

Computer Graphics 2015

4. Primitive Attributes

Hongxin Zhang

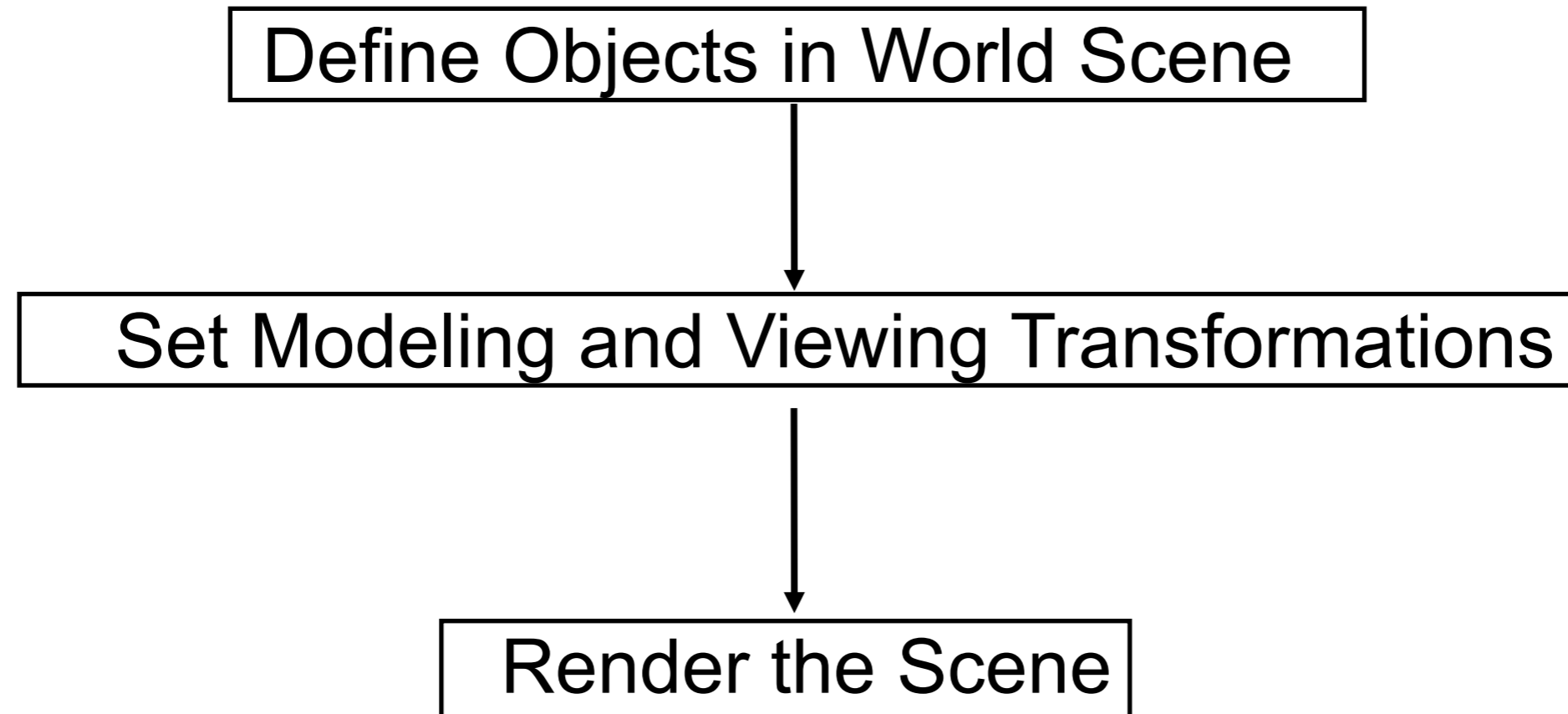
State Key Lab of CAD&CG, Zhejiang University

2016-10-17

Previous lessons

- Rasterization
 - line
 - circle /ellipse ? => homework
- OpenGL and its **rendering pipeline**

3 Stages in OpenGL



Example Code

```
int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (
        GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);

    glutInitWindowPosition(100,100);
    glutInitWindowSize(300,300);
    glutCreateWindow ("square");

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 10.0, 0.0, 10.0, -1.0, 1.0);

    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

```
void display(void)
{
    glClear( GL_COLOR_BUFFER_BIT);
    glColor3f(0.0, 1.0, 0.0);
    glBegin(GL_POLYGON);
        glVertex3f(2.0, 4.0, 0.0);
        glVertex3f(8.0, 4.0, 0.0);
        glVertex3f(8.0, 6.0, 0.0);
        glVertex3f(2.0, 6.0, 0.0);
    glEnd();
    glFlush();
}
```

Attribute parameters

- How to generate different display effects?
 - per primitive (C++)
 - system owns states (OpenGL)
- **OpenGL is a state machine!**

State parameters of OpenGL

- Attributes are assigned by OpenGL state functions:
 - color, matrix mode, buffer positions, Light ...
 - on state paras in this lesson

OpenGL Primitives

- GL_POINTS
- GL_LINES
- GL_LINE_STRIP
- GL_LINE_LOOP
- GL_TRIANGLES
- GL_QUADS
- GL_POLYGON
- GL_TRIANGLE_STRIP
- GL_TRIANGLE_FAN
- GL_QUAD_STRIP

1. GL_POLYGON and GL_TRIANGLE are the only ones in common usage

2. valid OpenGL polygons are closed, convex, co-planar and non-intersecting, which is always true for triangles!

Examples

```
glBegin(GL_POLYGON);  
    glVertex2i(0,0);  
    glVertex2i(0,1);  
    glVertex2i(1,1);  
    glVertex2i(1,0);  
glEnd();
```



```
glBegin(GL_POINTS);  
    glVertex2i(0,0);  
    glVertex2i(0,1);  
    glVertex2i(1,1);  
    glVertex2i(1,0);  
glEnd();
```



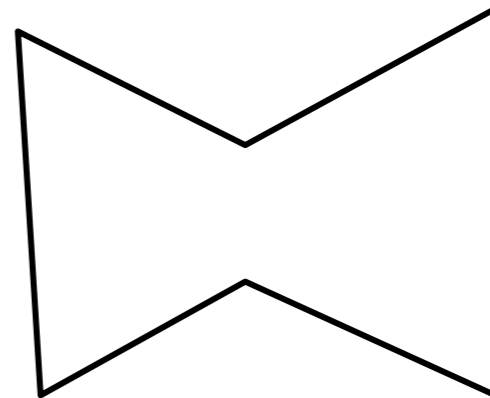
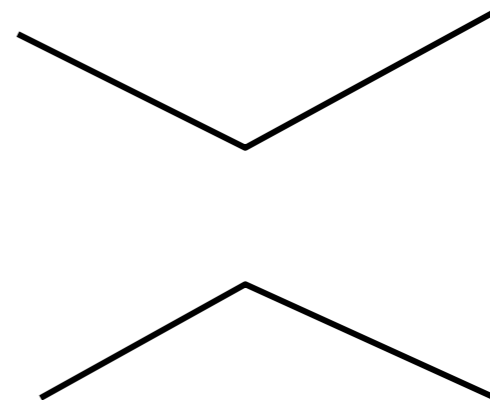
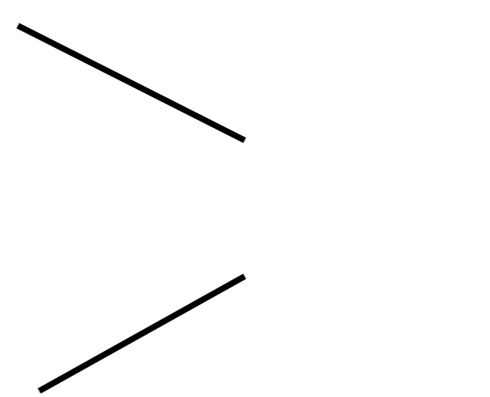
Examples

```
GLfloat list[6][2];
```

```
    glBegin(GL_LINES)  
    for (int i = 0 ; i < 6 ; i++)  
        glVertex2v(list[i]);  
    glEnd();
```

```
    glBegin(GL_LINE_STRIP)  
    for (int i = 0 ; i < 6 ; i++)  
        glVertex2v(list[i]);  
    glEnd();
```

```
    glBegin(GL_LINE_LOOP)  
    for (int i = 0 ; i < 6 ; i++)  
        glVertex2v(list[i]);  
    glEnd();
```

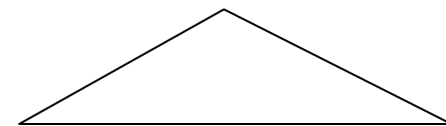
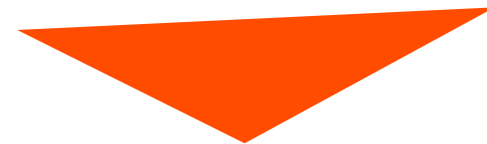
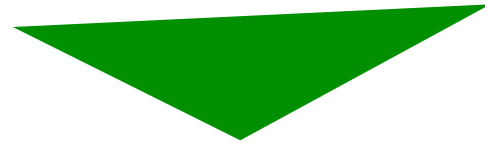


Examples

```
GLfloat list[6][2] ;
```

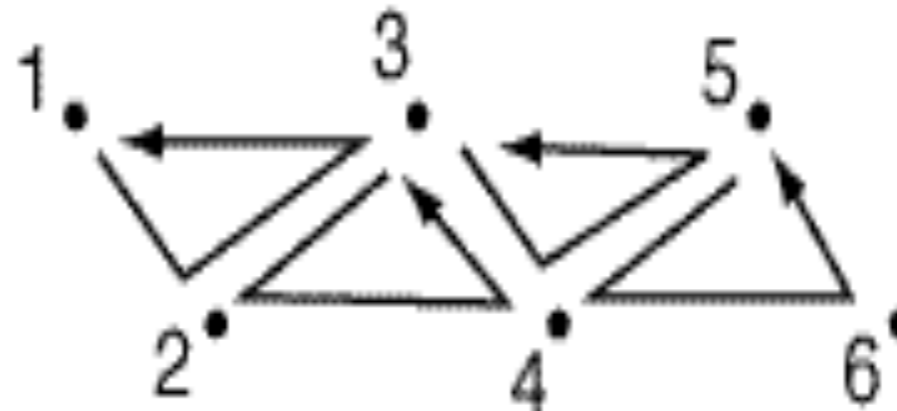
```
glColor3f(0.0, 1.0, 0.0);  
glBegin(GL_TRIANGLES)  
  for (int i = 0 ; i < 6 ; i++)  
    glVertex2v(list[i]);  
glEnd() ;
```

```
glBegin(GL_TRIANGLES)  
  glColor3f(1.0, 0.0, 0.0);  
  for ( i = 0 ; i < 3 ; i++)  
    glVertex2v(list[i]);  
  glColor3f(1.0, 1.0, 1.0);  
  for ( i = 3 ; i < 6 ; i++)  
    glVertex2v(list[i]);  
glEnd() ;
```

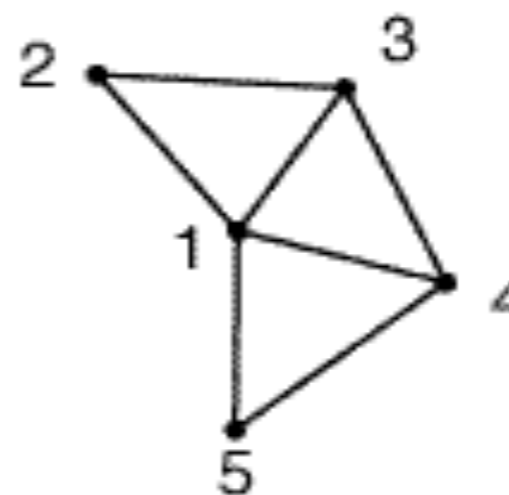


Examples

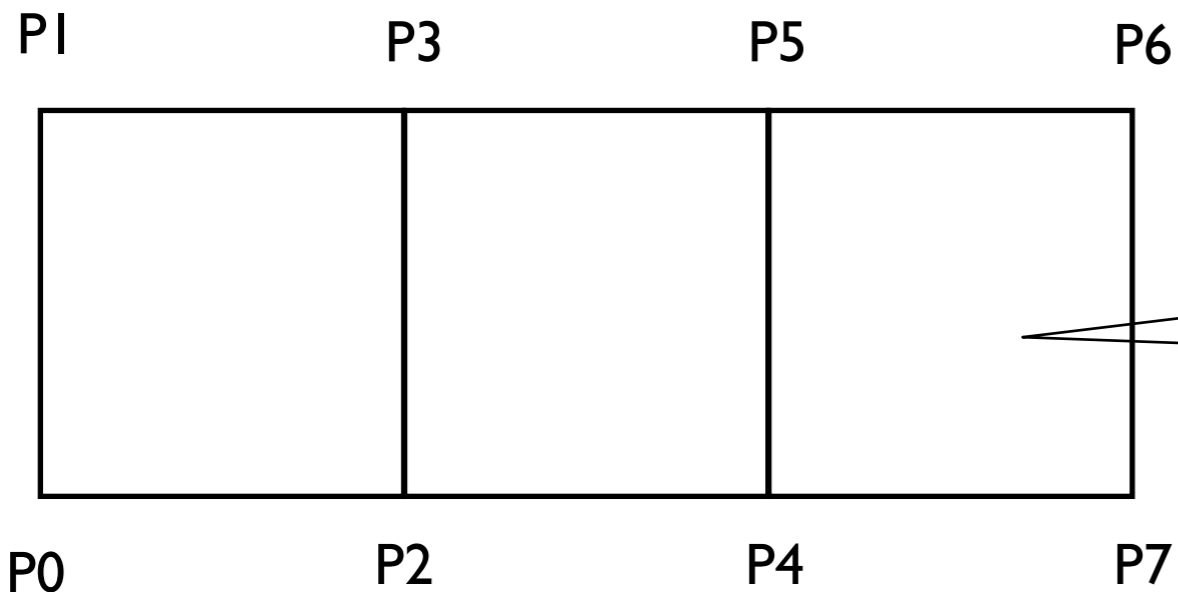
GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN



GL_QUAD_STRIP



Must be planar convex

OpenGL Command Syntax

- All command names begin with **gl**
 - Ex.: `glVertex3f(0.0, 1.0, 1.0);`
- Constant names are in all uppercase
 - Ex.: `GL_COLOR_BUFFER_BIT`
- Data types begin with **GL**
 - Ex.: `GLfloat onevertex[3];`
- Most commands end in two characters that determine the data type of expected arguments
 - Ex.: `glVertex3f(...) => 3 GLfloat arguments`

glVertex

- All primitives are defined in terms of vertices
 - glVertex2f(x, y);
 - glVertex3f(x, y, z);
 - glVertex4f(x, y, z, w);
 - glVertex3fv(a); // with a[0], a[1], a[2]

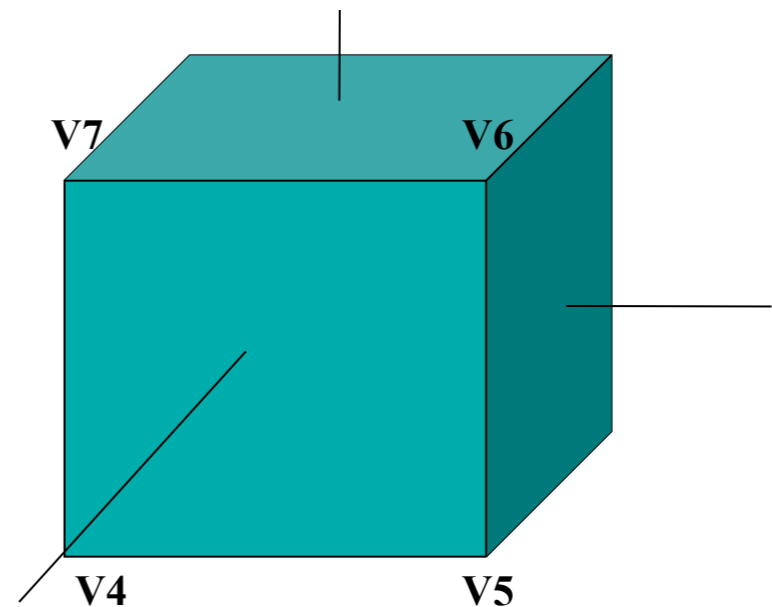
Building Objects From Vertices

- Specify a primitive mode, and enclose a set of vertices in a glBegin / glEnd block
- glBegin(GL_POLYGON);
- glVertex3f(1.0, 2.0, 0.0);
- glVertex3f(0.0, 0.0, 0.0);
- glVertex3f(3.0, 0.0, 0.0);
- glVertex3f(3.0, 2.0, 0.0);
- glEnd();

OpenGL Example

```
void drawOneCubeface(size)
{
    static GLfloat v[8][3];
    v[0][0] = v[3][0] = v[4][0] = v[7][0] = -size/2.0;
    v[1][0] = v[2][0] = v[5][0] = v[6][0] = size/2.0;
    v[0][1] = v[1][1] = v[4][1] = v[5][1] = -size/2.0;
    v[2][1] = v[3][1] = v[6][1] = v[7][1] = size/2.0;
    v[0][2] = v[1][2] = v[2][2] = v[3][2] = -size/2.0;
    v[4][2] = v[5][2] = v[6][2] = v[7][2] = size/2.0;

    glBegin(GL_POLYGON);
    glVertex3fv(v[0]);
    glVertex3fv(v[1]);
    glVertex3fv(v[2]);
    glVertex3fv(v[3]);
    glEnd();
}
```



Real examples in OpenGL|ES

```
float afVertices [] = {...};
```

```
glEnableVertexAttribArray(0);
```

```
glVertexAttribPointer(VERTEX_ARRAY, GL_FLOAT, GL_FALSE, afVertices);
```

```
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
```

Note: there is NO **glBegin/glVertex/glEnd** in OpenGL|ES

Colors

- OpenGL colors are typically defined as RGB components
 - each of which is a float in the range [0.0, 1.0]
- For the screen's background:
 - `glClearColor(0.0, 0.0, 0.0); // black color`
 - `glClear(GL_COLOR_BUFFER_BIT);`
- For objects:
 - `glColor3f(1.0, 1.0, 1.0); // white color`

Other Commands in glBegin / glEnd blocks

- Not every OpenGL command can be located in such a block. Those that can include, among others:
 - glColor
 - glNormal (to define a normal vector)
 - glTexCoord (to define texture coordinates)
 - glMaterial (to set material properties)

Example

```
glBegin( GL_POLYGON );  
    glColor3f( 1.0, 1.0, 0.0 ); glVertex3f( 0.0, 0.0, 0.0 );  
    glColor3f( 0.0, 1.0, 1.0 ); glVertex3f( 5.0, 0.0, 0.0 );  
    glColor3f( 1.0, 0.0, 1.0 ); glVertex3f( 0.0, 5.0, 0.0 );  
glEnd();
```

Polygon Display Modes

- `glPolygonMode(GLenum face, GLenum mode);`
 - Faces: `GL_FRONT`, `GL_BACK`, `GL_FRONT_AND_BACK`
 - Modes: `GL_FILL`, `GL_LINE`, `GL_POINT`
 - By default, both the front and back face are drawn filled
- `glFrontFace(GLenum mode);`
 - Mode is either `GL_CCW` (default) or `GL_CW`
- `glCullFace(GLenum mode);`
 - Mode is either `GL_FRONT`, `GL_BACK`, `GL_FRONT_AND_BACK`;
- You must enable and disable culling with
 - `glEnable(GL_CULL_FACE)` or `glDisable(GL_CULL_FACE);`

Drawing Other Objects

- GLU contains calls to draw cylinders, cones and more complex surfaces called NURBS
- GLUT contains calls to draw spheres and cubes

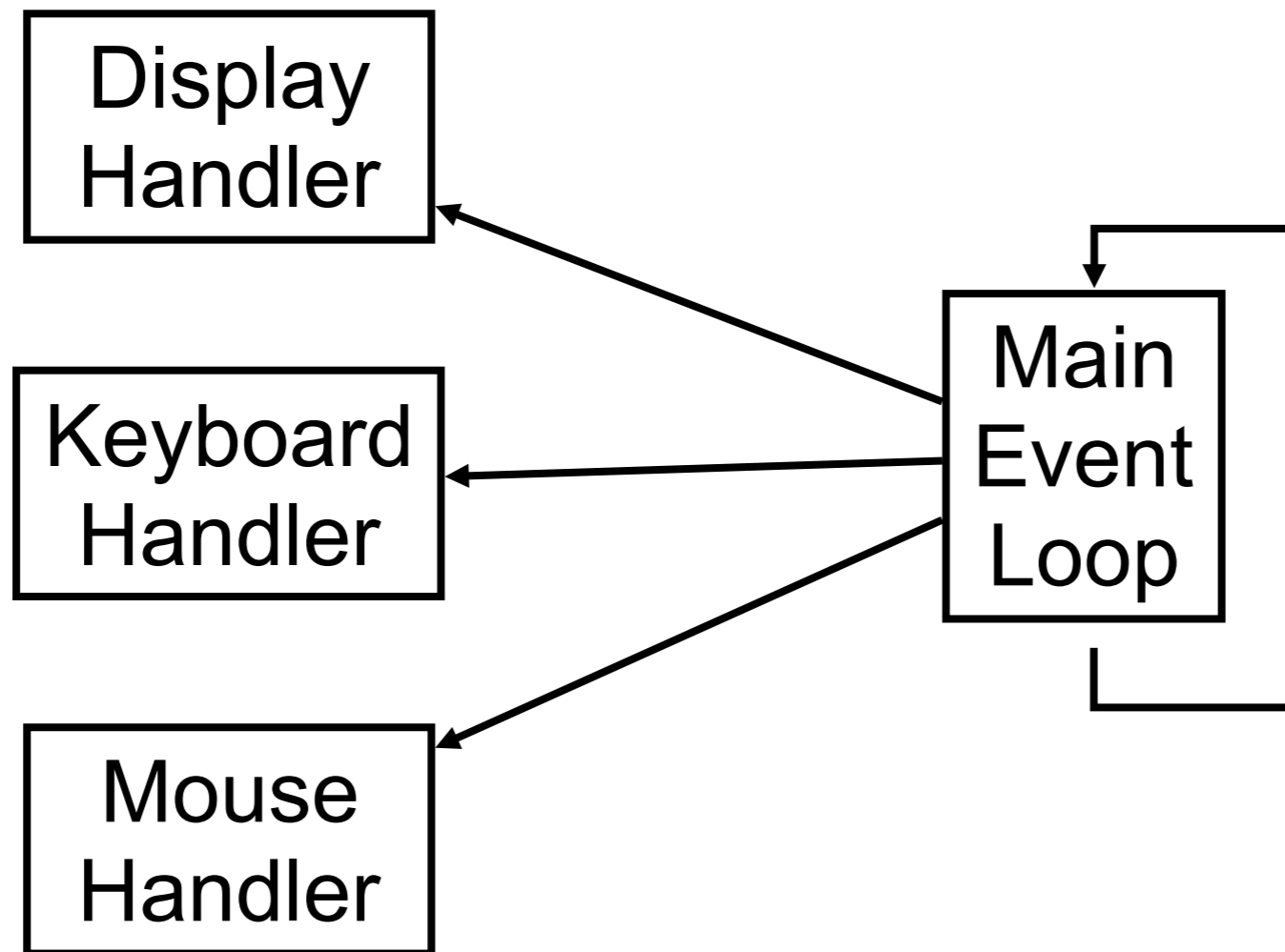
Compiling OpenGL Programs

- To use GLUT :
 - `#include <GL/glut.h>`
 - This takes care of every other include you need
 - Make sure that `glut.lib` (or `glut32.lib`) is in your compiler's library directory, and that the object module or DLL is also available
- See *OpenGL Game Programming* or online tutorials for details

Structure of GLUT-Assisted Programs

- GLUT relies on user-defined callback functions, which it calls whenever some event occurs
- Function to display the screen
- Function to resize the viewport
- Functions to handle keyboard and mouse events

Event Driven Programming



Simple GLUT Example

Displaying a square

```
int main (int argc, char *argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);

    int windowHandle
        = glutCreateWindow("Simple GLUT App");

    glutDisplayFunc(redraw);
    glutMainLoop();

    return 0;
}
```

Display Callback

Called when window is redrawn

```
void redraw()
{
    glClear(GL_COLOR_BUFFER_BIT);

    glBegin(GL_QUADS);
    glColor3f(1, 0, 0);
    glVertex3f(-0.5, 0.5, 0.5);
    glVertex3f( 0.5, 0.5, 0.5);
    glVertex3f( 0.5, -0.5, 0.5);
    glVertex3f(-0.5, -0.5, 0.5);
    glEnd(); // GL_QUADS

    glutSwapBuffers();
}
```

More GLUT

Additional GLUT functions

```
glutPositionWindow(int x,int y);  
glutReshapeWindow(int w, int h);
```

Additional callback functions

```
glutReshapeFunction(reshape);  
glutMouseFunction(mousebutton);  
glutMotionFunction(motion);  
glutKeyboardFunction(keyboardCB);  
glutSpecialFunction(special);  
glutIdleFunction(animate);
```

Reshape Callback

Called when the window is resized

```
void reshape(int w, int h)
{
    glViewport(0.0,0.0,w,h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0,w,0.0,h, -1.0, 1.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

Mouse Callbacks

Called when the mouse button is pressed

```
void mousebutton(int button, int state, int x, int y)
{
    if (button==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
    {
        rx = x; ry = windowHeight - y;
    }
}
```

Called when the mouse is moved with button down

```
void motion(int x, int y)
{
    rx = x; ry = windowHeight - y;
}
```

Keyboard Callbacks

Called when a button is pressed

```
void keyboardCB(unsigned char key, int x, int y)
{
    switch(key)
    { case 'a': cout<<"a Pressed"<<endl; break; }
}
```

Called when a special button is pressed

```
void special(int key, int x, int y)
{
    switch(key)
    { case GLUT_F1_KEY:
      cout<<"F1 Pressed"<<endl; break; }
}
```

OpenGL – GLUT Example

```
#include <gl/glut.h>
#include <stdlib.h>
static GLfloat spin = 0.0;
void init( void )
{
    glClearColor( 0.0, 0.0, 0.0, 0.0 );
    glShadeModel( GL_FLAT );
}
```

```
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT );
    glPushMatrix();
    glRotatef( spin, 0.0, 0.0, 1.0 );
    glColor3f( 1.0, 1.0, 1.0 );
    glRectf( -25.0, -25.0, 25.0, 25.0 );
    glPopMatrix();
    glutSwapBuffers();
}
```

OpenGL – GLUT Example

```
void spinDisplay( void )
```

```
{
```

```
    spin += 2.0;
```

```
    if( spin > 360.0 )
```

```
        spin -= 360.0;
```

```
    glutPostRedisplay();
```

```
}
```

```
void reshape( int w, int h )
```

```
{
```

```
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
```

```
    glMatrixMode( GL_PROJECTION );
```

```
    glLoadIdentity();
```

```
    glOrtho( -50.0, 50.0, -50.0, 50.0, -1.0, 1.0 );
```

```
    glMatrixMode( GL_MODELVIEW );
```

```
    glLoadIdentity();
```

```
}
```


OpenGL – GLUT Example

```
void mouse( int button, int state, int x, int y )
{
    switch( button )
    {
        case GLUT_LEFT_BUTTON:
            if( state == GLUT_DOWN )
                glutIdleFunc( spinDisplay );
            break;
        case GLUT_RIGHT_BUTTON:
            if( state == GLUT_DOWN )
                glutIdleFunc( NULL );
            break;
        default:    break;
    }
}
```

OpenGL – GLUT Example

```
int main( int argc, char ** argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGB );
    glutInitWindowSize( 250, 250 );
    glutInitWindowPosition( 100, 100 );
    glutCreateWindow( argv[ 0 ] );

    init();
    glutDisplayFunc( display );
    glutReshapeFunc( reshape );
    glutMouseFunc( mouse );
    glutMainLoop();
    return 0;
}
```

Web Resources

<http://www.opengl.org>

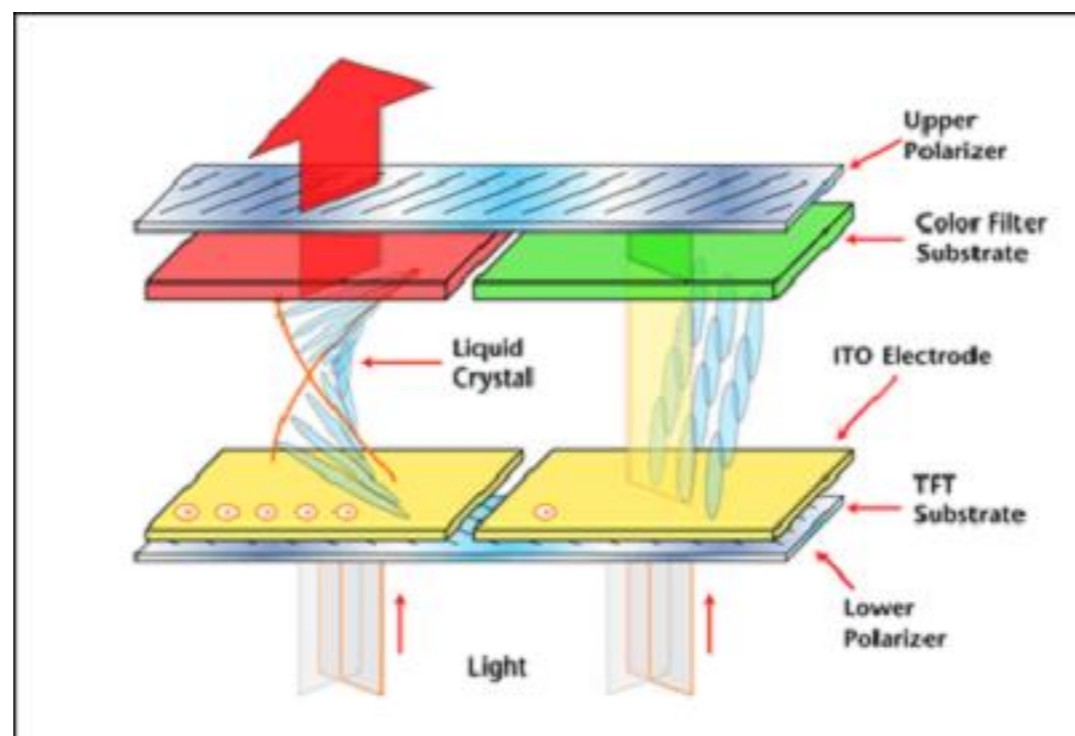
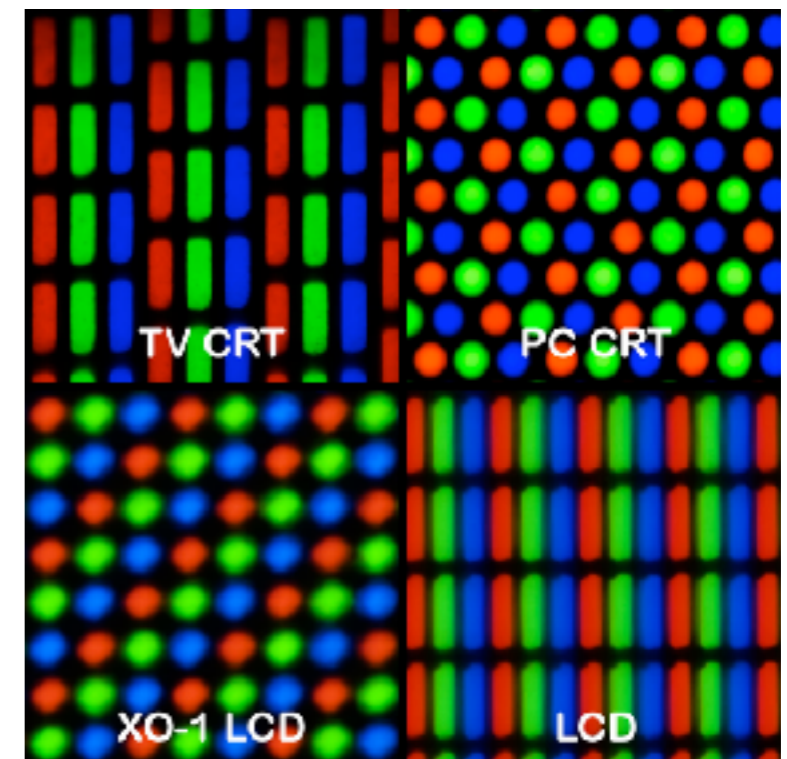
<http://nehe.gamedev.net>

<http://www.xmission.com/~nate/glut.html>

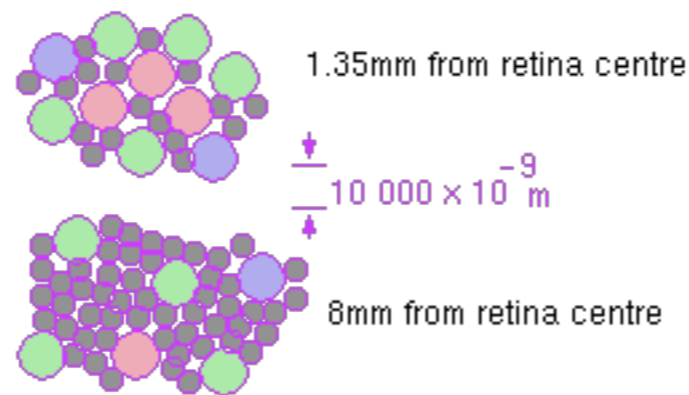
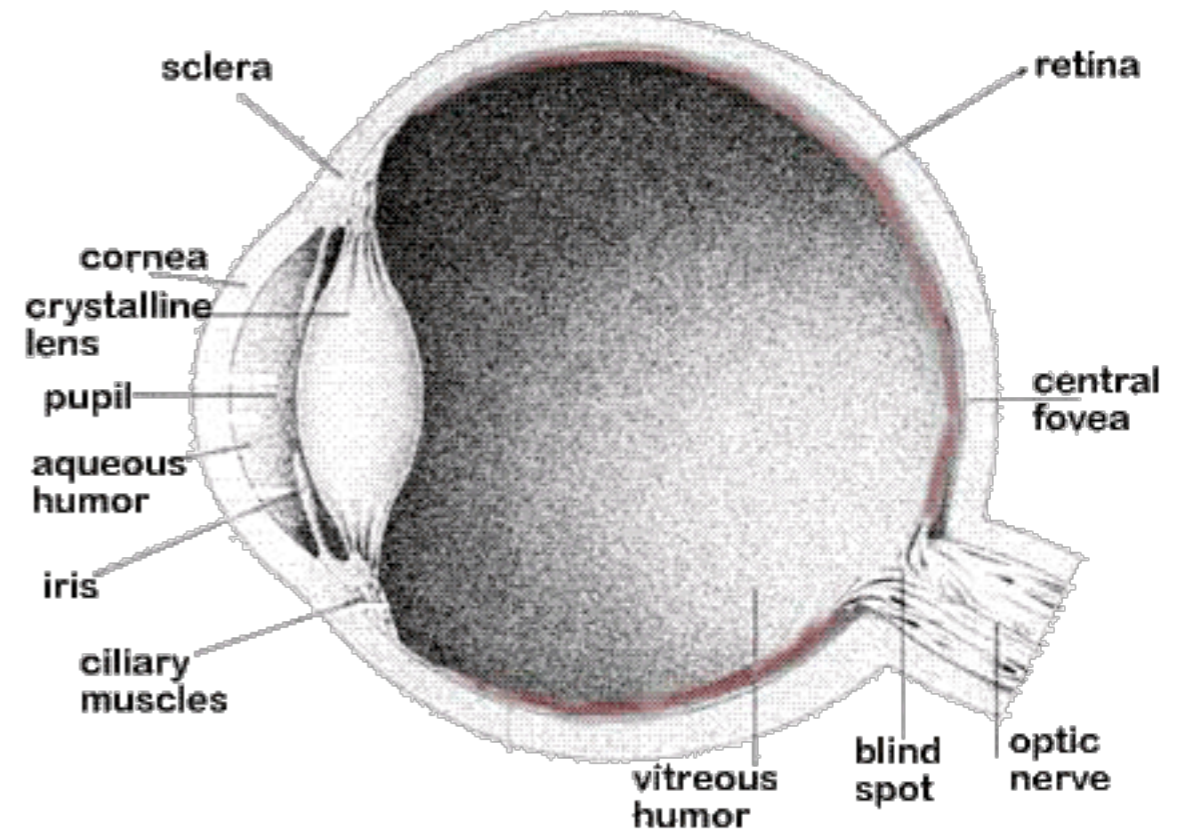
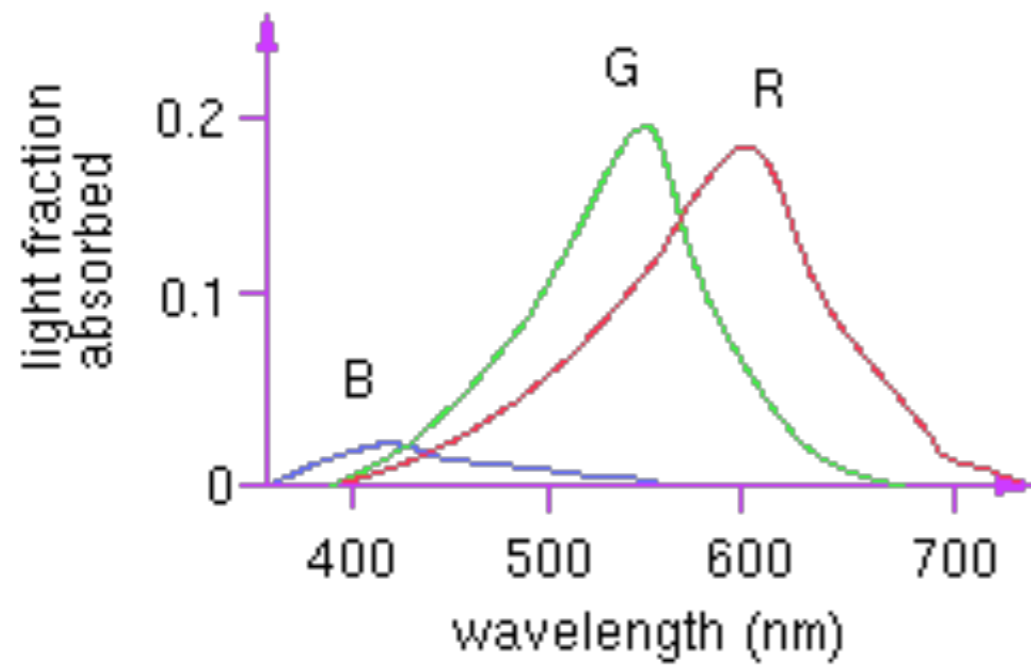
Color and greyscale

- Color is a fundamental primitive attribute
- RGB color model
- Color lookup table / Color map
- Greyscale

Why RGB?



Color Model



Color perception

- Three types of cones:

S	M	L	
Blue	Green	Red	roughly approximate
430nm	560nm	610nm	peak sensitivities

- Colorblindness results from a deficiency of one cone type.

OpenGL Color function

- GLUT_RGB and
- GLUT_RGBA with alpha channel

- glColor3f (1.0, 1.0, 1.0);
- glColor3i (0, 255, 255);
- glColor3fv (colorArray);

OpenGL Color function

- Color index mode
 - `glIndexi (196);`
- Color blending function
 - `glEnable (GL_BLEND);`
 - `glDisable (GL_BLEND);`
 - `glBlendFunc (sFactor, dFactor);`

OpenGL Color Array

- Defined in the latest OpenGL standard
- `glEnableClientState (GL_COLOR_ARRAY);`
- `glColorPointer (...);`
- `glEnableClientState (GL_VERTEX_ARRAY);`
- `glVertexPointer (...);`

Attributes of

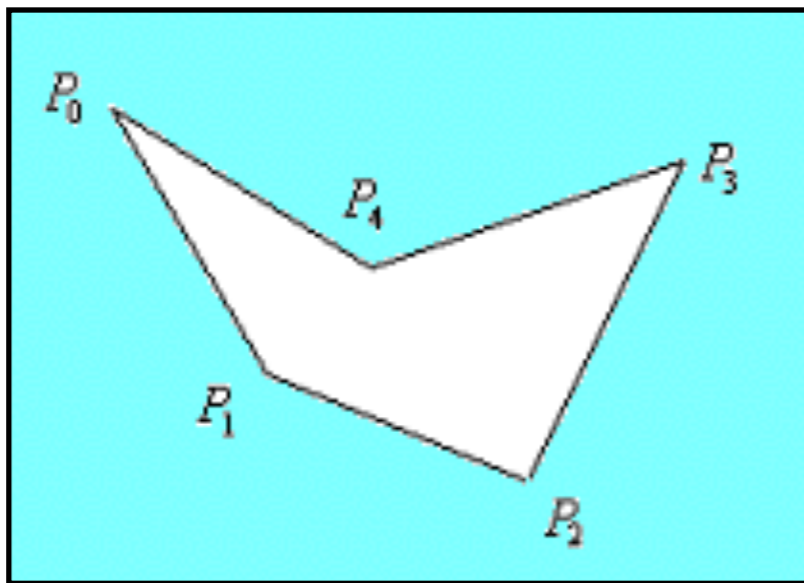
- Point
 - Size and Color
- Line
 - line width
 - line style
 - brush

Region attributes

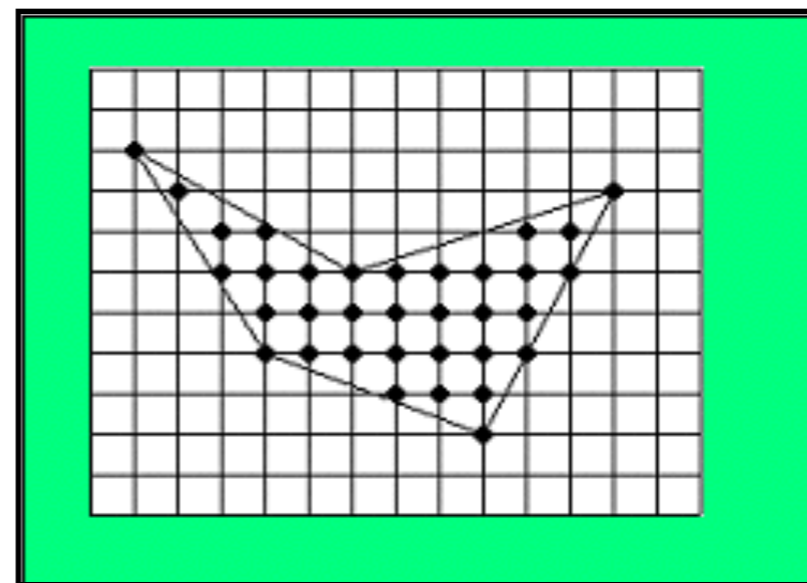
- defined by a planar polygon
 - filling style:
 - wireframe,
 - fill,
 - tiling pattern

Polygon filling

- Polygon representation



By vertex

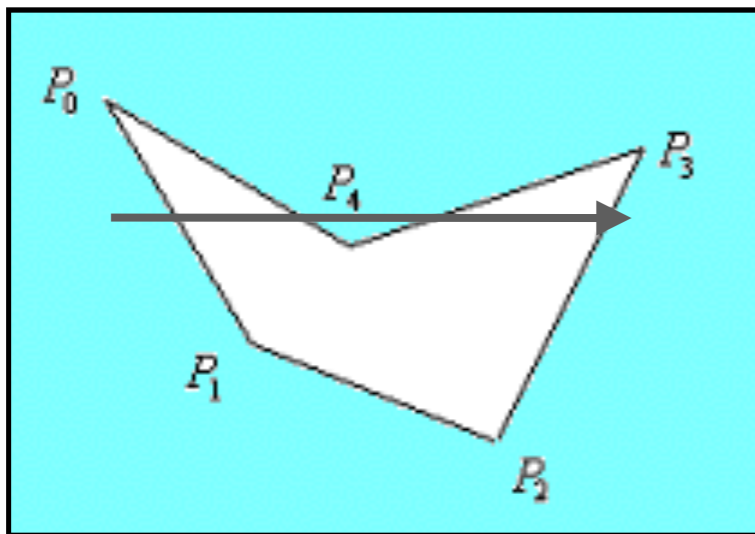


By lattice

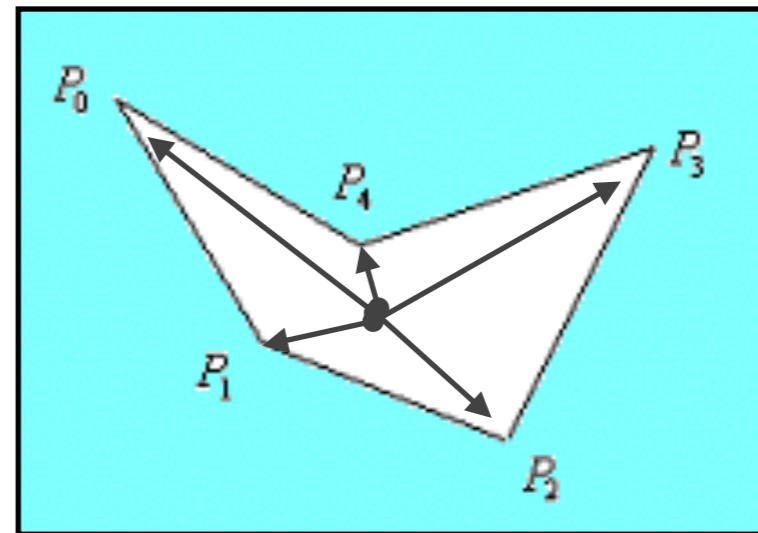
- Polygon filling:
- vertex representation vs lattice representation

Polygon filling

- fill a polygonal area \rightarrow test every pixel in the raster to see if it lies inside the polygon.



even-odd test

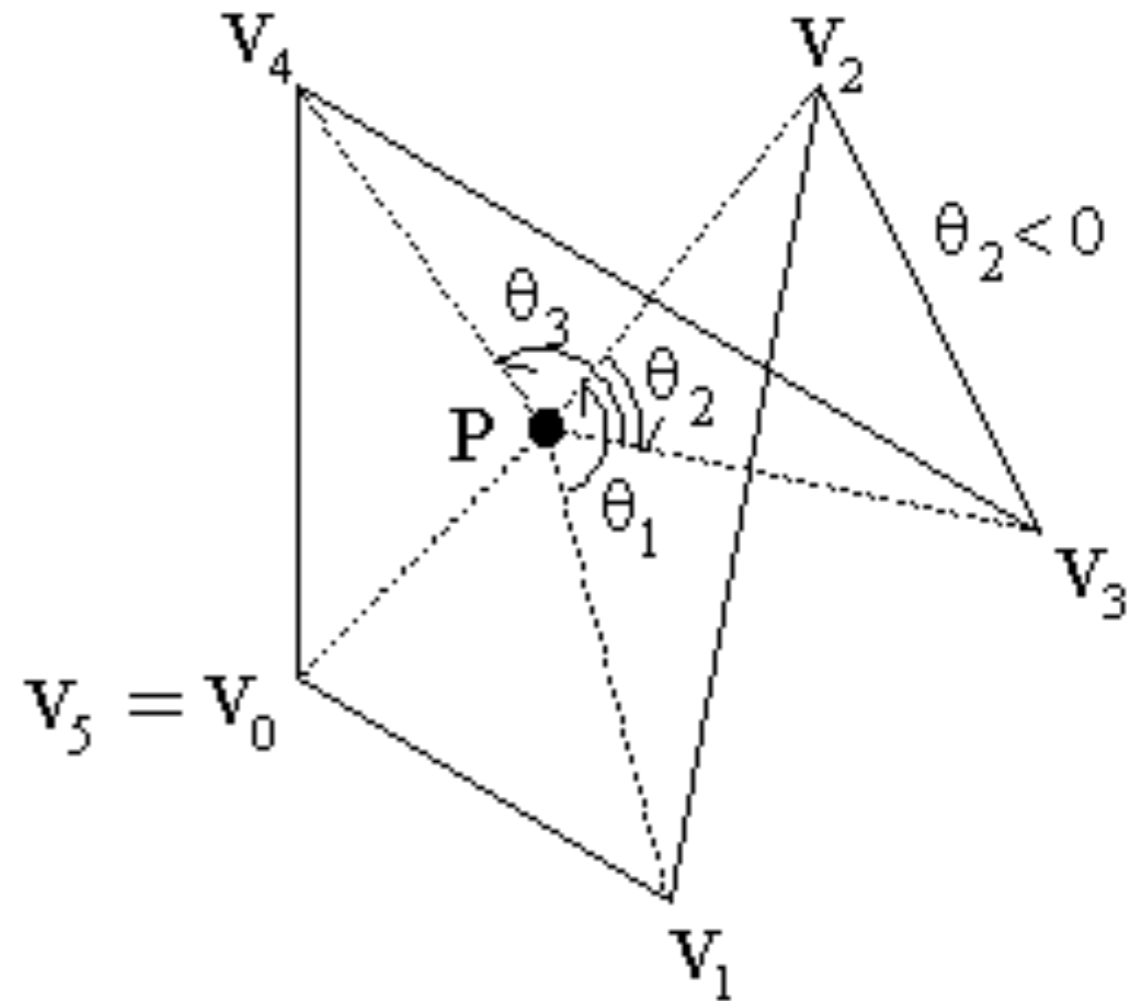


winding number test

Question5: How to Judge...?

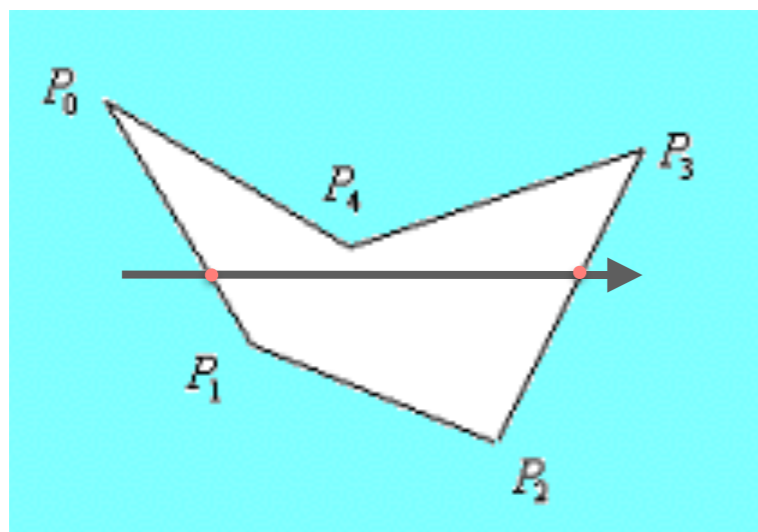
Inside check

$$\begin{aligned} \mathbf{wn} &= \frac{1}{2\pi} \sum_{i=0}^{n-1} \theta_i \\ &= \frac{1}{2\pi} \sum_{i=0}^{n-1} \arccos \left(\frac{\mathbf{PV}_i \cdot \mathbf{PV}_{i+1}}{|\mathbf{PV}_i| |\mathbf{PV}_{i+1}|} \right) \end{aligned}$$

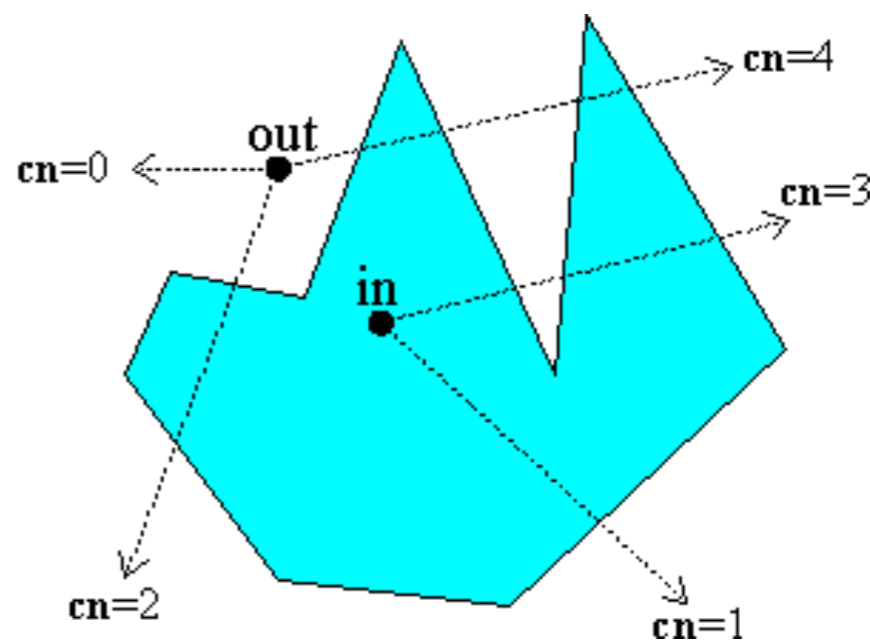
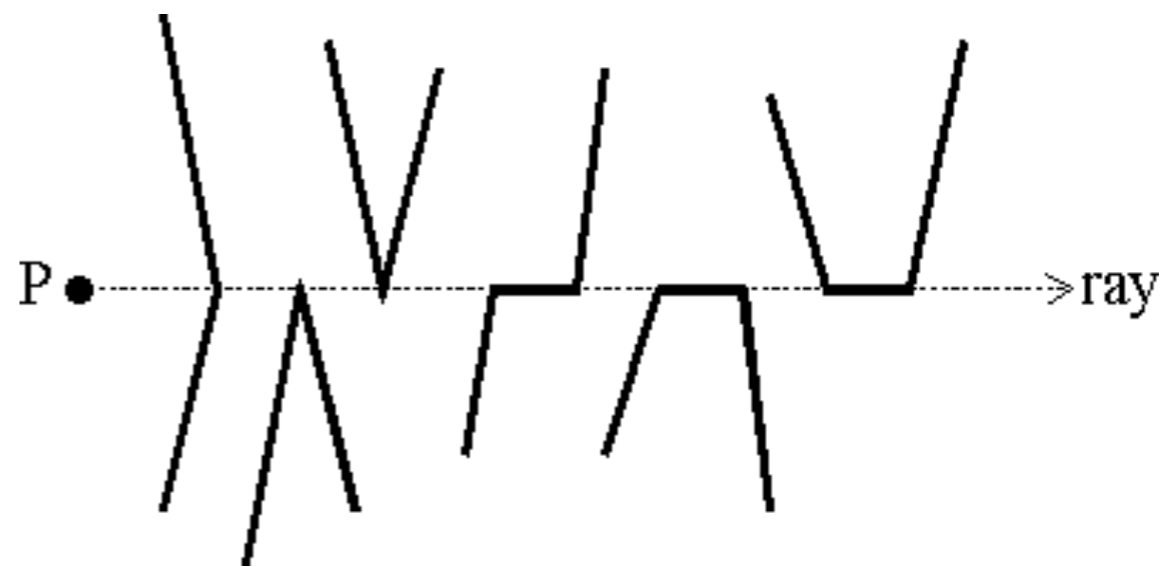


Question6: How to improve ...?

Inside check



even-odd test



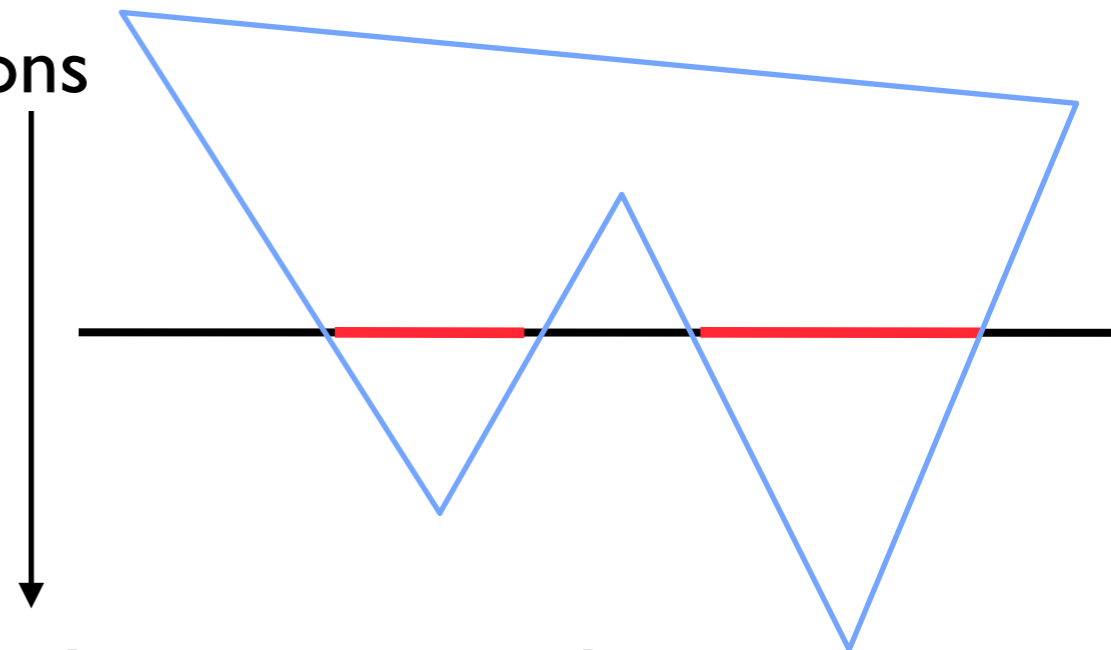
Scan Line Methods

- Makes use of the *coherence* properties
 - Spatial coherence : Except at the boundary edges, adjacent pixels are likely to have the same characteristics
 - Scan line coherence : Pixels in the adjacent scan lines are likely to have the same characteristics
- Uses intersections between area boundaries and scan lines to identify pixels that are inside the area

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



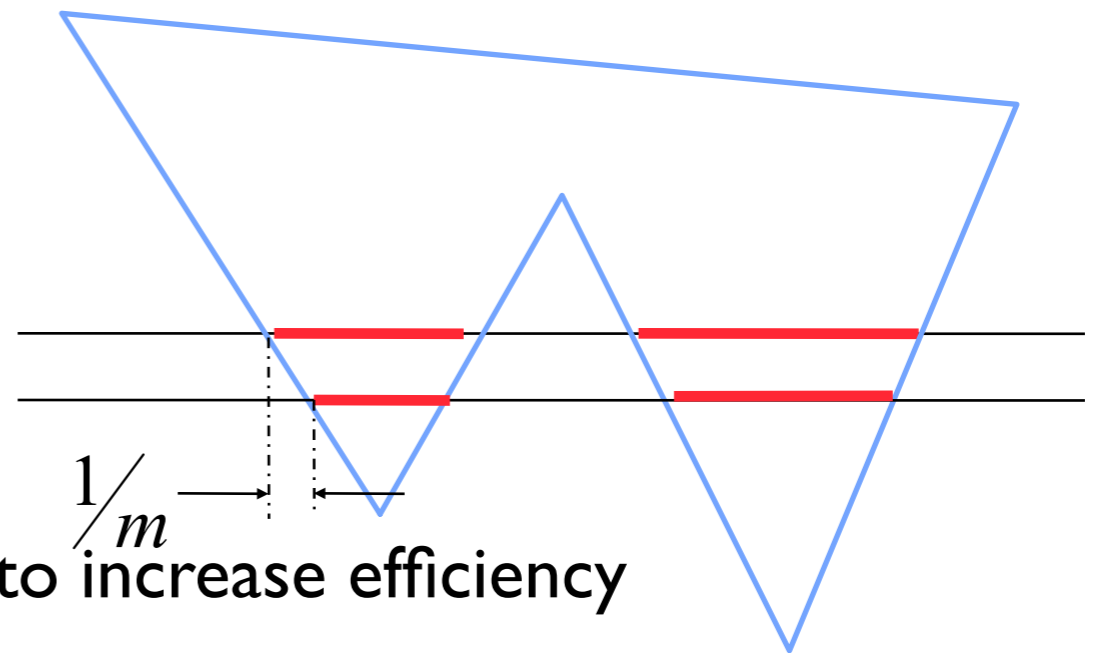
Discussion 5 : How to speed up, or how to avoid calculating intersection

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 scanline as

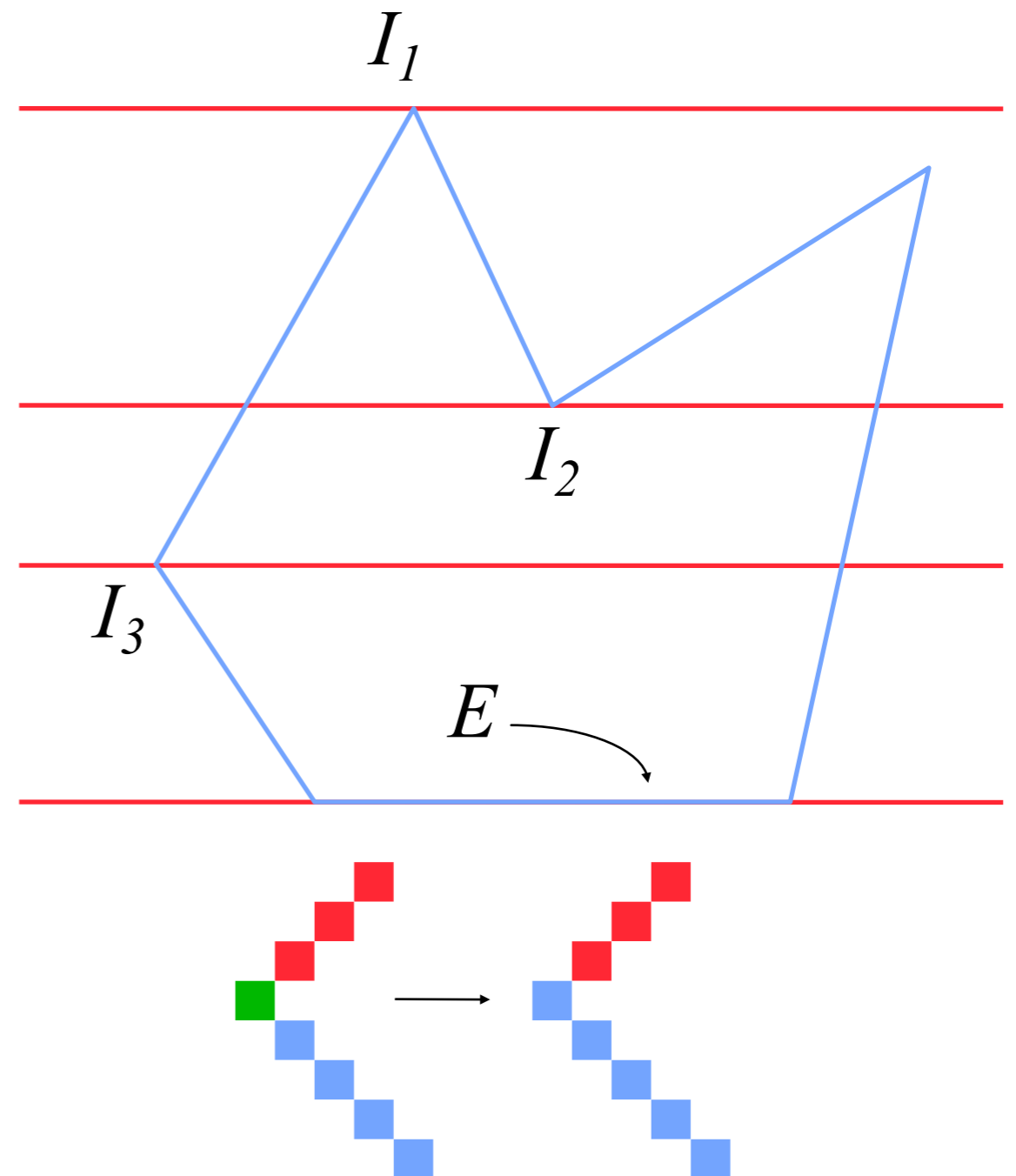
$$x_{i+1} = x_i + \frac{1}{m}$$



- List of active edges could be maintained to increase efficiency
- Efficiency could be further improved if polygons are convex, much better if they are only triangles

Special cases for Scan Line Method

- Overall topology should be considered for intersection at the vertices
- Intersections like I_1 and I_2 should be considered as two intersections
- Intersections like I_3 should be considered as one intersection
- Horizontal edges like E need not be considered

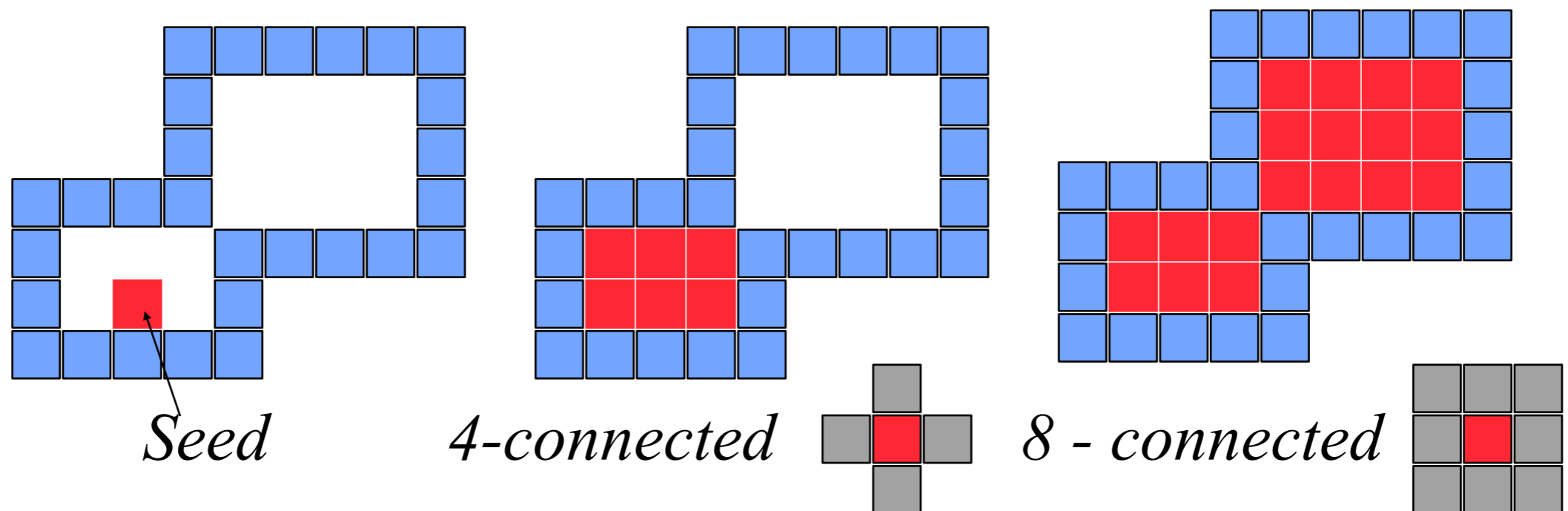


Advantages of Scan Line method

- The algorithm is efficient
- Each pixel is visited only once
- Shading algorithms could be easily integrated with this method to obtain shaded area

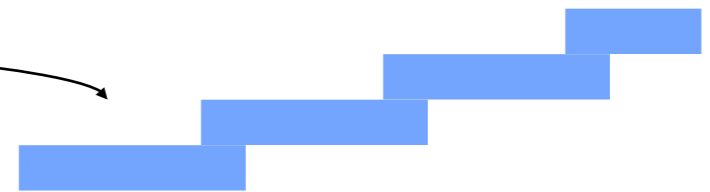
Seed Fill Algorithms

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

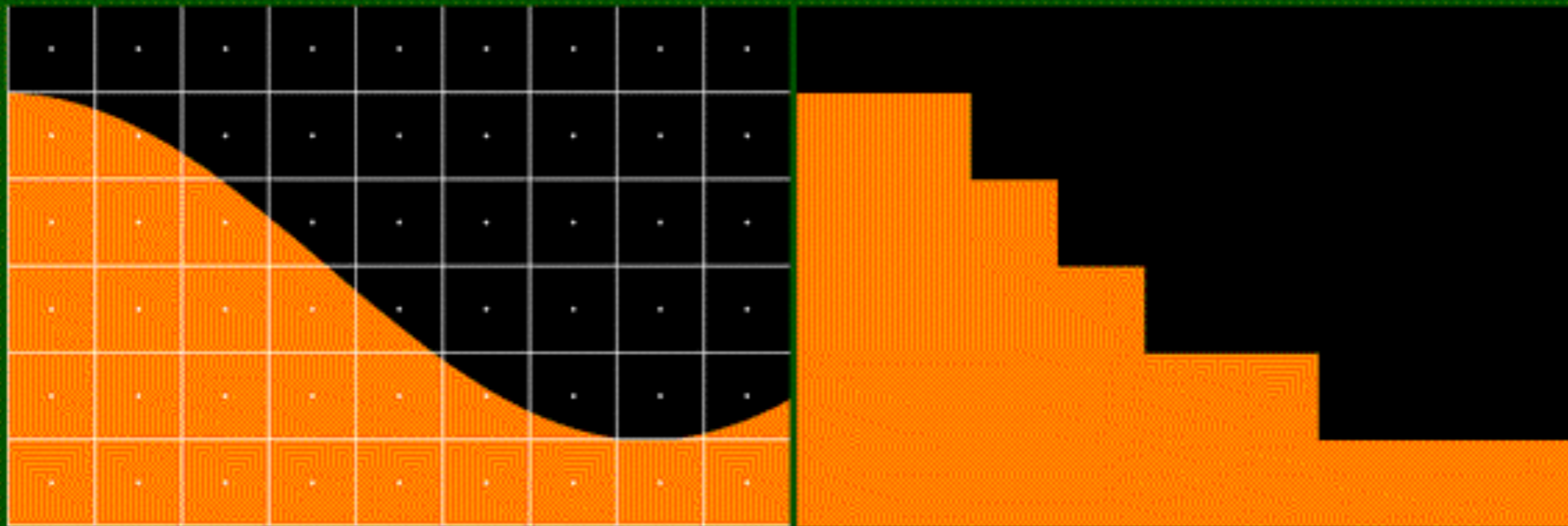


Aliasing

- 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)

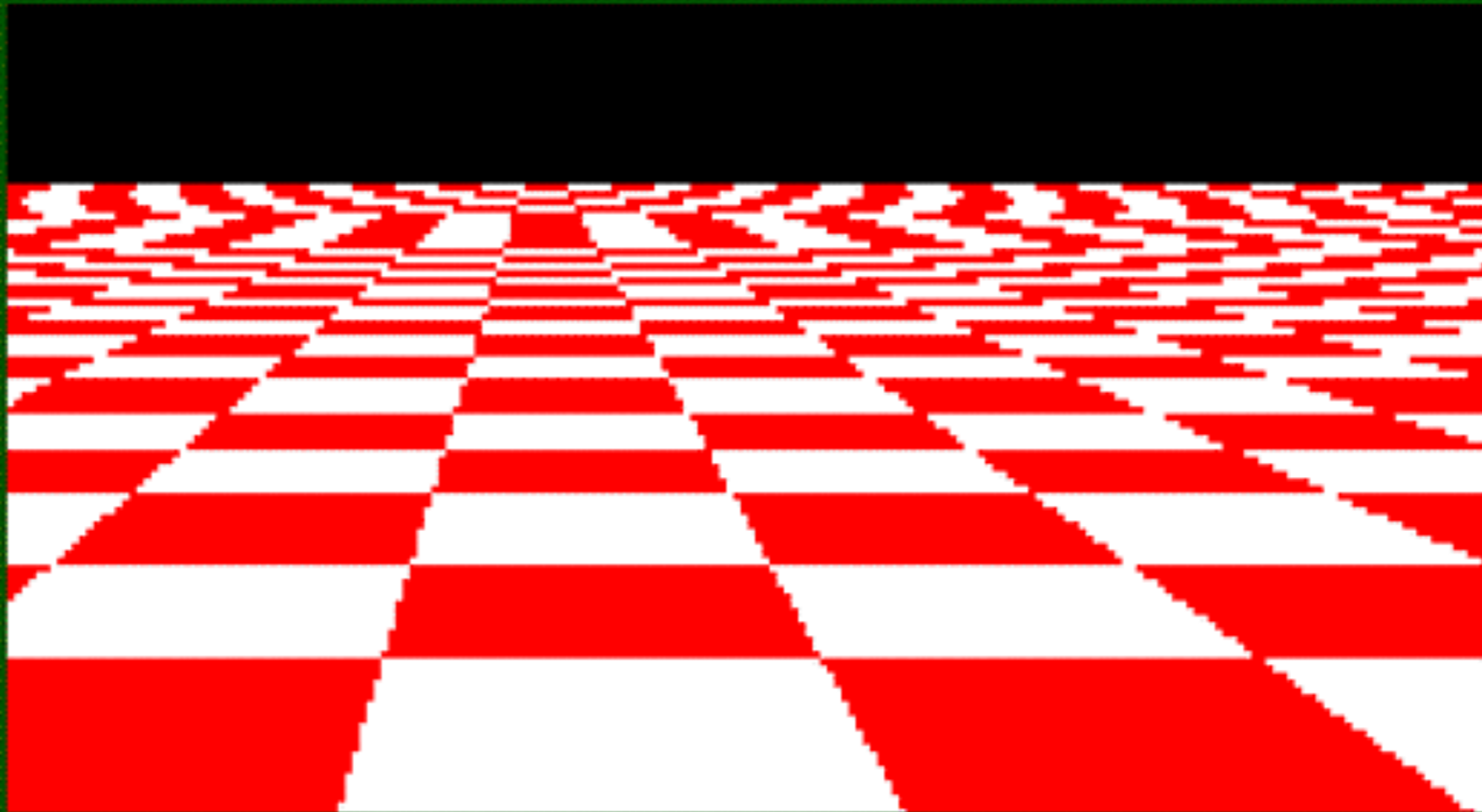


Original

Rendered

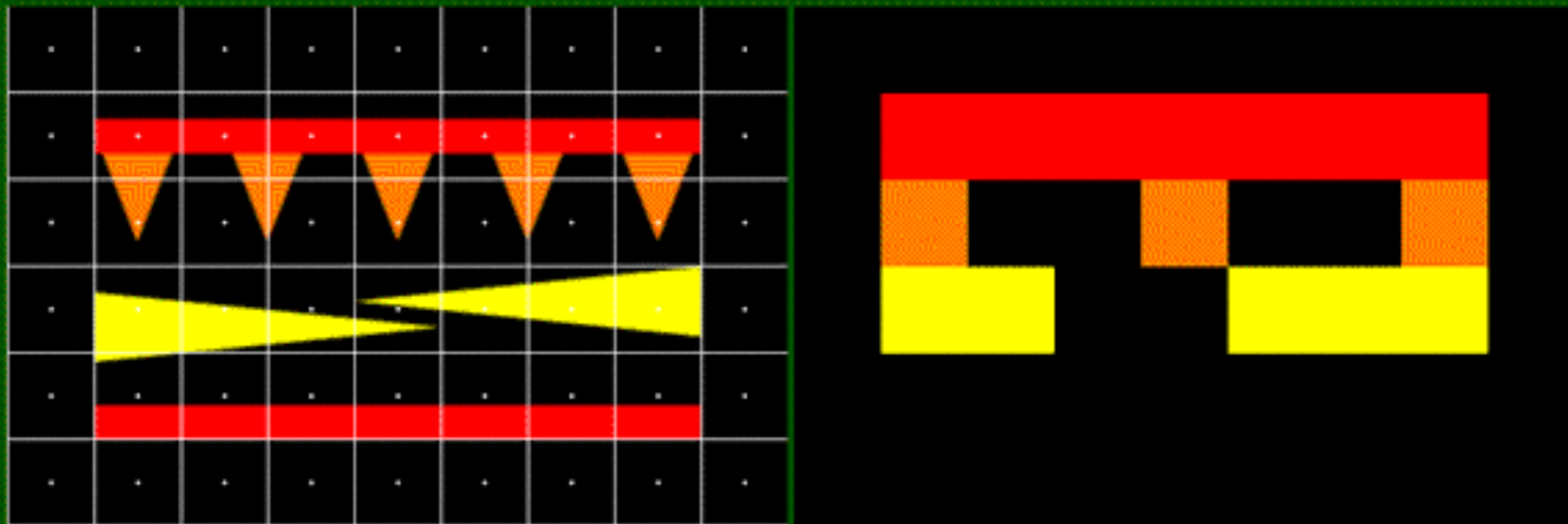
Jagged profiles

Aliasing(examples)



Disintegrating textures

Aliasing(examples)



Original

Rendered

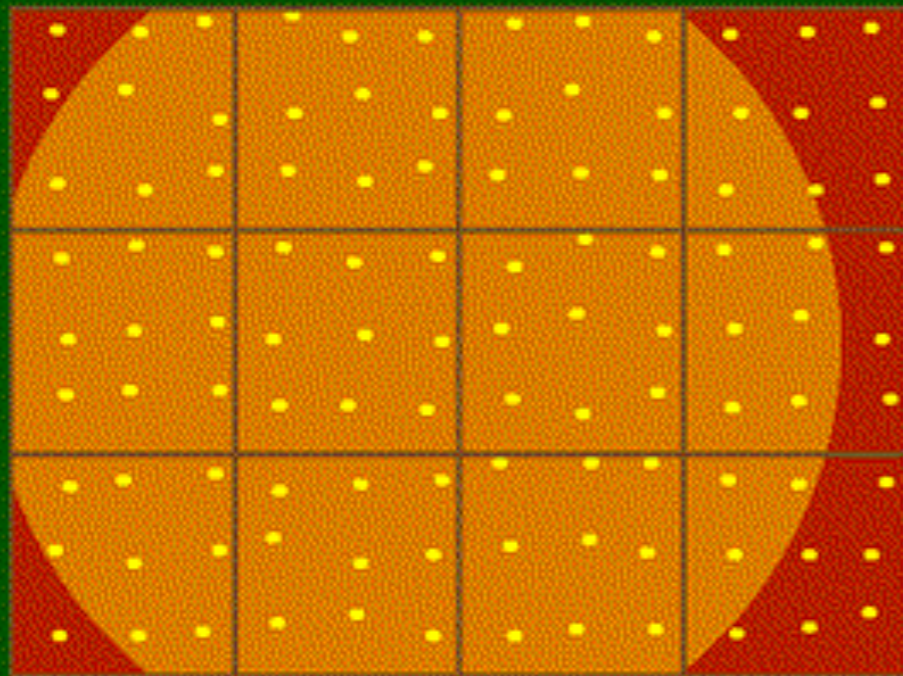
Loss of detail

Antialiasing

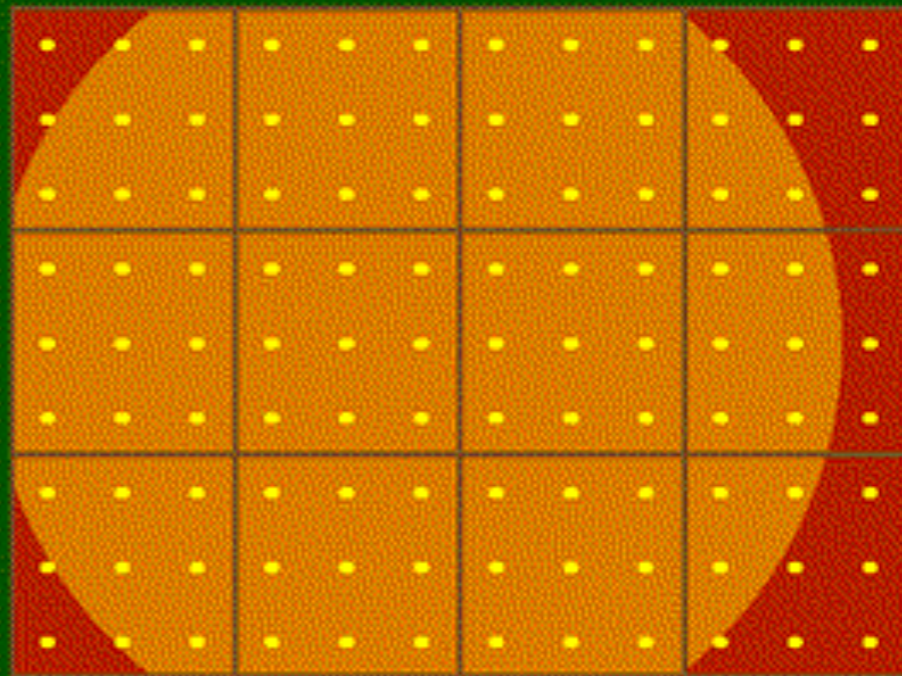
- Application of techniques to reduce/eliminate aliasing artifacts
- Some of the methods are
 - increasing sampling rate by increasing the resolution. Display memory requirements increases four times if the resolution is doubled
 - averaging methods (post processing). Intensity of a pixel is set as the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

Antialiasing(postfiltering)

How should one supersample?



Jittered

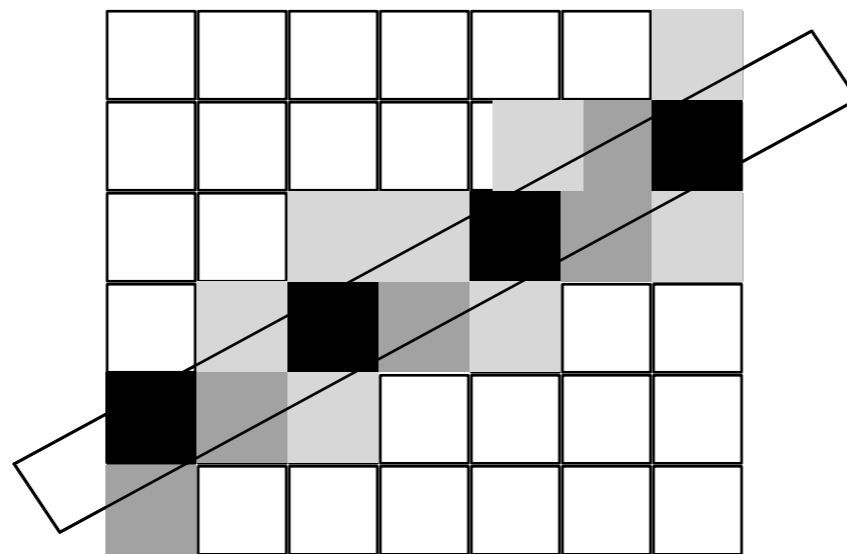


Regular

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

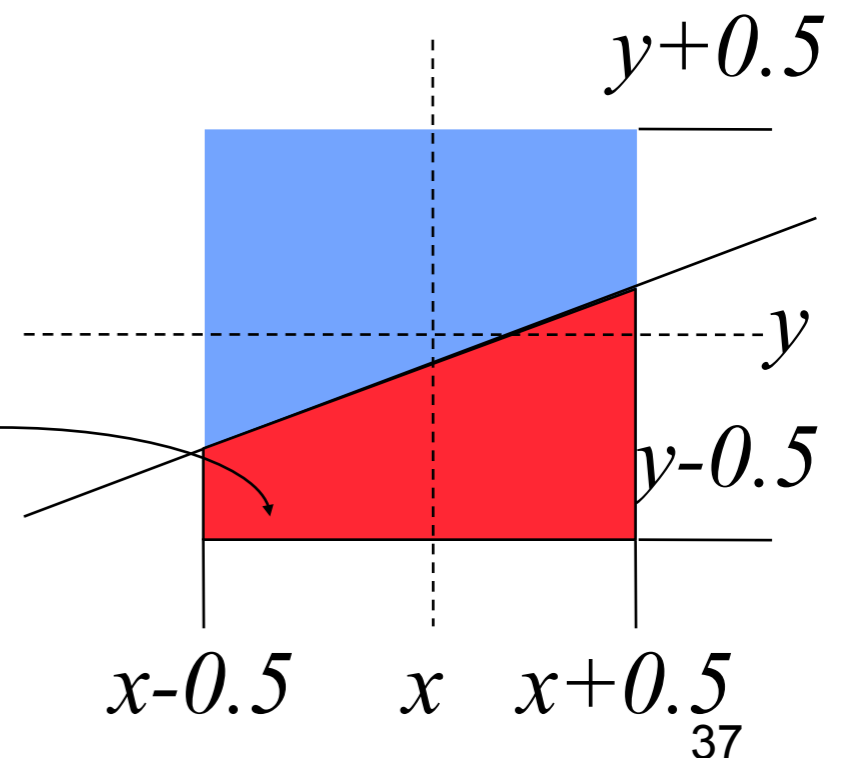


Area Sampling

- 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

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

filled area



Clipping

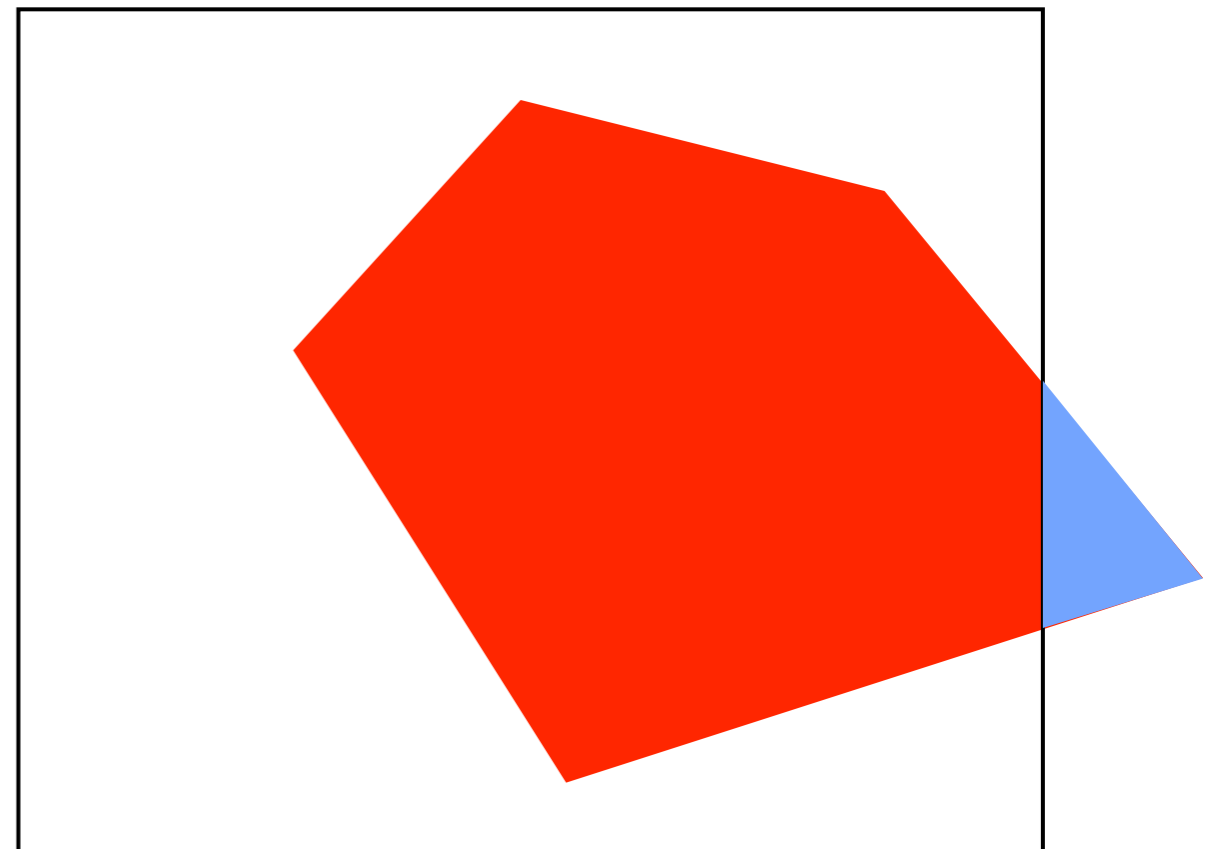
- Clipping of primitives is done usually before scan converting the primitives
- Reasons being
 - scan conversion needs to deal only with the clipped version of the primitive, which might be much smaller than its unclipped version
 - Primitives are usually defined in the real world, and their mapping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

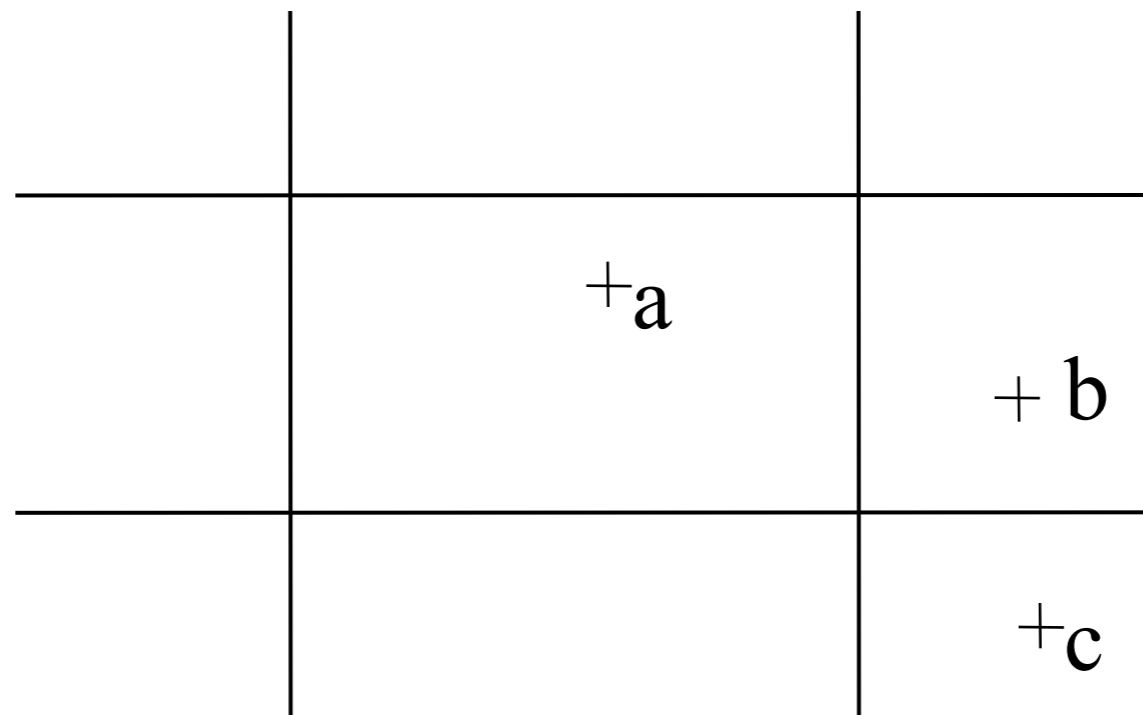


How would we clip?

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

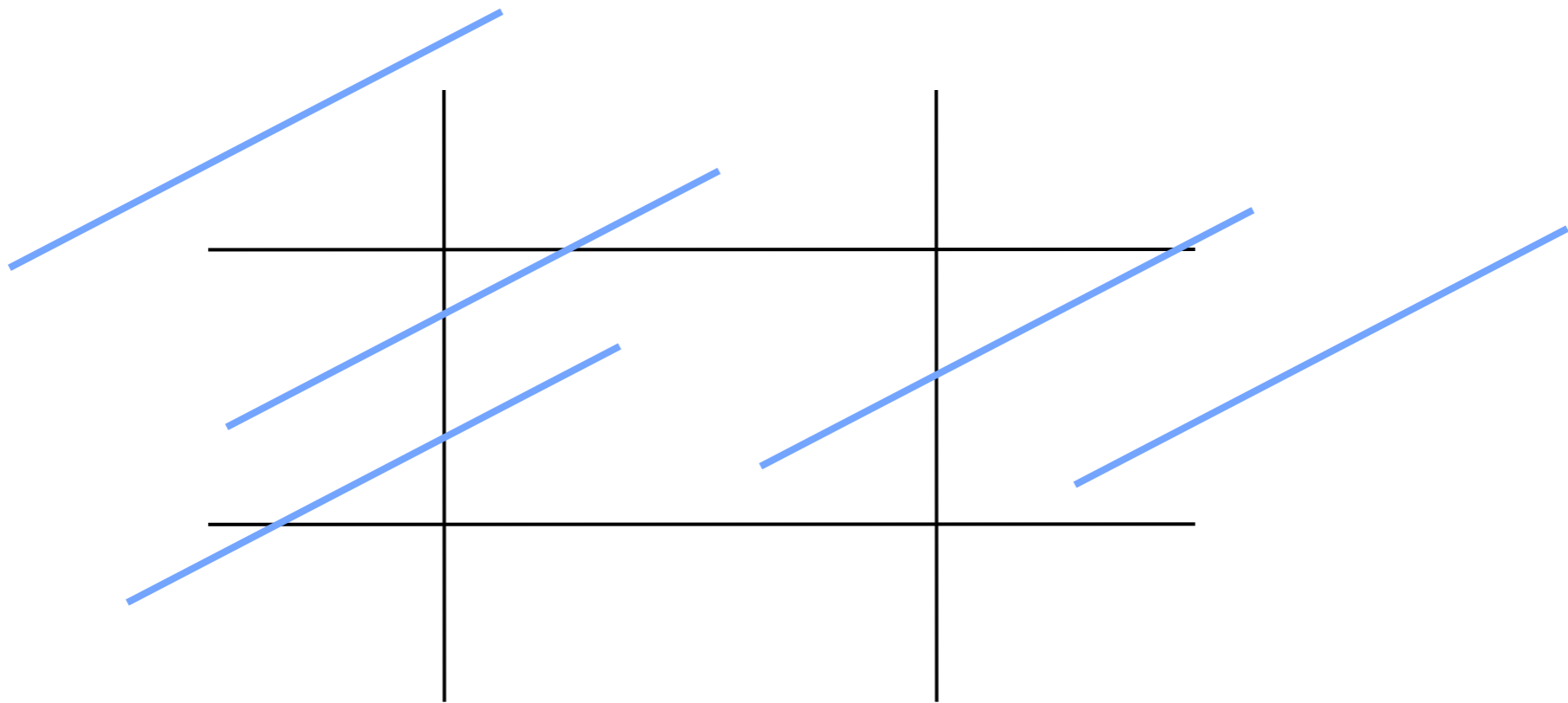
We'll start in 2D

- 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 $y \leq \max_y$



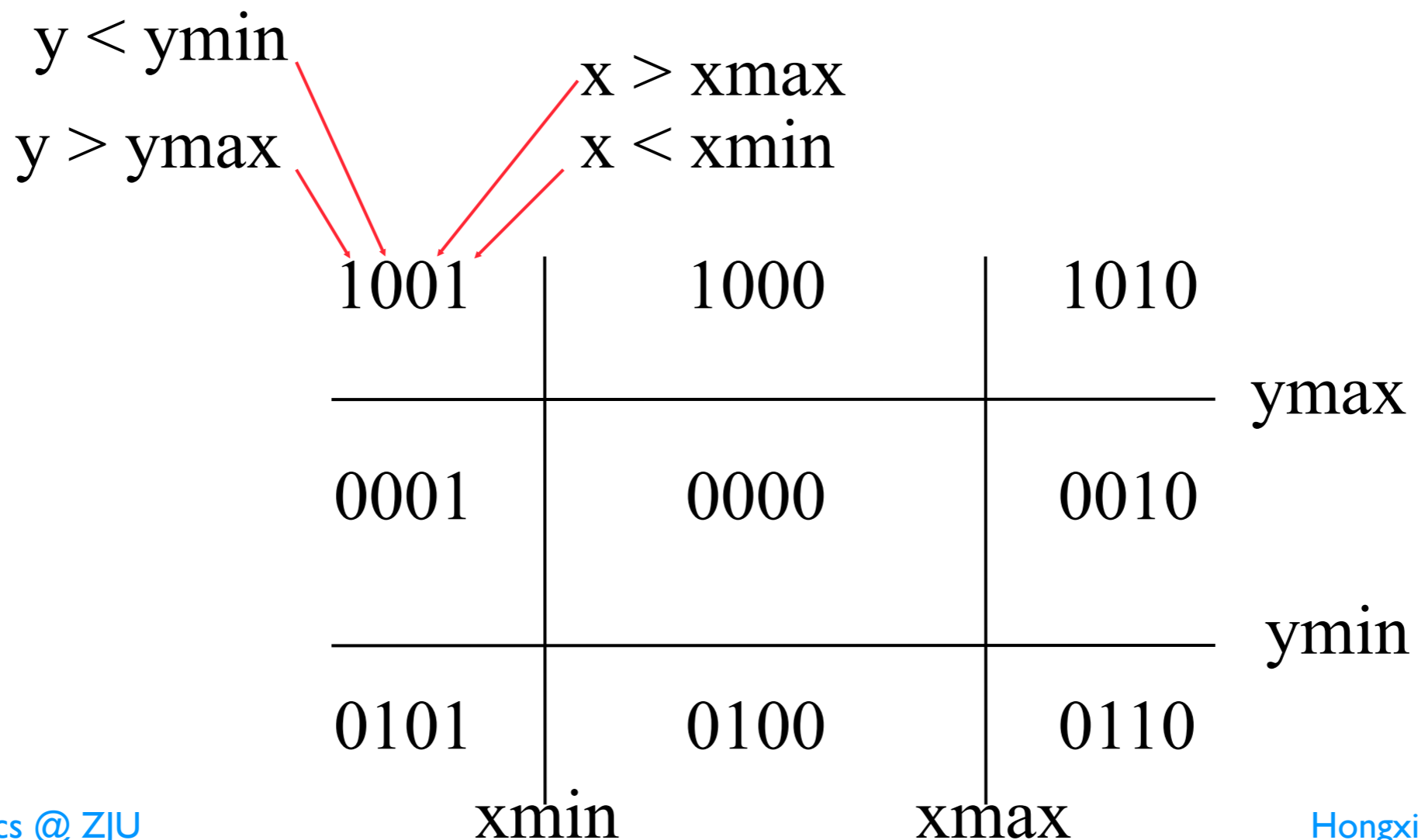
Line Segments

- What can happen when a line segment is clipped?



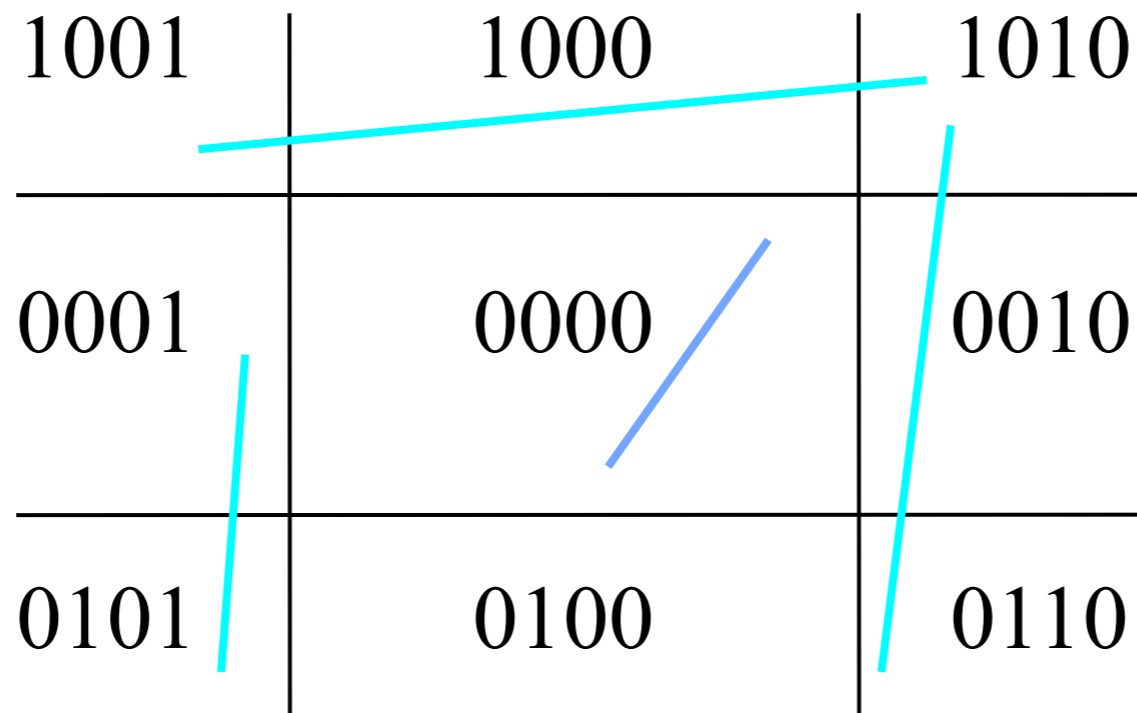
Cohen-Sutherland Line Clipping

- 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.



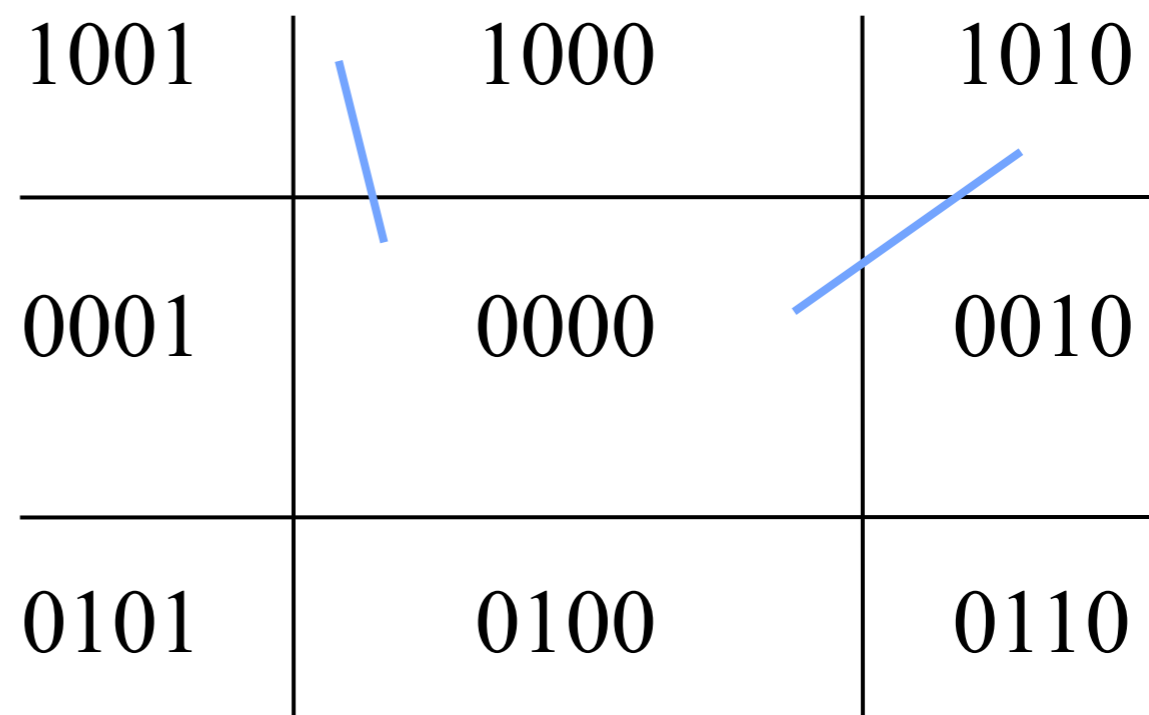
More cases

- 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.

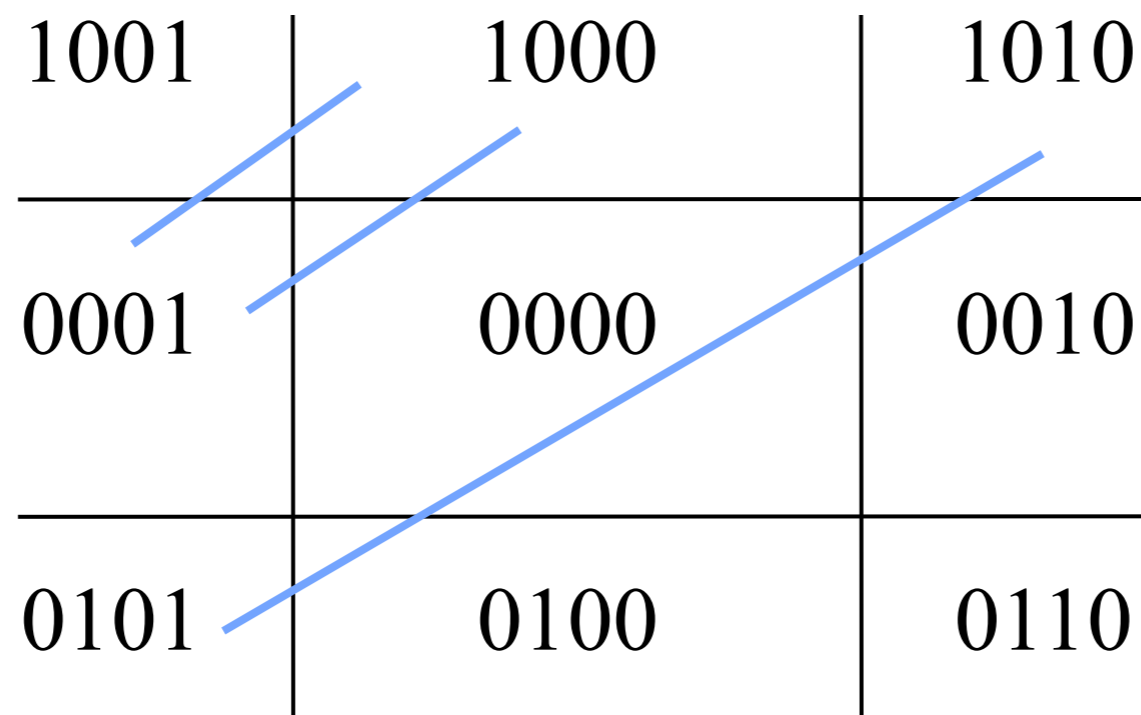
– Clip edges with bits set...

– May require two clip computations



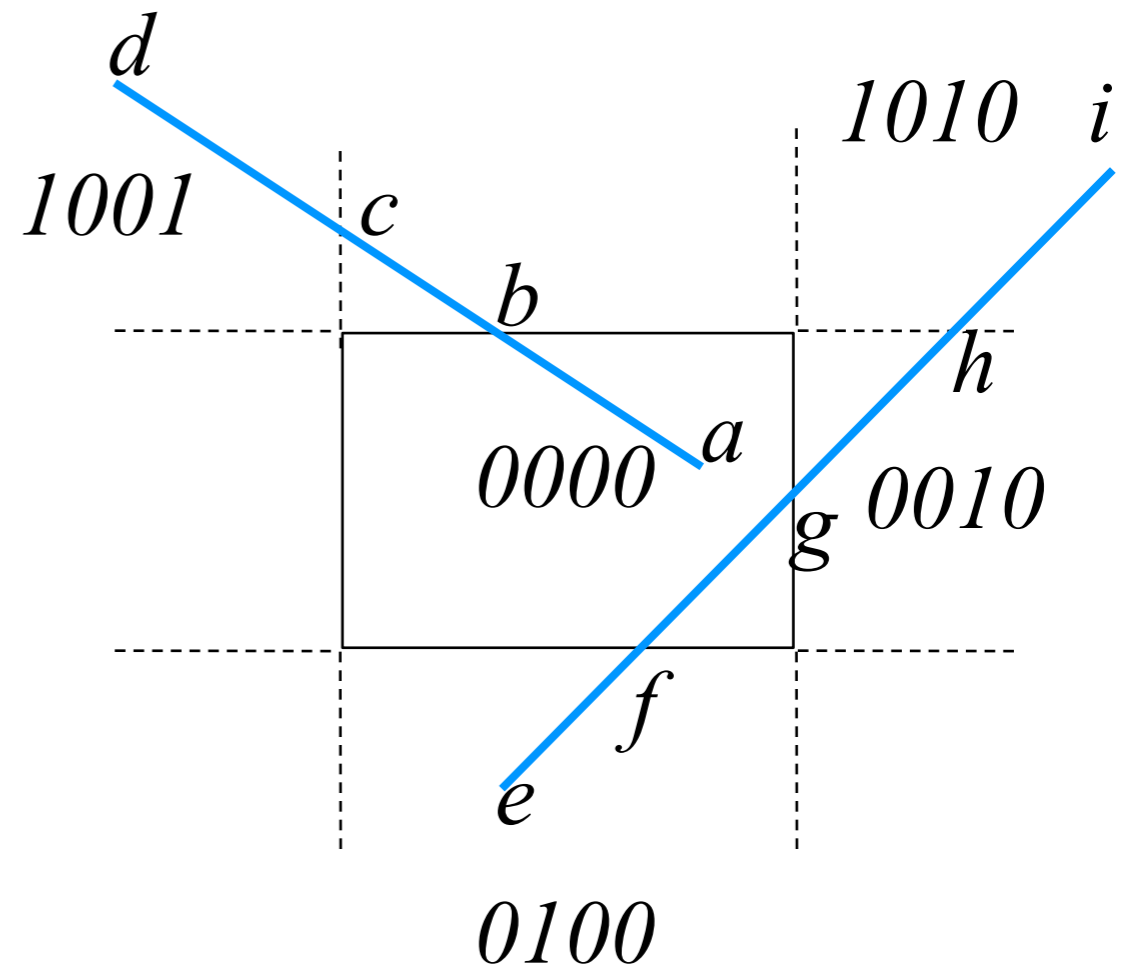
Last case...

- 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

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

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

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

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

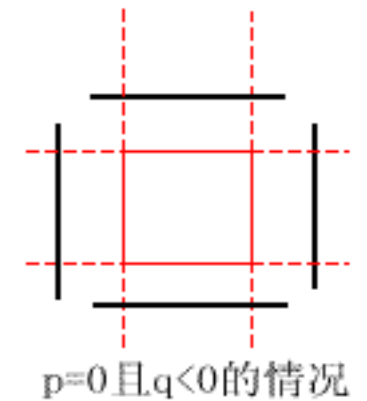
Liang–Barsky algorithm

To compute the final line segment:

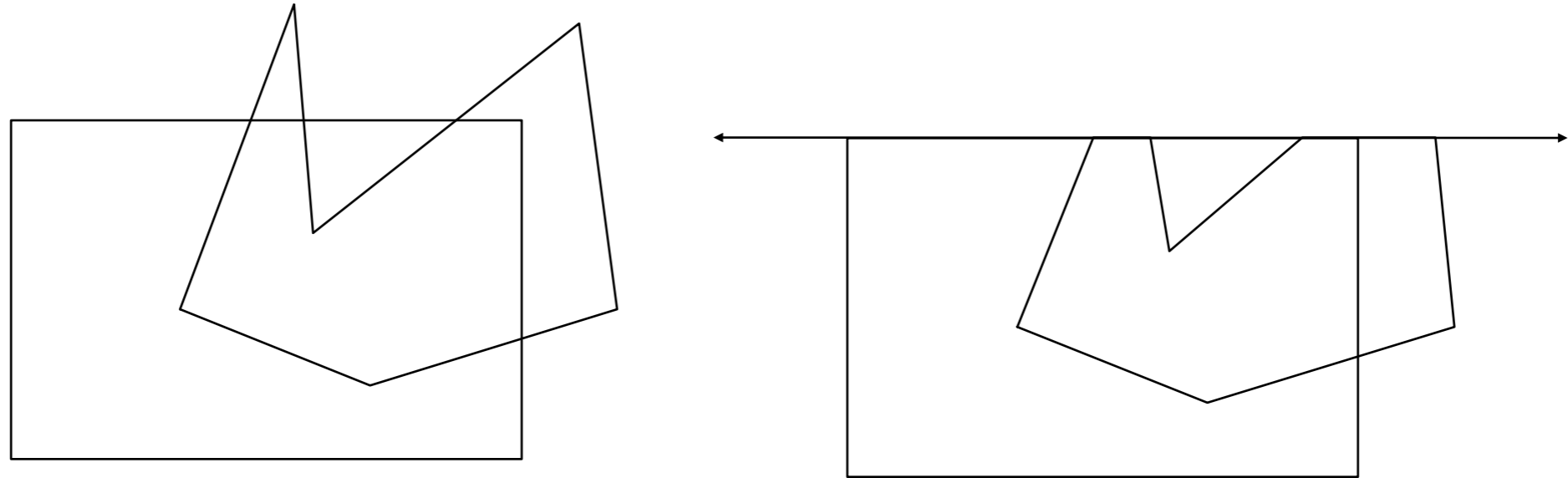
1. A line parallel to a clipping window edge has $p_k = 0$ for that boundary.
2. If for that k , $q_k < 0$, the line is completely outside and can be eliminated.
3. When $p_k < 0$ the line proceeds outside to inside the clip window and when $p_k > 0$, the line proceeds inside to outside.
4. For nonzero 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。



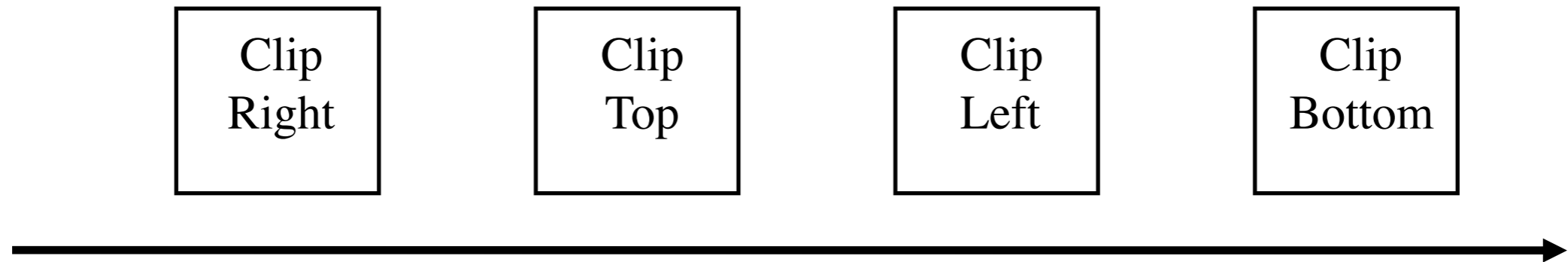
Sutherland-Hodgeman Polygon-Clipping Algorithm



- 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.

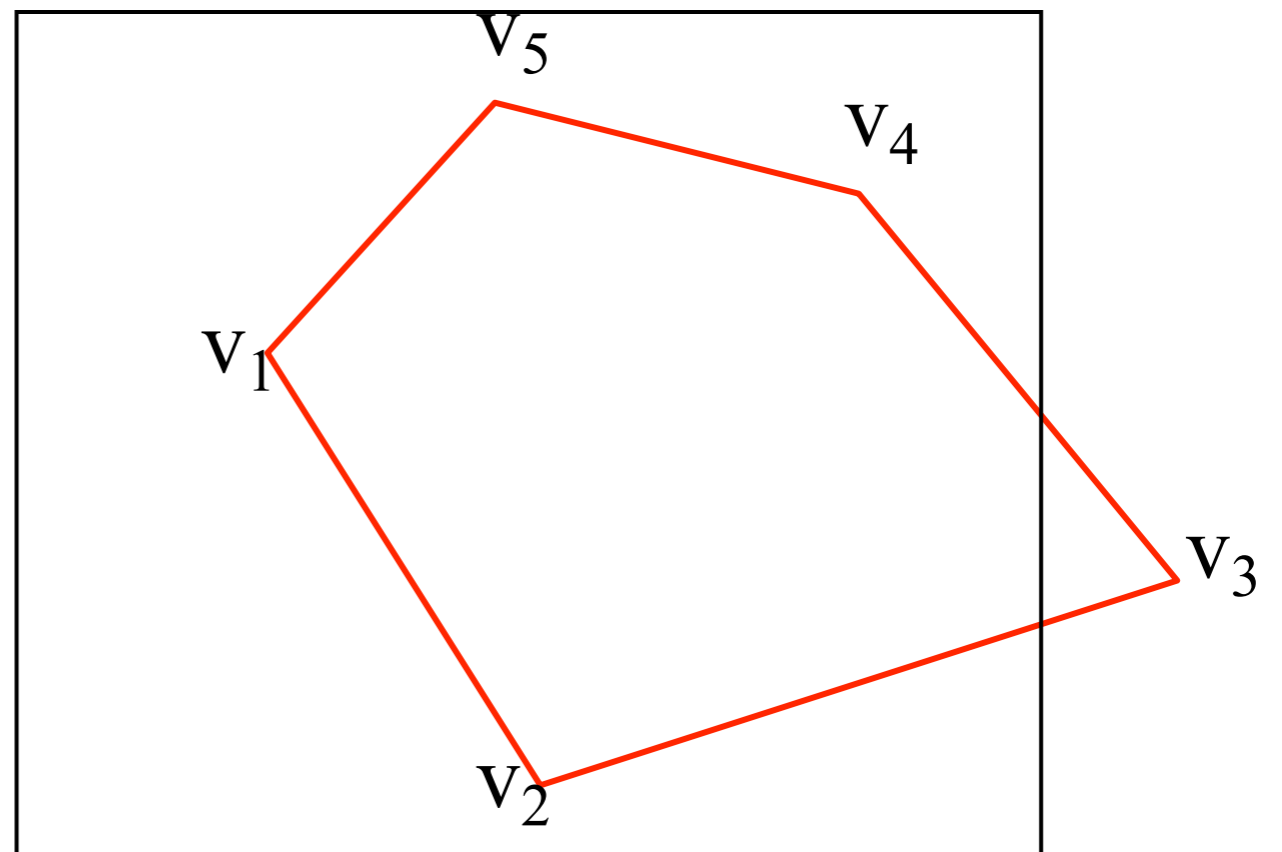
Pipelined Polygon Clipping

- 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.



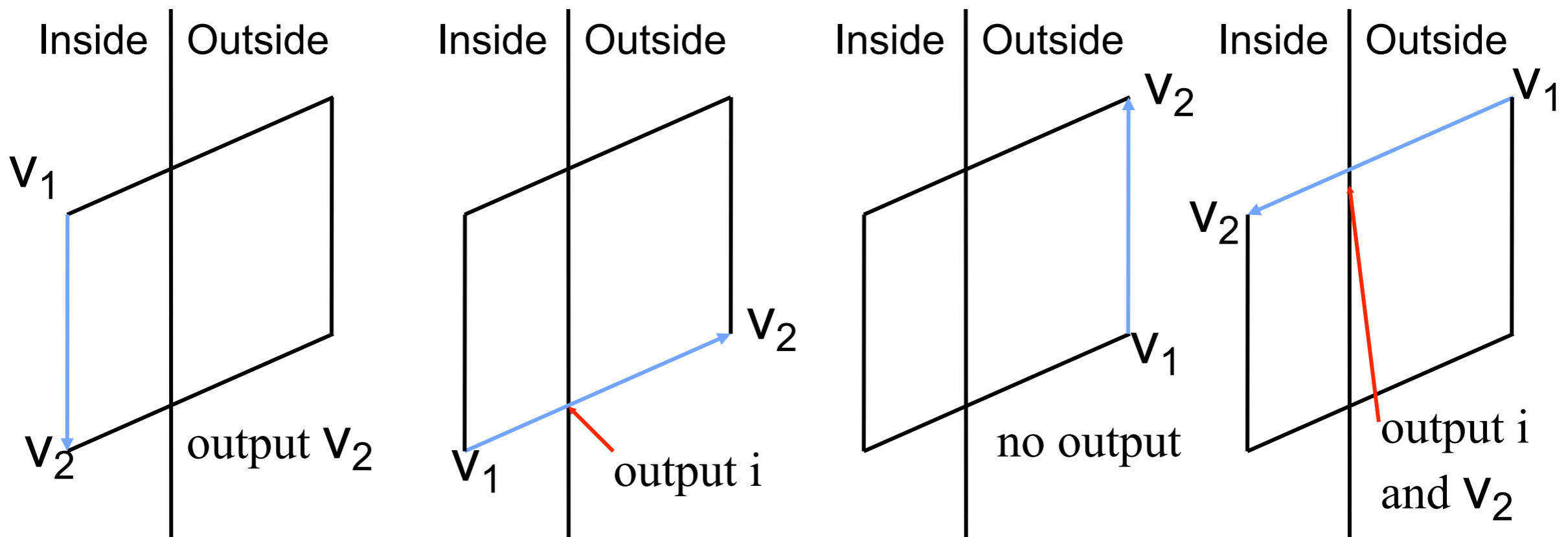
Sutherland-Hodgeman Polygon Clipping Algorithm

- 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...

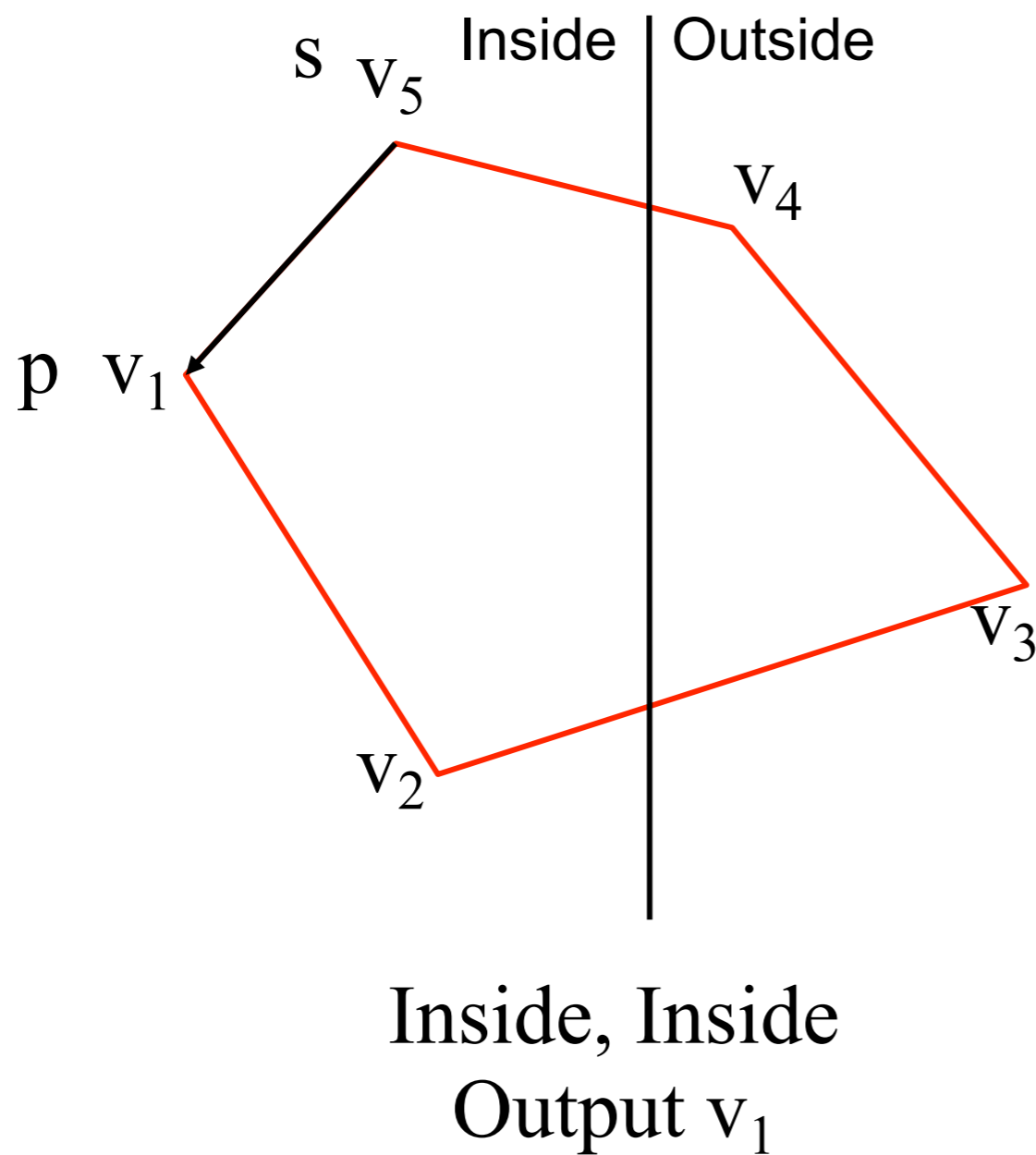


Sutherland-Hodgeman algorithm

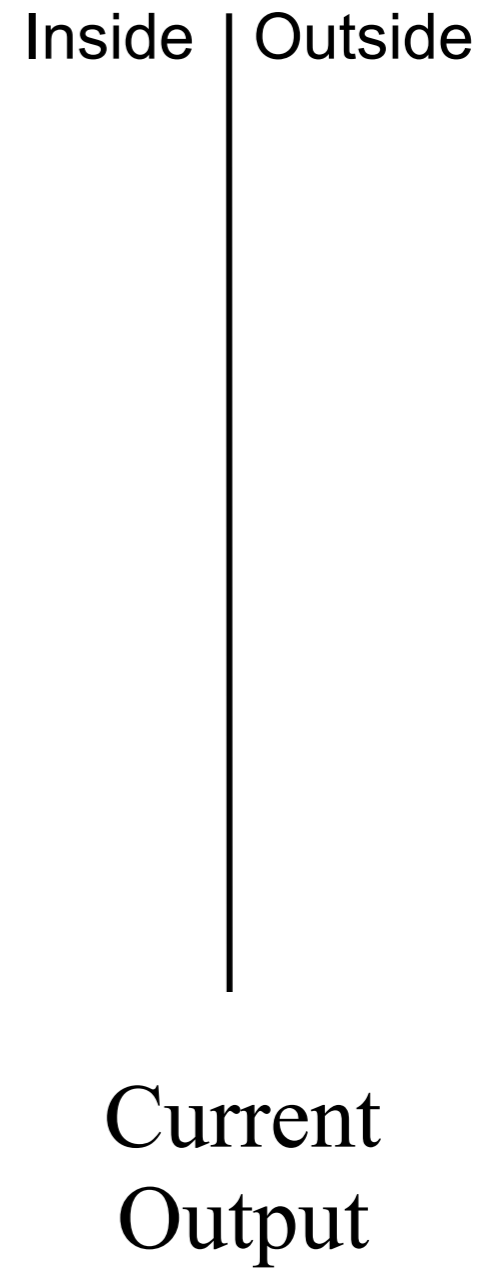
- Present the vertices in pairs
 - $(v_n, v_1), (v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)$
 - For each pair, what are the possibilities?
 - Consider v_1, v_2



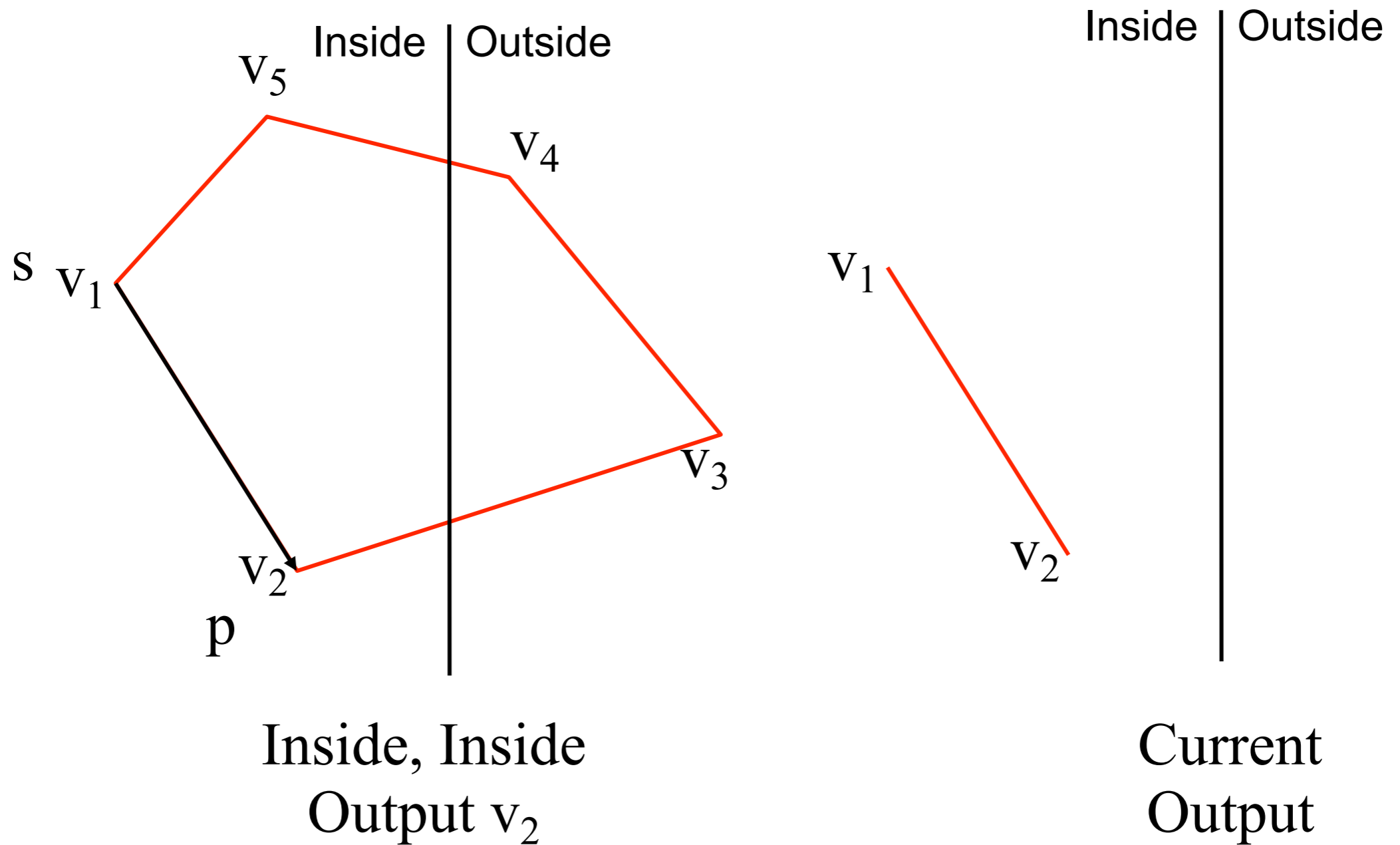
Example: v_5, v_1



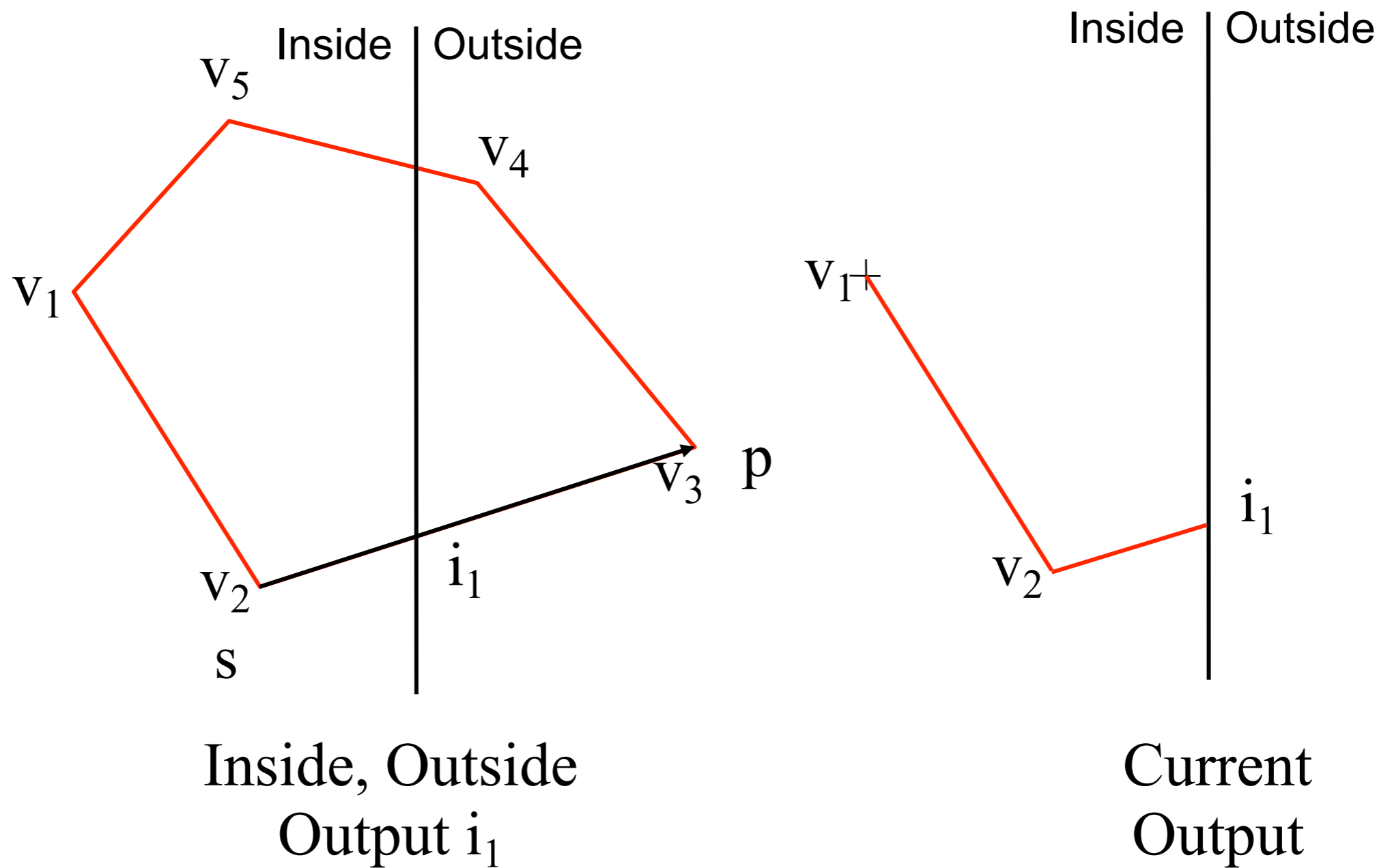
v_1^+



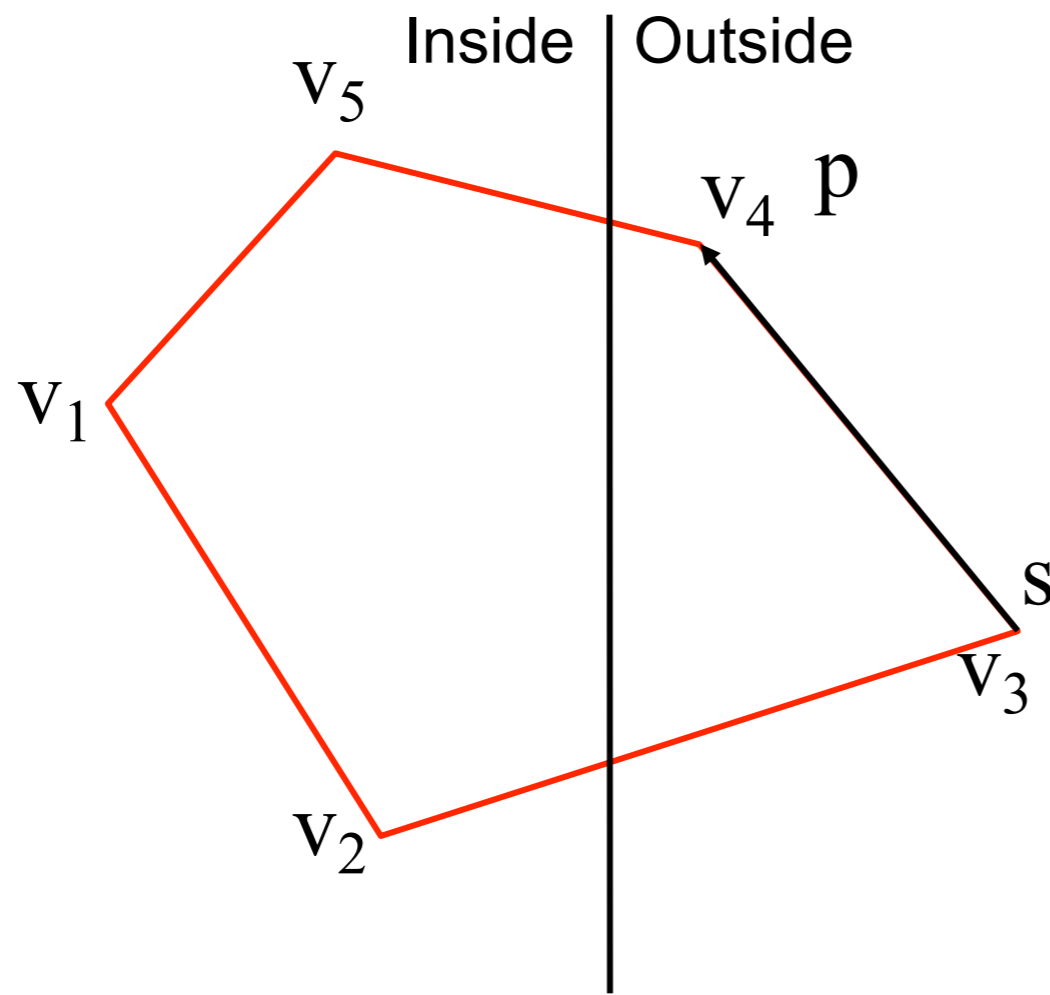
V_1, V_2



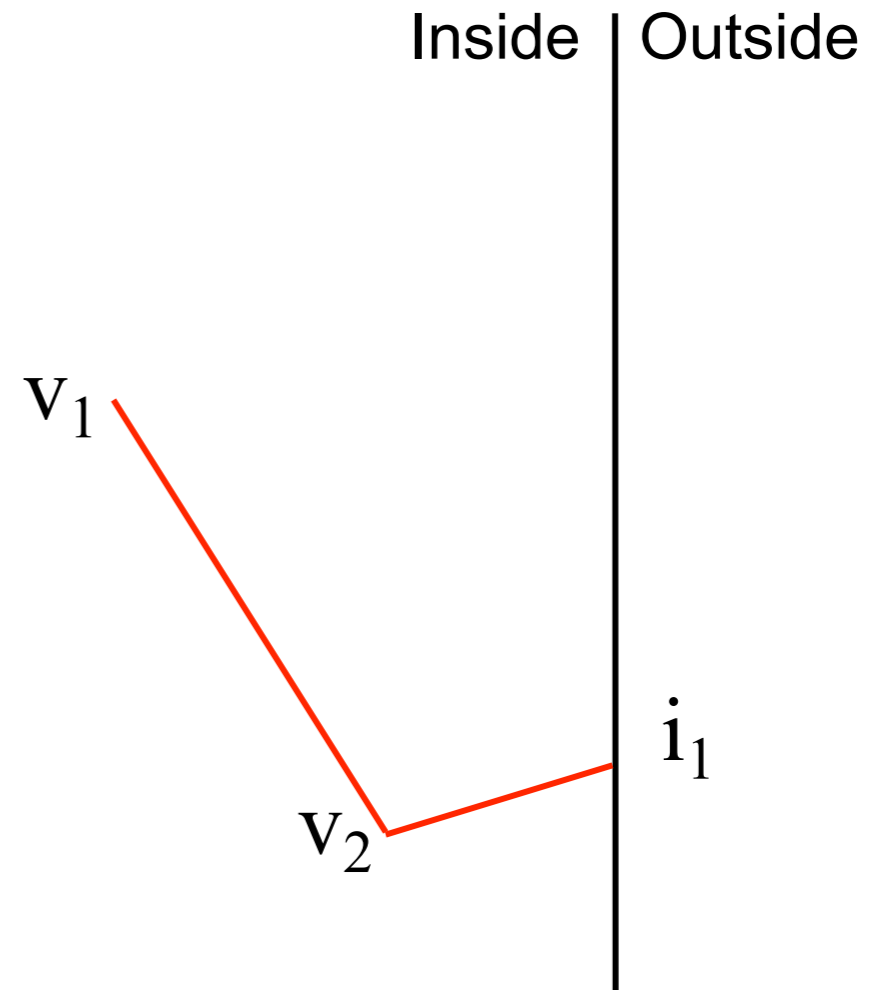
V_2, V_3



V_3, V_4

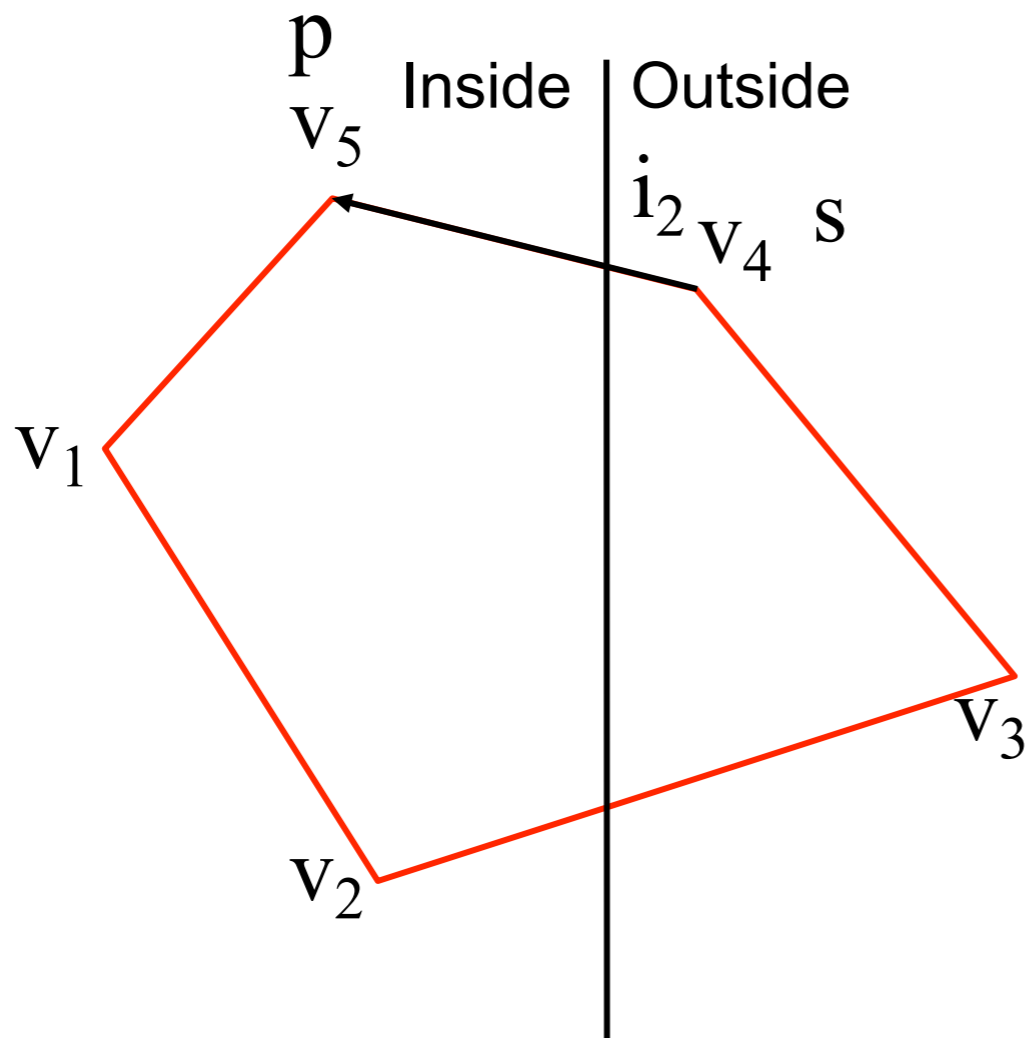


Outside, Outside
No output

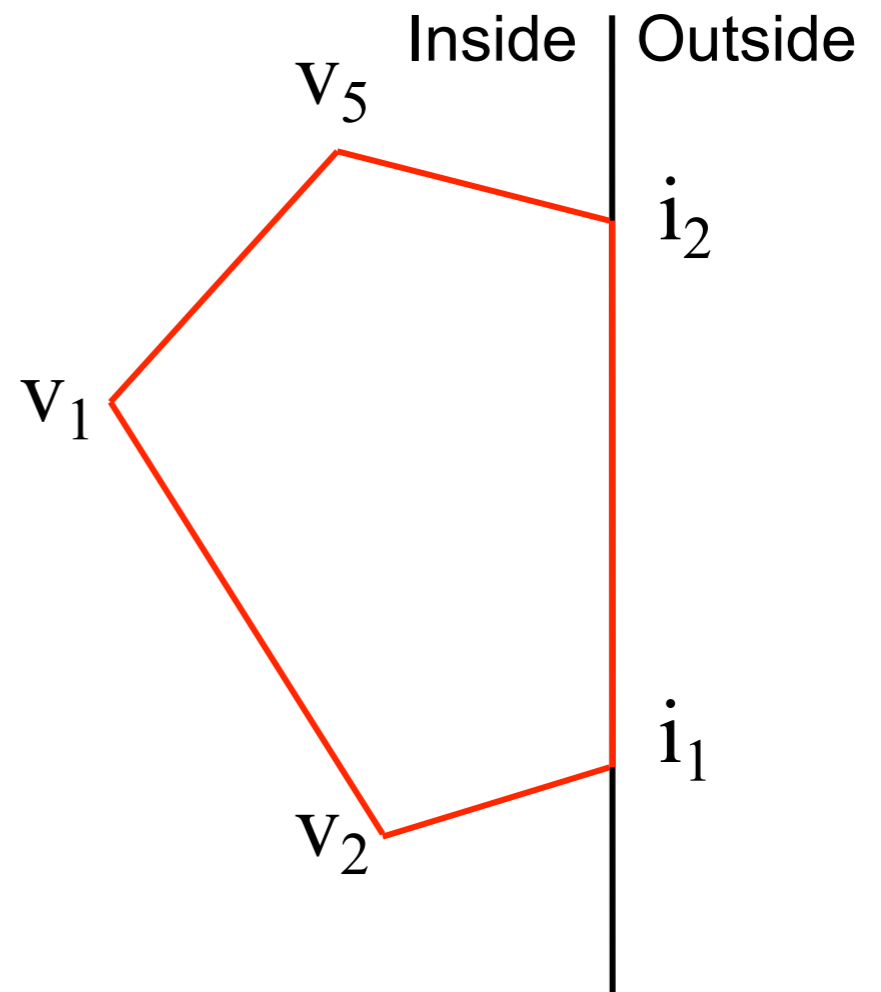


Current
Output

V_4, V_5 – last edge...



Outside, Inside
Output i_2, v_5



Current
Output