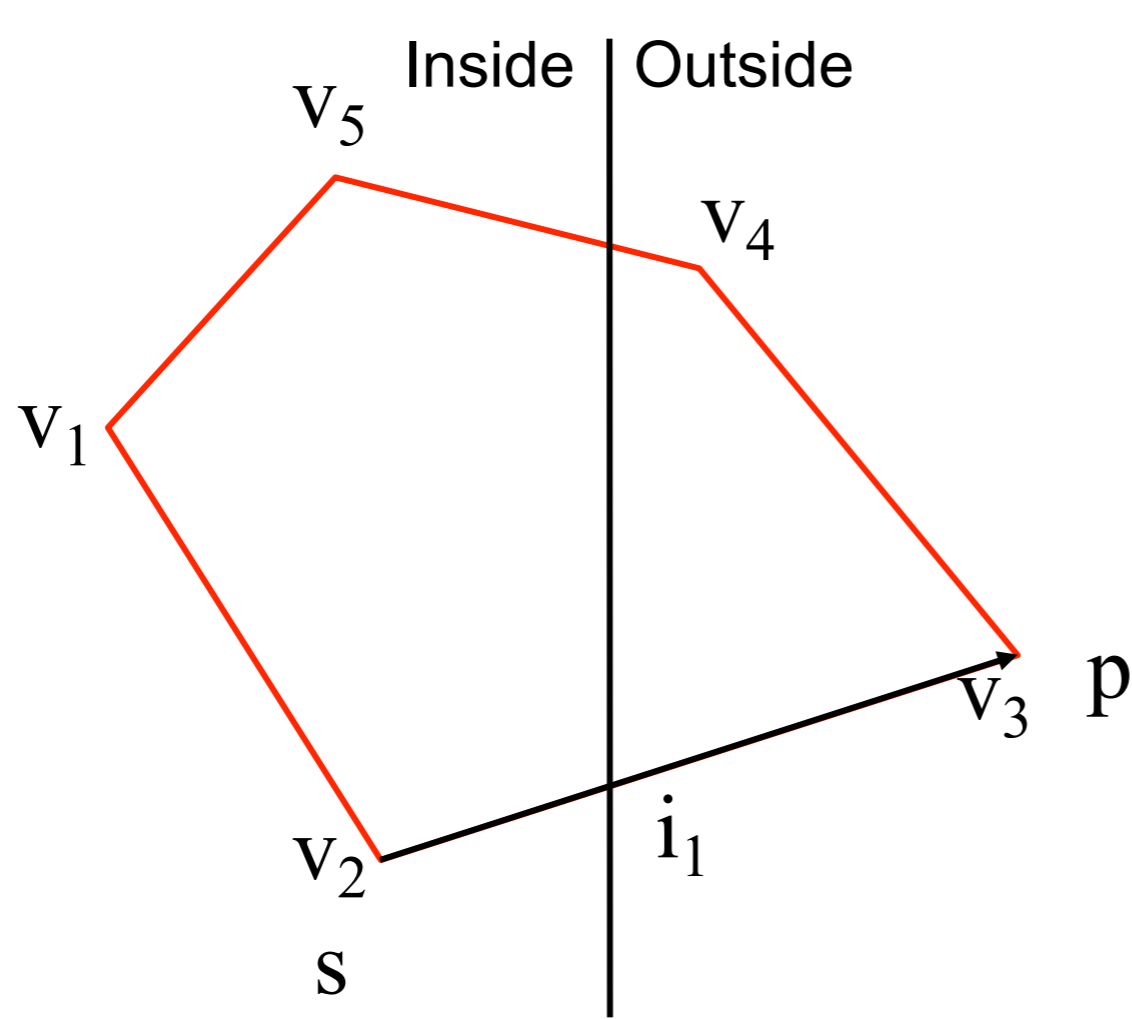
Inside, Outside
Output i_1

Inside | Outside

v_1+

i_1

v_2

Current
Output

Zhejiang University

Computer Graphics 2016, ZJU

v_3, v_4

Inside | Outside

v_5

v_4 p

v_1

s

v_3

v_2
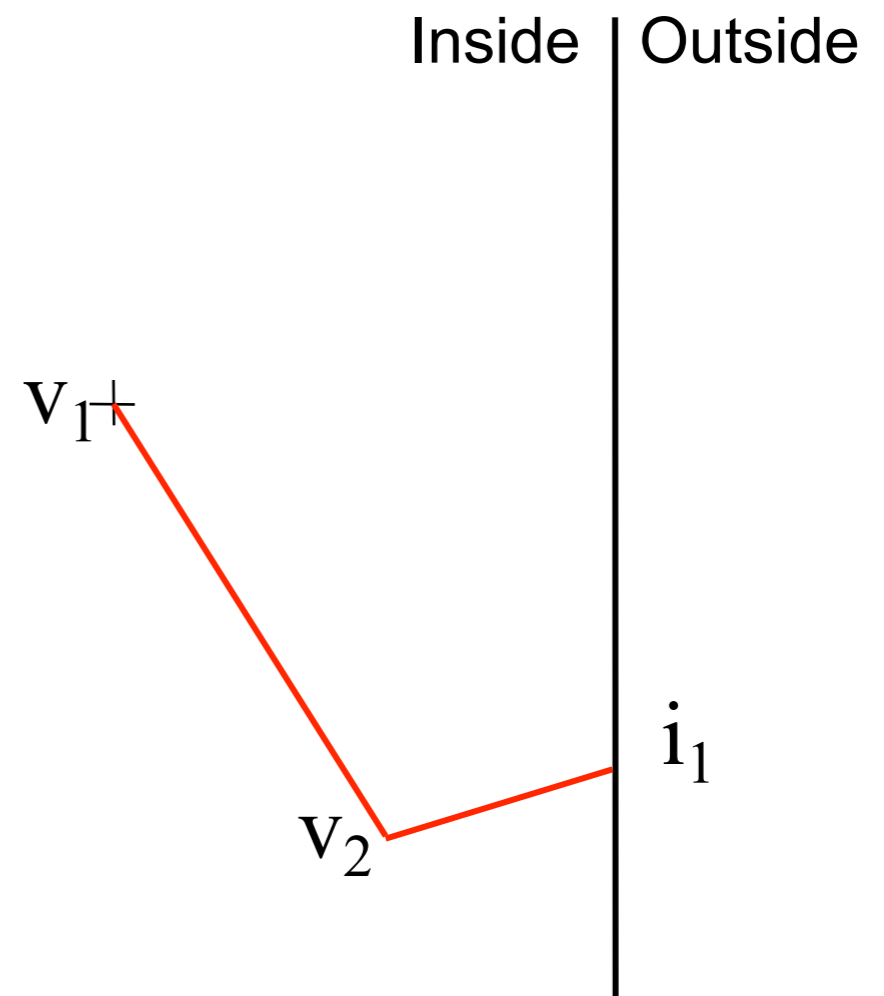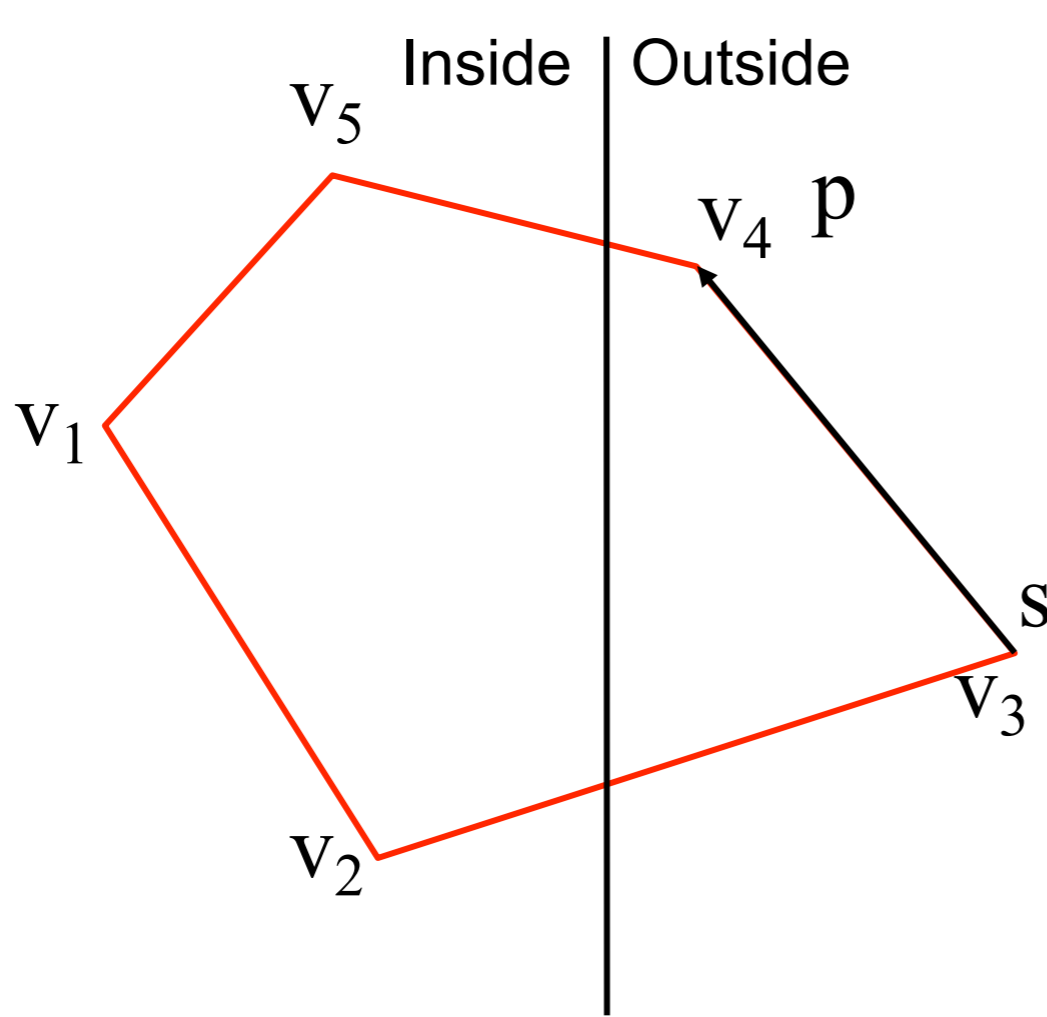
Outside, Outside
No output

Inside | Outside

v_1

i_1

v_2
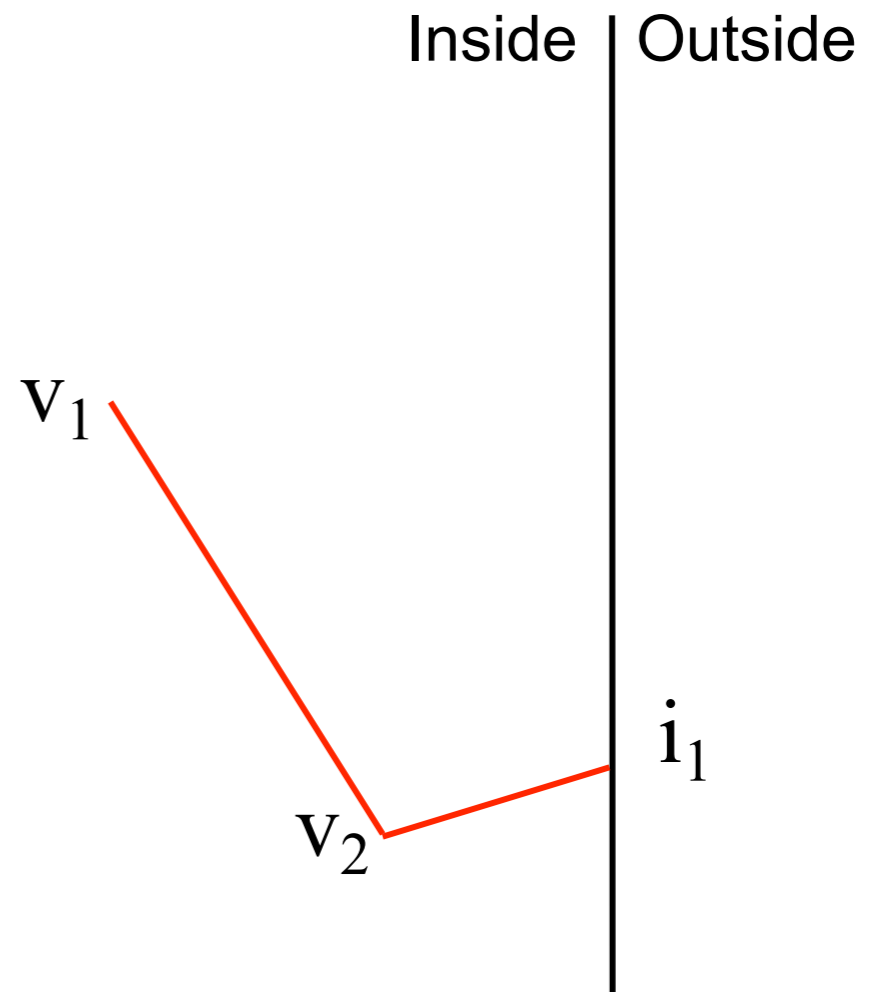
Current
Output

v_4, v_5 – last edge...
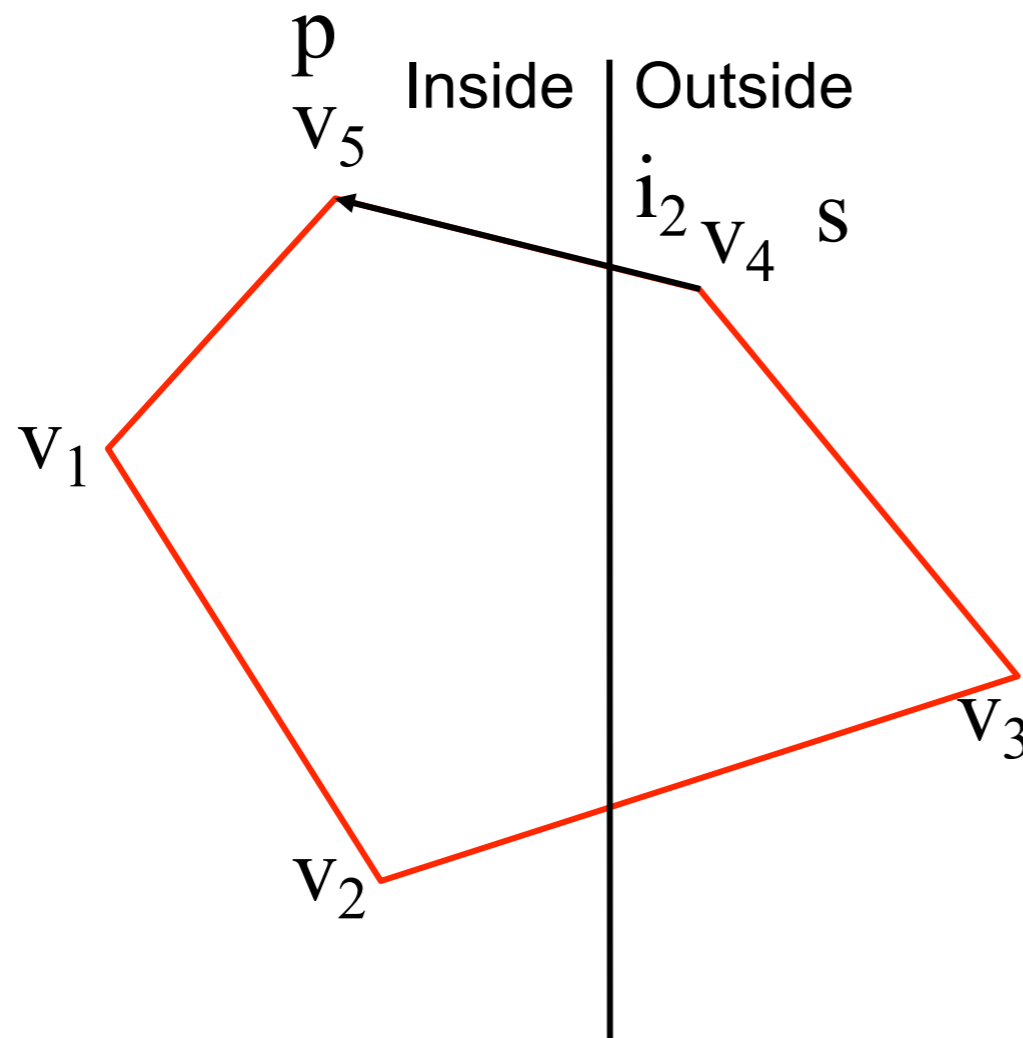
Outside, Inside
Output i_2, v_5
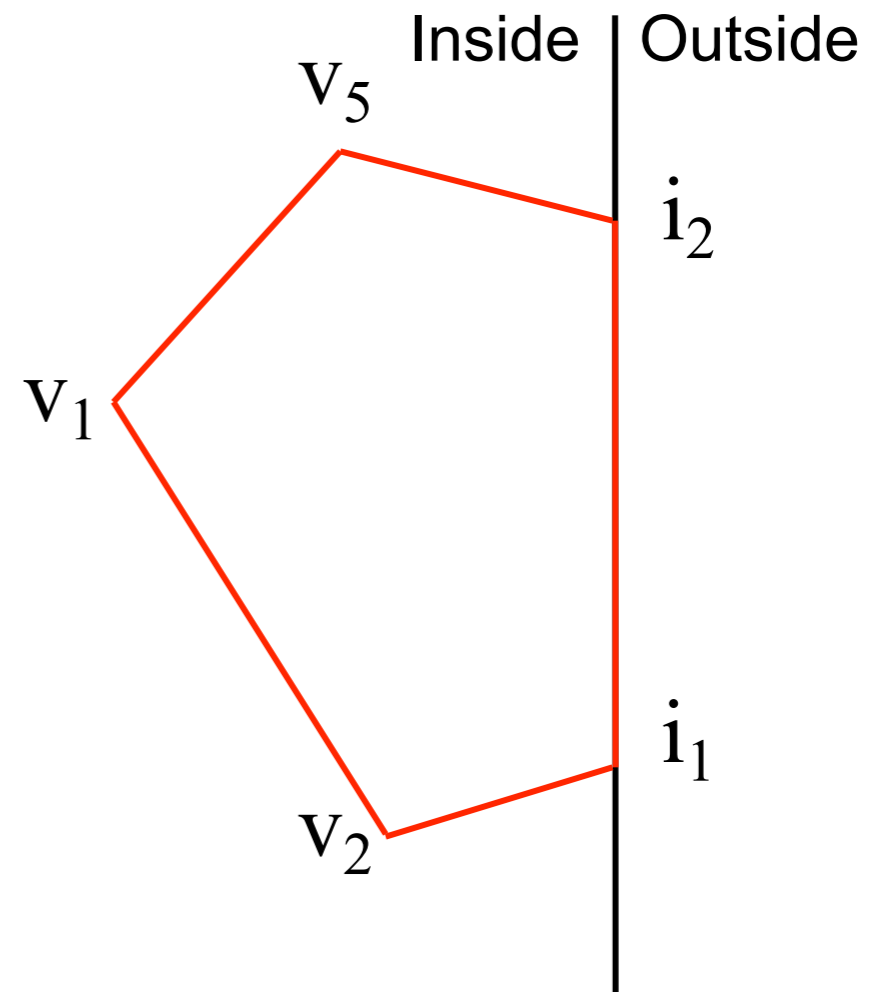
Current
Output

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

$$x => x + T_x$$

$$y => y + T_y$$

$$z => z + T_z$$
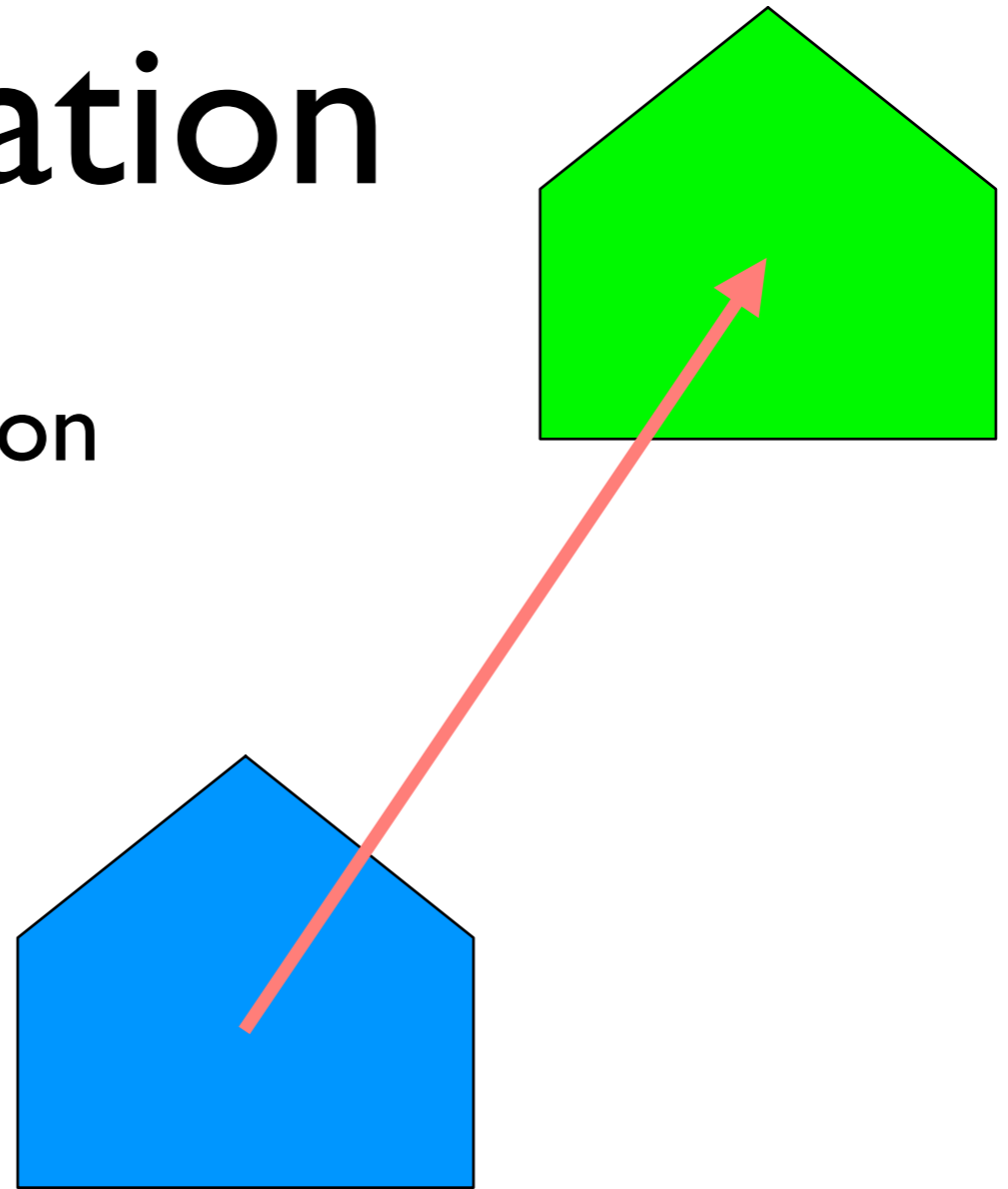
- Translation vector (T_x, T_y, T_z) or shift vector

Translating an Object

(T_x, T_y)
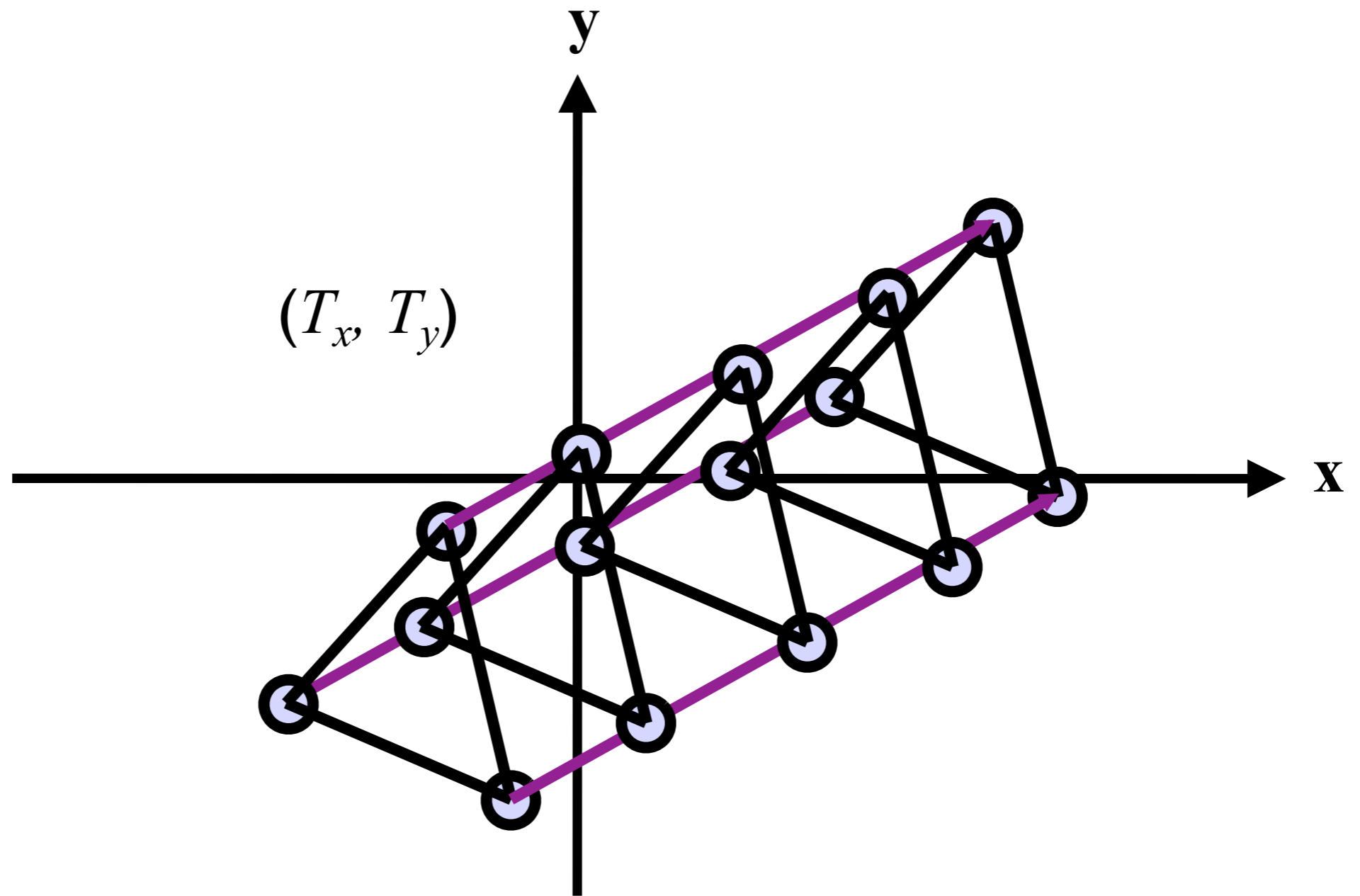
Scaling
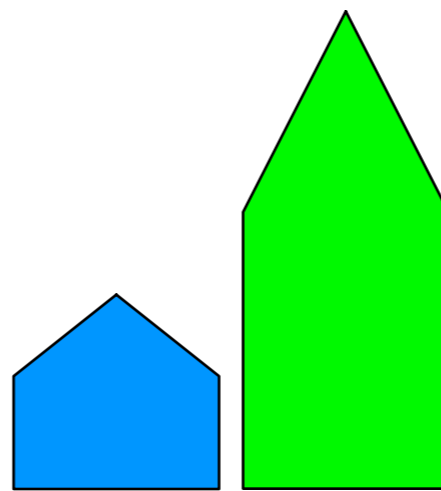
- Changing the size of an object

$x \Rightarrow x * S_x$
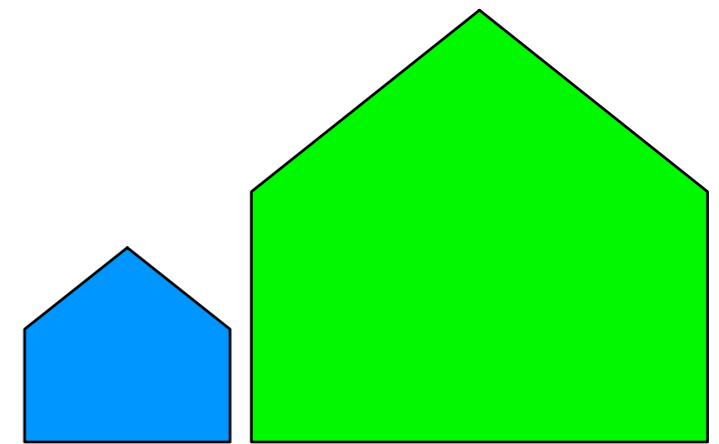
$y \Rightarrow y * S_y$

$z \Rightarrow z * S_z$
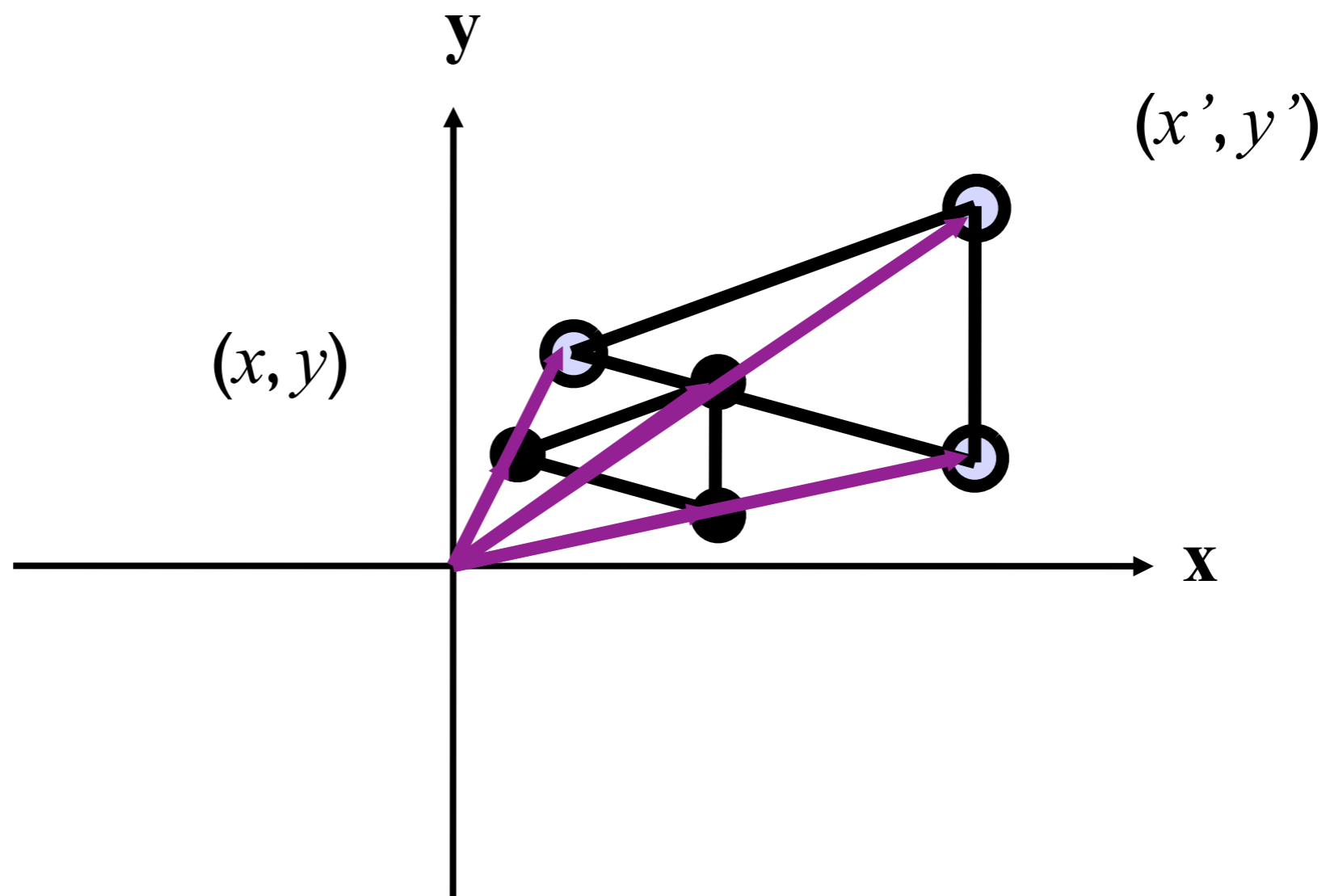
- Scale factor (S_x, S_y, S_z)
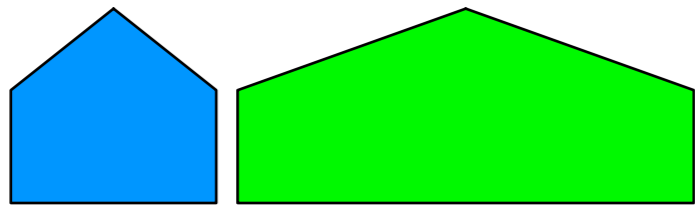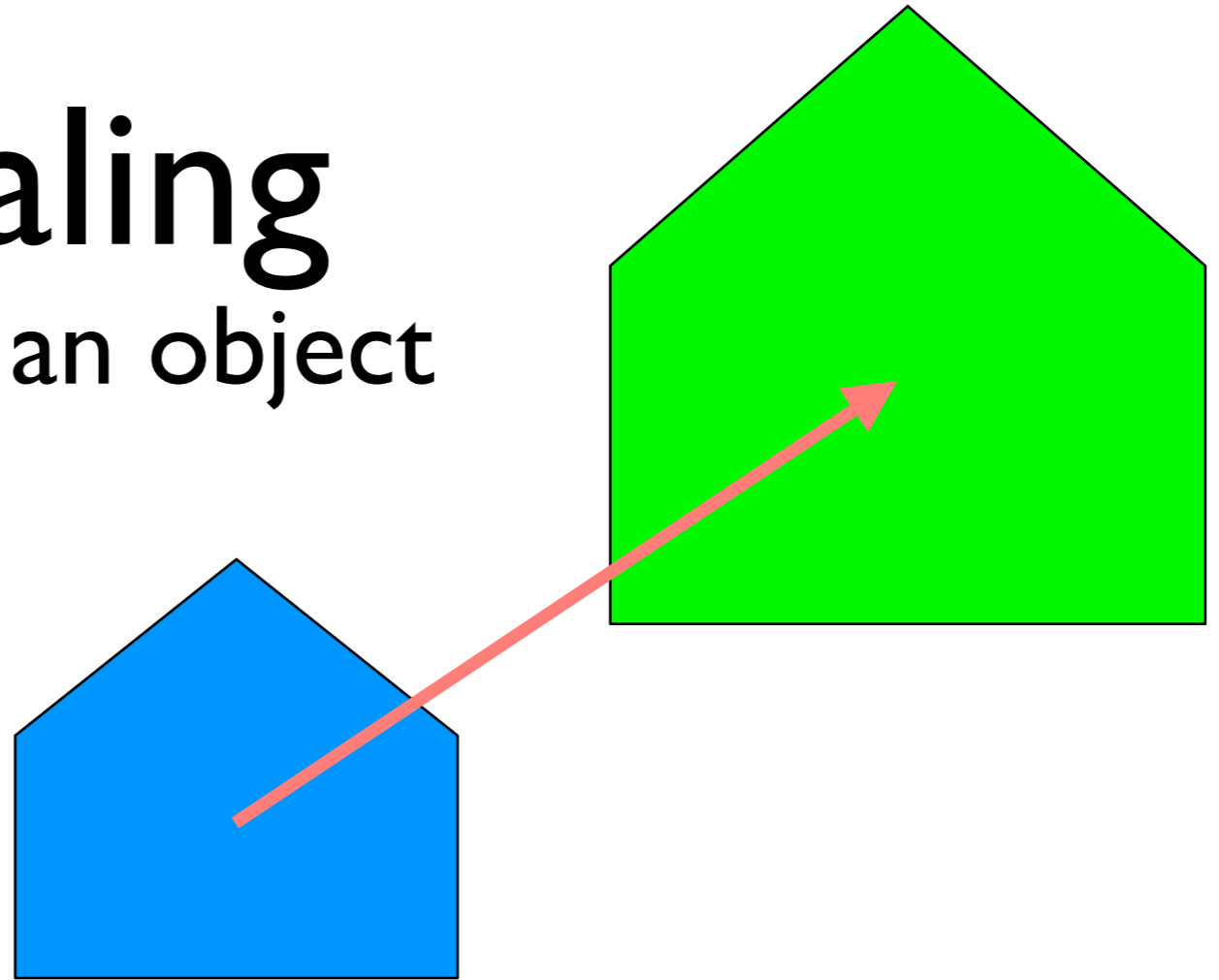
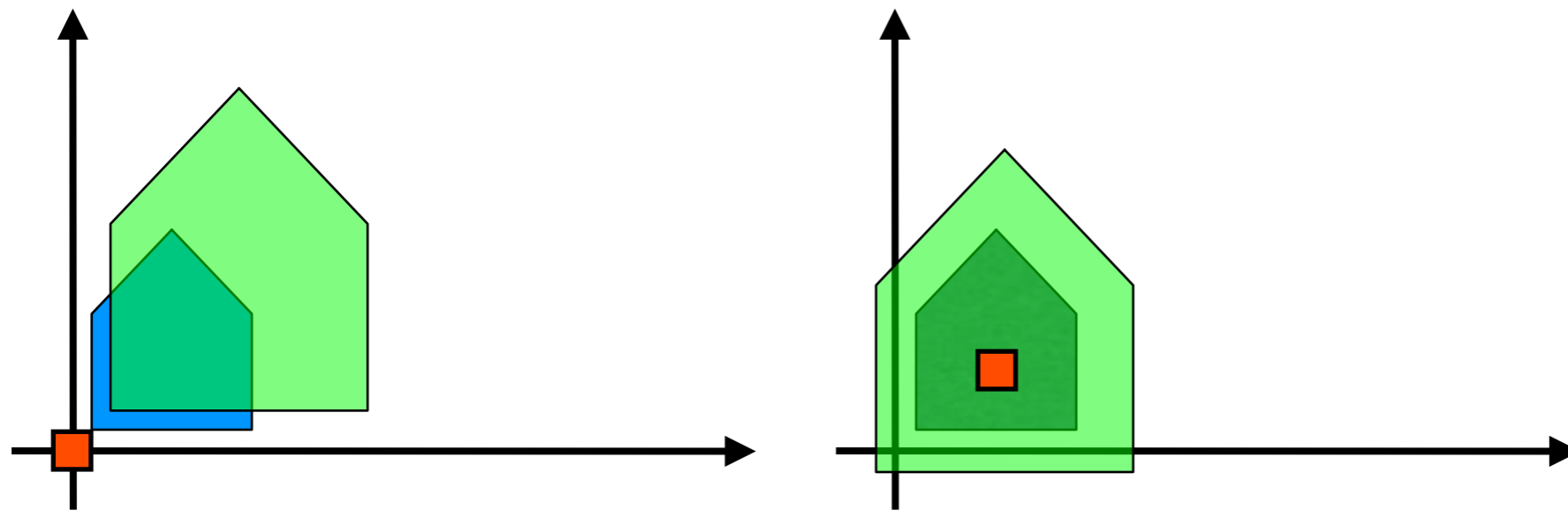$S_y = 1$

$S_x = 1$

$S_x = S_y$

Scaling an Object
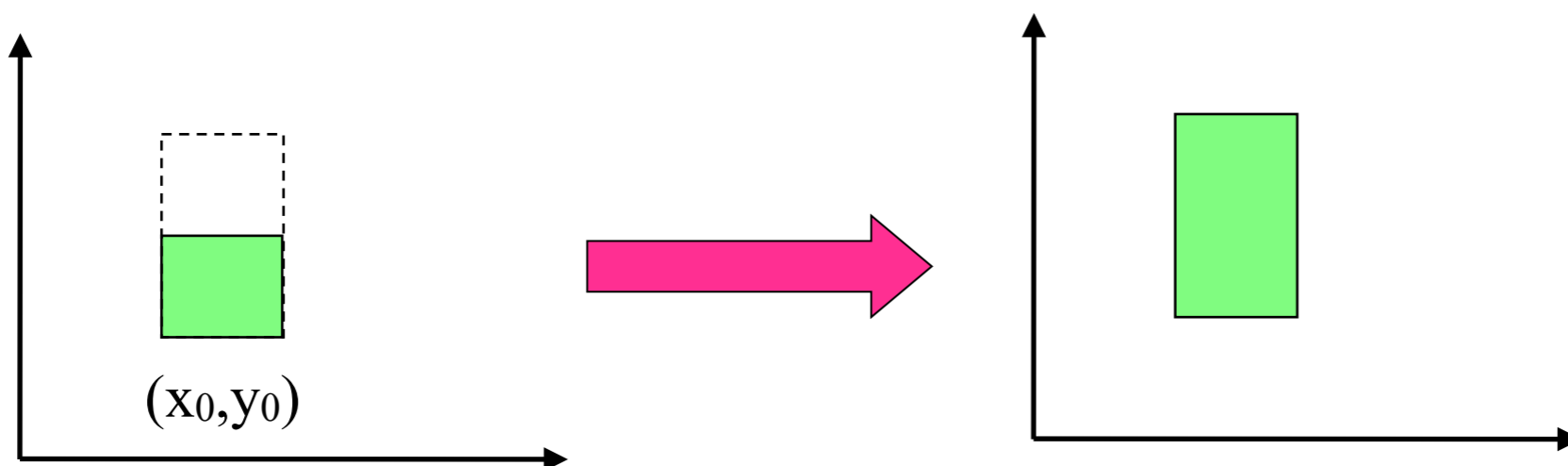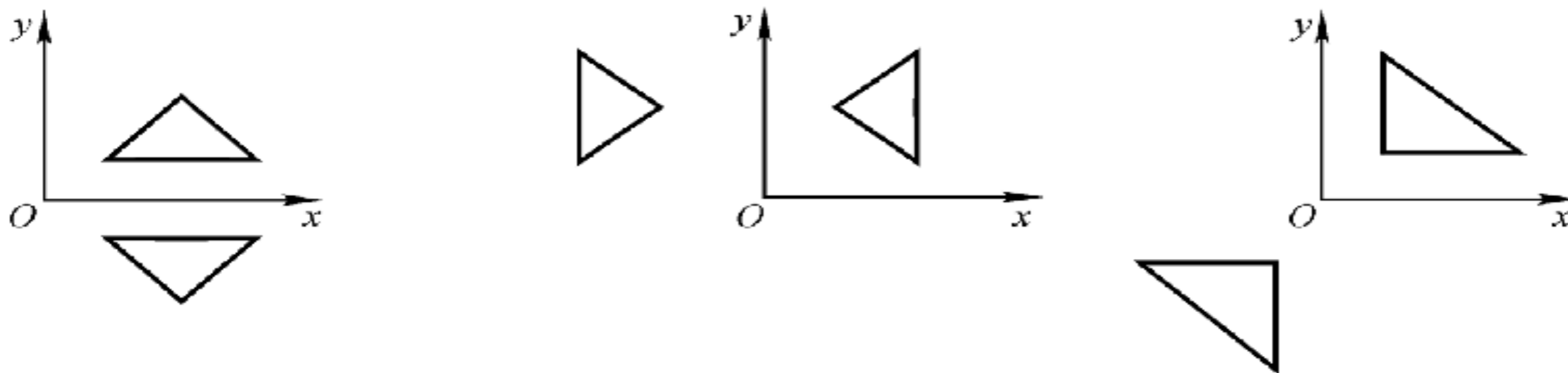

(x',y')

(x,y)

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**
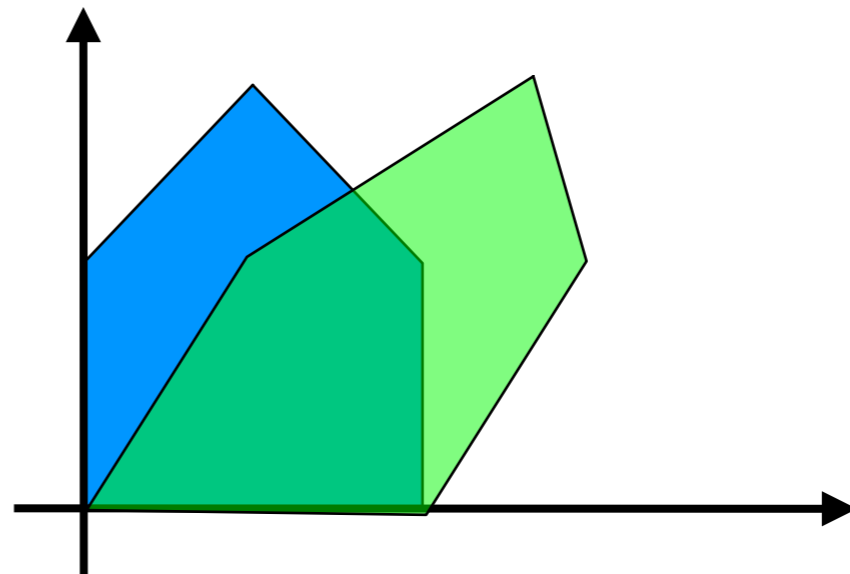
Scaling an Object

(x_0,y_0)

Question8 : How ?

Shearing

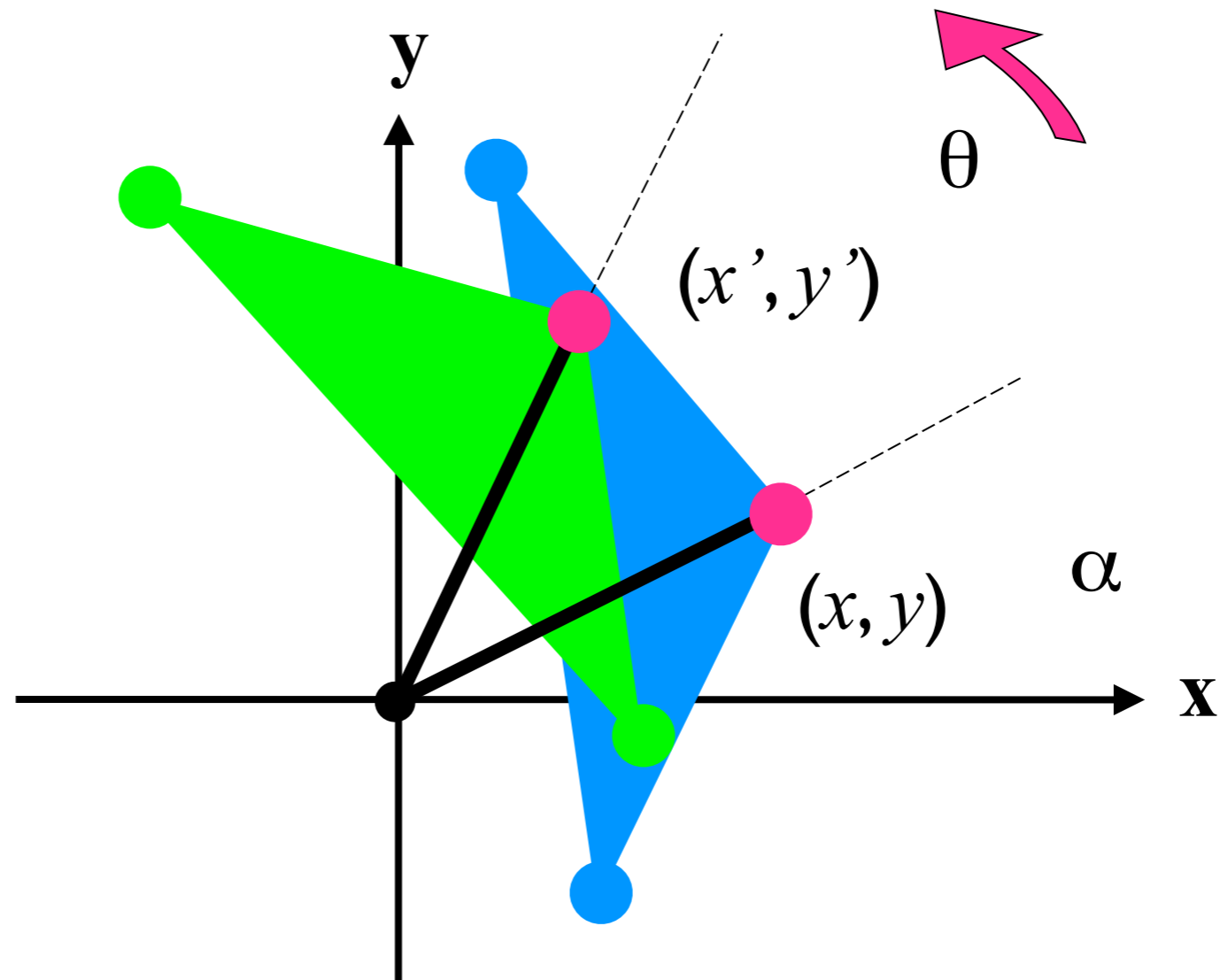- Produces shape distortions
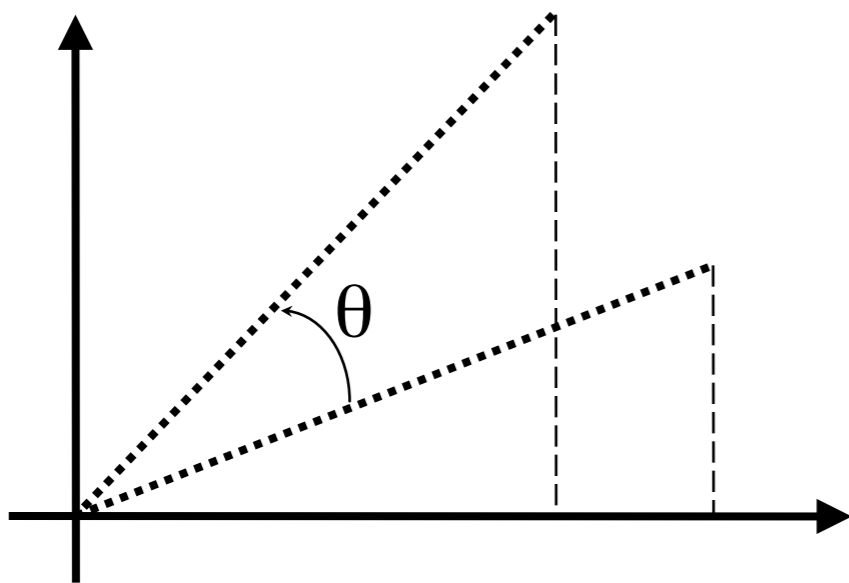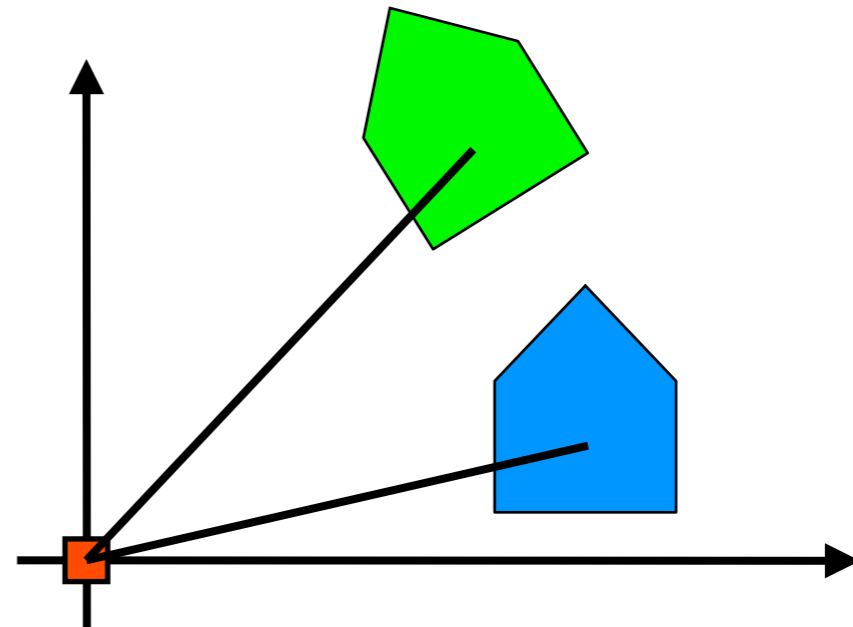
- Shearing in x-direction

$x => x + a*y$

$y => y$

$z => z$

Rotation

Rotation

- Is a Rigid Body Transformation

$x = x * \cos(\theta) - y * \sin(\theta)$

$y = x * \sin(\theta) + y * \cos(\theta)$

$z = z$

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

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)

$newx = x - x_r$

$newy = y - y_r$

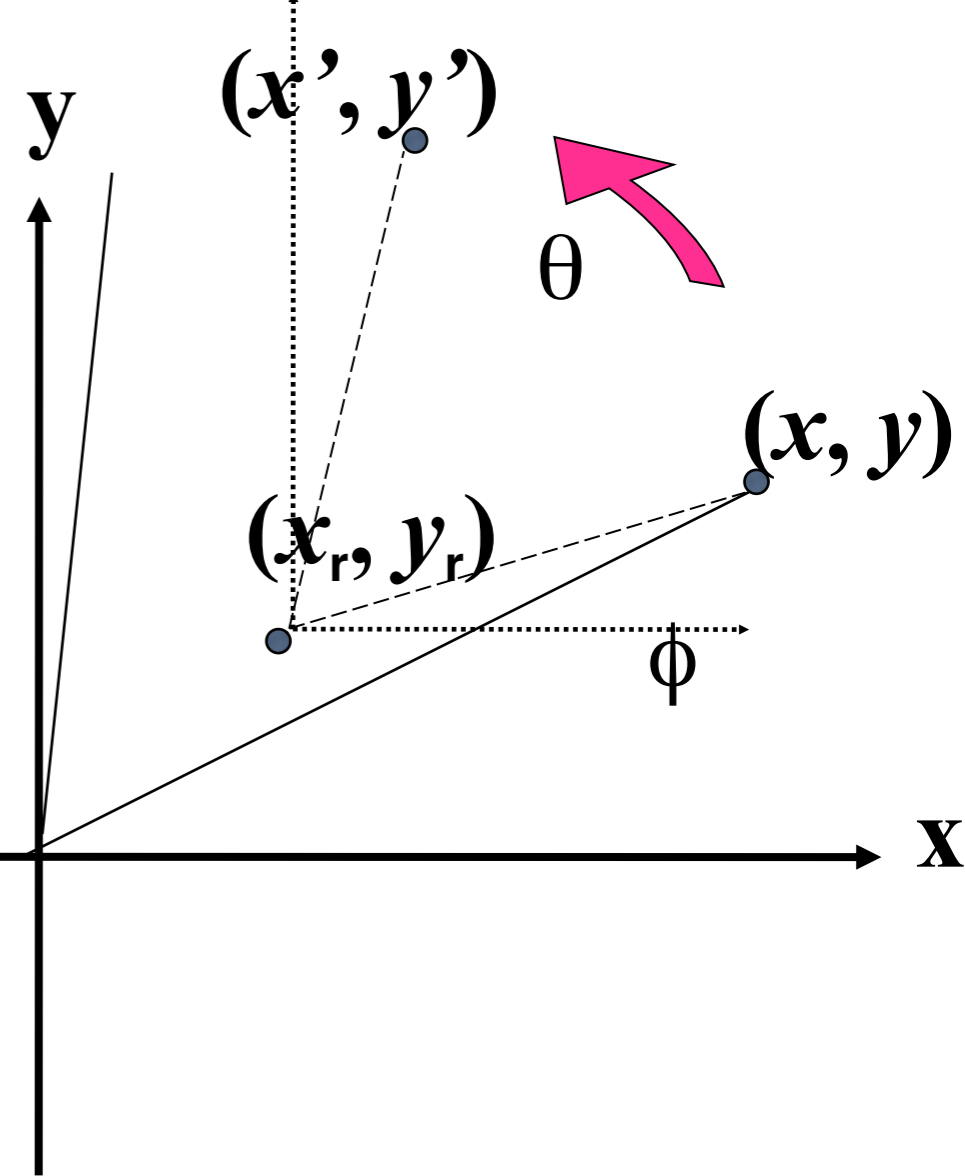$newx' = newx \cos\theta - newy \sin\theta$

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

$x' = newx' + x_r$

$y' = newy' + y_r$

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

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

General Linear Transformation

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

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

  - Translation

  - Scaling

  - Rotation