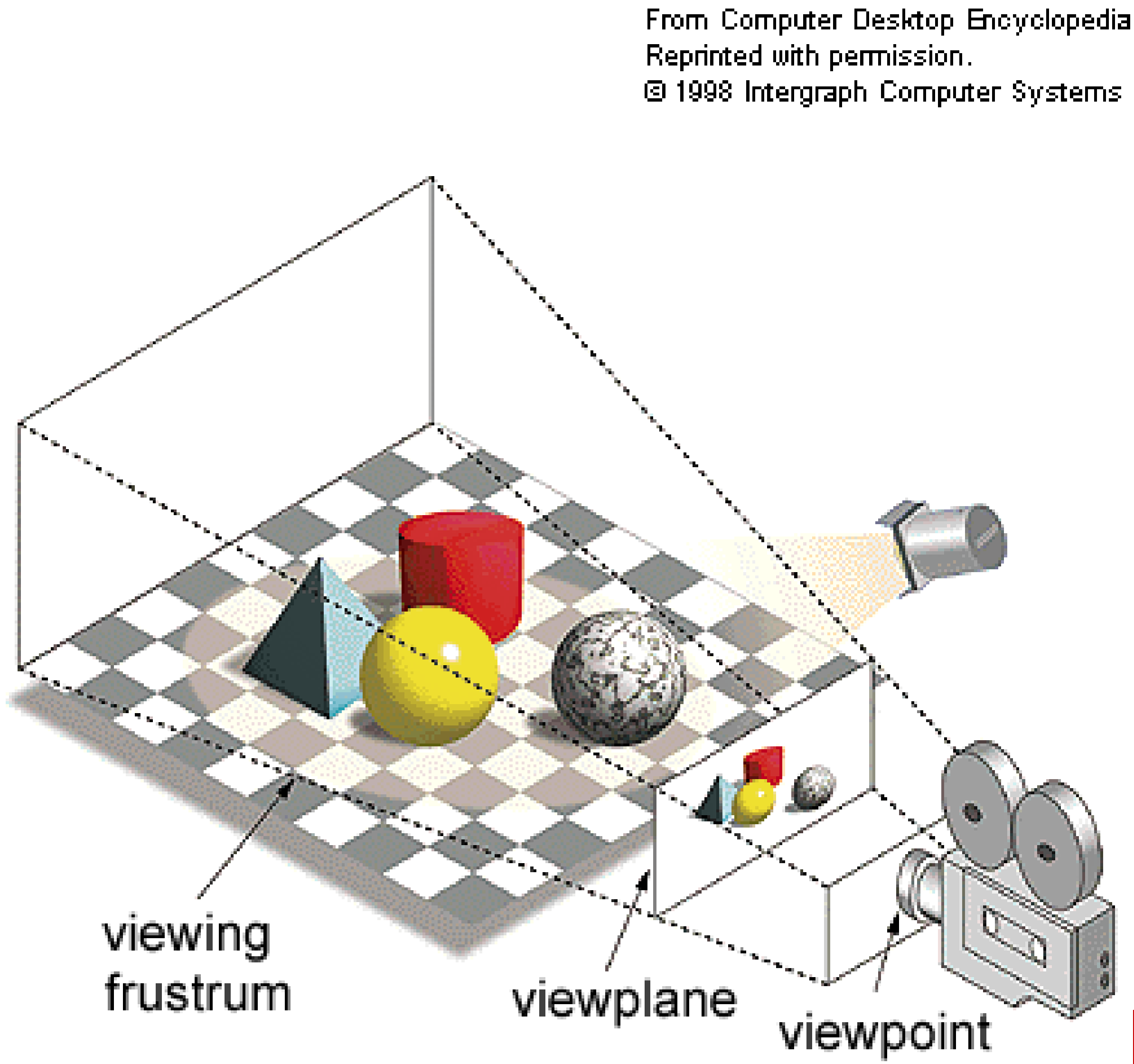
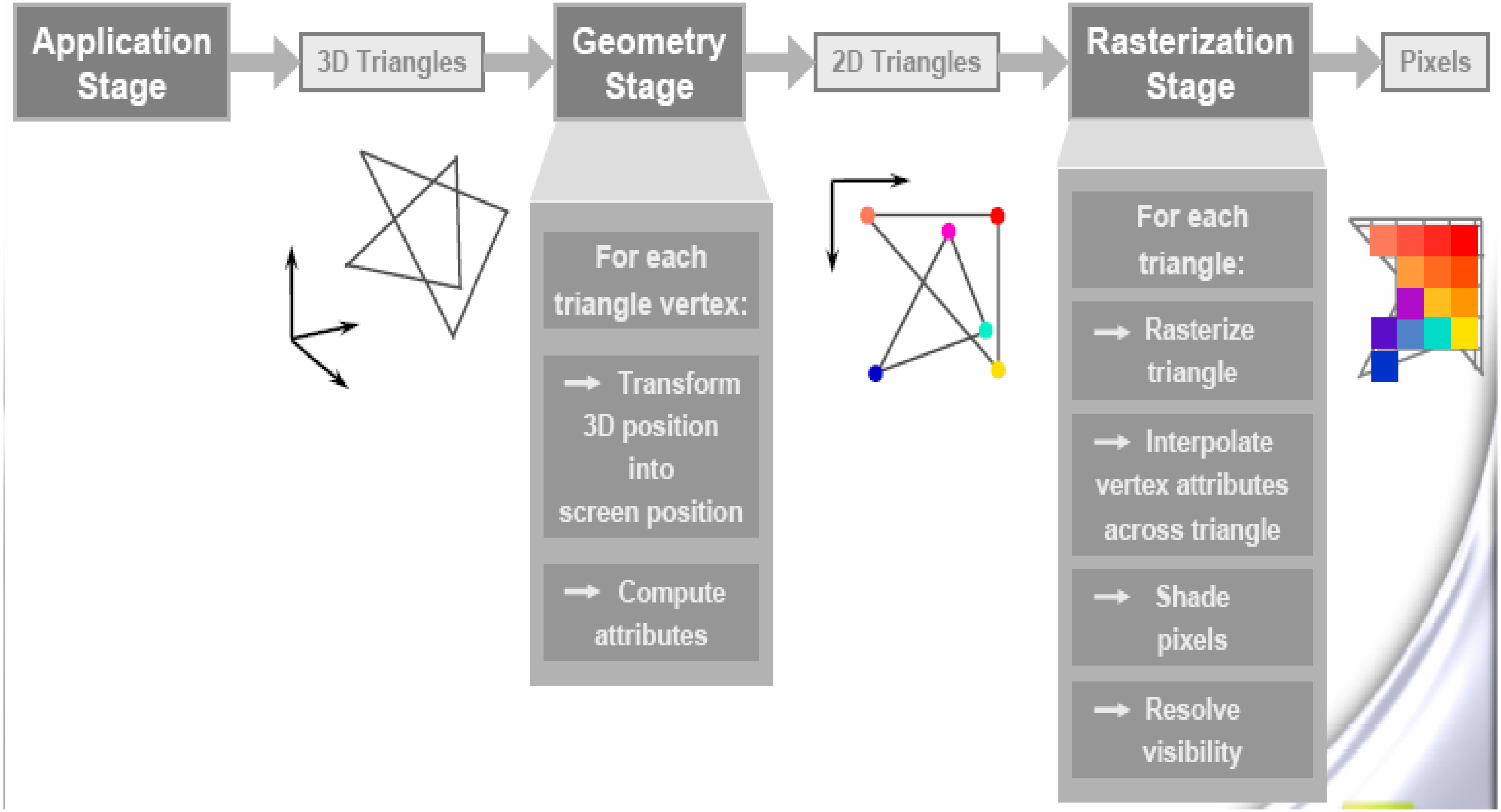
Rendering: part II

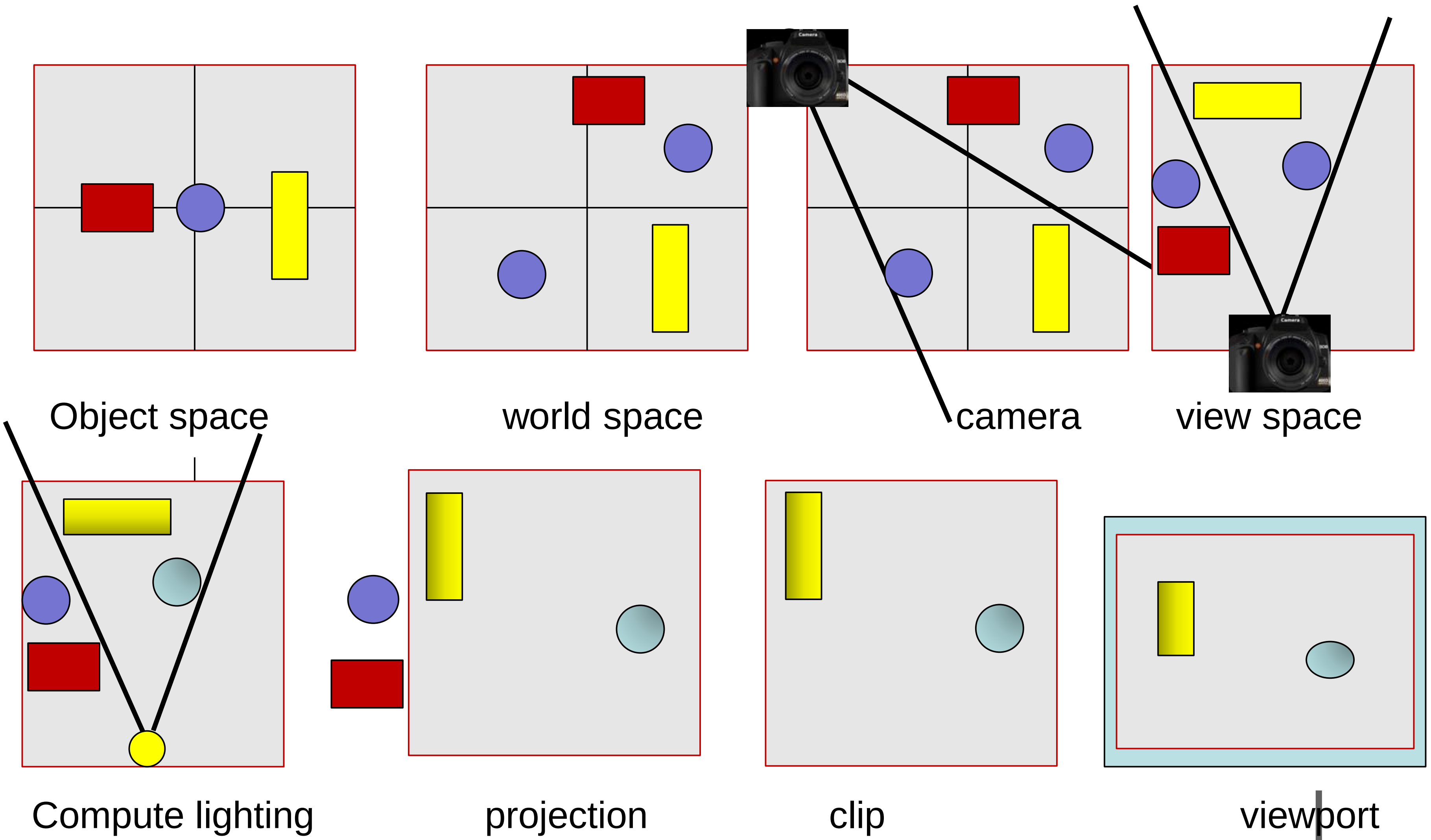


Rendering
is the "engine" that creates images from 3D scenes and a virtual camera

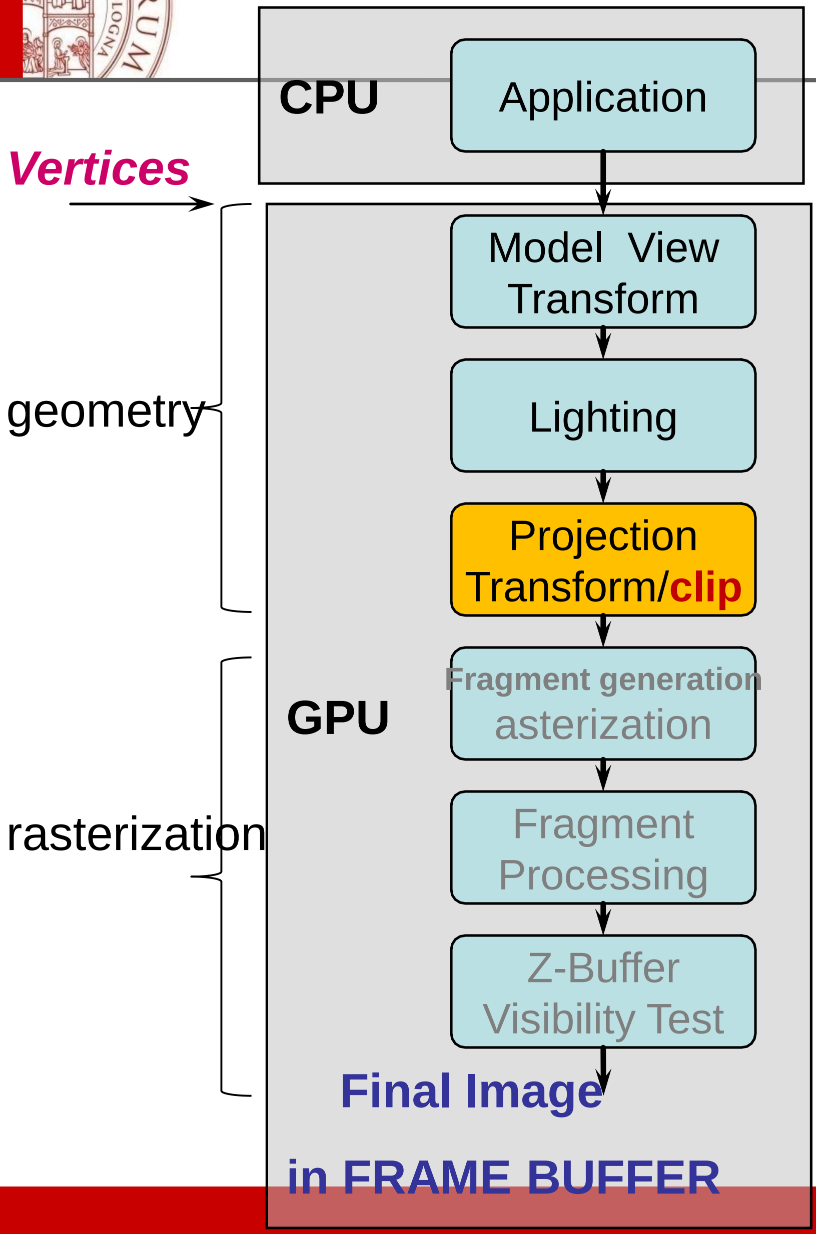
Rendering Pipeline



Geometry stage

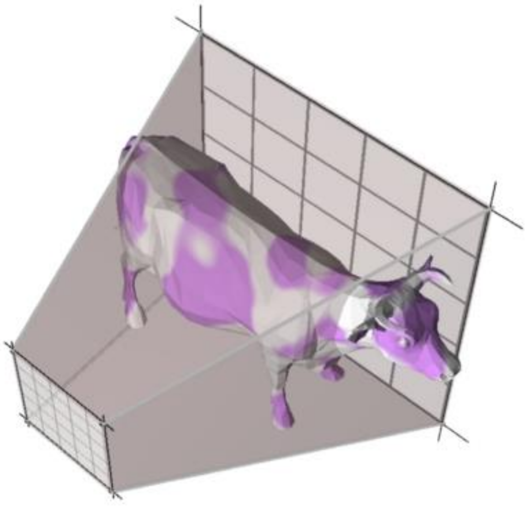


Geometry stage: Clipping



We need to clip scene against sides of view volume

Portions of the object outside the view volume are removed



Clipping: when?

CPU

Application

Vertices

geometry

Model View Transform

Lighting

Projection Transform/**clip**

rasterization

GPU

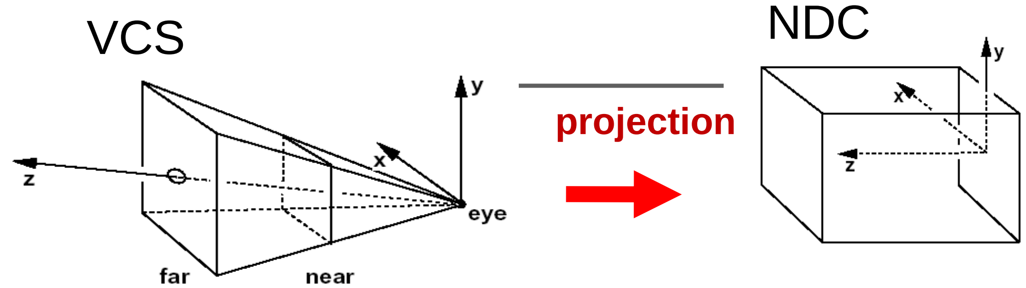
Fragment generation Rasterization

Fragment Processing

Z-Buffer Visibility Test

Final Image

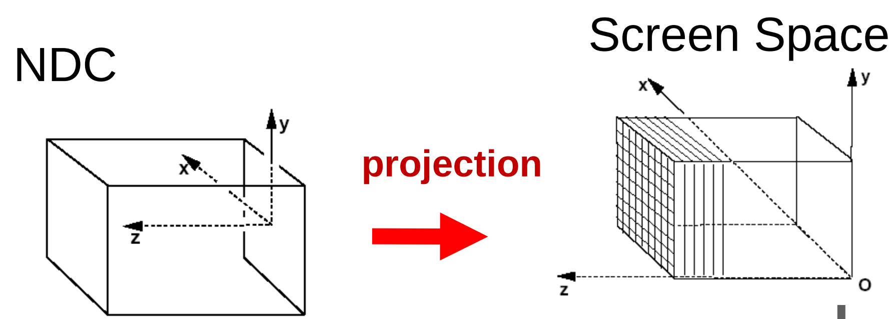
in FRAME BUFFER



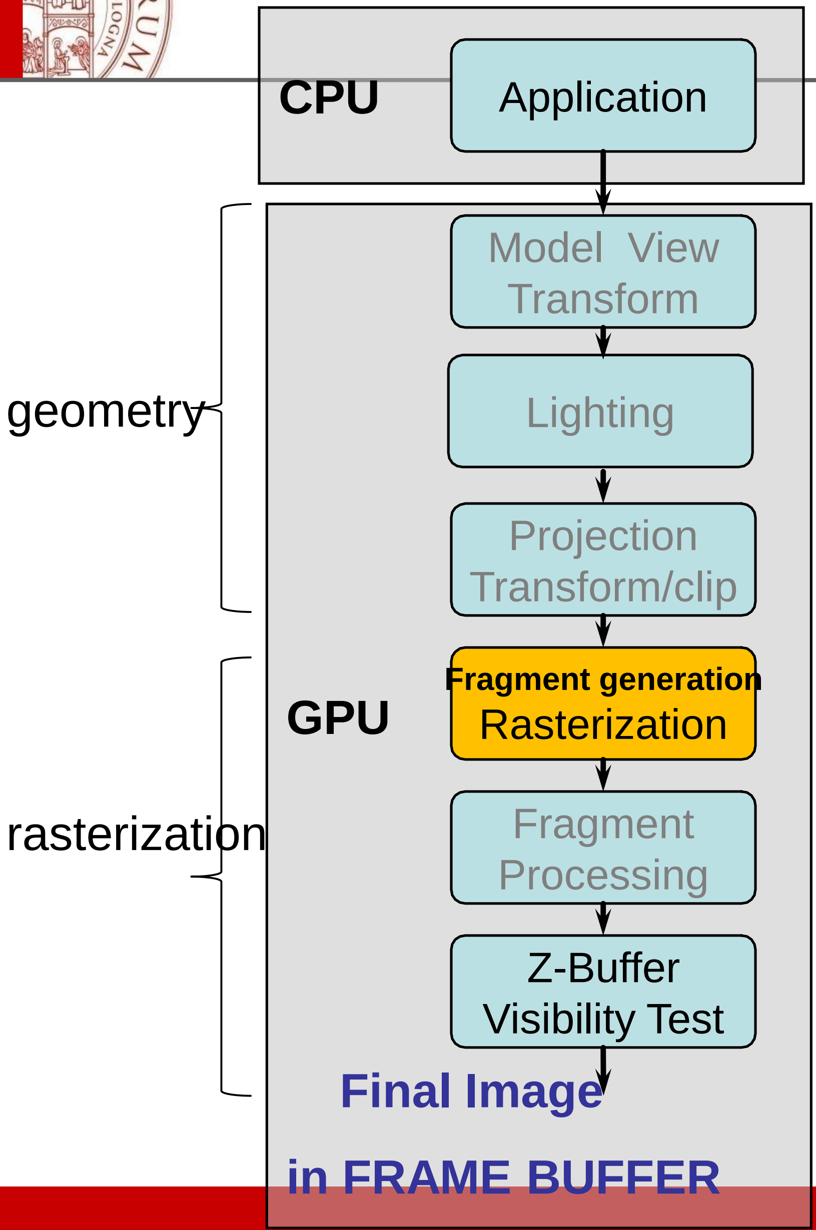
Proj. Transformation (Perspective / Orthographic)

Clipping

Projection (to Screen Space)

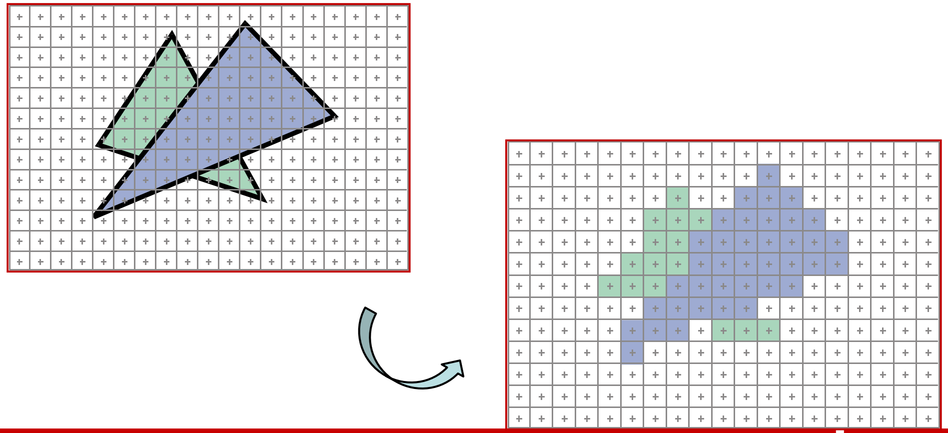


Scan Conversion (Rasterization)

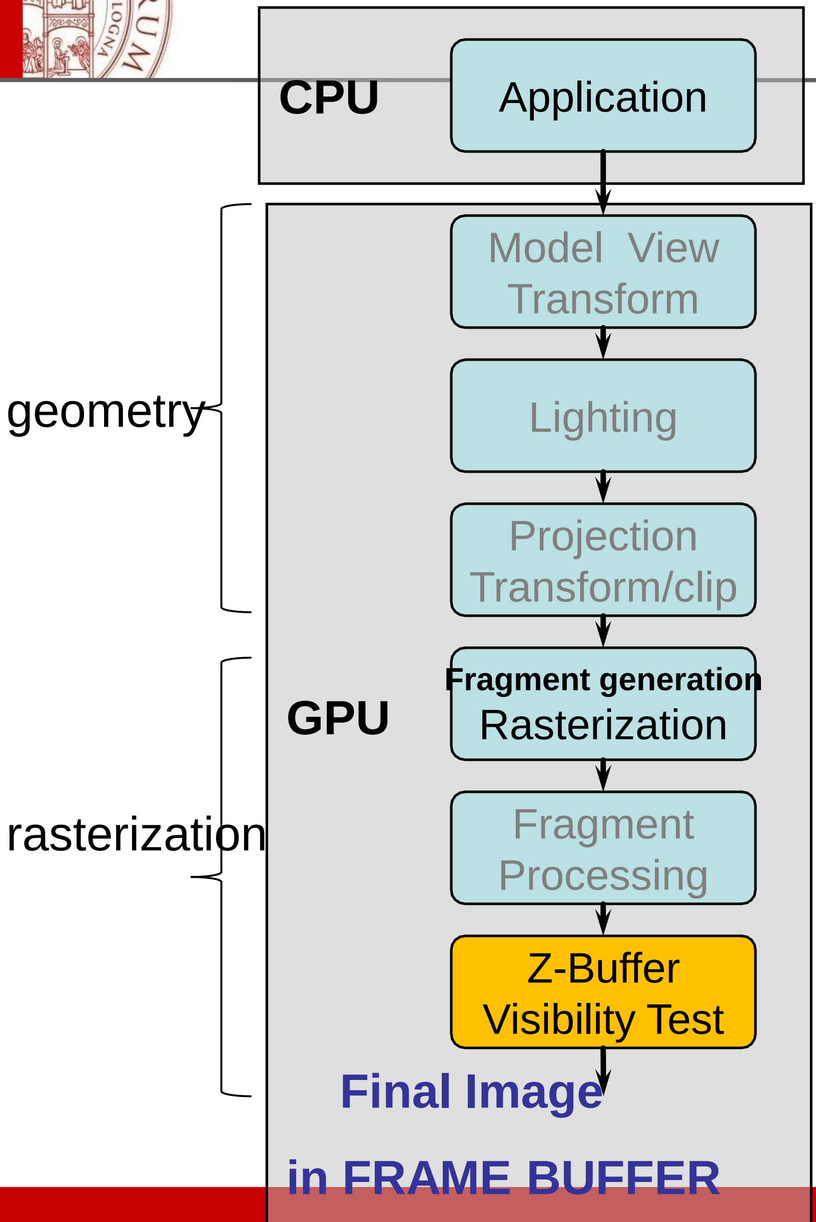


Vertices

- Determine which pixels that are inside primitive specified by a set of vertices
- Produces a set of fragments
- **Fragments** have a location (pixel location) and other attributes such color and texture coordinates that are determined by interpolating values at vertices



Visibility / Display

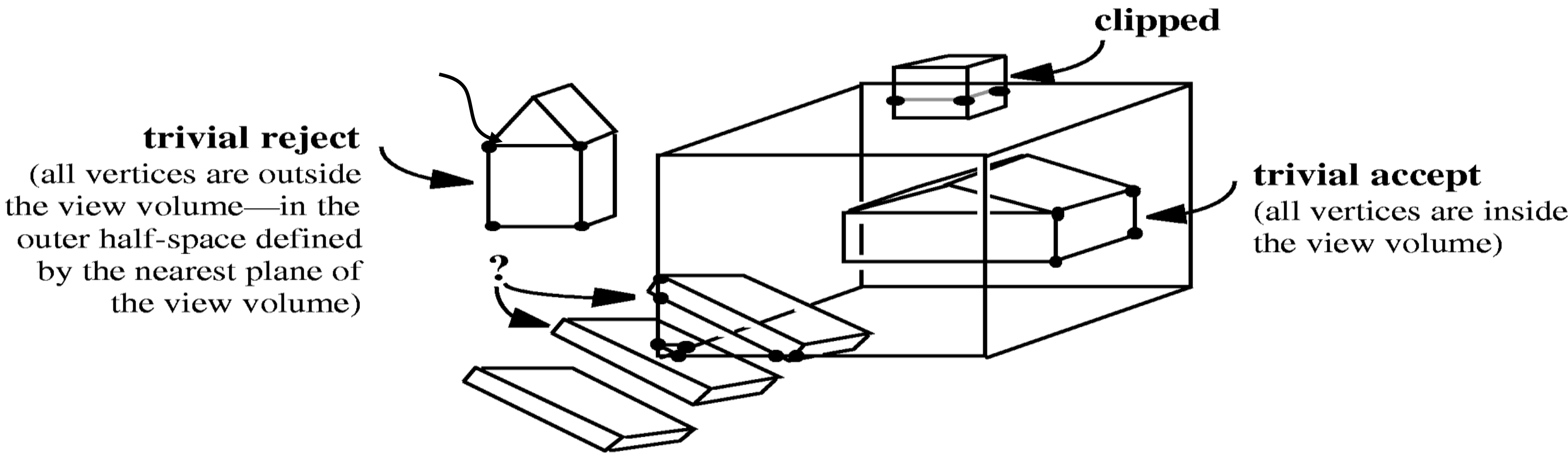


Vertices

- Fragment processing (texturing)
- Draw the closest object
  (depth buffer)

Clipping Against View Volume

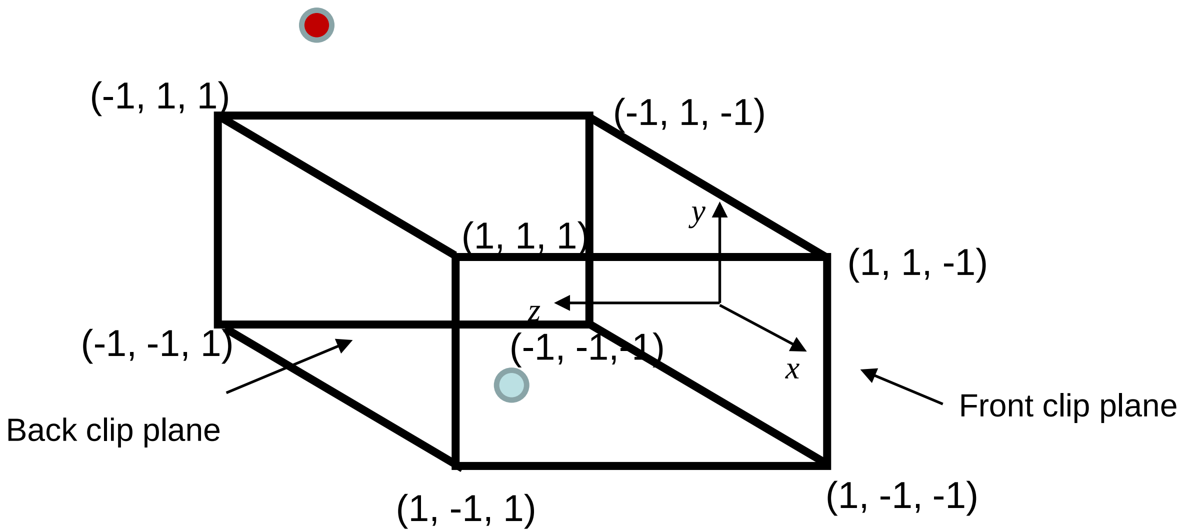
- **Polyhedra** transformed to the normalized world are clipped against bounds of canonical view volume, one polygon at a time
- **Polygons** are clipped one edge at a time
- Intersection calculations are trivial because of normalized planes of **canonical view volume**
- New vertices are created where objects are clipped



- Use **bounding volumes** to trivially reject groups of objects at a time

Point Clipping

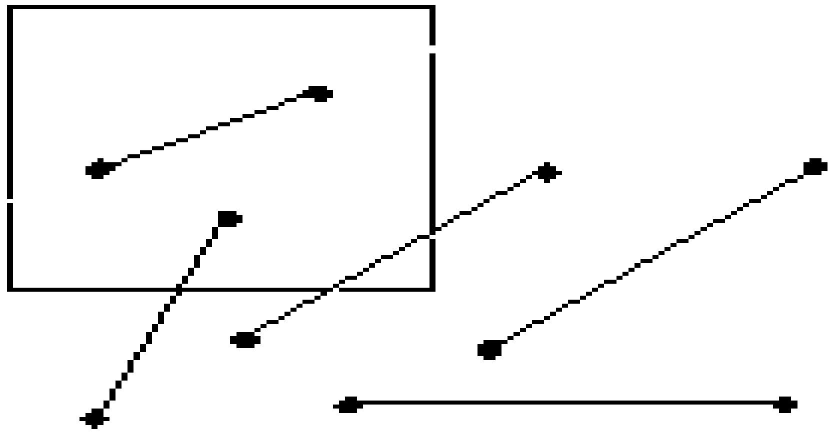
- The view volume is a cuboid that extends from -1 to 1 in *x* and *y* and z
- Test components of vertices -1 <= *x,y,z* <=1



- Points falling within these values are saved, and vertices falling outside get clipped

Line Clipping

Endpoint analysis for lines:



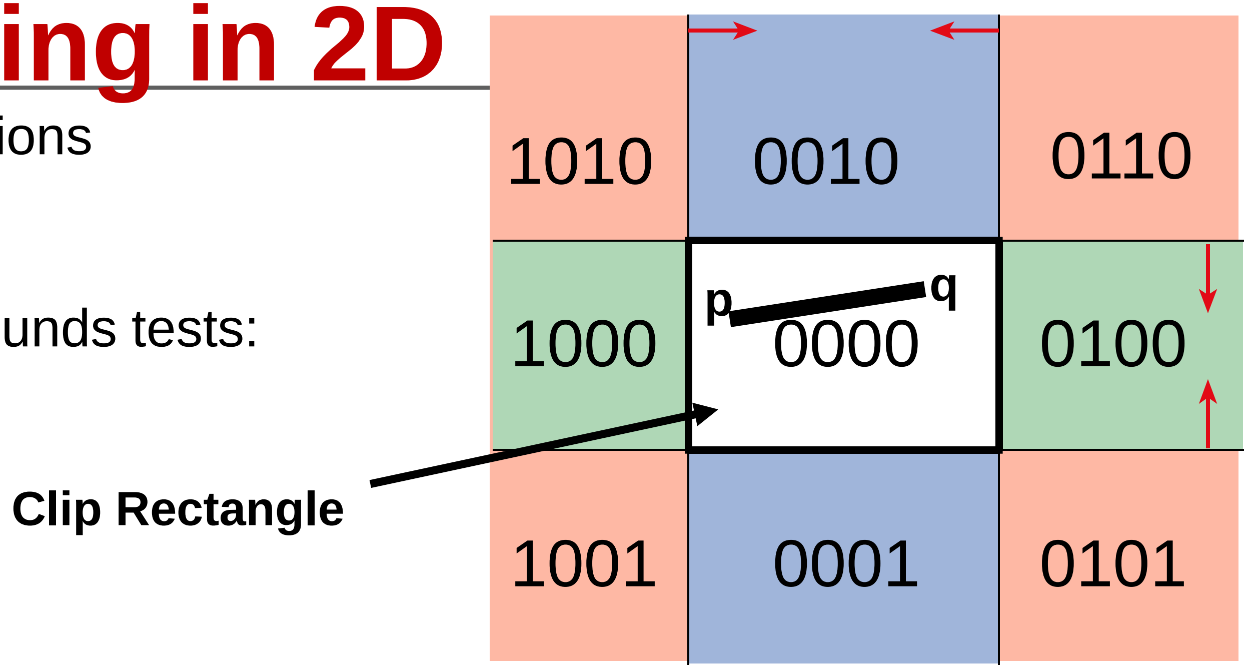
- if both endpoints in, can do “trivial acceptance”
- if one endpoint is inside, one outside, must clip
- if both endpoints out, don’t know

Cohen-Sutherland Line Clipping in 2D

- Divide plane into 9 regions

- **4 bit outcode OC**

Records results of four bounds tests:



First bit: $x<x_{min}$ outside halfplane of left edge, to left of left edge
Second bit: $x>x_{max}$ outside halfplane of right edge, to right of right edge
Third bit: $y>y_{max}$ outside halfplane of top edge, above top edge
Fourth bit: $y<y_{min}$ outside halfplane of bottom edge, below bottom edge

- Determine OC for the line vertices (OC_p , OC_q)
- **CASE 1:**

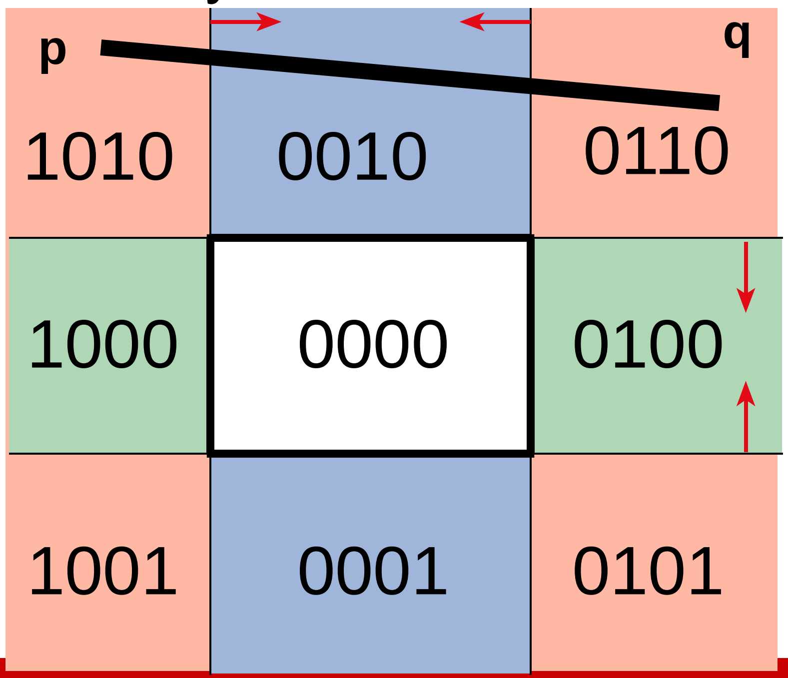
Lines with *OCp* = 0000 and *OCq* = 0000 can be *trivially* ***accepted***

- **CASE 2**:

Lines lying entirely in a half plane on outside of an edge can be *trivially* ***rejected***:

$$OCp \text{ AND } OC_q \neq 0000$$

(i.e., they share an “outside” bit)



Outcode of p : 1010

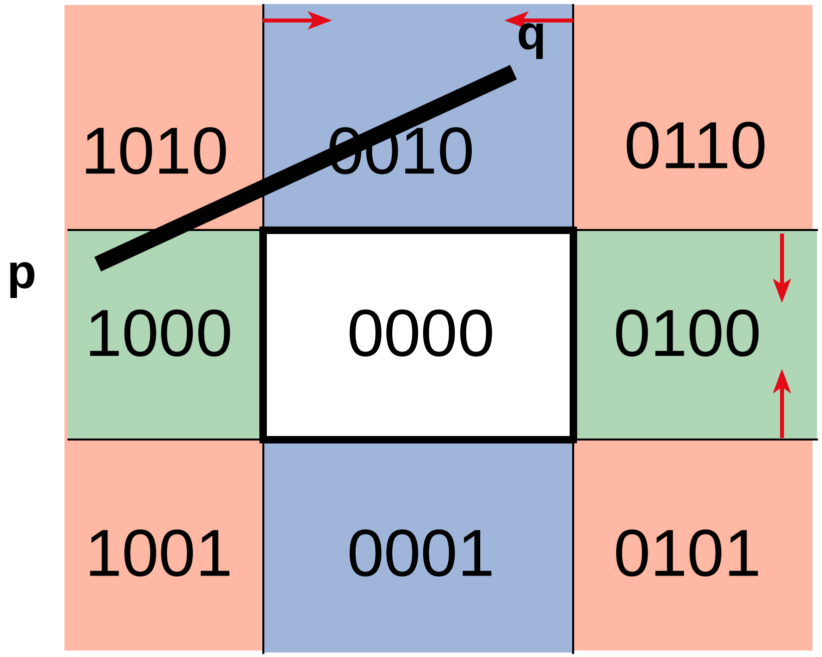
Outcode of q : 0110

Outcode of [pq] : 0010

Clipped because there is a 1

- **CASE 3:**

External vertices but the line can not be rejected



$$OCp \text{ AND } OC_q = 0000$$

Outcode of p : 1000

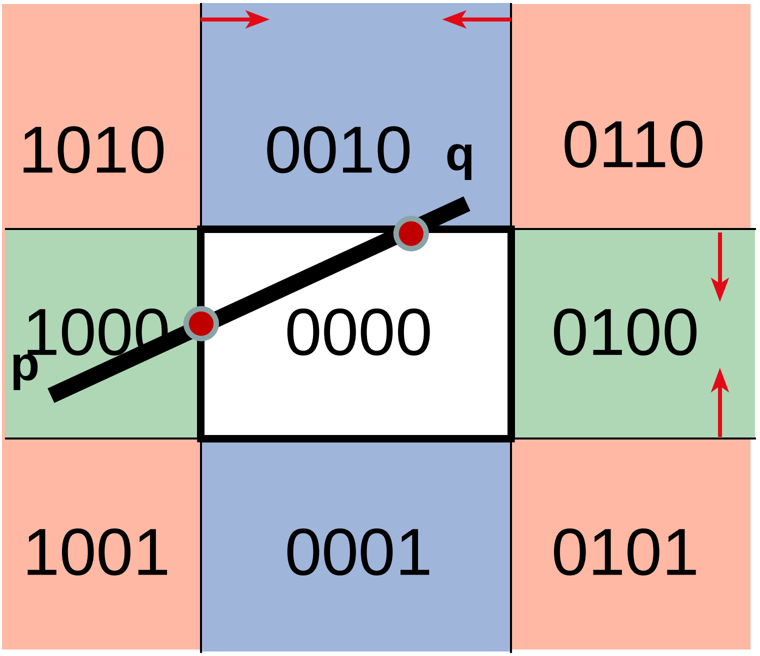
Outcode of q : 0010

Outcode of [pq] : 0000

Not clipped

External vertices but the line can not be rejected OC_p AND OC_q = 0000

- Consider one vertex with OC $\neq$0000;
- Determine the intersection between the line and the window edge (corresponding to the bit $\neq$ 0 in the OC);
- If the intersection has OC=0000 then update the vertex with the computed intersection vertex, otherwise reject segment
- Repeat test

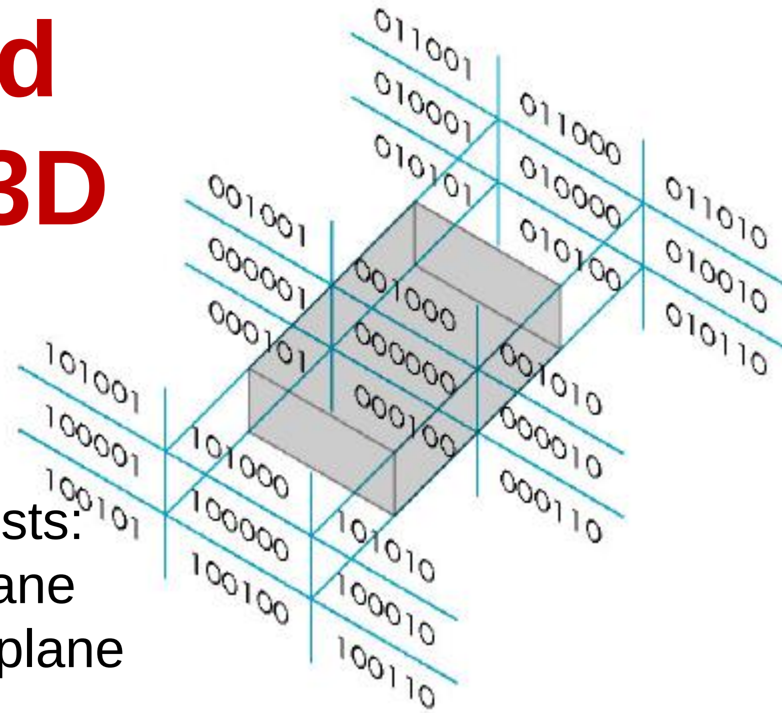


Outcode of p : 1000

Outcode of q : 0010

Outcode of [pq] : 0000

Cohen-Sutherland Line Clipping in 3D



- Very similar to 2D
- Divide volume into 27 regions

- **6 bit outcode** records results of 6 bounds tests:

First bit: outside back plane, behind back plane
Second bit: outside front plane, in front of front plane
Third bit: outside top plane, above top plane
Fourth bit: outside bottom plane, below bottom plane
Fifth bit: outside right plane, to right of right plane
Sixth bit: outside left plane, to left of left plane

- Lines with $OC_0 = 000000$ and $OC_1 = 000000$ can be ***trivially accepted***
- Lines lying entirely in a volume on outside of a plane can be ***trivially rejected***:

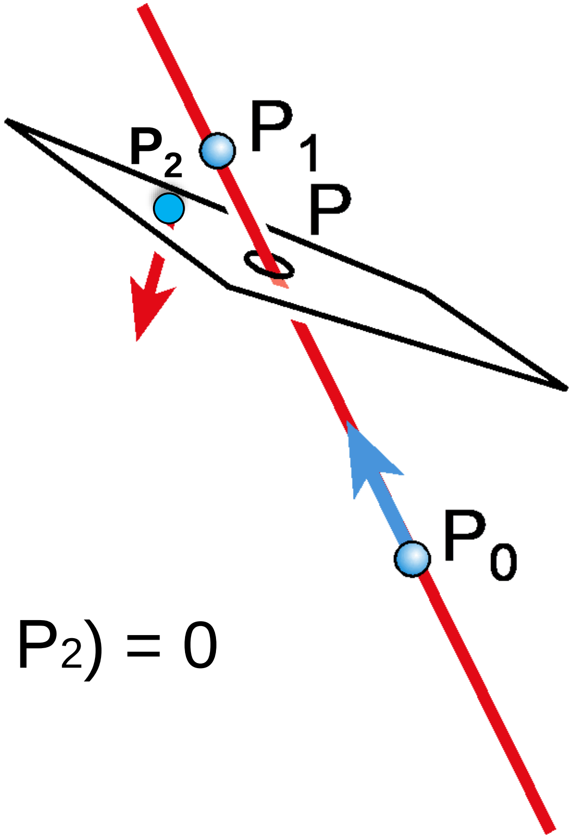
$$OC_0 \text{ AND } OC_1 \neq 0 \text{ (i.e., they share an “outside” bit)}$$

- Otherwise CLIP

Line – Plane Intersection

- Explicit (Parametric) Line Equation

$$P(t) = P_0 + t\,(P_1 - P_0)$$
$$P(t) = (1-t)\,P_0 + t\,P_1$$



- How do we intersect?

  Insert explicit equation of line into the plane equation

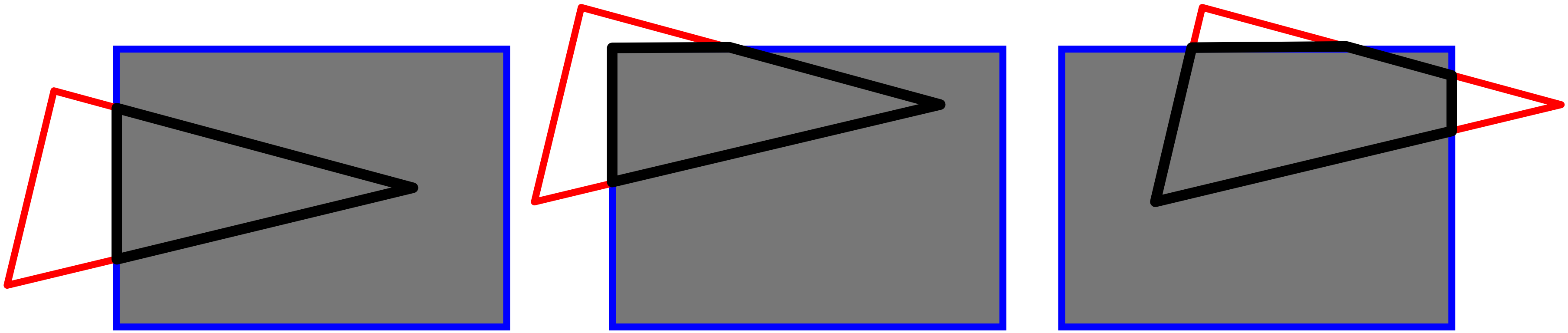
- $P(t)= (1 - t)P_0 + tP_1$ $n \cdot (P(t) - P_2) = 0$

$$\mathbf{t= n\cdot(P_2-P_0)/n\cdot(P_1-P_0)}$$

Then find the intersection point and shorten the line

Polygon Clipping

- What happens to a triangle during clipping?
  - possible outcomes:

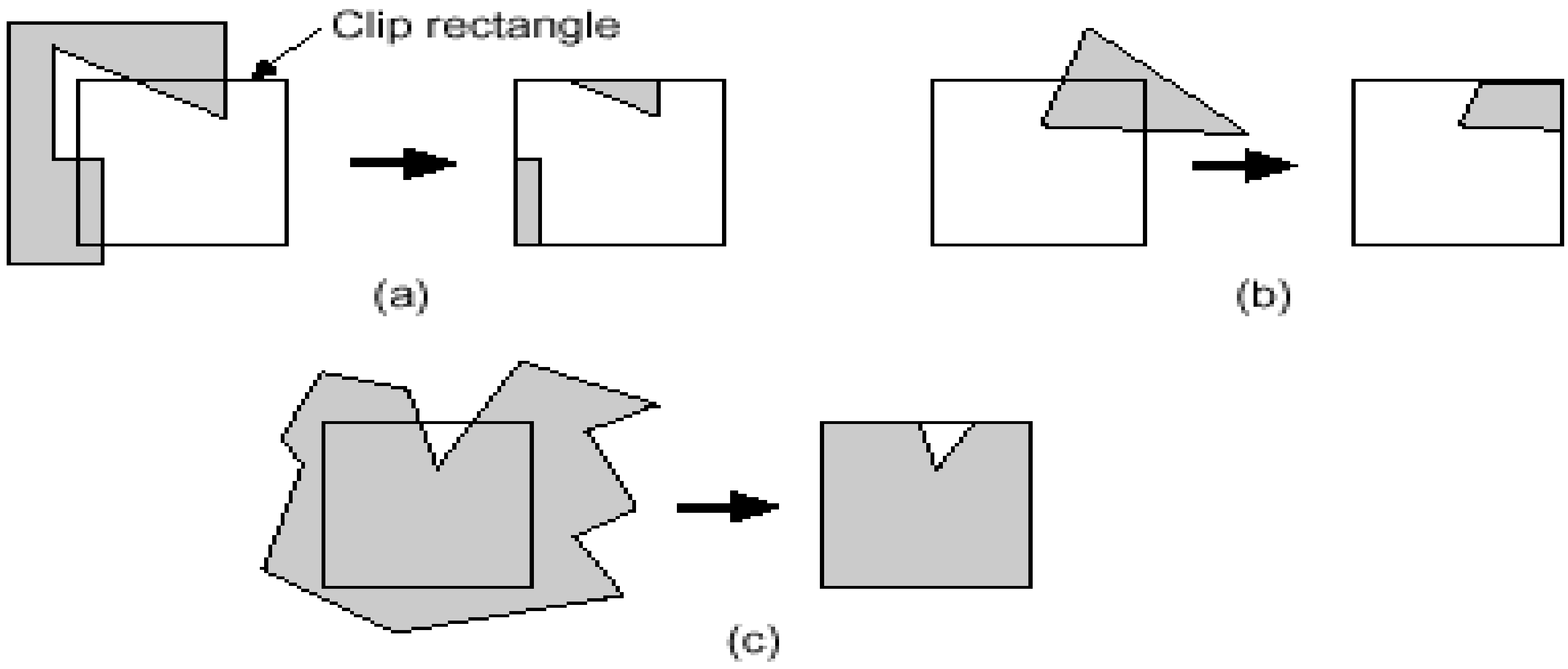


triangle → triangle   triangle → quad   triangle → 5-gon

How many sides could a clipped triangle have?

Sutherland-Hodgman Polygon Clipping

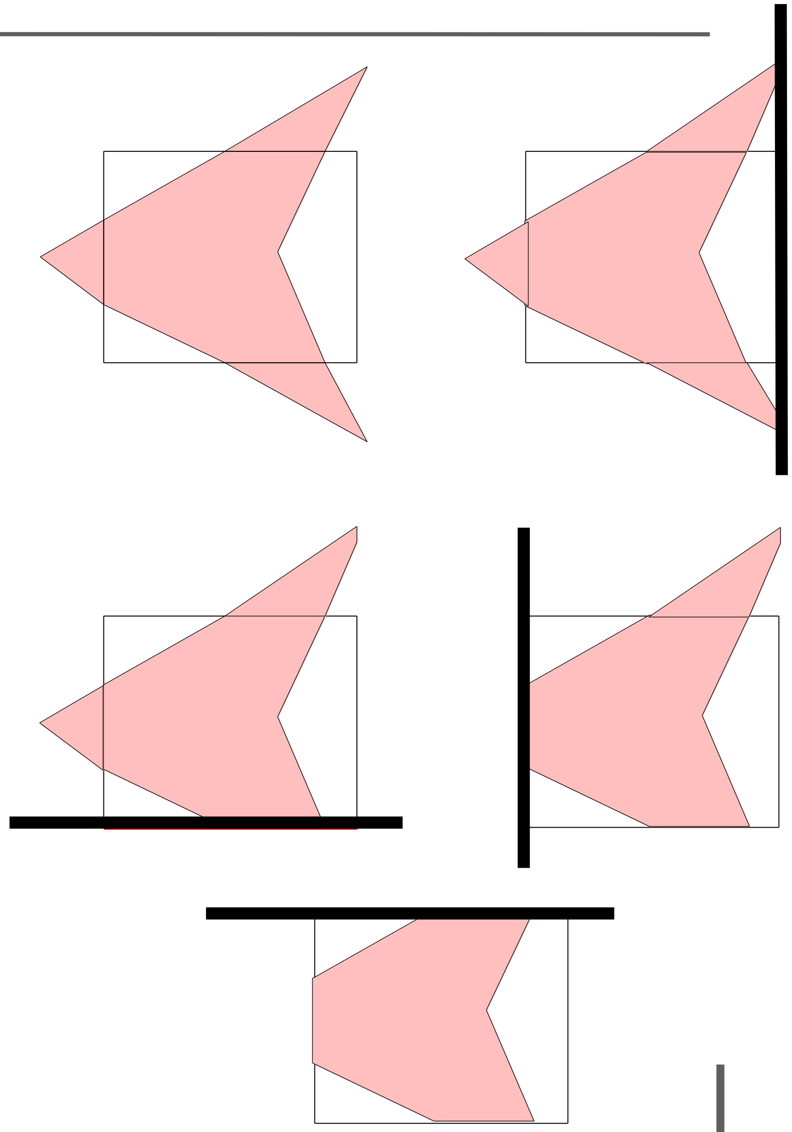
INPUT: v1, v2, ..vn ordered polygon vertices
OUTPUT: one or more polygons (for nonconvex)



Sutherland-Hodgman Polygon Clipping

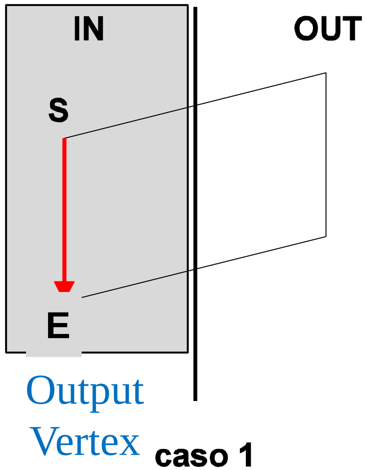
Decompose the problem in simple clip pipelined subproblems:

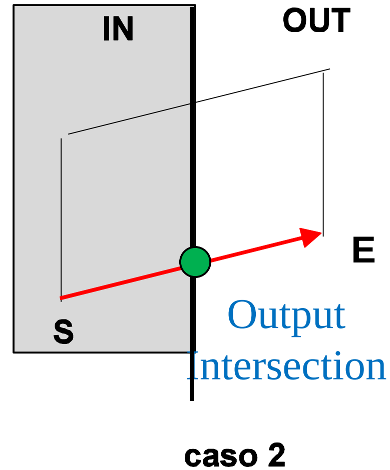
-Clip right window boundary
-Clip bottom window boundary
-Clip left window boundary
-Clip top window boundary



For each window boundary:

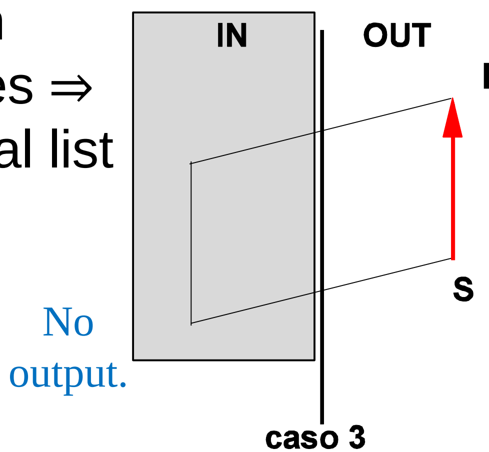
- **CASE 1**: internal polygon edge ⇒ add E to the final vertex list

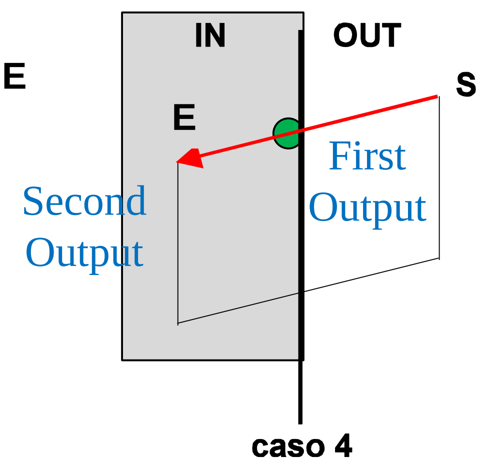




- **CASE 2**: add the intersection point to the final vertex list

- **CASE 3**: both external vertices ⇒ no vertex in final list

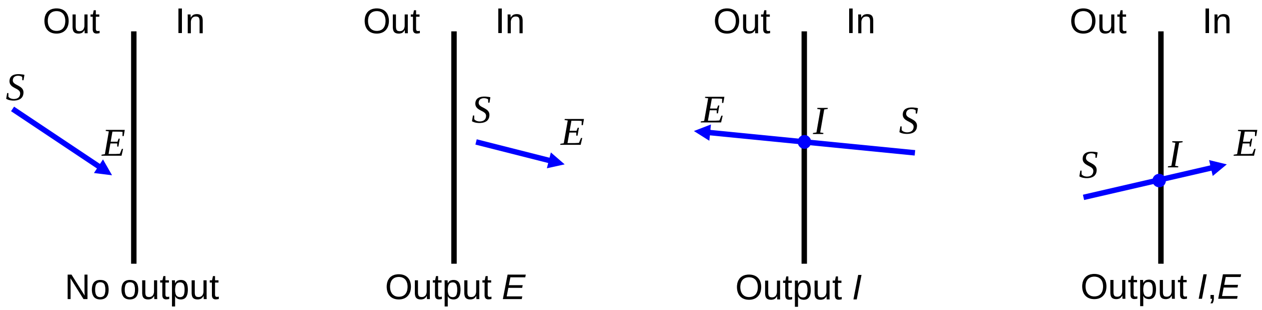


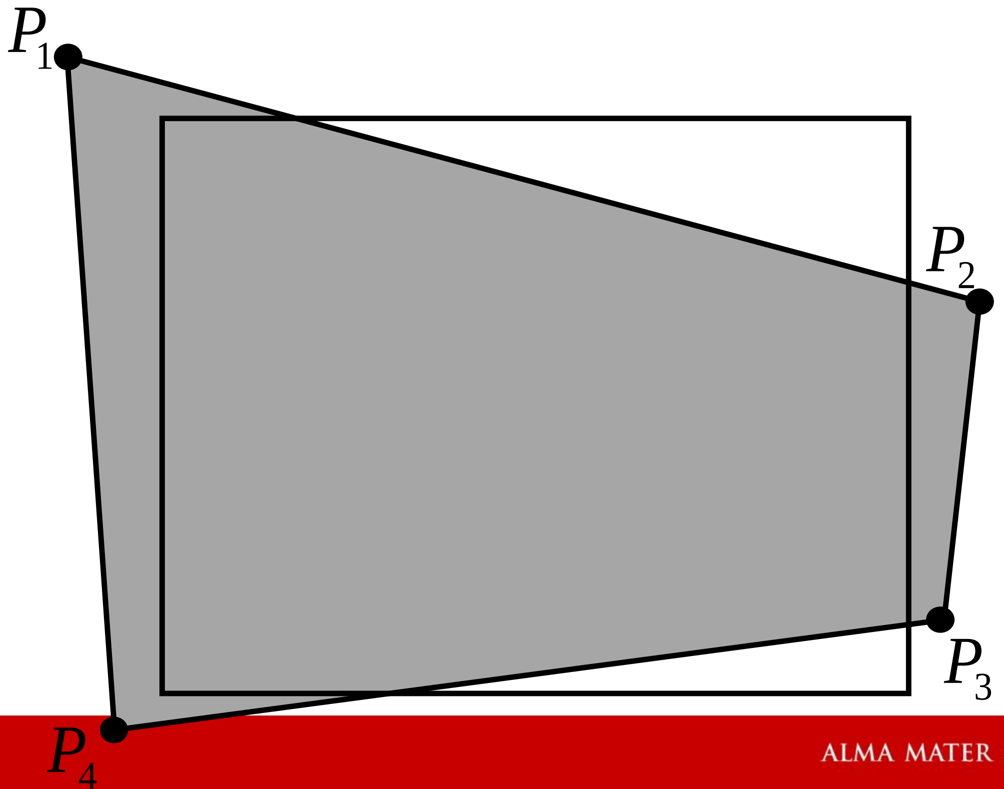


- **CASE 4**: add intersection point and E to the final vertex list

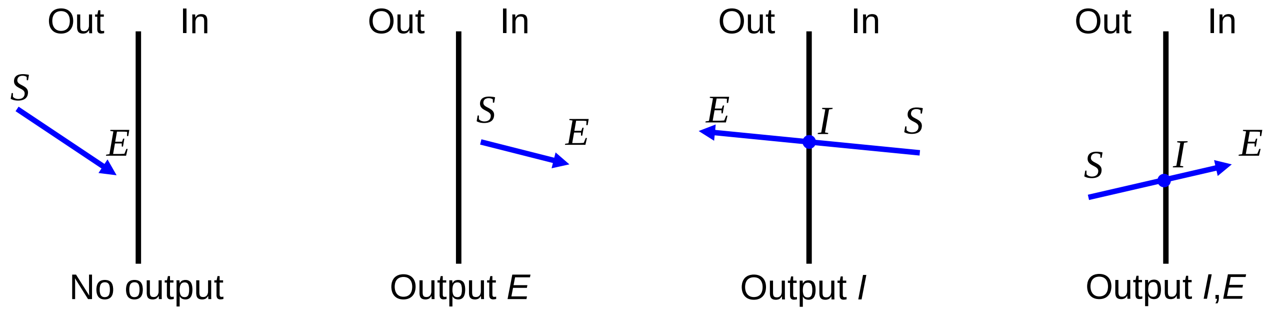
Clipping pipeline on the other window boundaries

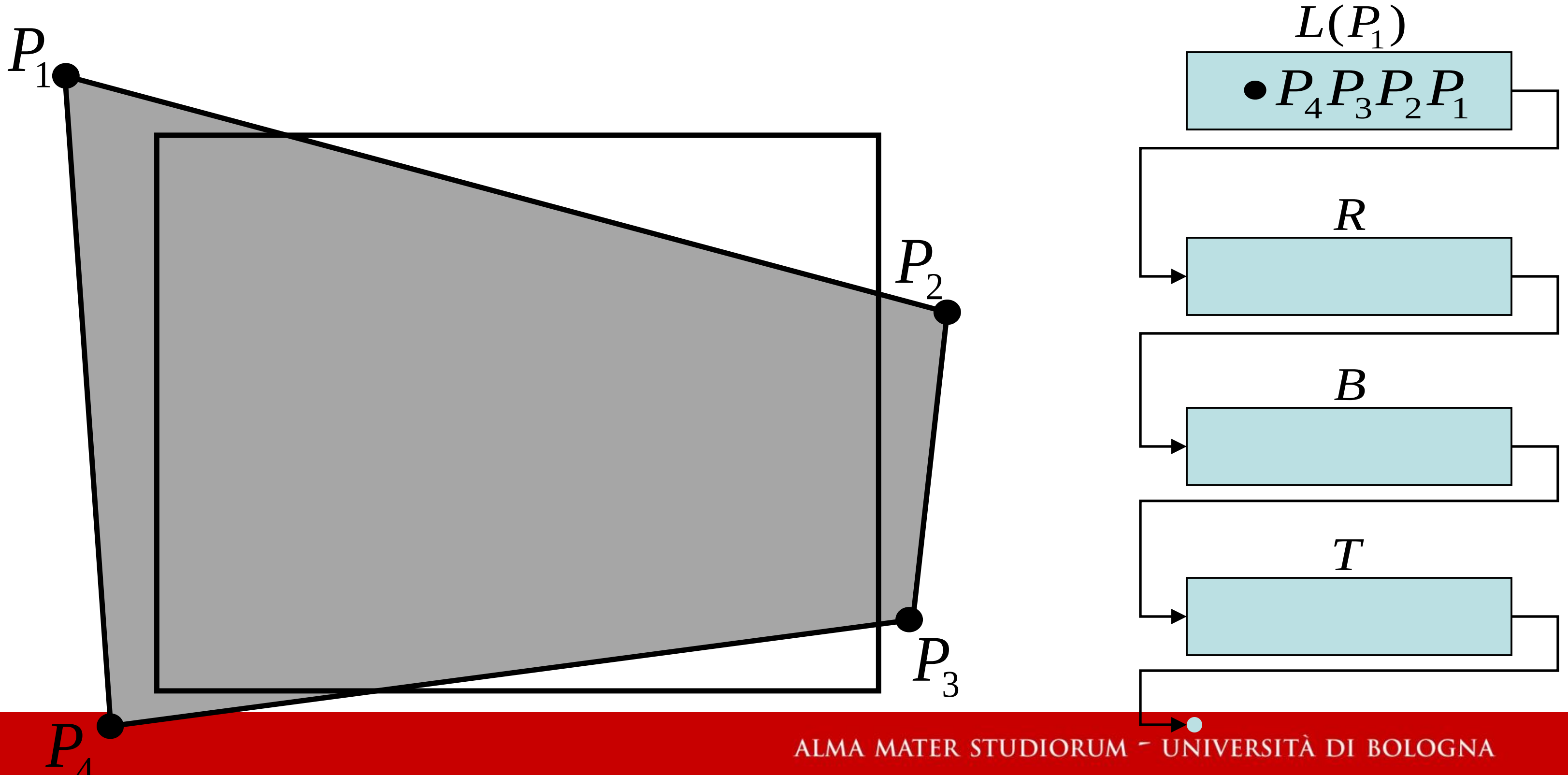
Sutherland-Hodgman Algorithm



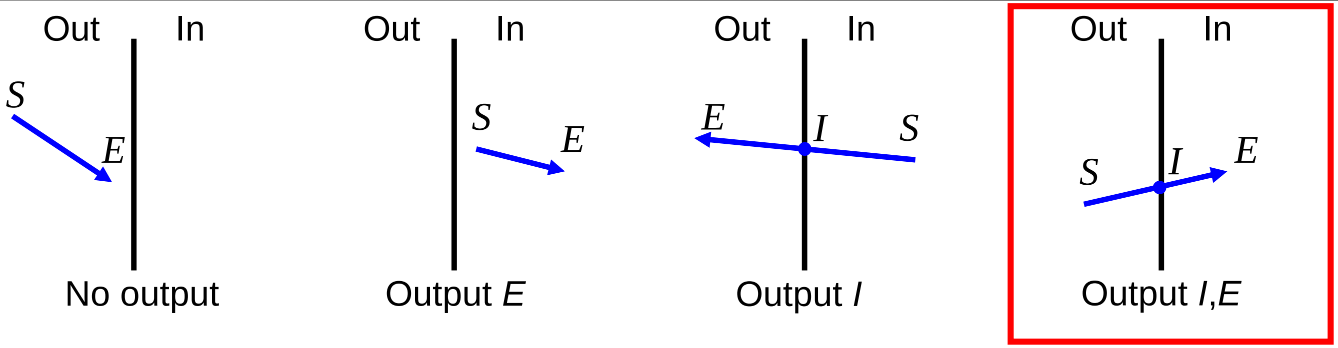


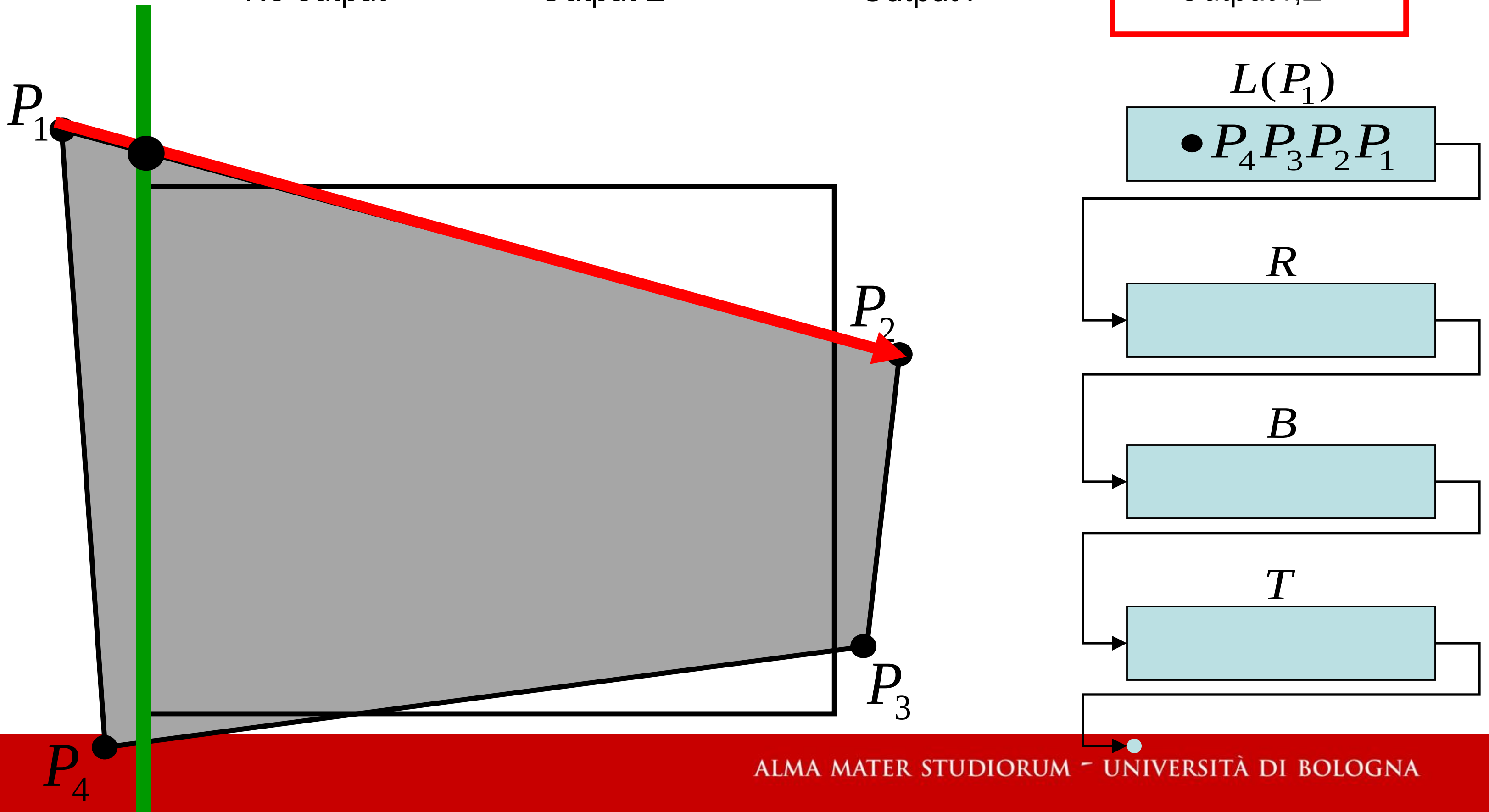
Sutherland-Hodgman Algorithm



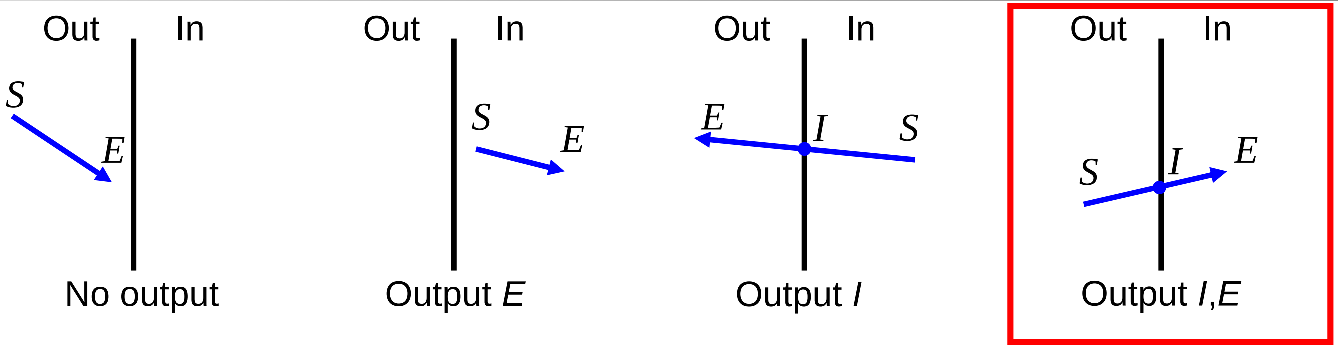


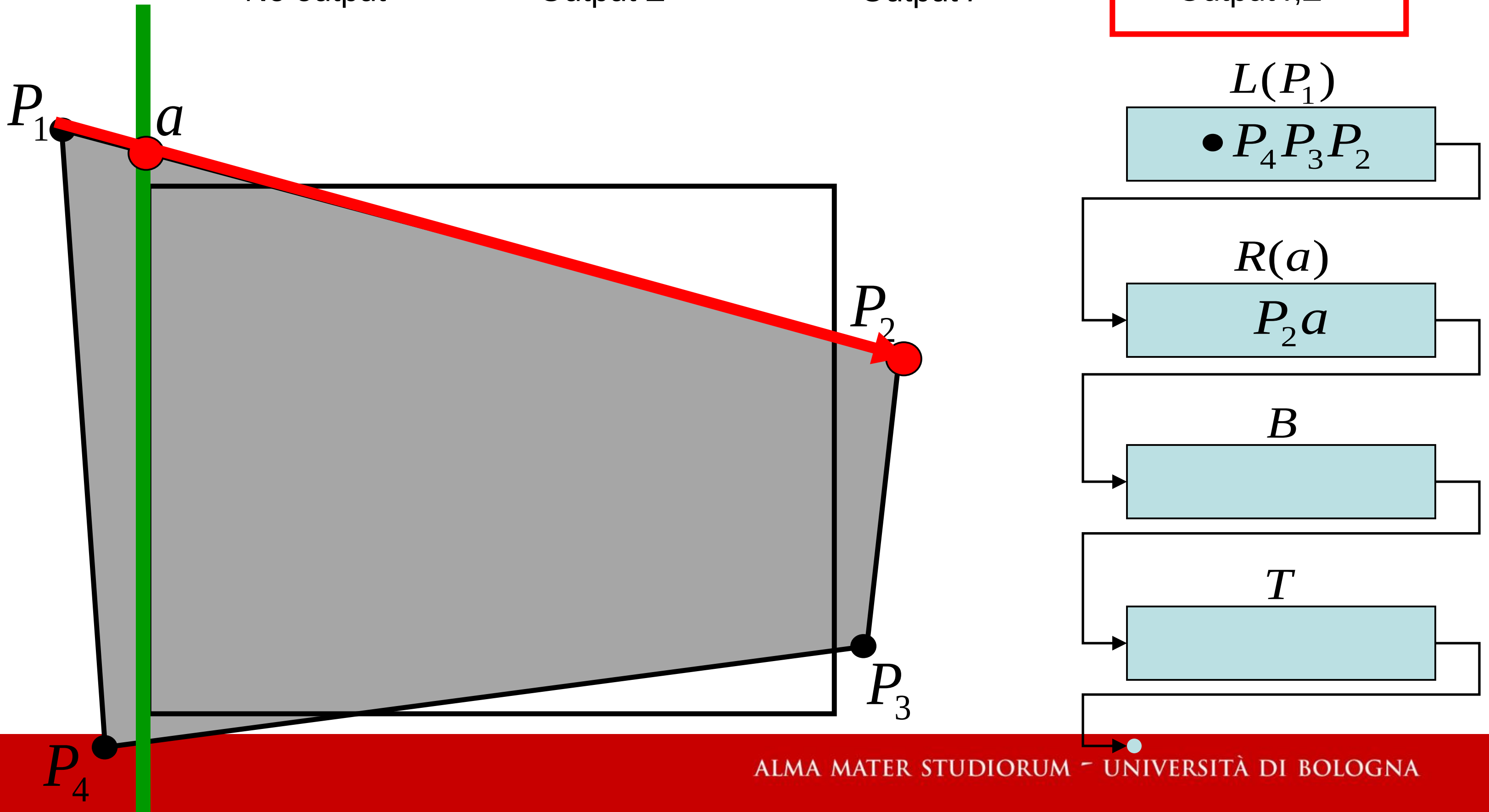
Sutherland-Hodgman Algorithm



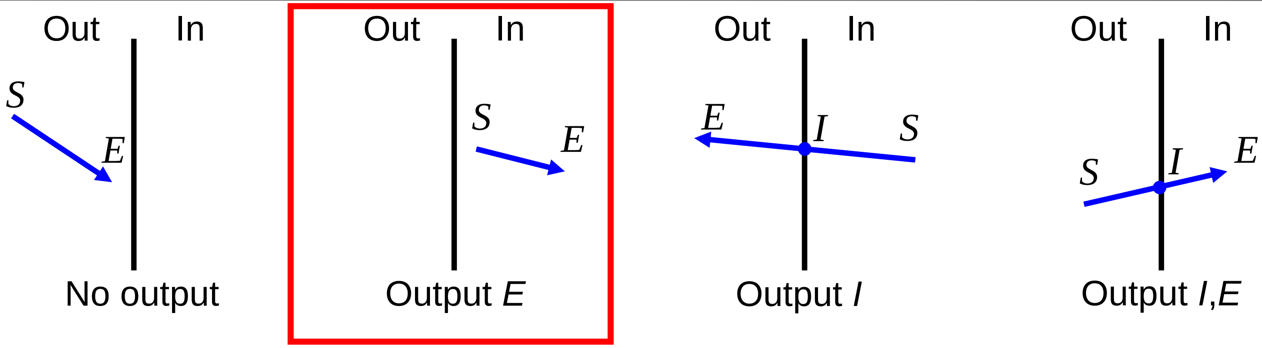


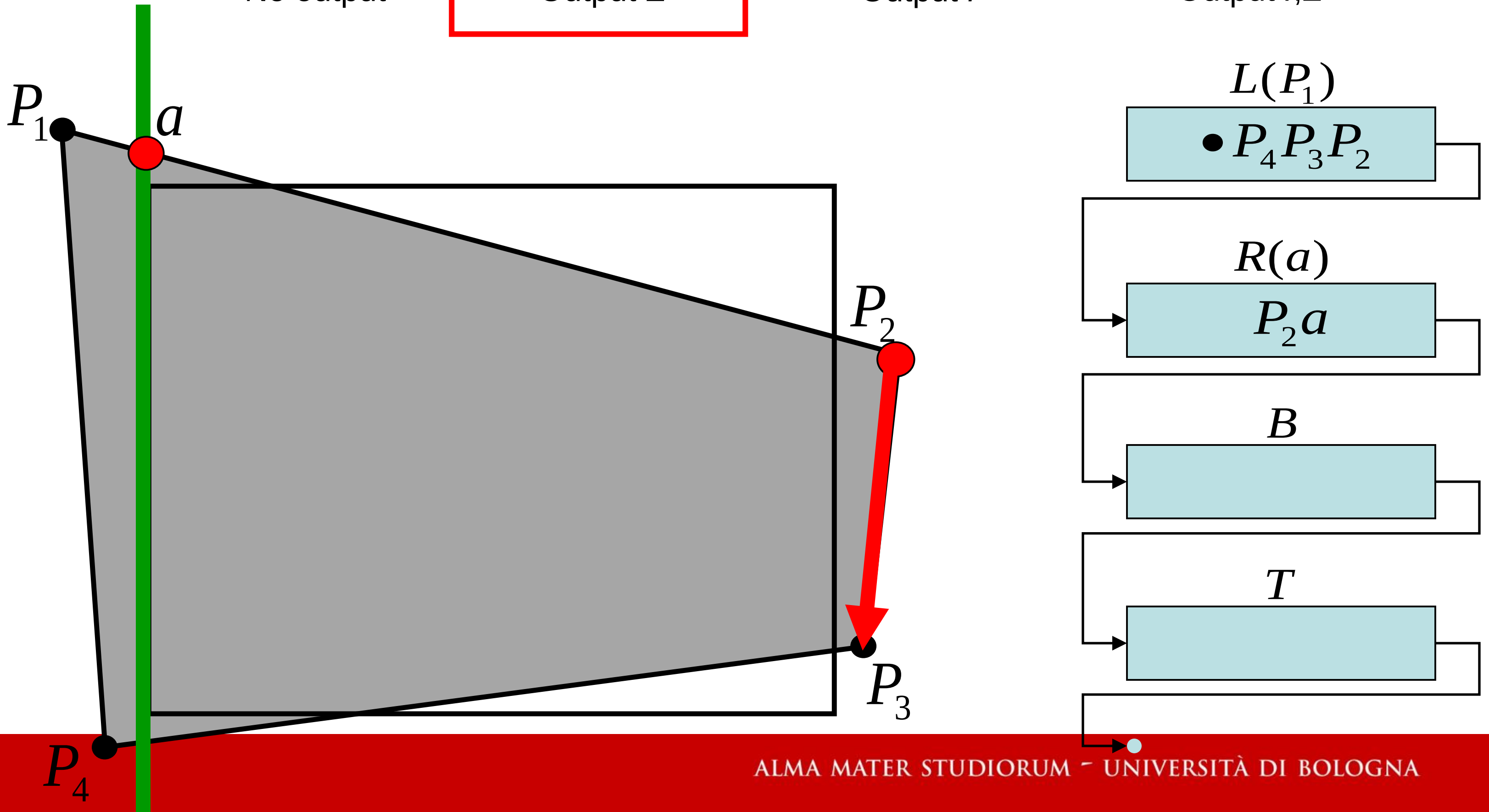
Sutherland-Hodgman Algorithm



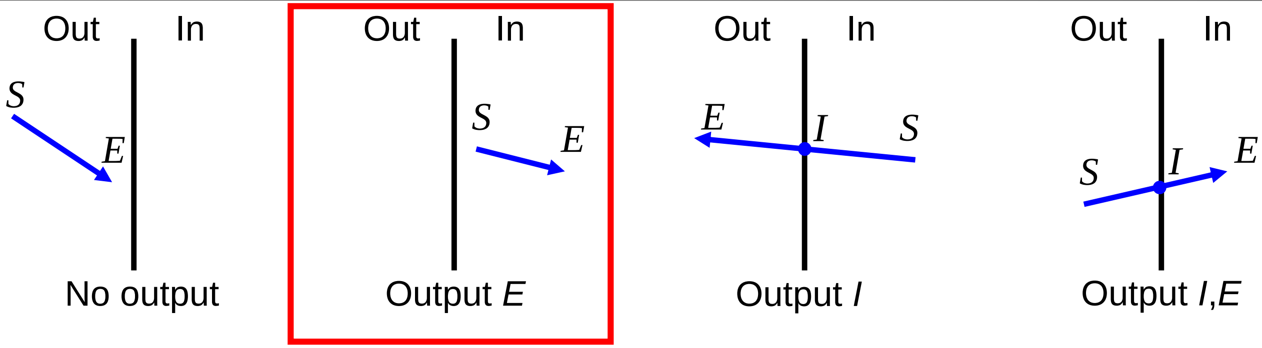


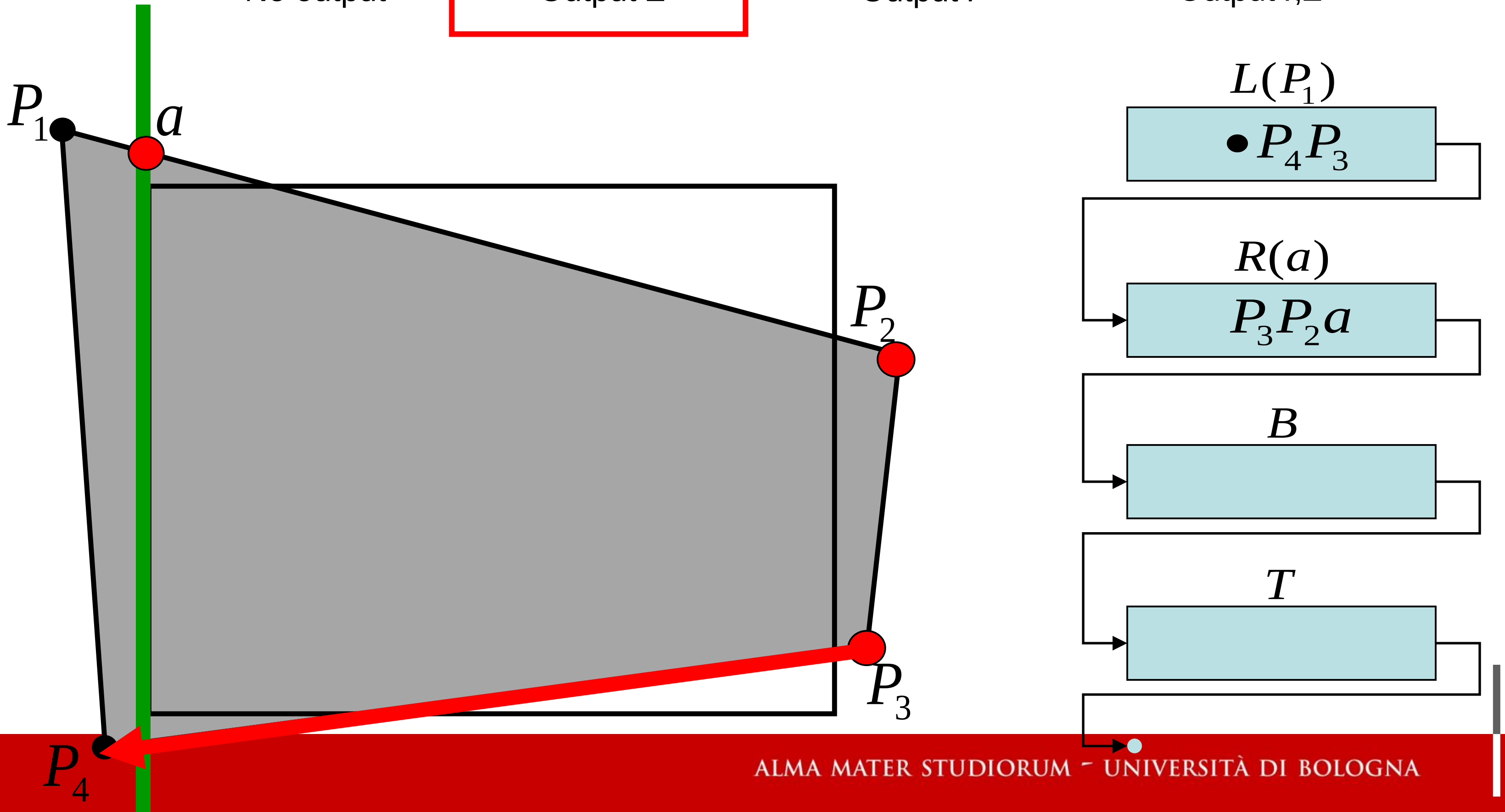
Sutherland-Hodgman Algorithm



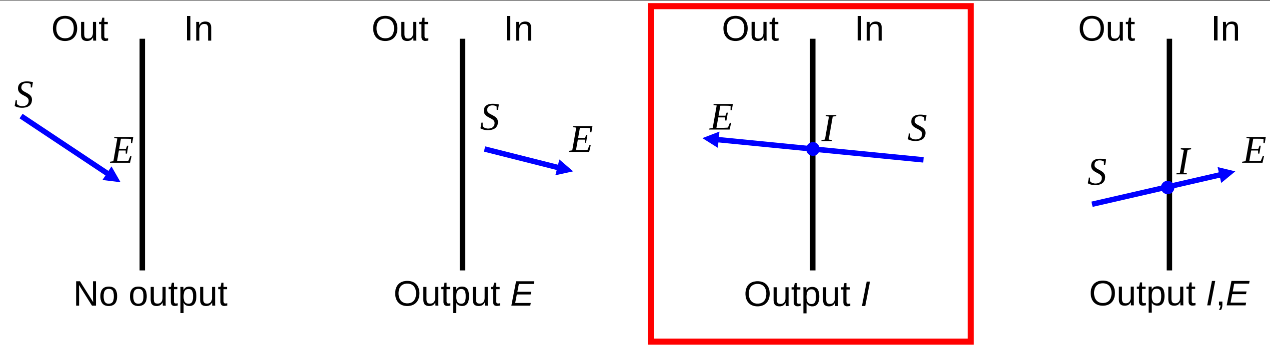


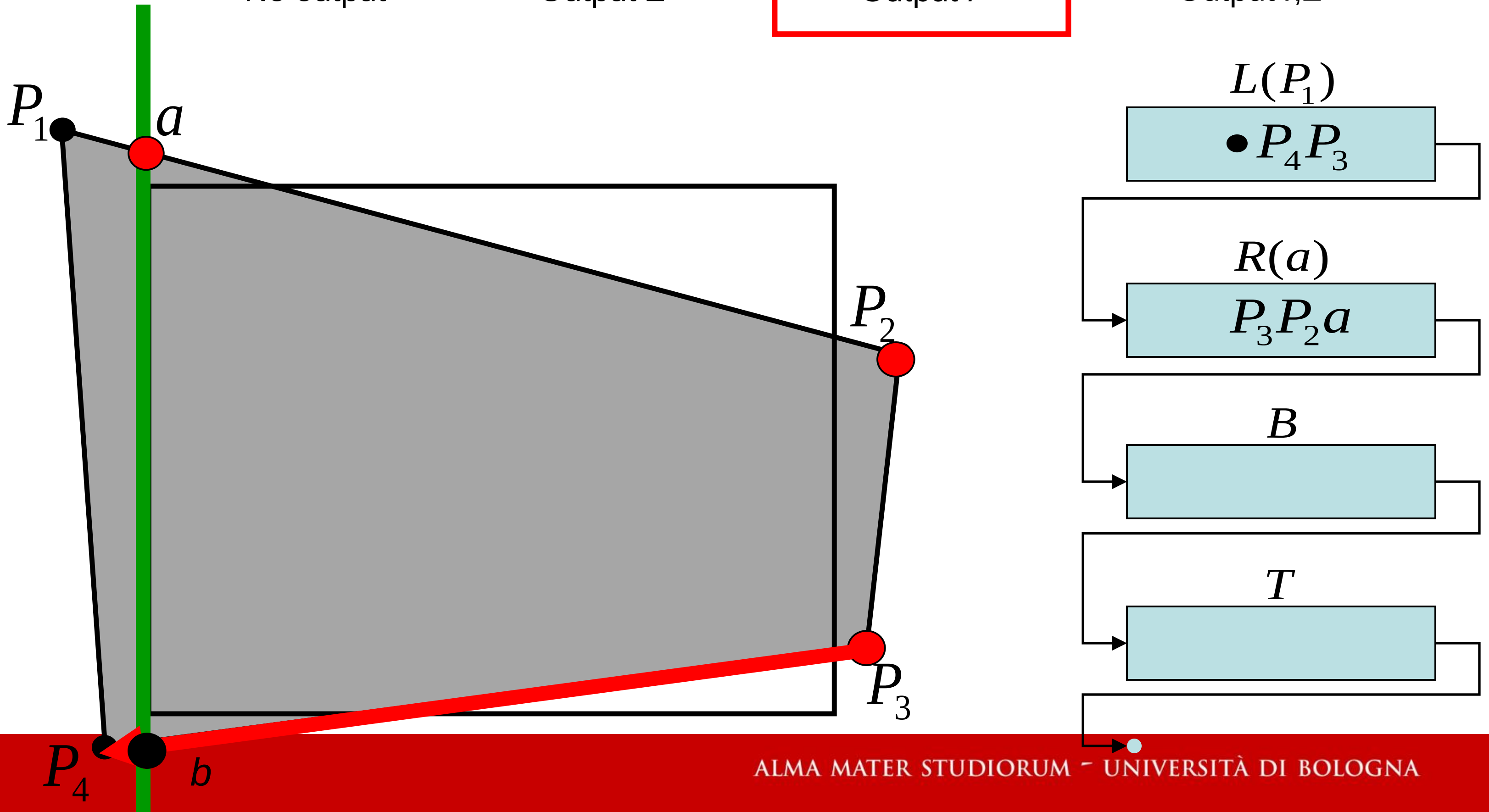
Sutherland-Hodgman Algorithm



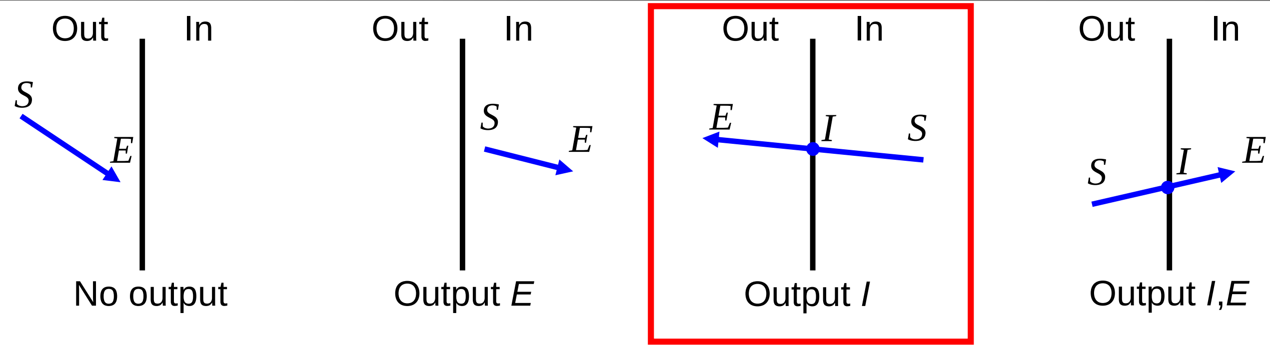


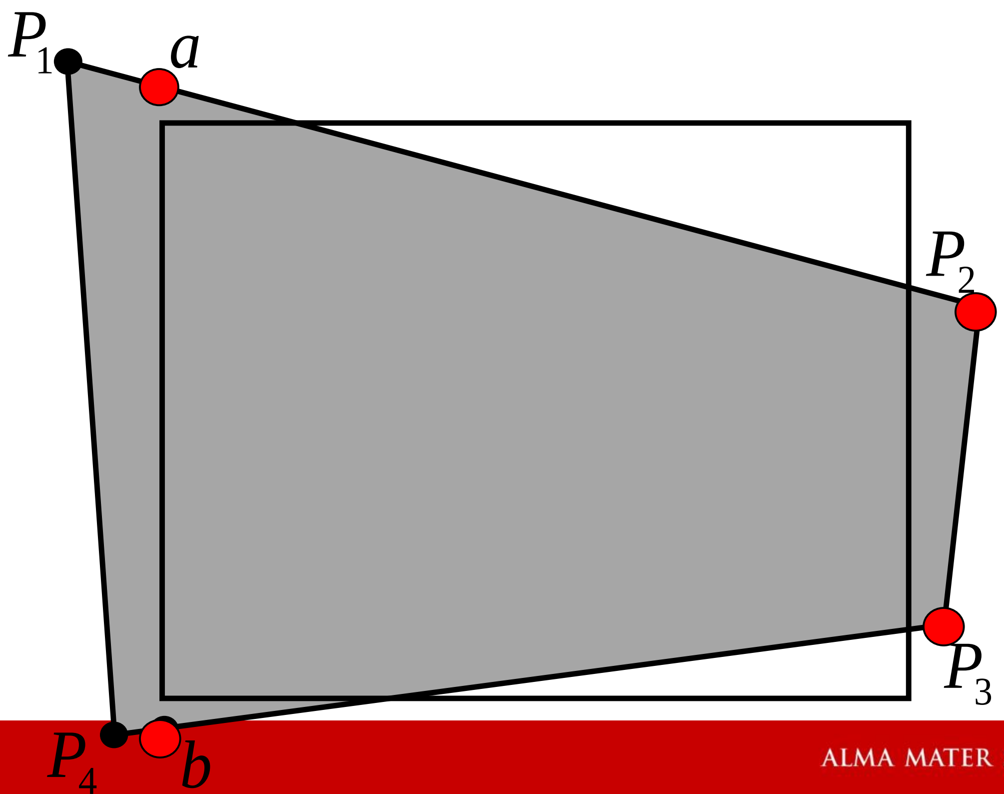
Sutherland-Hodgman Algorithm

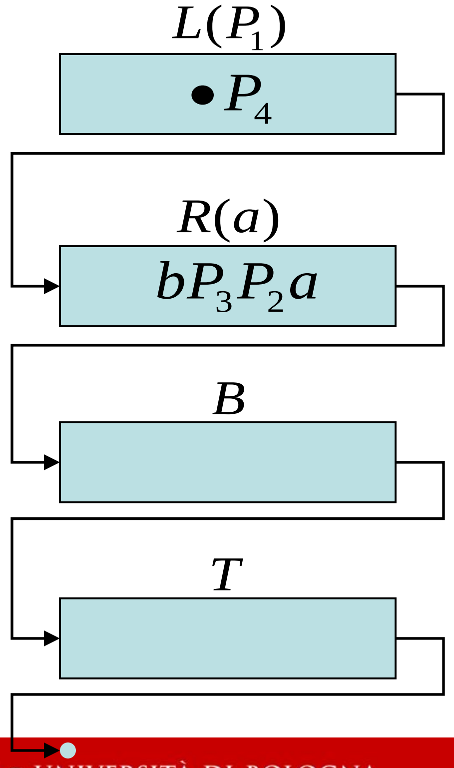




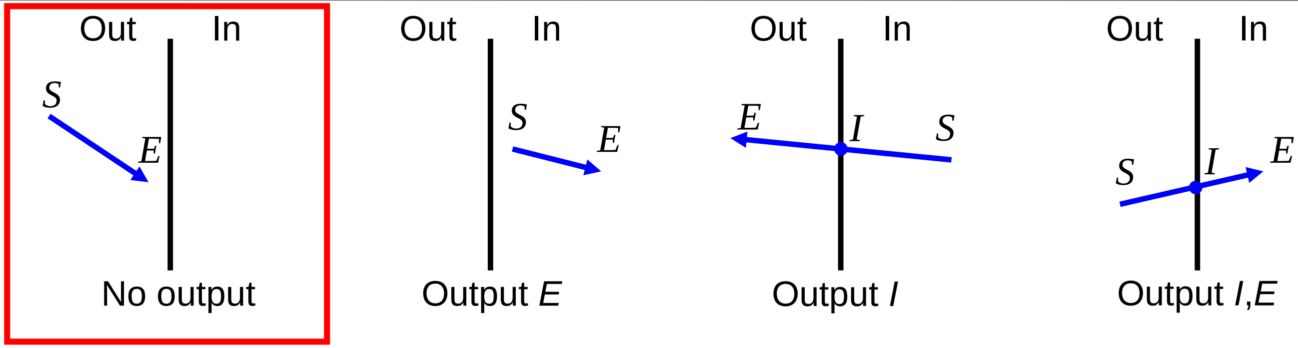
Sutherland-Hodgman Algorithm

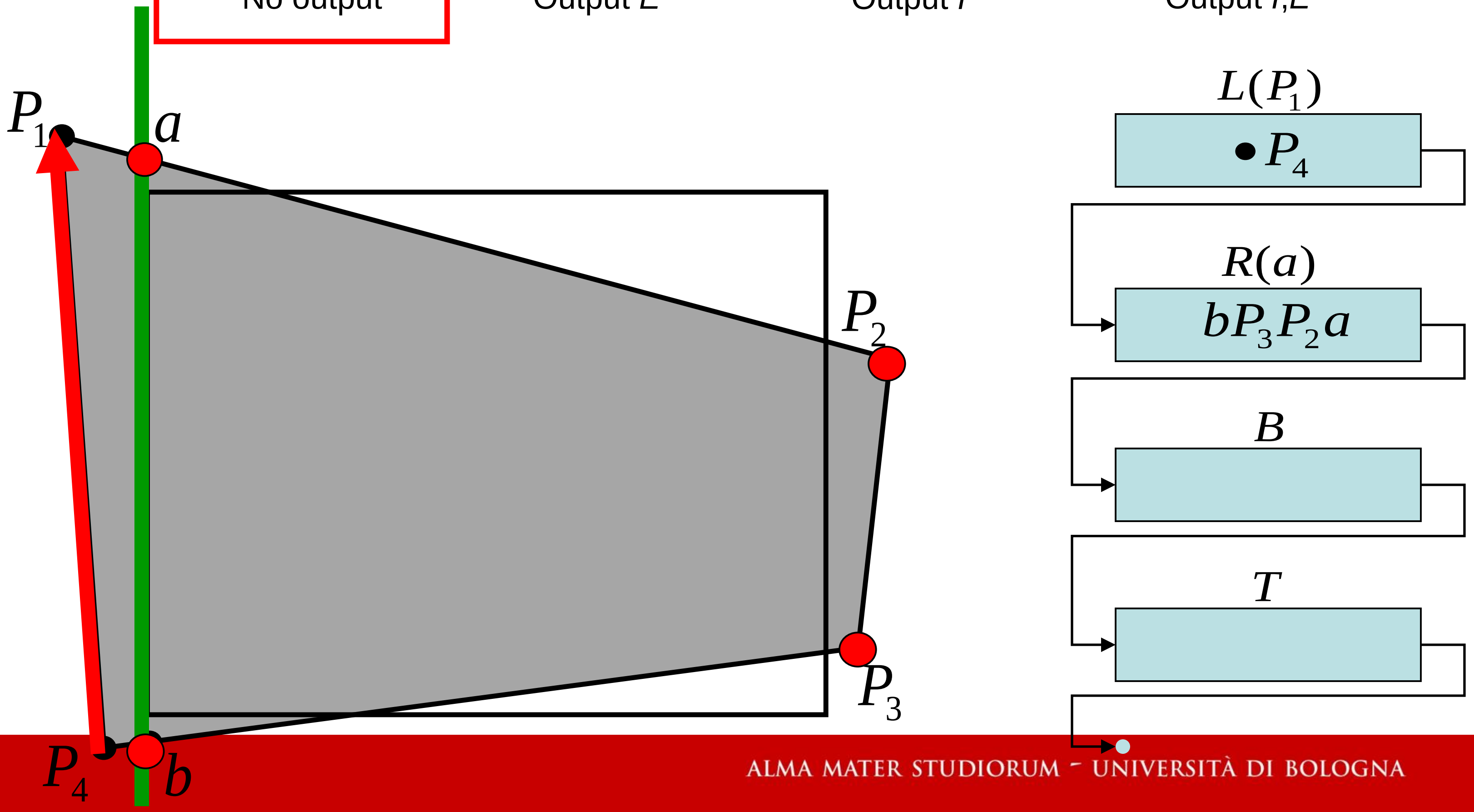




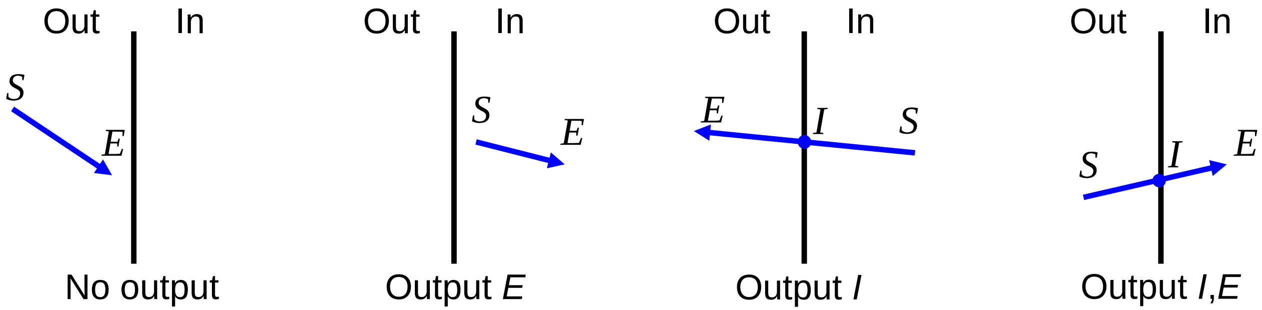


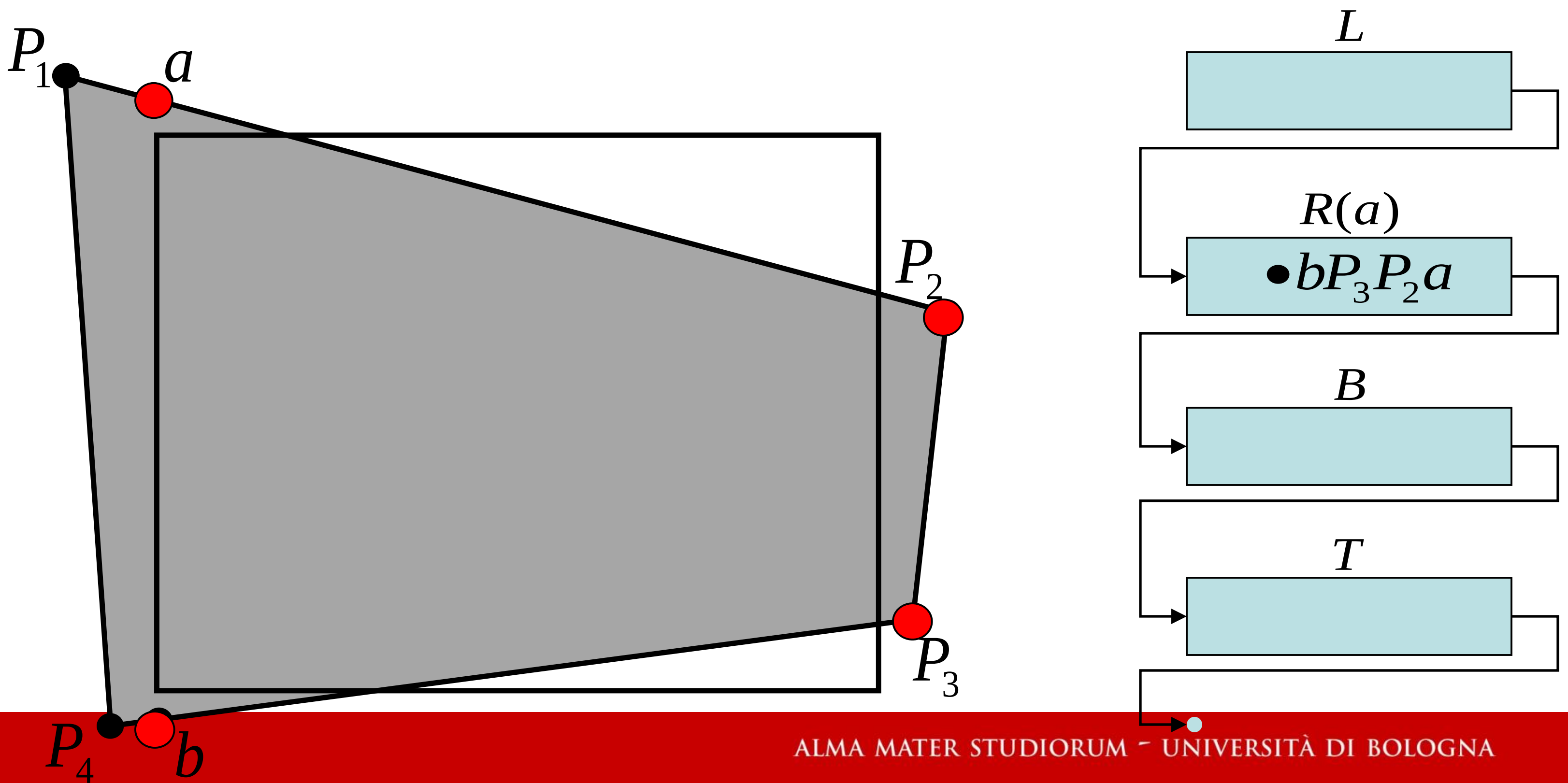
Sutherland-Hodgman Algorithm



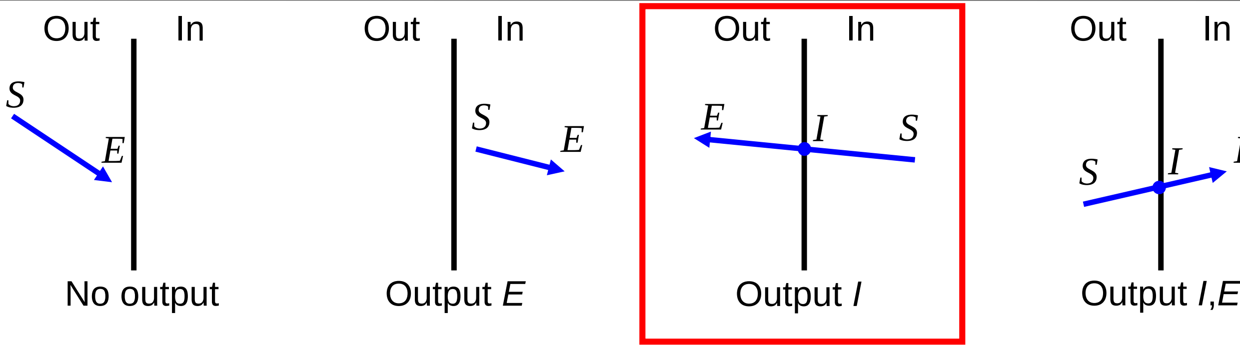


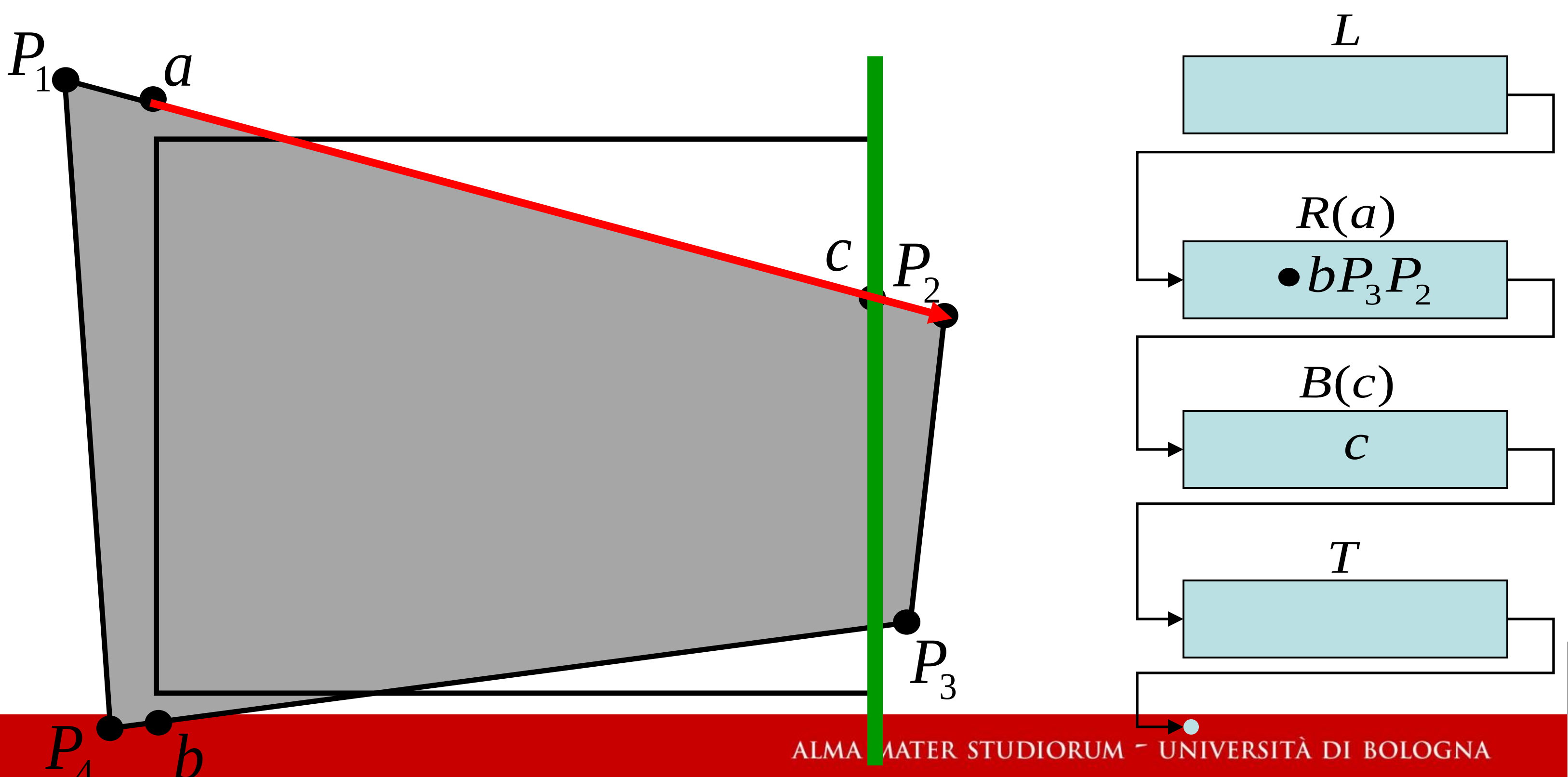
Sutherland-Hodgman Algorithm



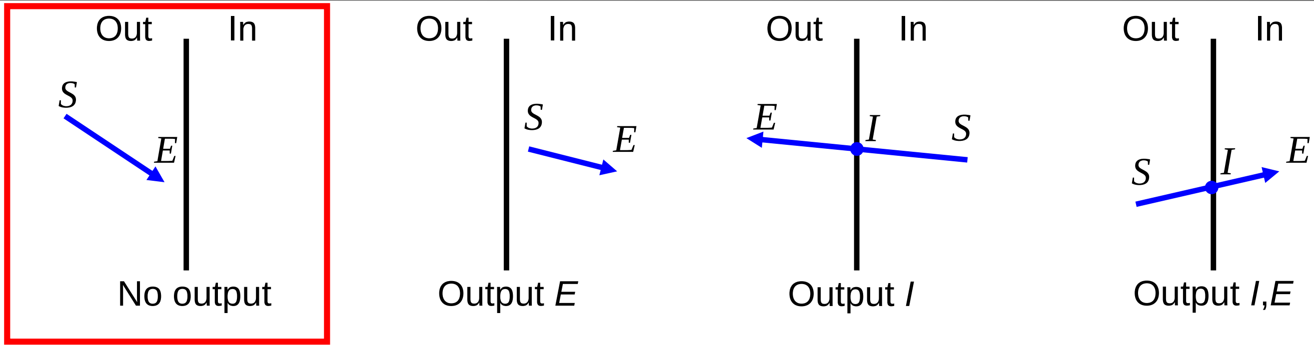


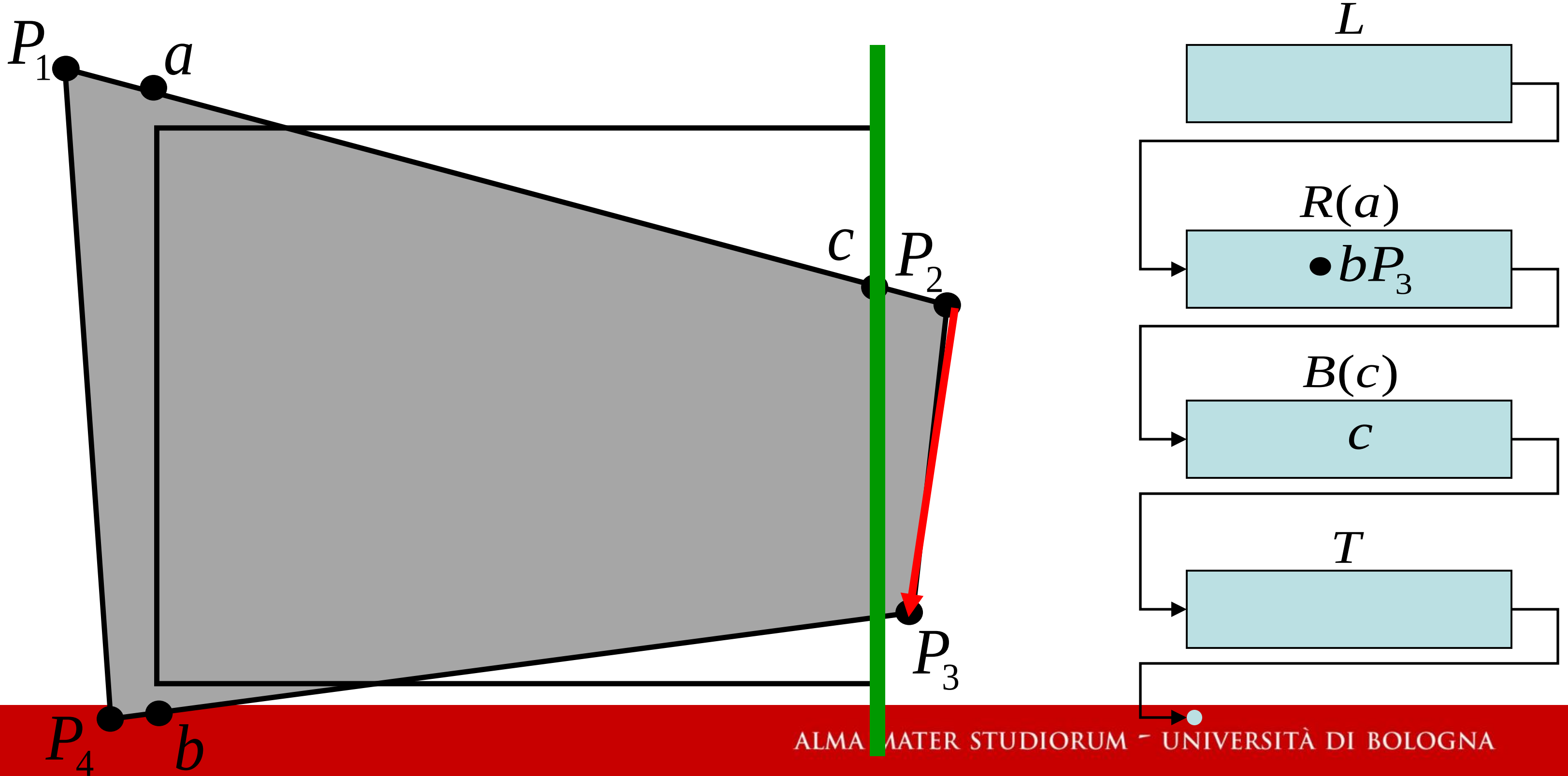
Sutherland-Hodgman Algorithm



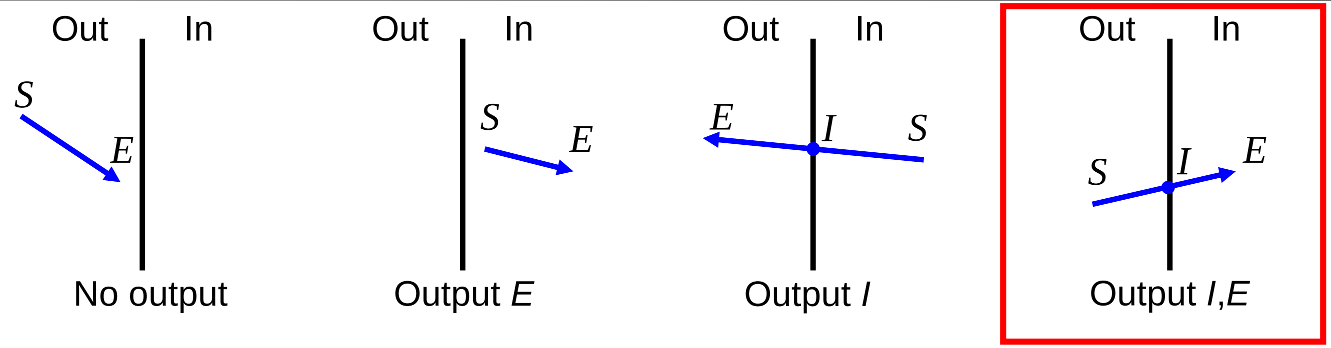


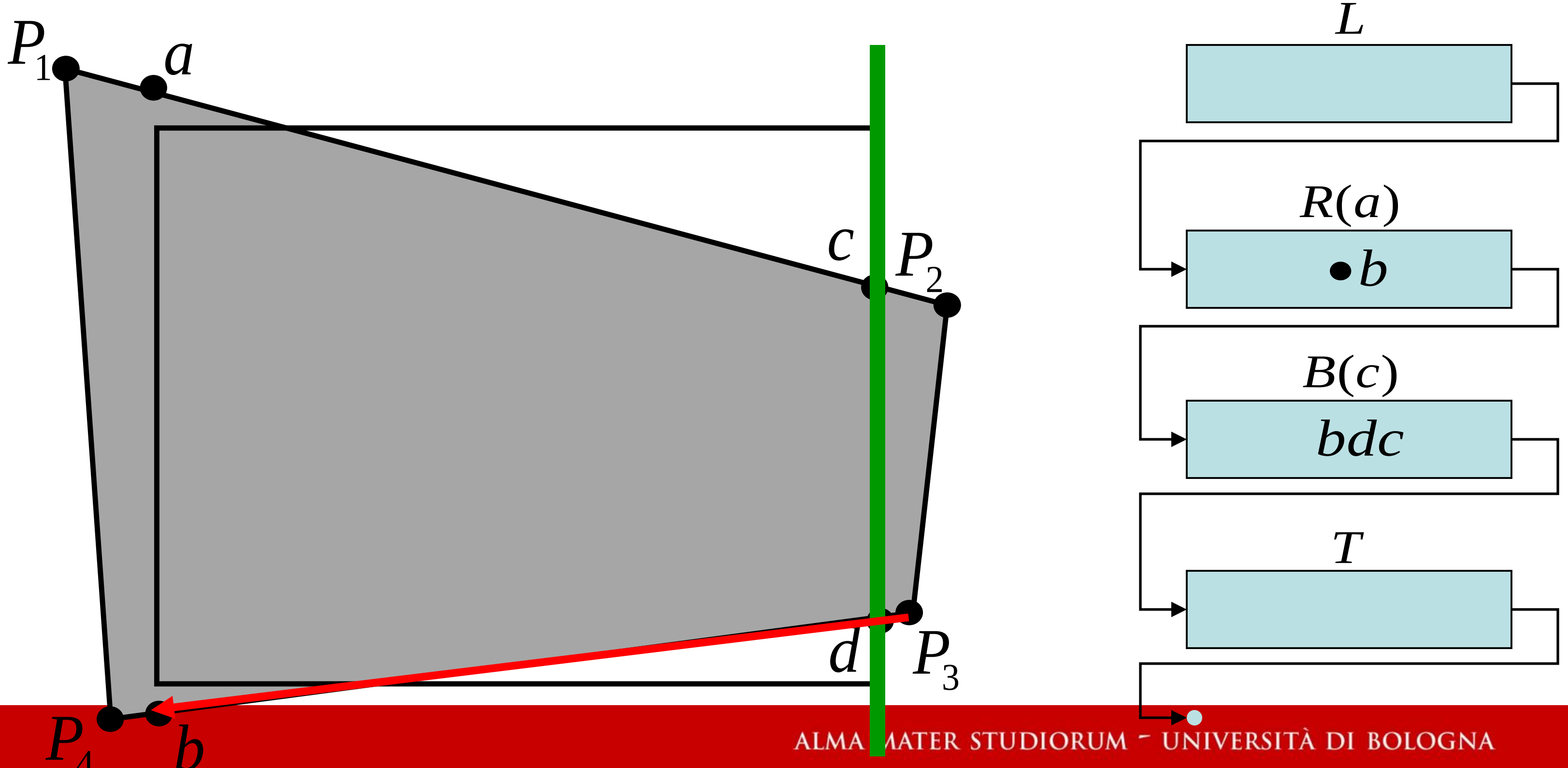
Sutherland-Hodgman Algorithm



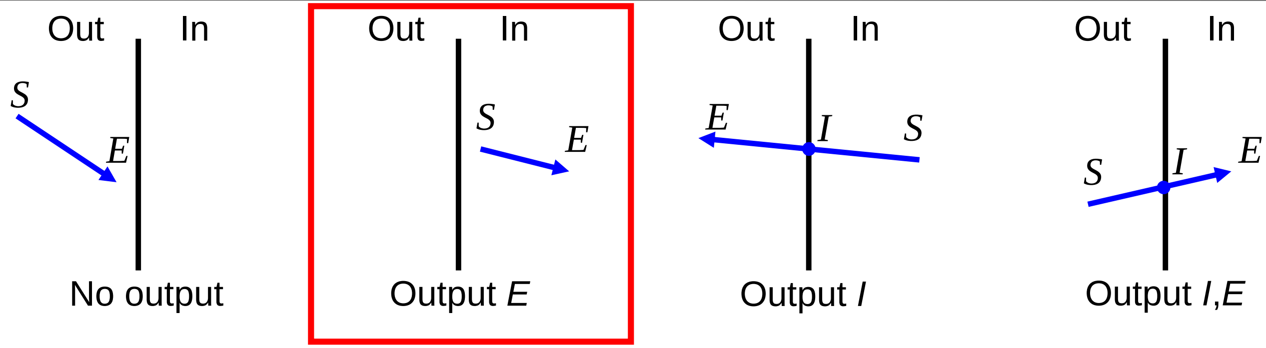


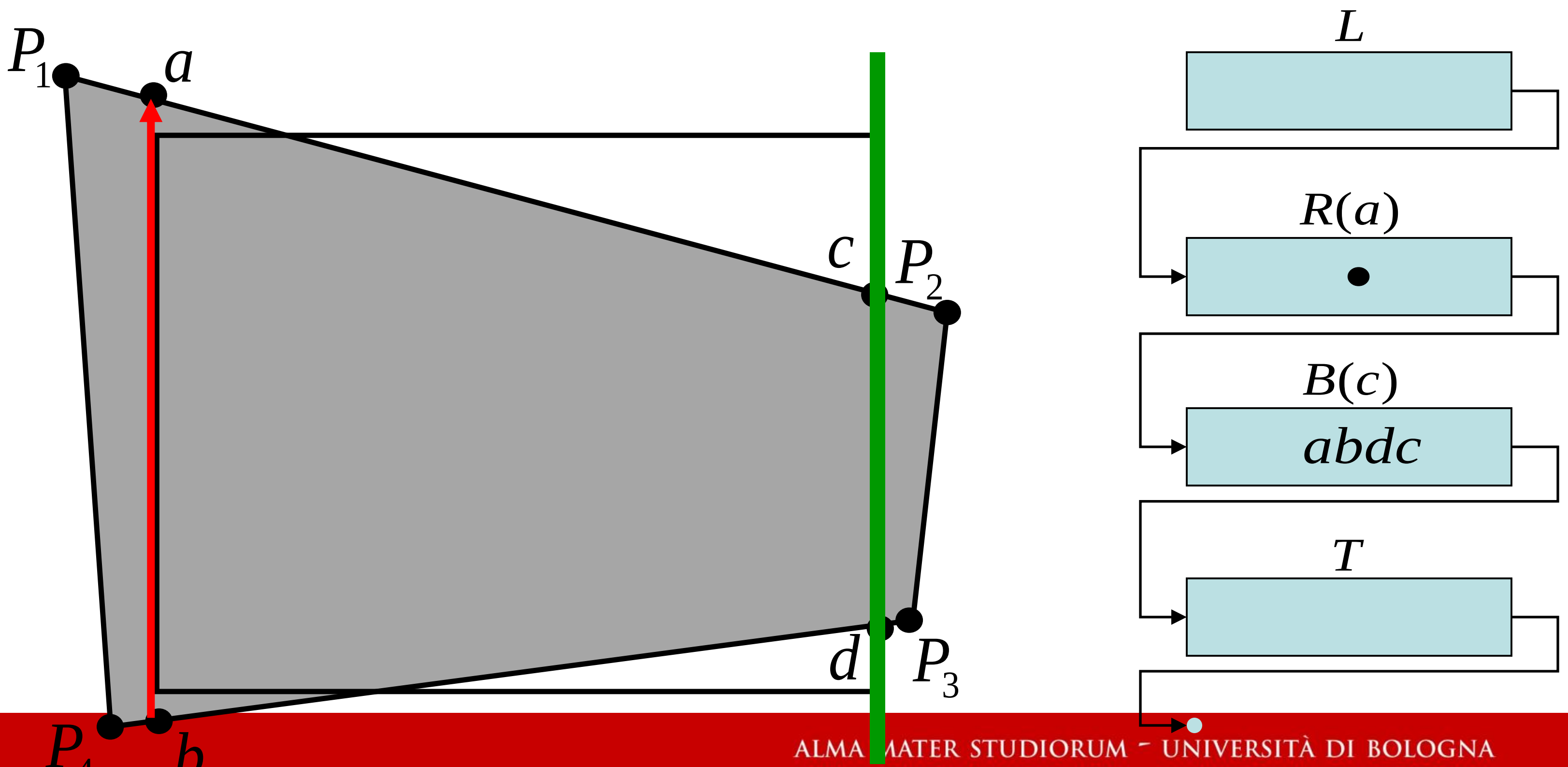
Sutherland-Hodgman Algorithm



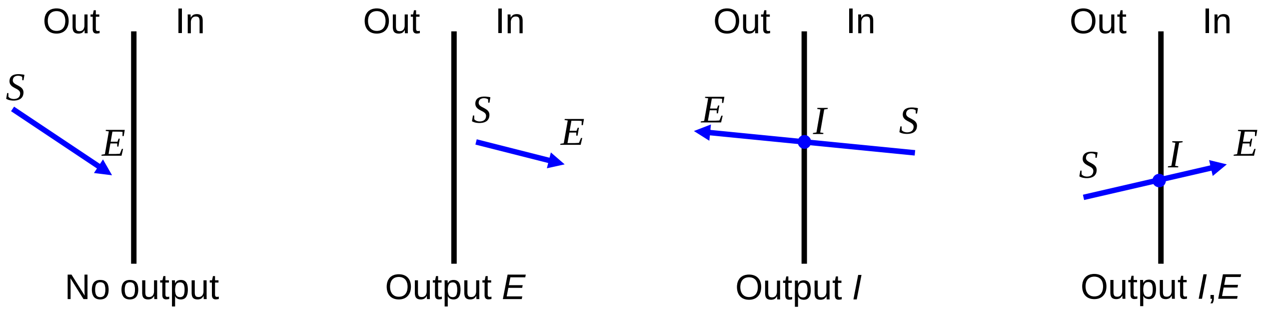


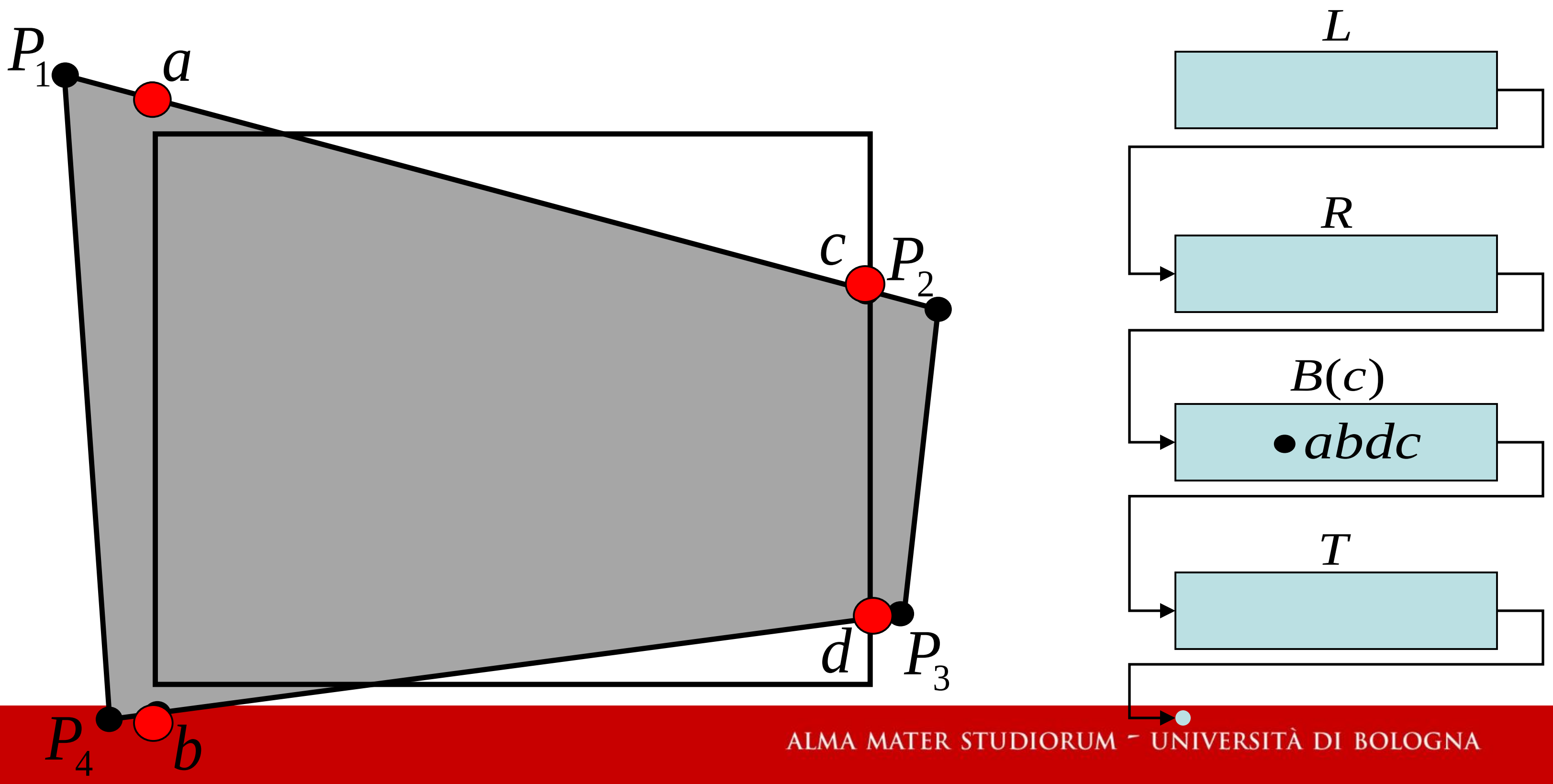
Sutherland-Hodgman Algorithm



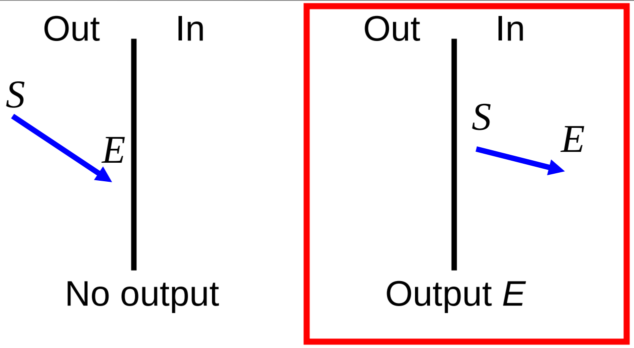


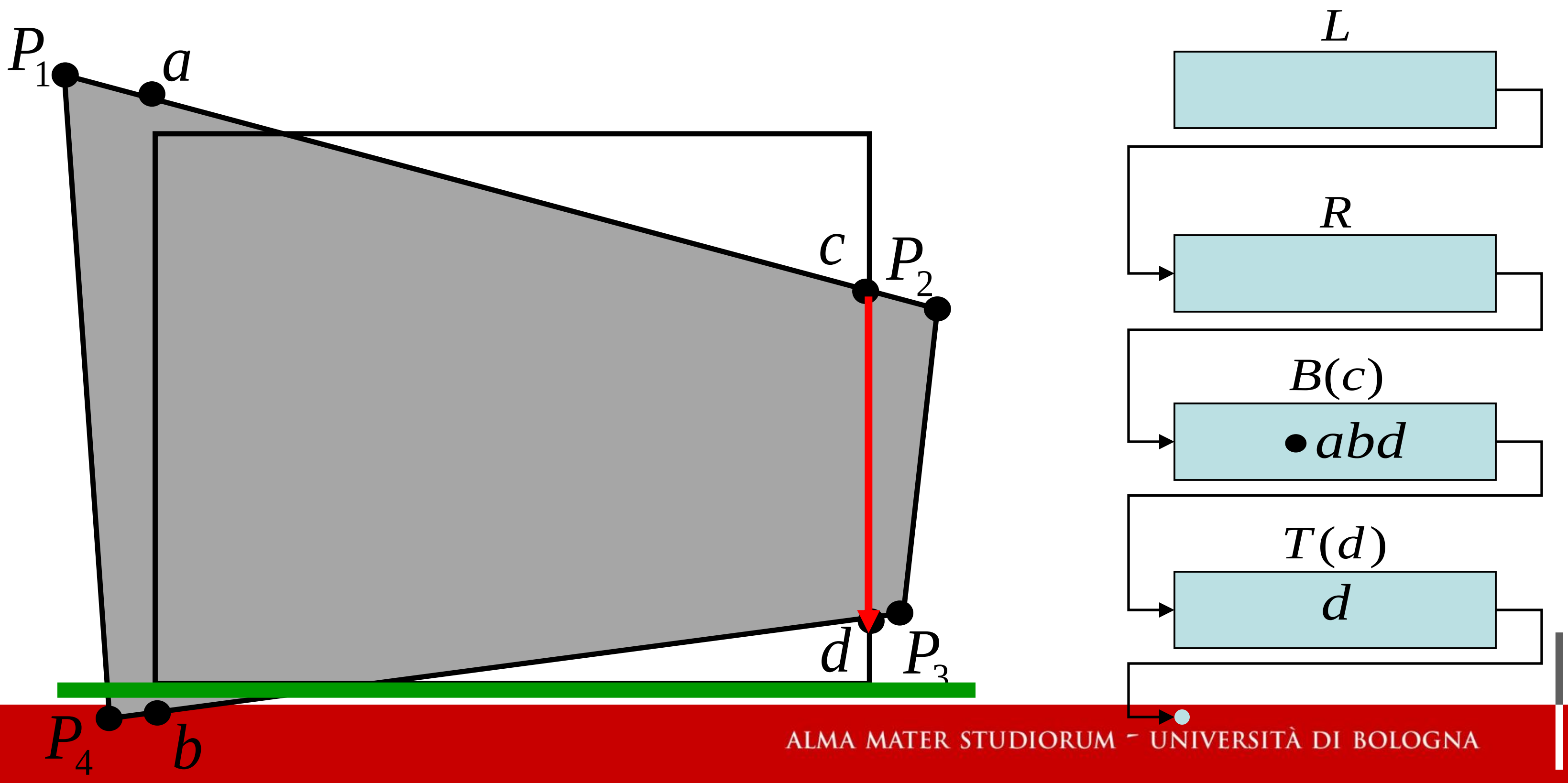
Sutherland-Hodgman Algorithm



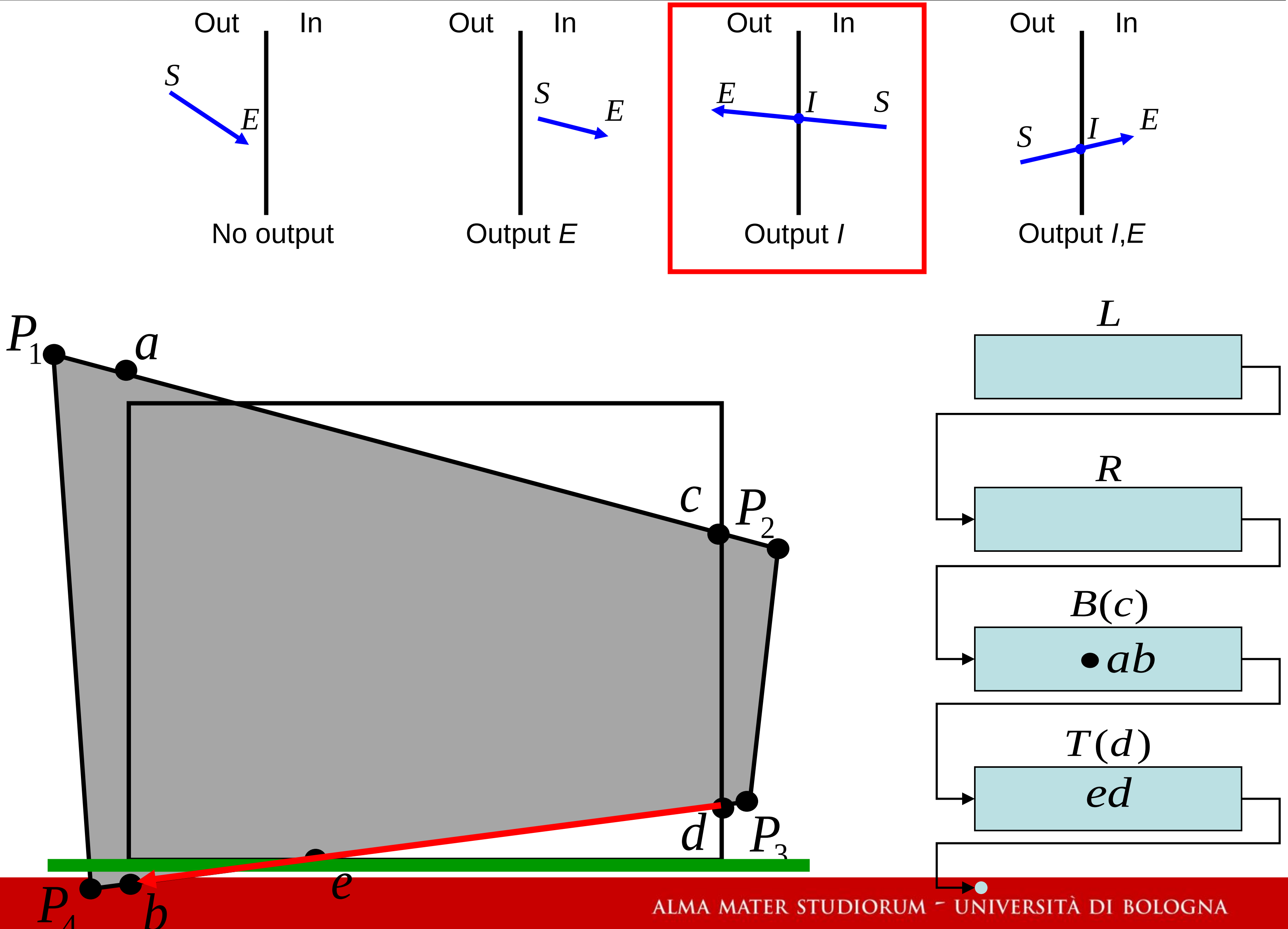


Sutherland-Hodgman Algorithm

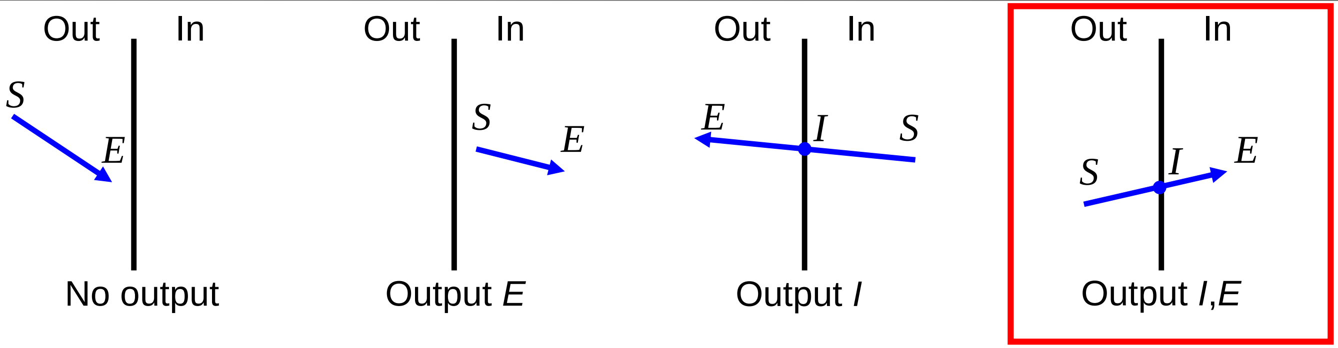


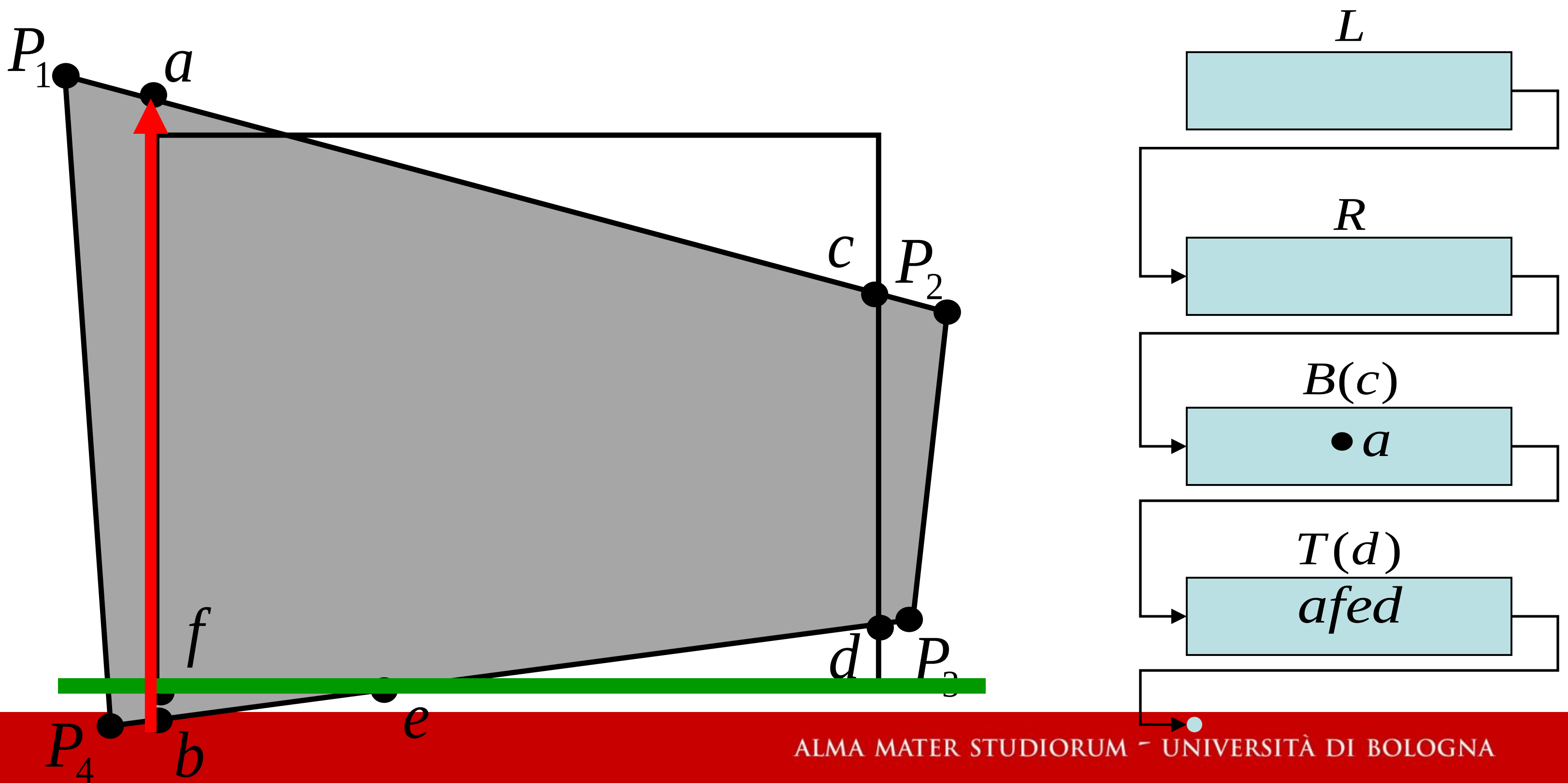


Sutherland-Hodgman Algorithm

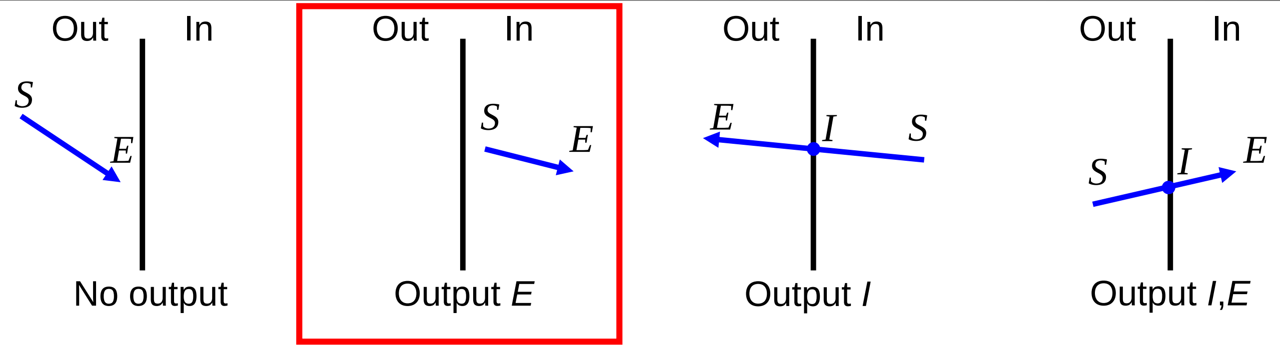


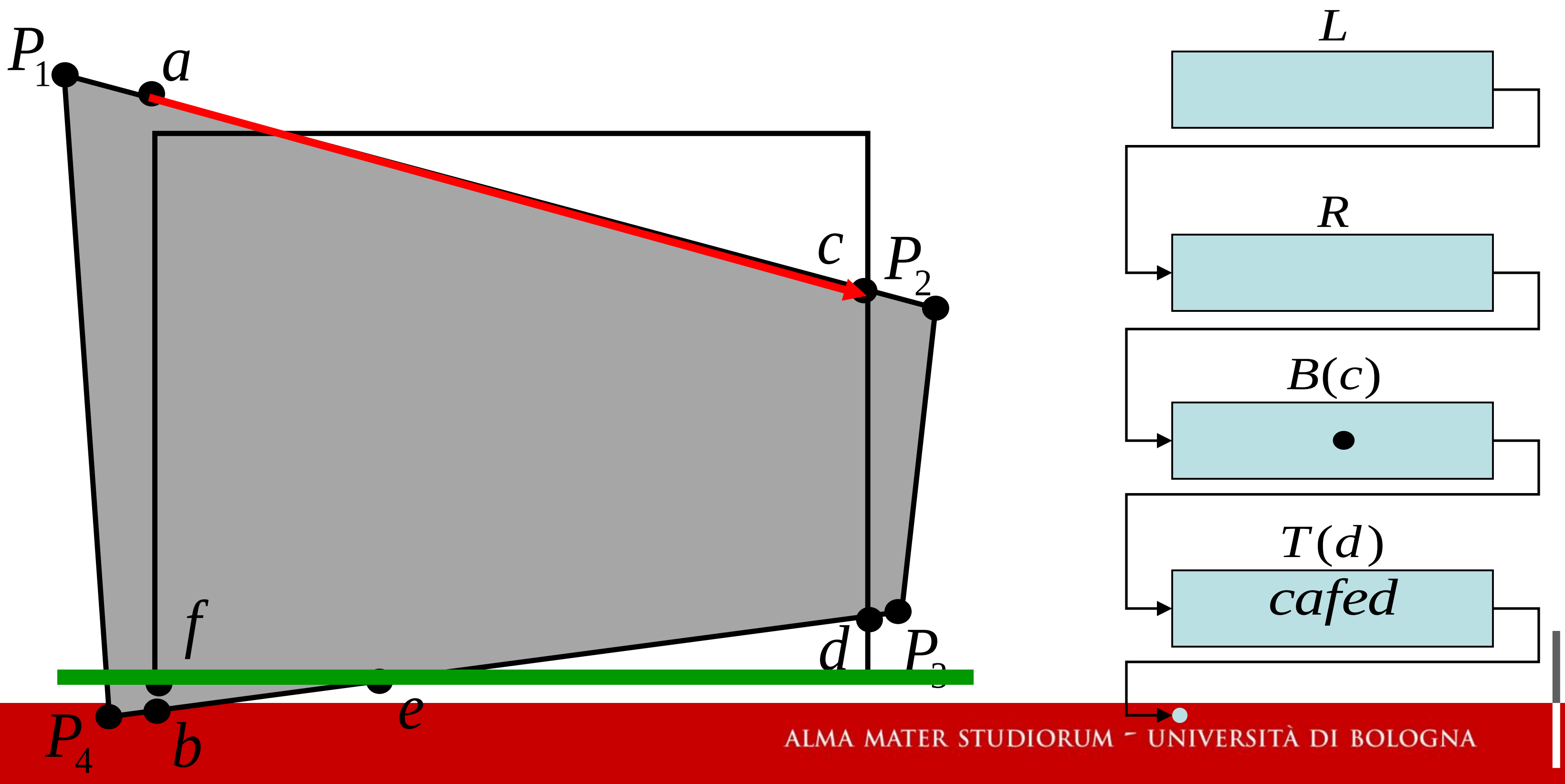
Sutherland-Hodgman Algorithm



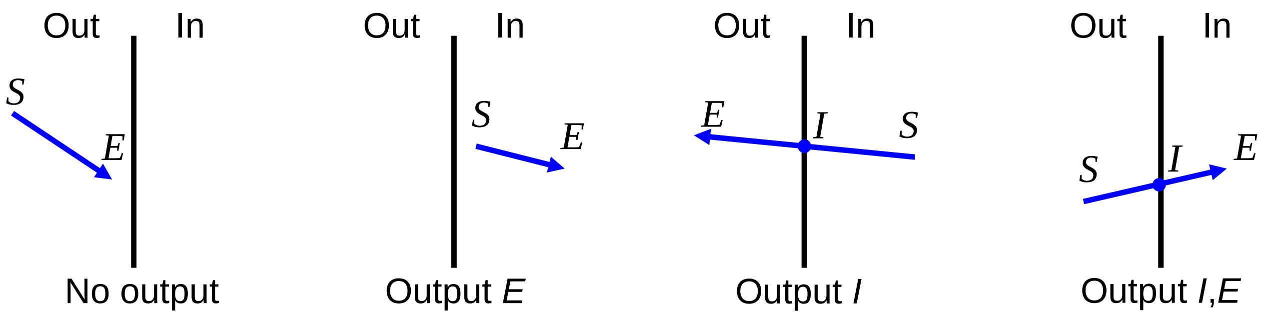


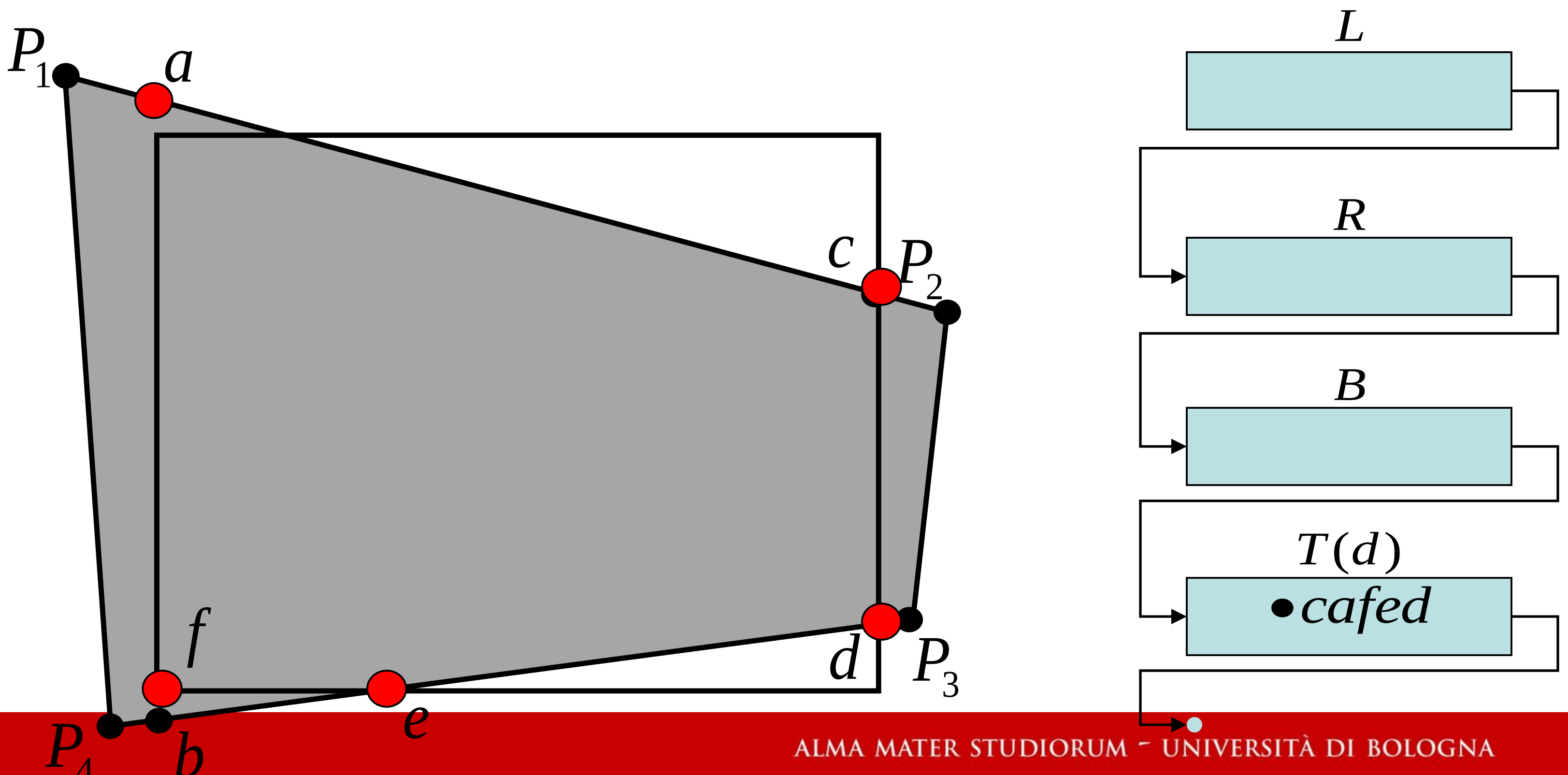
Sutherland-Hodgman Algorithm



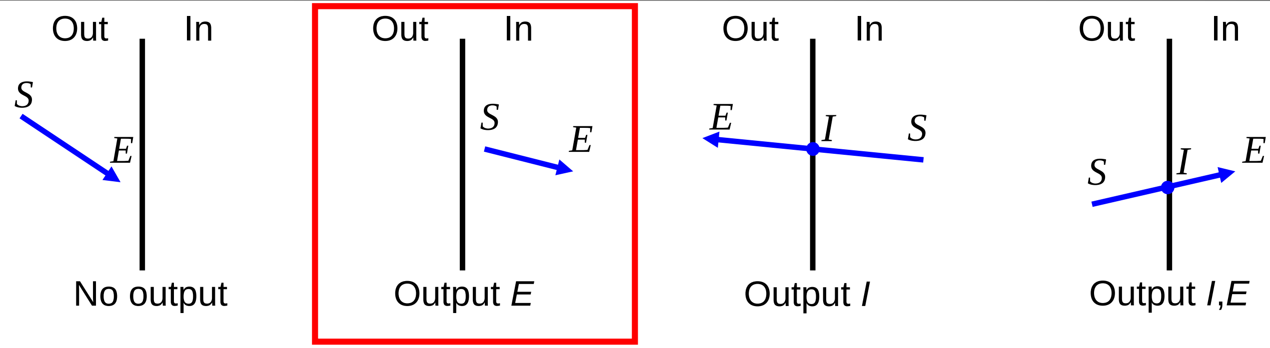


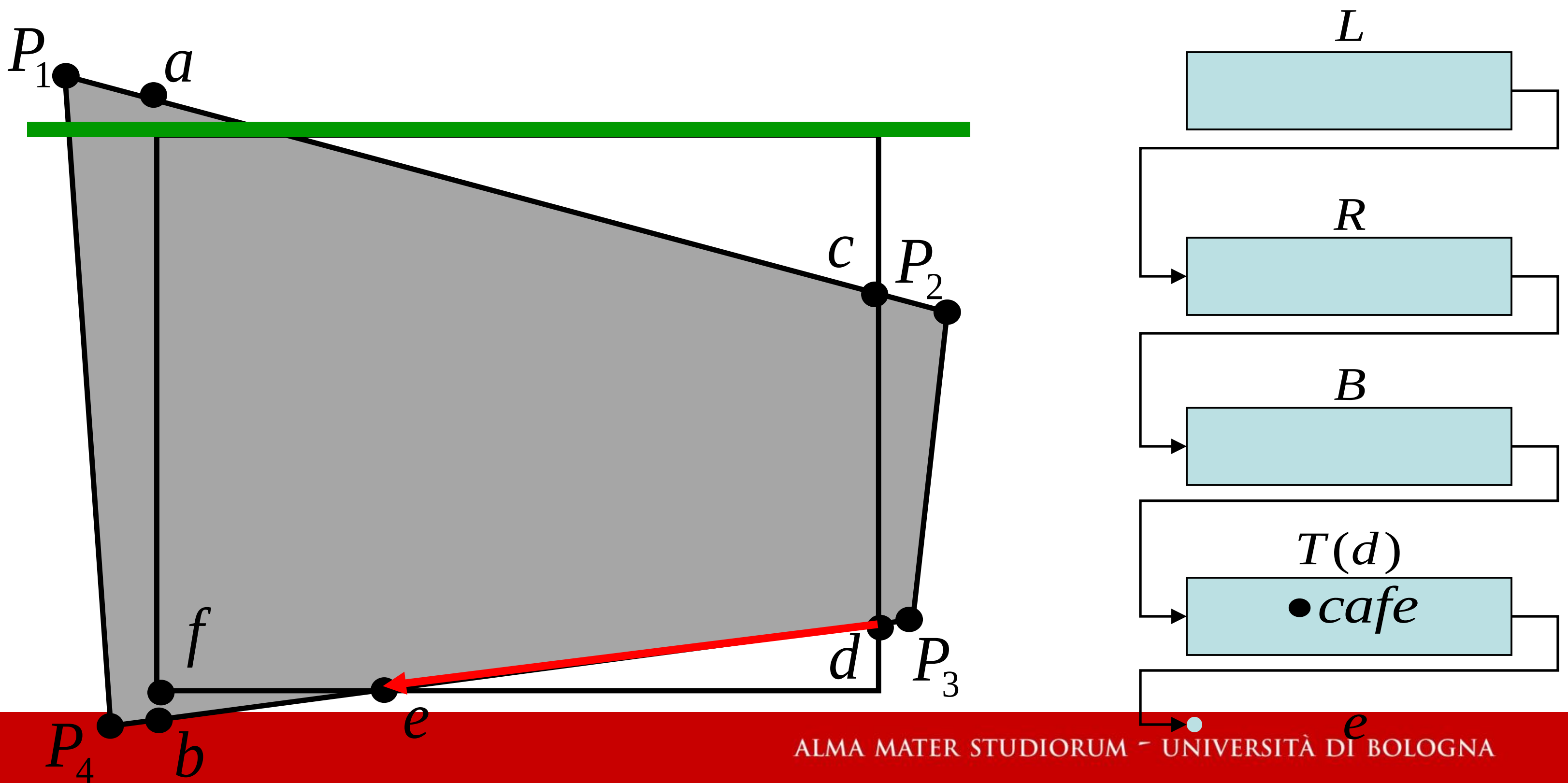
Sutherland-Hodgman Algorithm



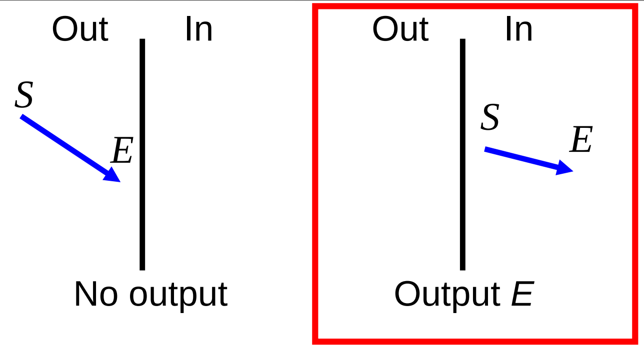


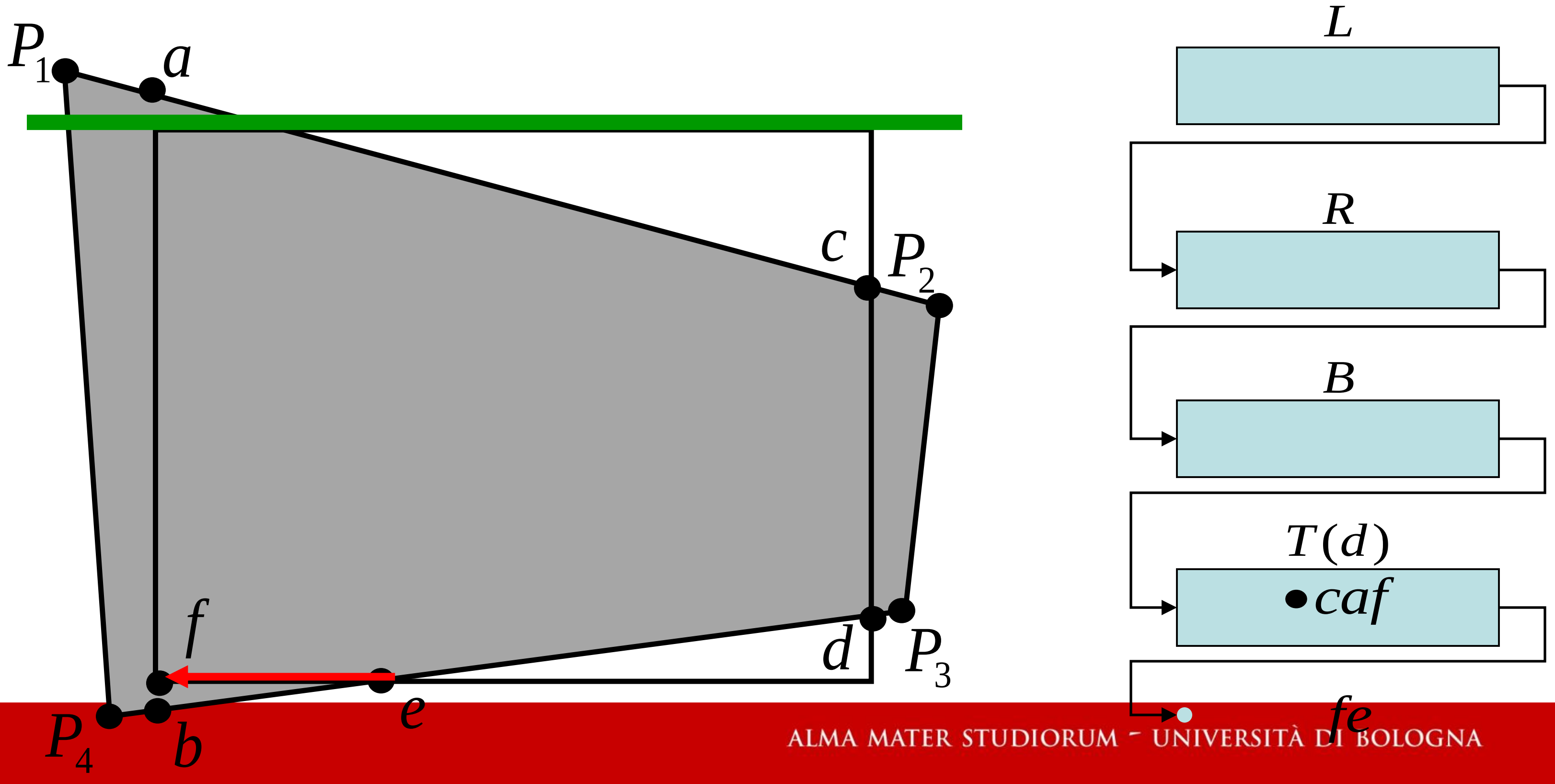
Sutherland-Hodgman Algorithm



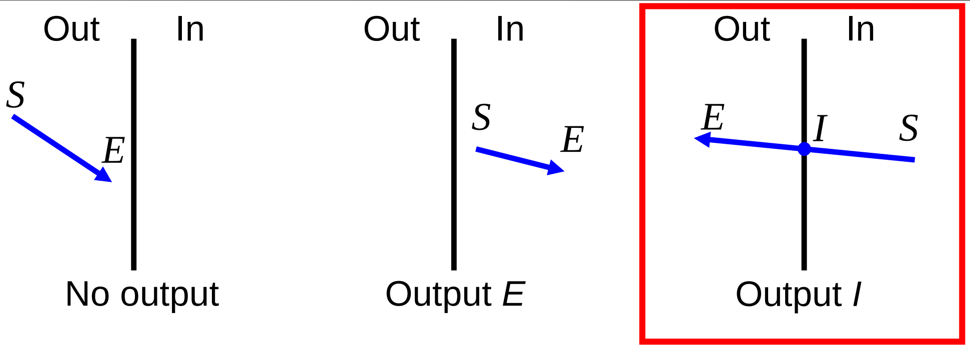


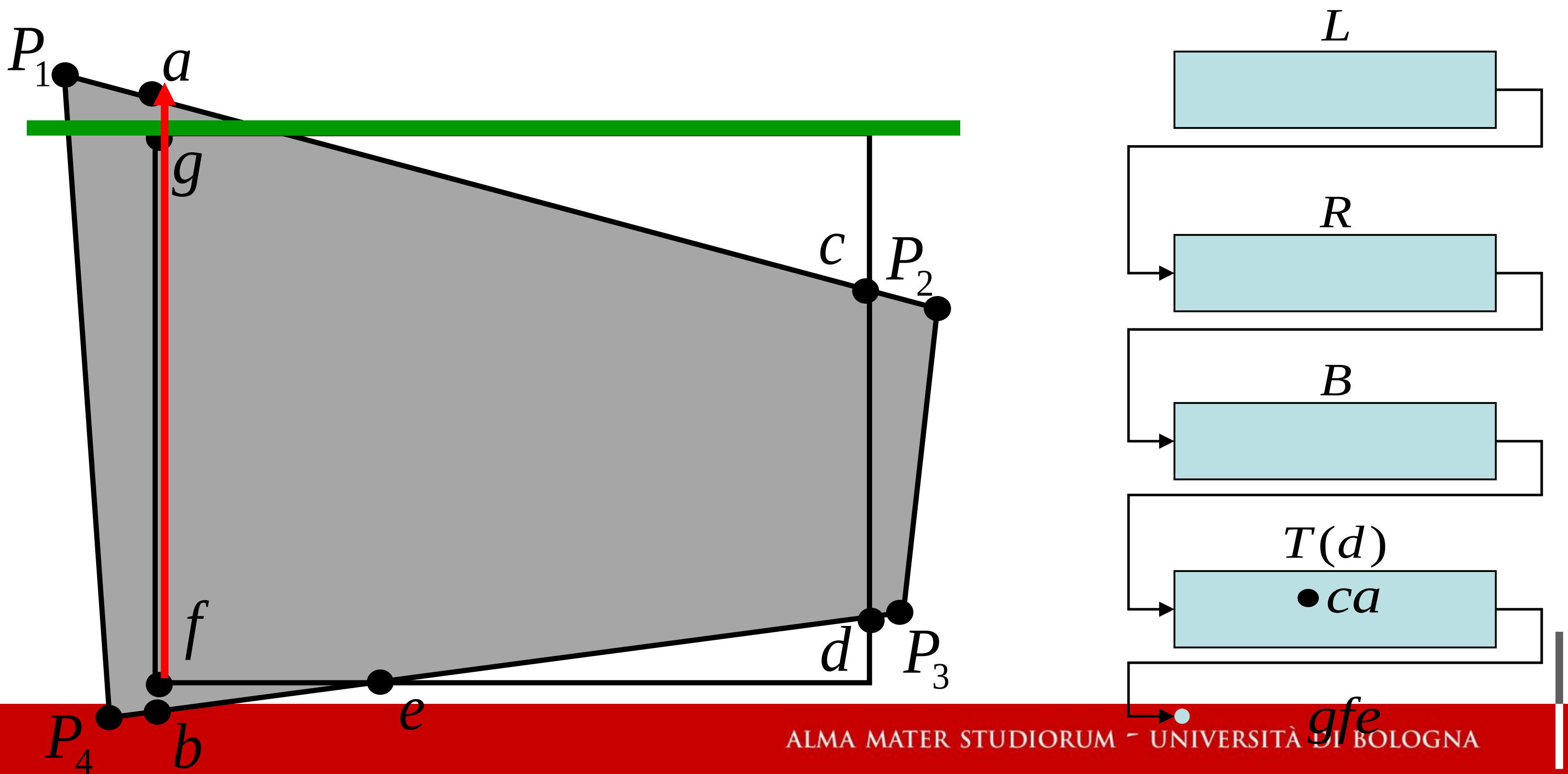
Sutherland-Hodgman Algorithm



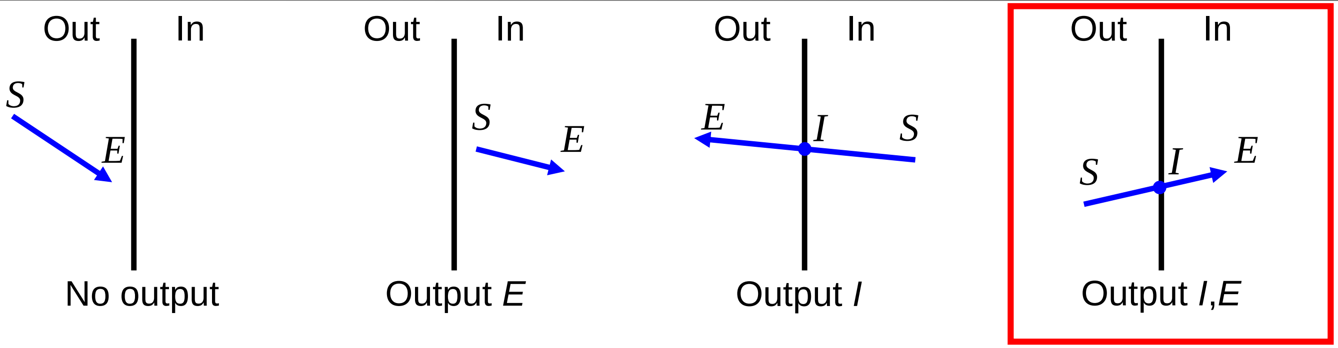


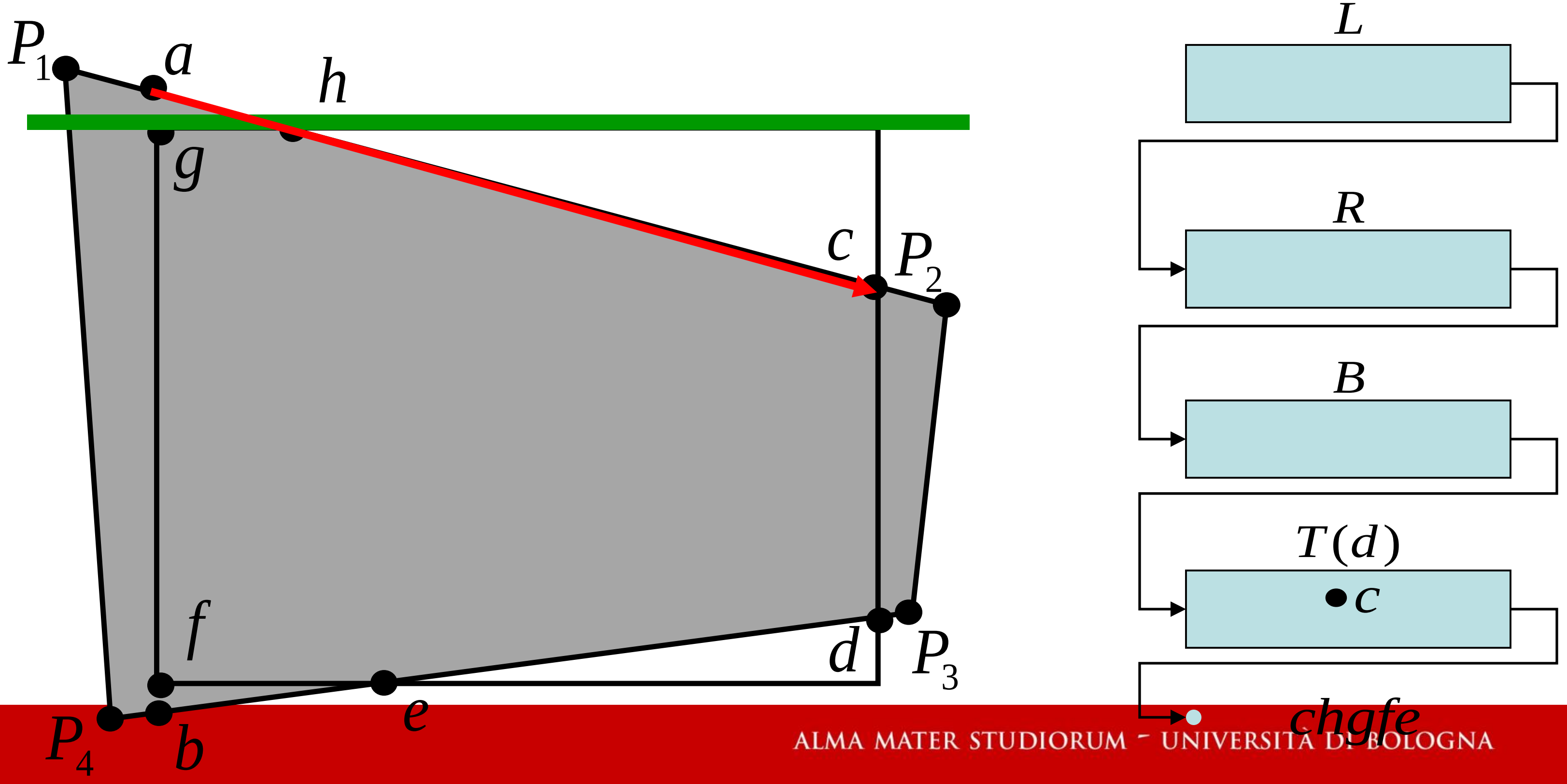
Sutherland-Hodgman Algorithm



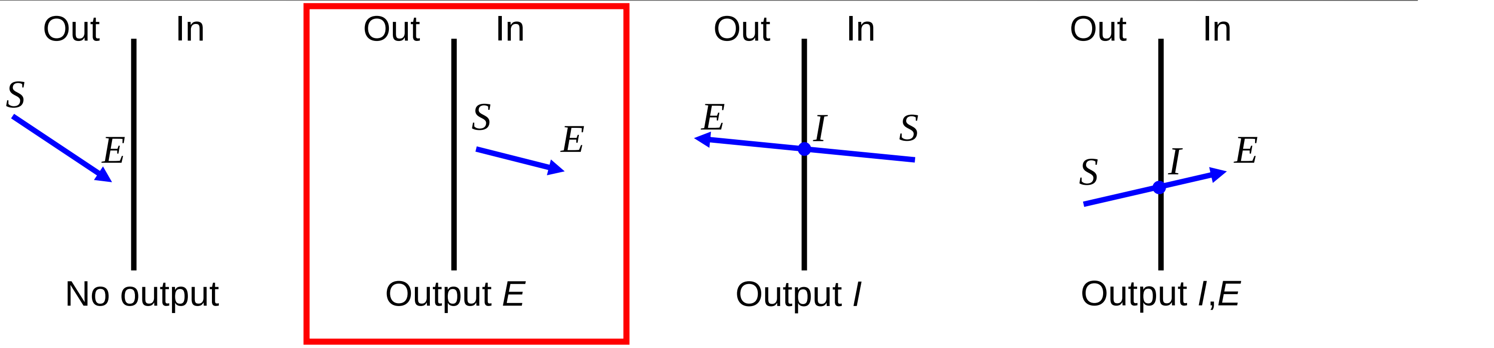


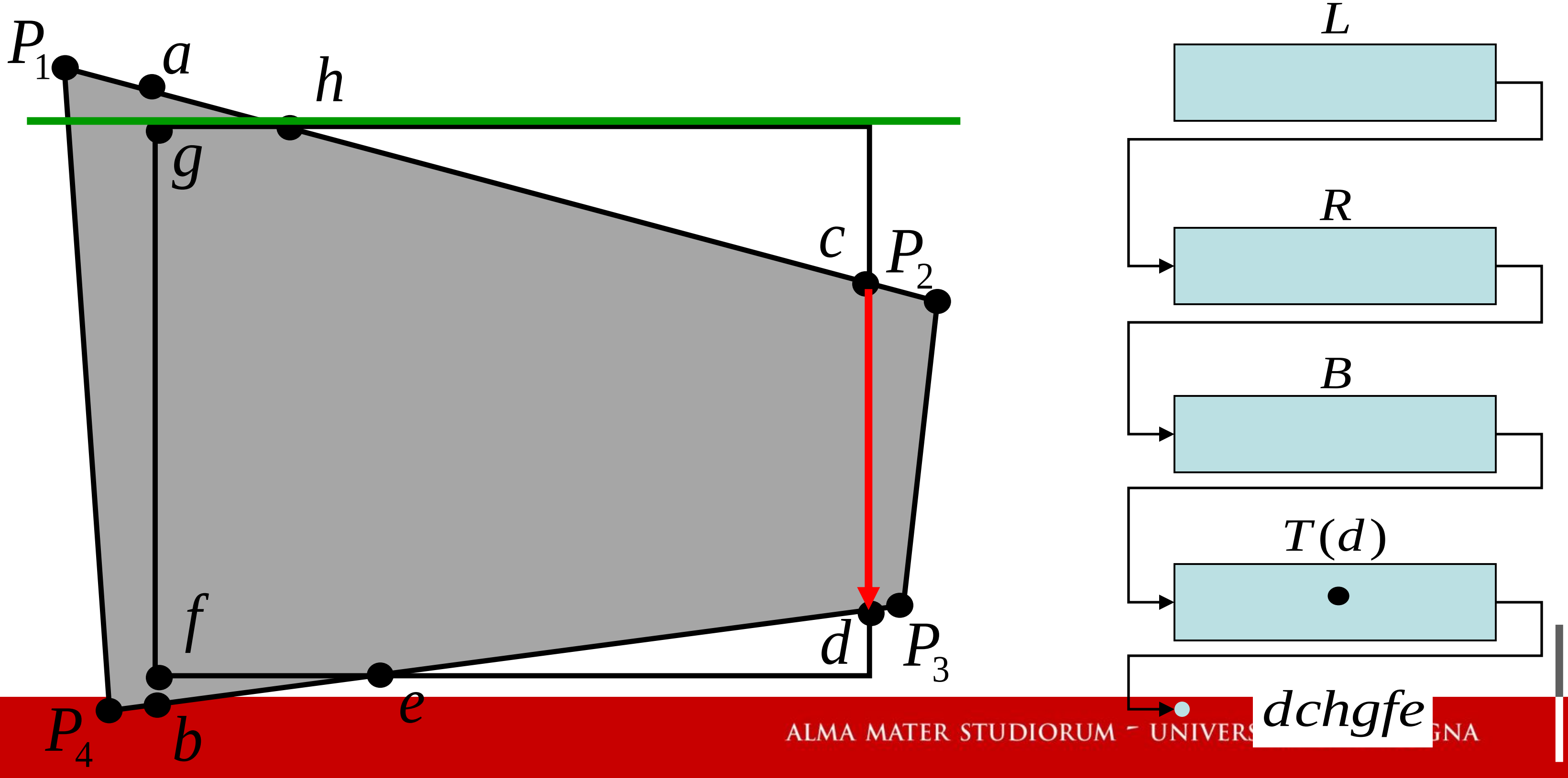
Sutherland-Hodgman Algorithm



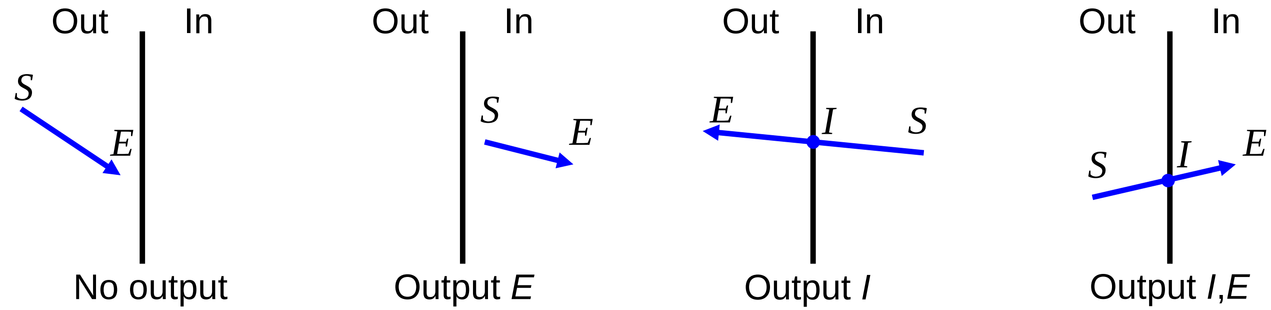


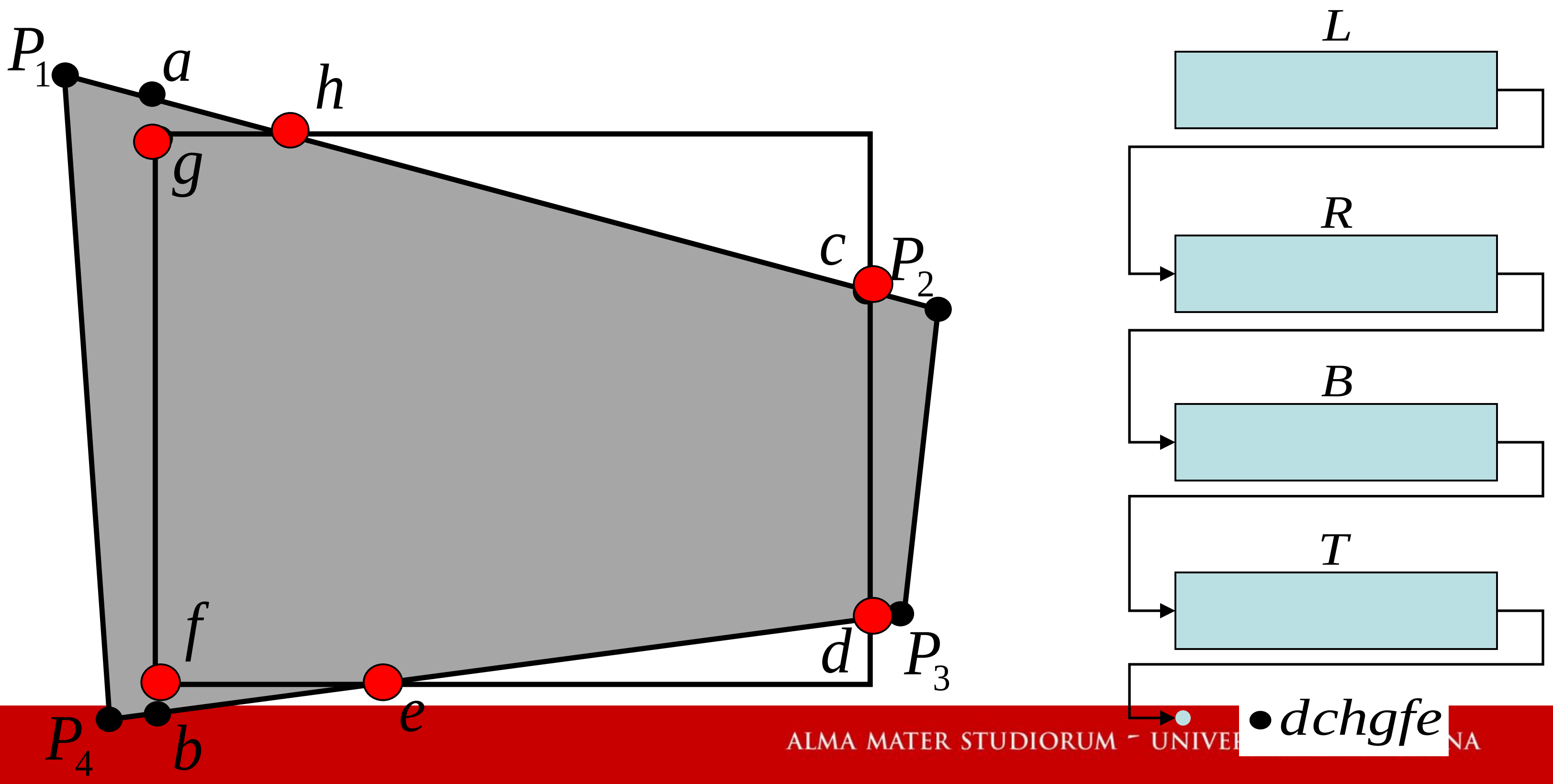
Sutherland-Hodgman Algorithm



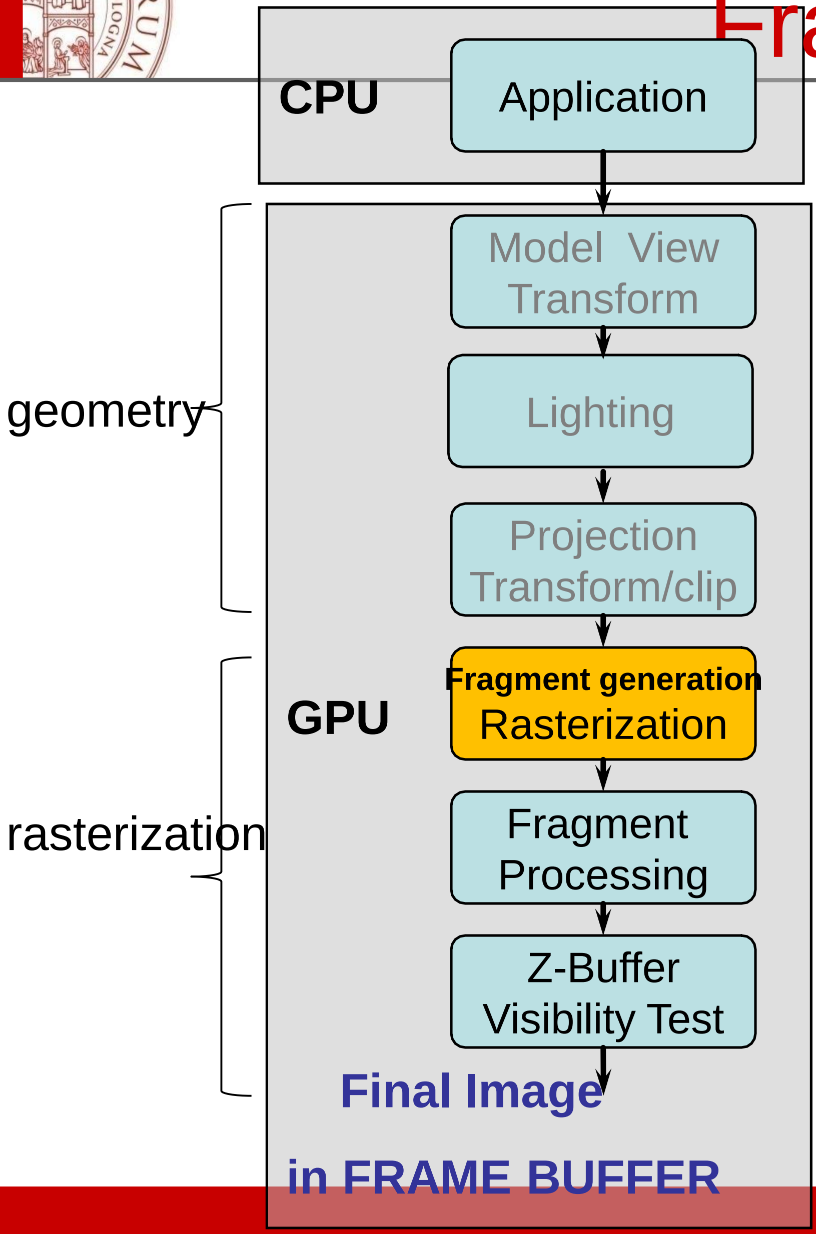


Sutherland-Hodgman Algorithm





Rasterization (Scan Conversion Fragment generation)



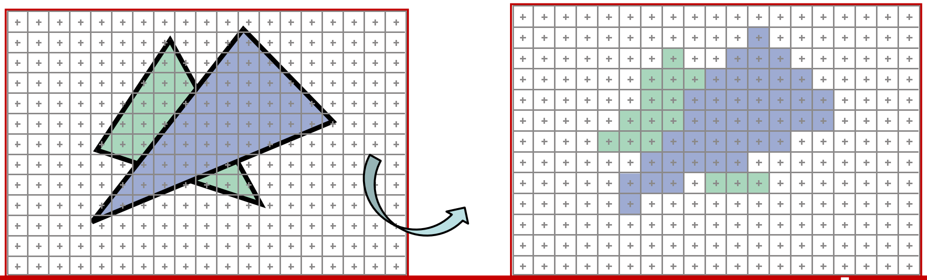
Vertices

- Determine which pixels that are inside primitive specified by a set of vertices

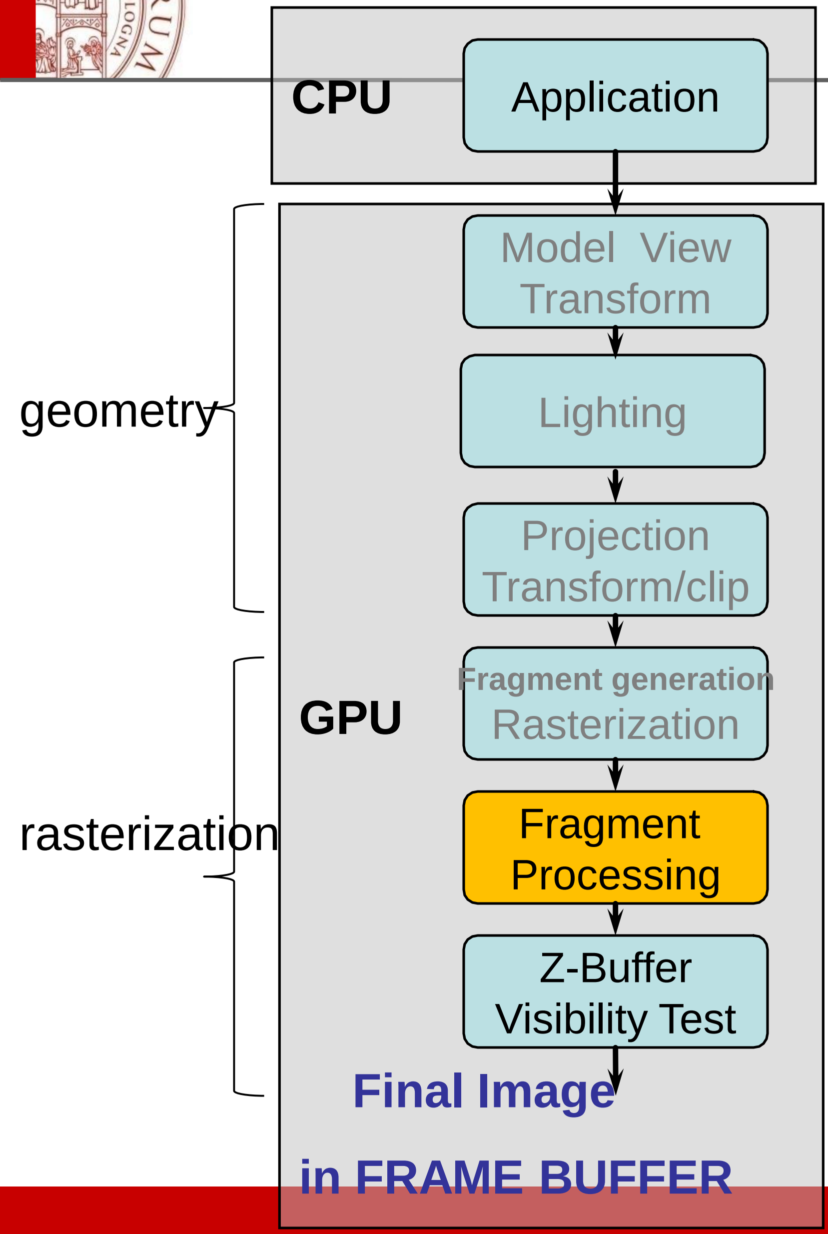
  OUTPUT: Produces a set of fragments

- **Fragments** have a location (pixel location) and other attributes such color and texture coordinates that are determined by interpolating values at vertices

Fragments



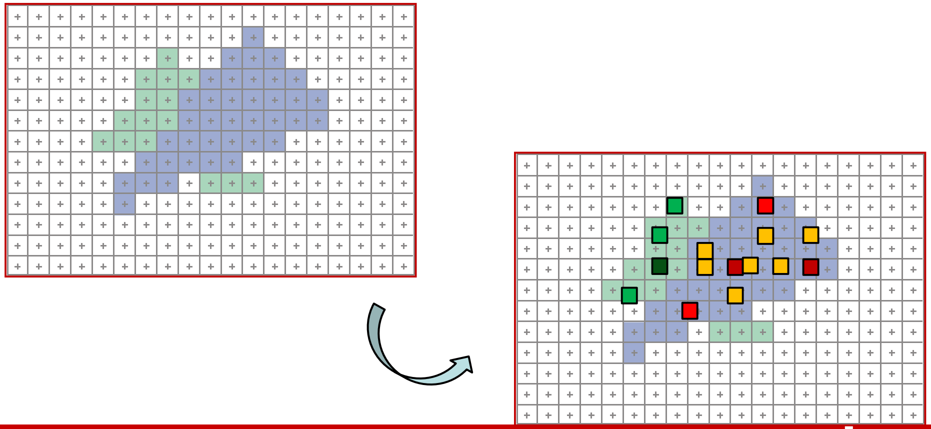
Fragment processing



– **Fragment Processing**

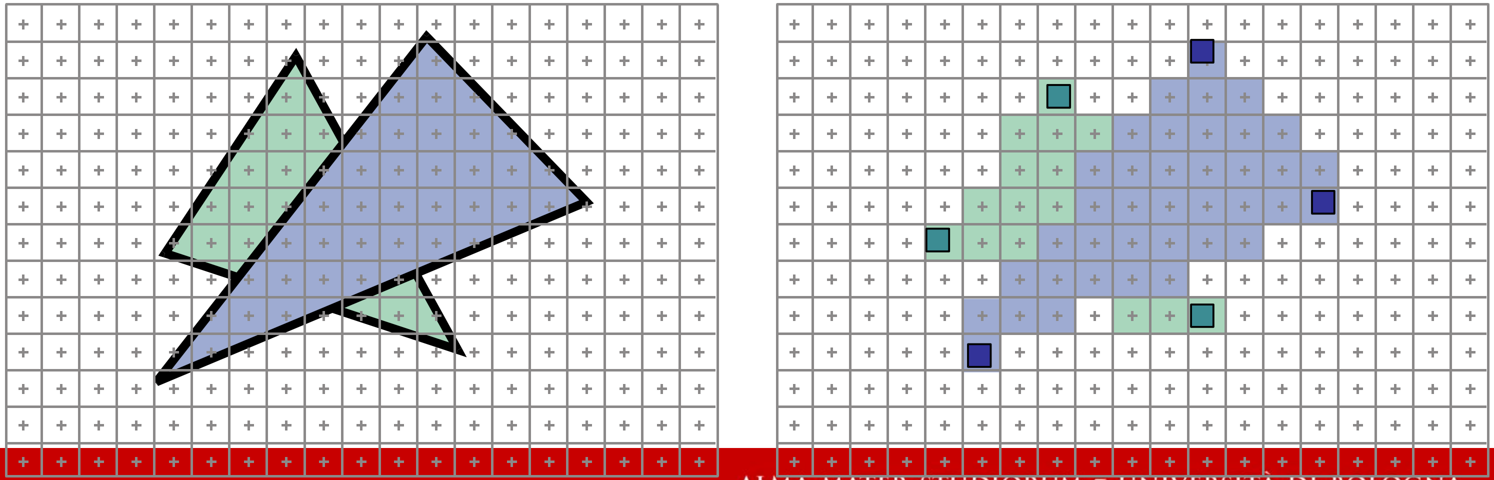
Pixel colors determined later using color, texture, and other vertex properties

(Texture Mapping)



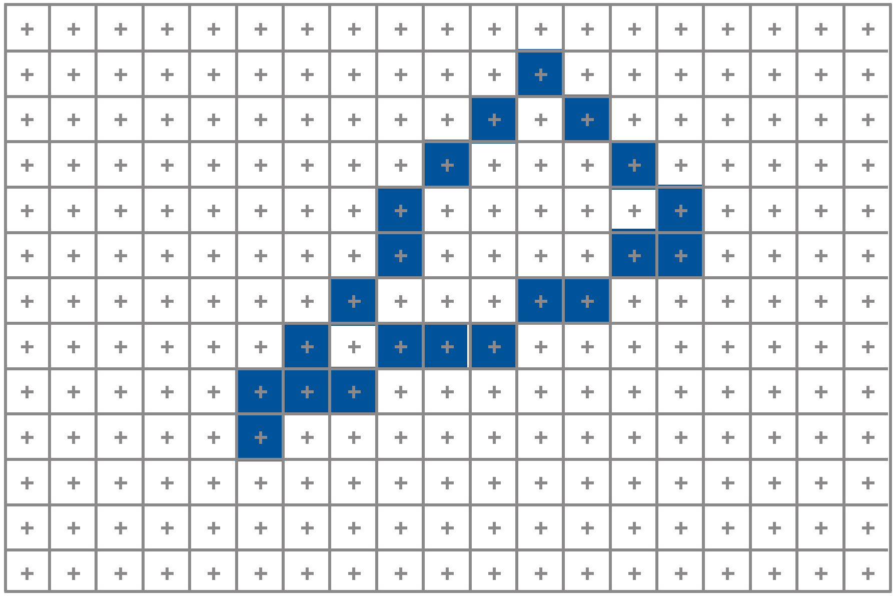
Rasterization

- Geometric primitives
  (point, line, polygon, circle, polyhedron, sphere...)
- Primitives are continuous; screen is discrete
- Rasterization: algorithms for *efficient* generation of the samples occupied by a geometric primitive
  - It enumerates the fragments covered by the primitive
  - It interpolates values, called attributes, across the primitive



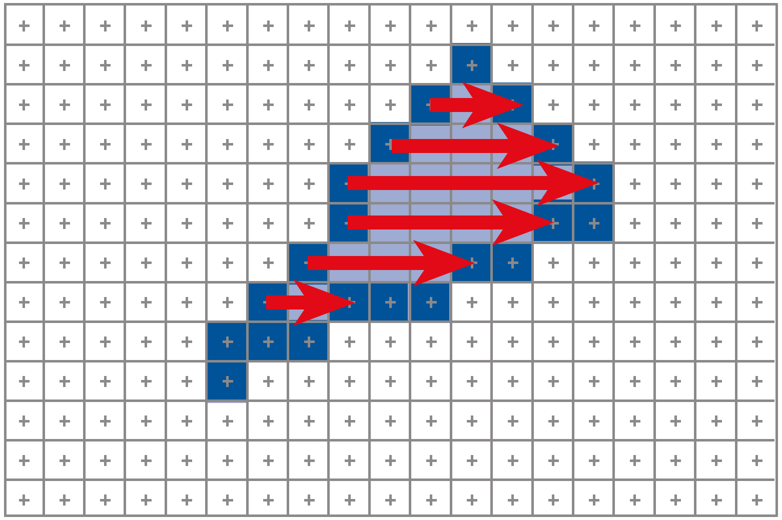
Line rasterization

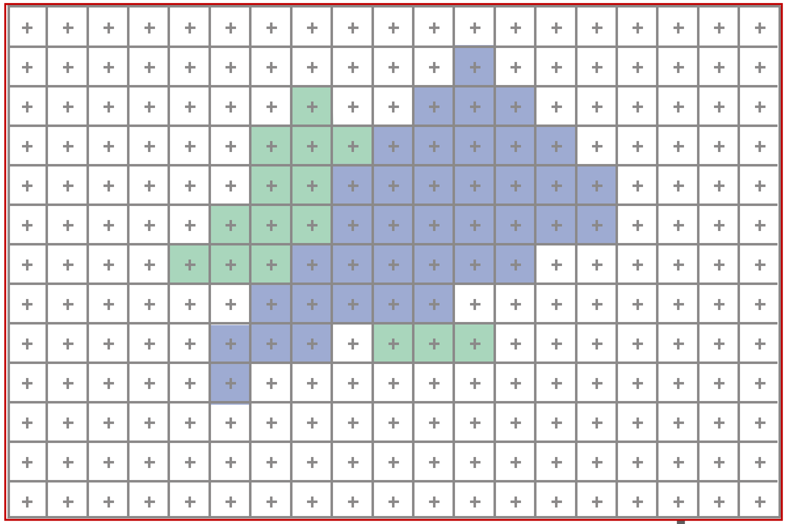
- Compute the boundary pixels



Polygon Rasterization

- Compute the boundary pixels
- Polygon Filling:
  - Fill the spans
  - Flood fill

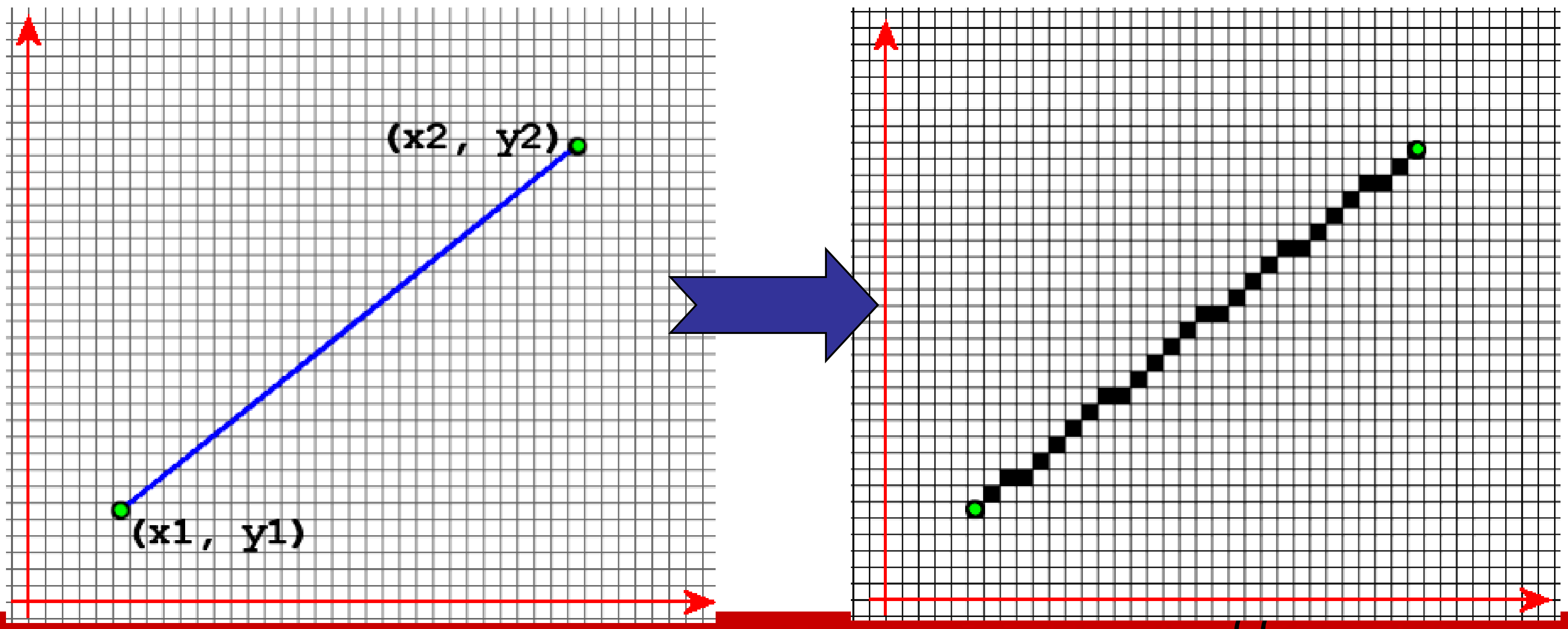




Scan Converting: how to draw 2D Line Segments

1. Given two points on the screen (with integer coords.)
2. Determine which pixels should be drawn in between these to display a unit width line…
   - Line-Drawing Algorithms:

     DDA, Midpoint (Bresenham's) Algorithm



Line Rasterization Requirements

Transform continuous primitive into discrete samples

- Uniform thickness & brightness
- Continuous appearance
- No gaps
- Accuracy
- Speed

Finding next pixel:

Special case:

- Horizontal Line:

  Draw pixel *P* and increment *x* coordinate value by 1 to get next pixel.

- Vertical Line:

  Draw pixel *P* and increment *y* coordinate value by 1 to get next pixel.

- Diagonal Line:

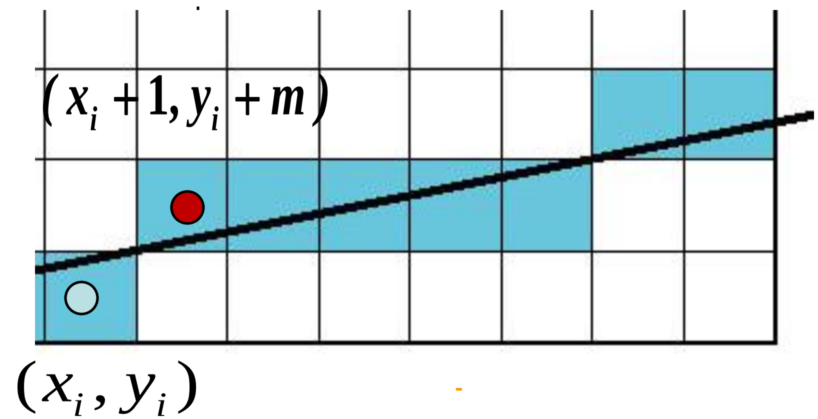
  Draw pixel *P* and increment both *x* and *y* coordinate by 1 to get next pixel.

- What should we do in general case?
  - Increment x coordinate by 1 and choose point closest to line.
  - But how do we measure "closest"?

Strategy 1 – Digital Differential Analyzer (DDA)

Equation of line that connects two points

$$y = mx + B$$



Starting with leftmost point,

- Increment x_i by 1

$$\begin{aligned} y_{i+1} &= mx_{i+1} + B \\ &= m(x_i + 1) + B \\ &= mx_i + B + m \\ &= y_i + m \end{aligned}$$

Float Increment

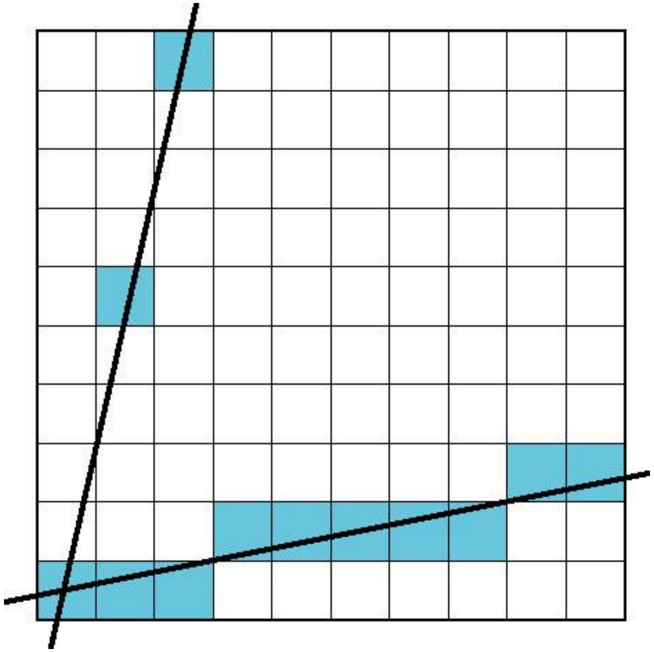
$$m = \Delta y / \Delta x$$

$$\begin{cases} \Delta x = x_2 - x_1 \\ \Delta y = y_2 - y_1 \end{cases}$$

- Color pixel at $drawpixel[x, round(y)]$

Problem

- DDA = for each x plot pixel at closest y
  - Problems for steep lines



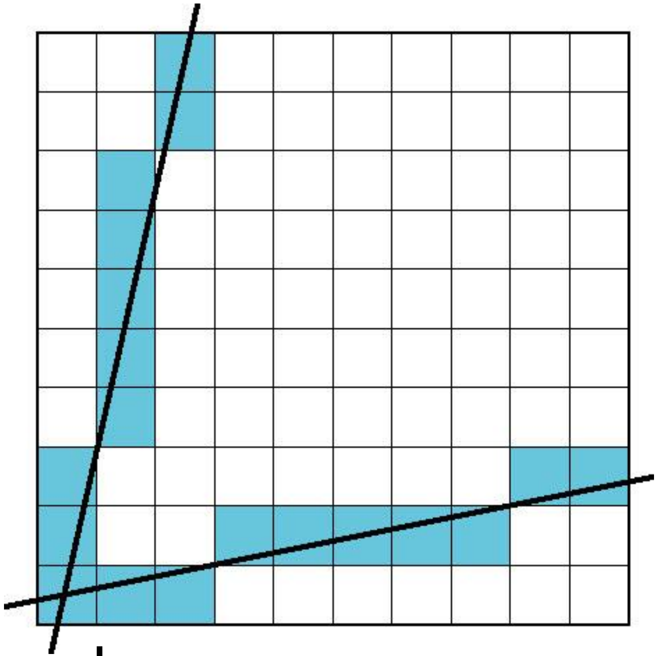
Solution: if the slope is >1, step for y instead

Using Symmetry

- For m > 1, swap role of x and y

$y_{i+1} = mx_{i+1} + B = m(x_i + 1) + B = y_i + m$ $(0<m \leq 1)$

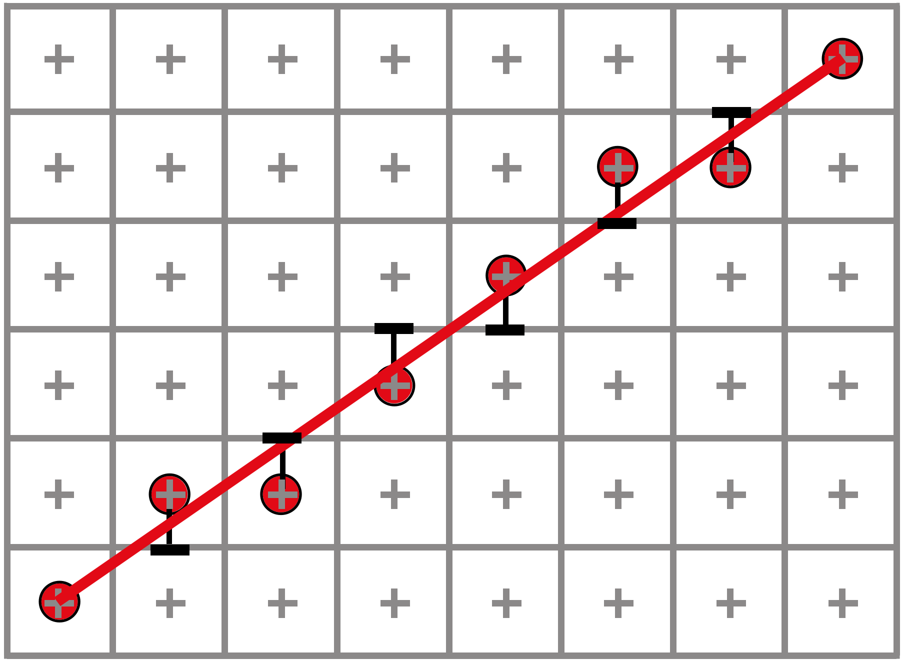
$y_{i+1} = y_i+1$ $x_{i+1} = (y_{i+1} - B)/m = x_i + 1/m$ $(m>1)$



y and m are float type numbers,
it is hard to be implemented by hardware.

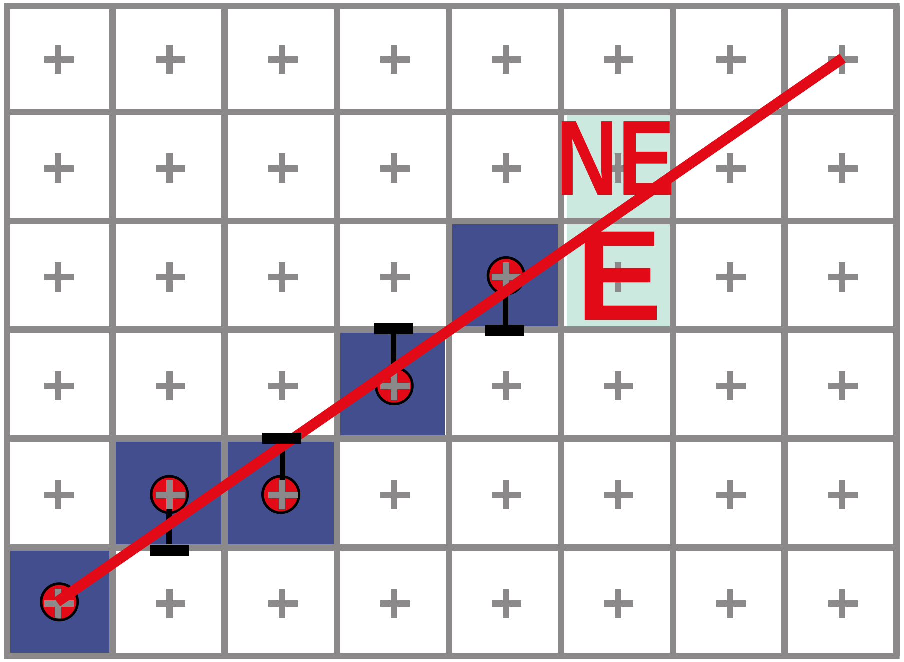
Strategy 2: Midpoint Algorithm (1985) (Bresenham's Algorithm (1965))

- Select pixel vertically closest to line segment
  - intuitive, efficient, pixel center always within 0.5 vertically



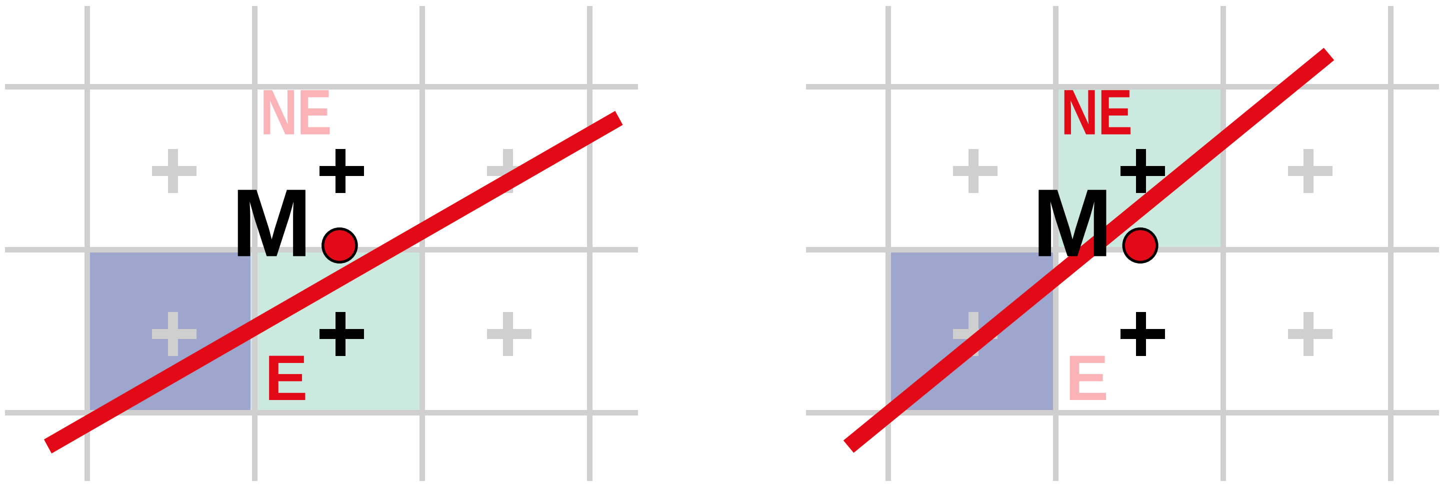
Bresenham's Algorithm

- Observation:
  - If we're at pixel P (x_p, y_p), the next pixel must be either E (x_p+1, y_p) or NE (x_p+*1*, y_p+1)



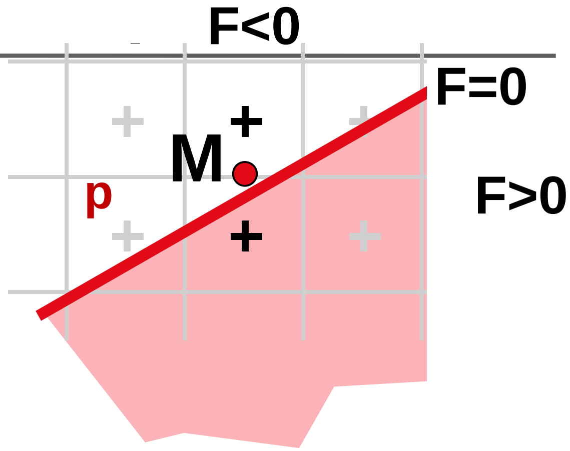
Bresenham Step

- Which pixel to choose: E or NE?
  - Choose E if segment passes below or through middle point M
  - Choose NE if segment passes above M



Now, find a way to calculate on which side of line midpoint lies

Line



Line equation as function y:

- $y = m\,x + B = dy/dx\;\; x + B$

Line equation as implicit function:

- $F(x, y) = a\,x + b\,y + c = 0$

  from above, $y\,dx = dy\,x + B\,dx$ *or*

  $$\boldsymbol{F(x, y) = dy\,x - y\,dx + B\,dx = 0}$$

  so $\boldsymbol{a = dy}$, $\boldsymbol{b = -dx}$, $\boldsymbol{c = B\,dx}$, $a > 0$ for $y_1 < y_2$

- $F(x_m, y_m) = 0$ when any point M is on line
- $F(x_m, y_m) < 0$ when any point M is above line
- $F(x_m, y_m) > 0$ when any point M is below line
- Our decision will be based on value of function at midpoint M (x_m, y_m) where ($x_m$$\mathbf{=x_p+1}$, y_m=$\mathbf{y_p+½}$)

Decision Variable

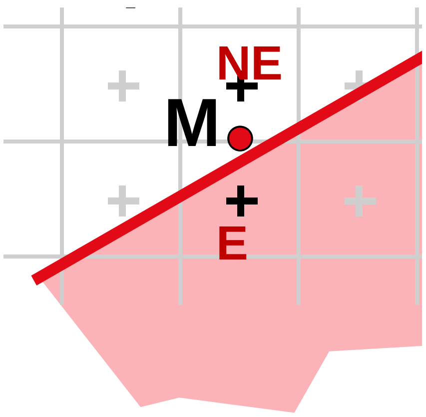
Decision Variable *d*:

- midpoint M at $(x_p + 1, y_p + ½)$
- We only need sign of $F(x_p + 1, y_p + ½)$ to see where line lies, and then pick nearest pixel
- $d = F(x_p + 1, y_p + ½)$

  - if $d > 0$ choose pixel NE

  - if $d < 0$ choose pixel E

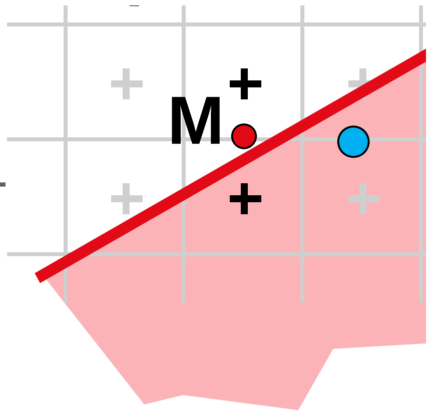
  - if $d = 0$ choose either one consistently



How do we incrementally update *d* ?

- On basis of picking E or NE, figure out location of M for that pixel, and corresponding value of d for next grid line

If E was chosen:



Increment *M* by one in *x* direction

$d_{new} = F(x_p + 2, y_p + ½)$
$\quad = a(x_p + 2) + b(y_p + ½) + c$
$d_{old} = a(x_p + 1) + b(y_p + ½) + c$

- Subtract d_{old} from d_{new} to get incremental difference ΔE

  $d_{new} = d_{old} + a$

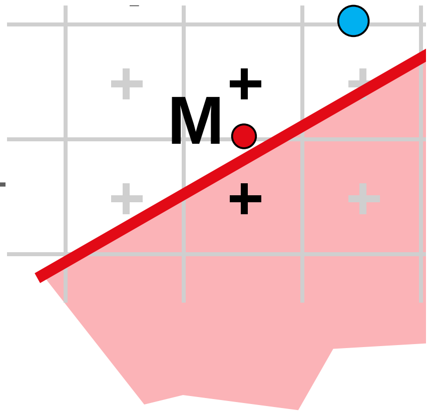
  $\Delta E = a = dy$

- Derive value of decision variable at next step incrementally without computing *F(M)* directly

  $d_{new} = d_{old} + \Delta E = d_{old} + dy$

- ΔE can be thought of as correction or update factor to take d_{old} to d_{new}
- It is referred to as forward difference

If NE was chosen:



Increment M by one in both *x* and *y* directions

$$d_{new} = F(x_p + 2, y_p + 3/2)$$
$$= a(x_p + 2) + b(y_p + 3/2) + c$$

- Subtract d_{old} from d_{new} to get incremental difference

$$d_{new} = d_{old} + a + b$$
$$\Delta NE = a + b = dy - dx$$

- Thus, incrementally,

$$\mathbf{d_{new} = d_{old} + \Delta NE = d_{old} + dy - dx}$$

Summary

- At each step, algorithm chooses between 2 pixels based on sign of decision variable calculated in previous iteration.

- It then updates decision variable **d** by adding either ΔE or ΔNE to old value depending on choice of pixel. Simple additions only!

- First midpoint for first $d = d_{start}$ is at $(x_0 + 1, y_0 + ½)$:

  $$F(x_0 + 1, y_0 + ½) = F(x_0, y_0) + a + b/2 = a+b/2$$

  To eliminate fraction in d_{start} :

  =0, (x_0, y_0) is on the line

  redefine F by multiplying it by 2; F(x,y) = 2(ax + by + c)

Example Code

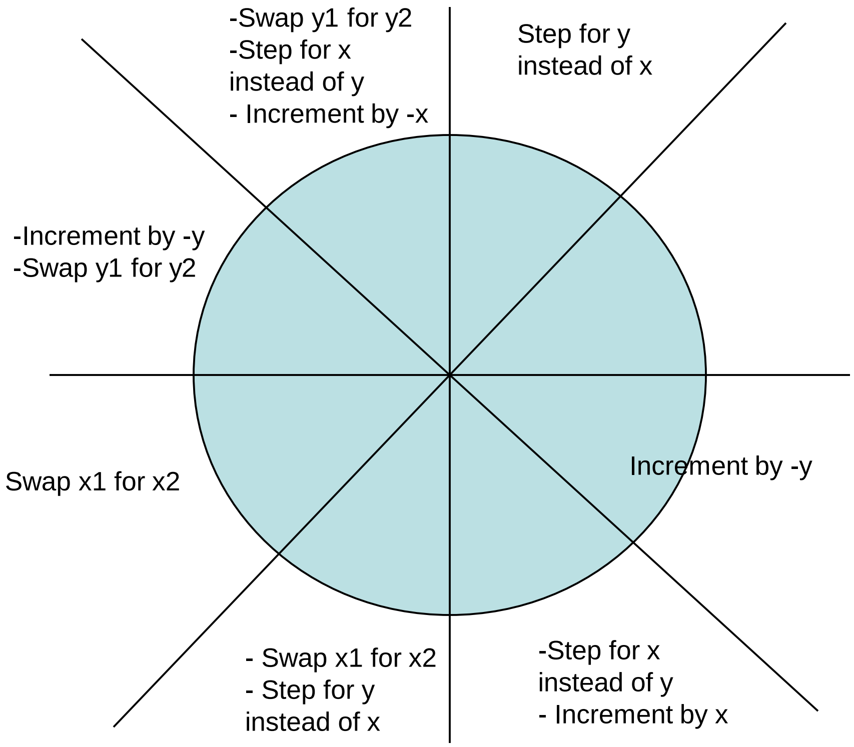
```
void MidpointLine(int x1, int y1,
                  int x2, int y2, int value) {
        int     dx = x2 - x1;
        int     dy = y2 - y1;
        int     d = 2 * dy - dx;
        int     incrE = 2 * dy;
        int     incrNE = 2 * (dy - dx);
        int     x = x1;
        int     y = y1;

        writePixel(x, y, value);

        while (x < x2) {
                if (d <= 0) {  // East Case
                        d = d + incrE;
                } else {               // Northeast Case
                        d = d + incrNE;
                        y++;
                }
                x++;
                writePixel(x, y, value);
        }               /* while */
}                       /* MidpointLine */
```

Other Quadrants

- Note that this only applies to lines with a positive gradient
- But you can easily write a separate case for each other case
- Also if the gradient is too steep you need to step for x instead of y (as we saw in DDA)



Polygon Rasterization
Polygon Filling

Polygon Rasterization
Polygon Filling

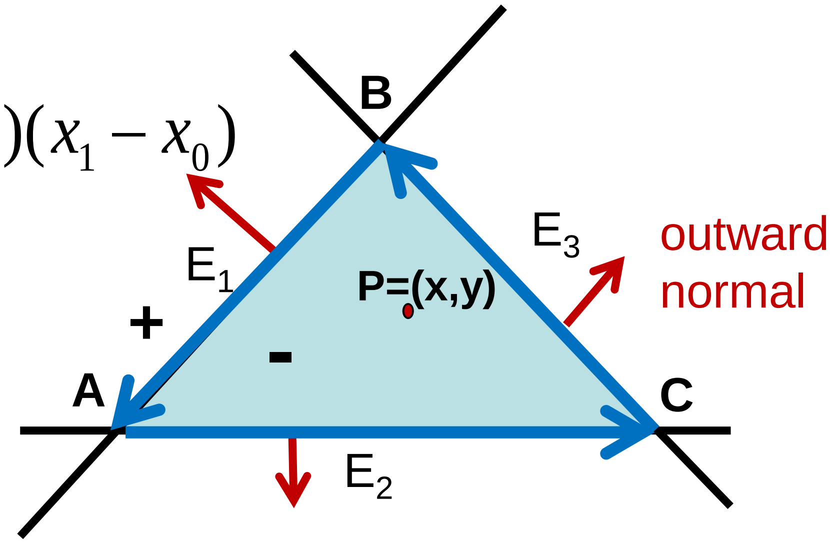
- The basic rule for filling a polygon is:
  - If a point is inside the polygon, color it with the inside color.
  - **INSIDE/OUTSIDE TEST**: triangle test/odd-even test
- Polygon fill is a sorting problem, where we sort all the pixels in the frame buffer into those that are inside the polygon, and those that are not.
- Polygon filling Algorithms:
  - Flood fill
  - Scan-line fill

Inside Triangle Test

A point is inside a triangle if it is in the negative half-space of all three boundary lines

- Triangle vertices are ordered counter-clockwise
- Point must be on the left side of every boundary line

$$E(x,y) = ax + by + c =$$
$$= (x - x_0)(y_1 - y_0) - (y - y_0)(x_1 - x_0)$$
$$= x\underbrace{(y_1 - y_0)}_{a} + y\underbrace{(x_0 - x_1)}_{b} + \underbrace{-x_0(y_1 - y_0) + y_0(x_1 - x_0)}_{c}$$

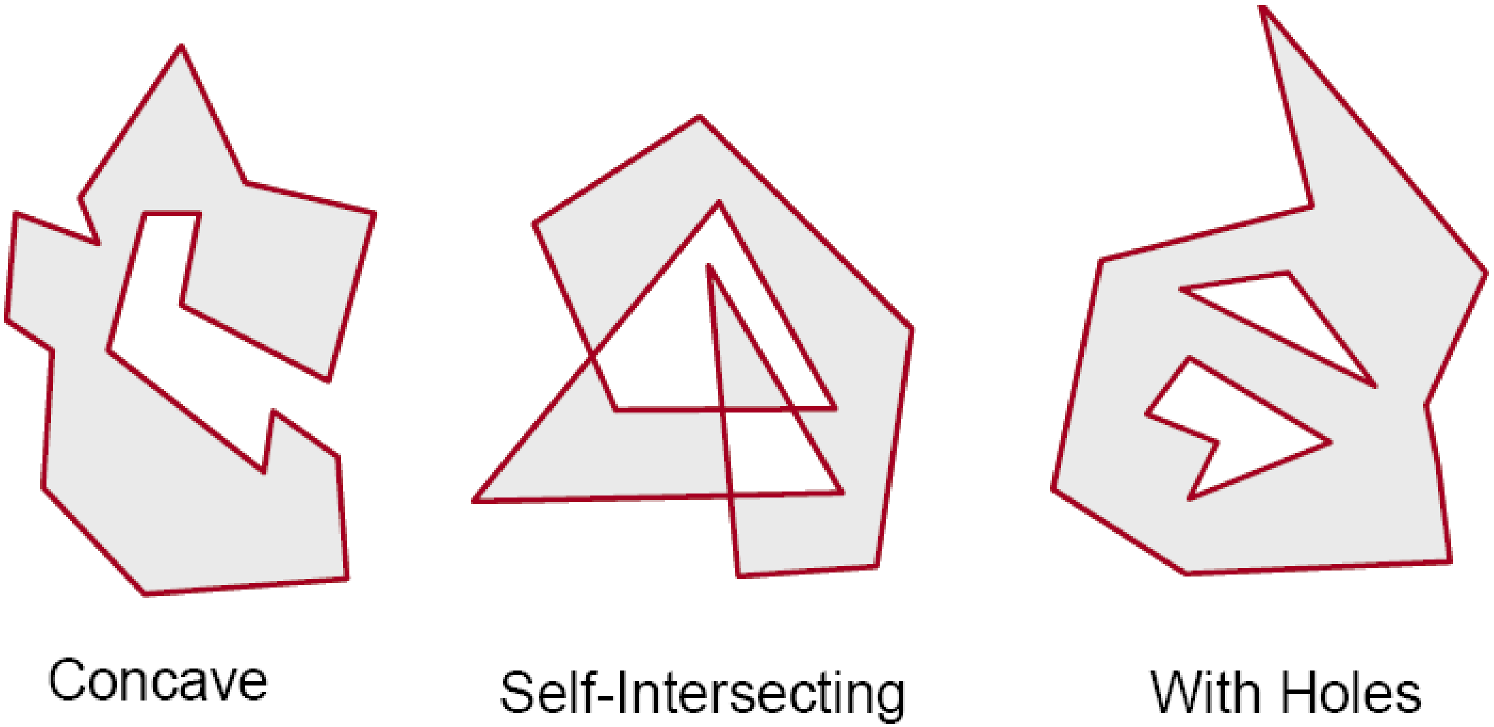


$$(x,y) \text{ within triangle } \Leftrightarrow \mathrm{E}_i(x,y) \le 0, \forall i = 1,2,3$$

Triangle method works only for convex polygons!

Inside Polygon Rule

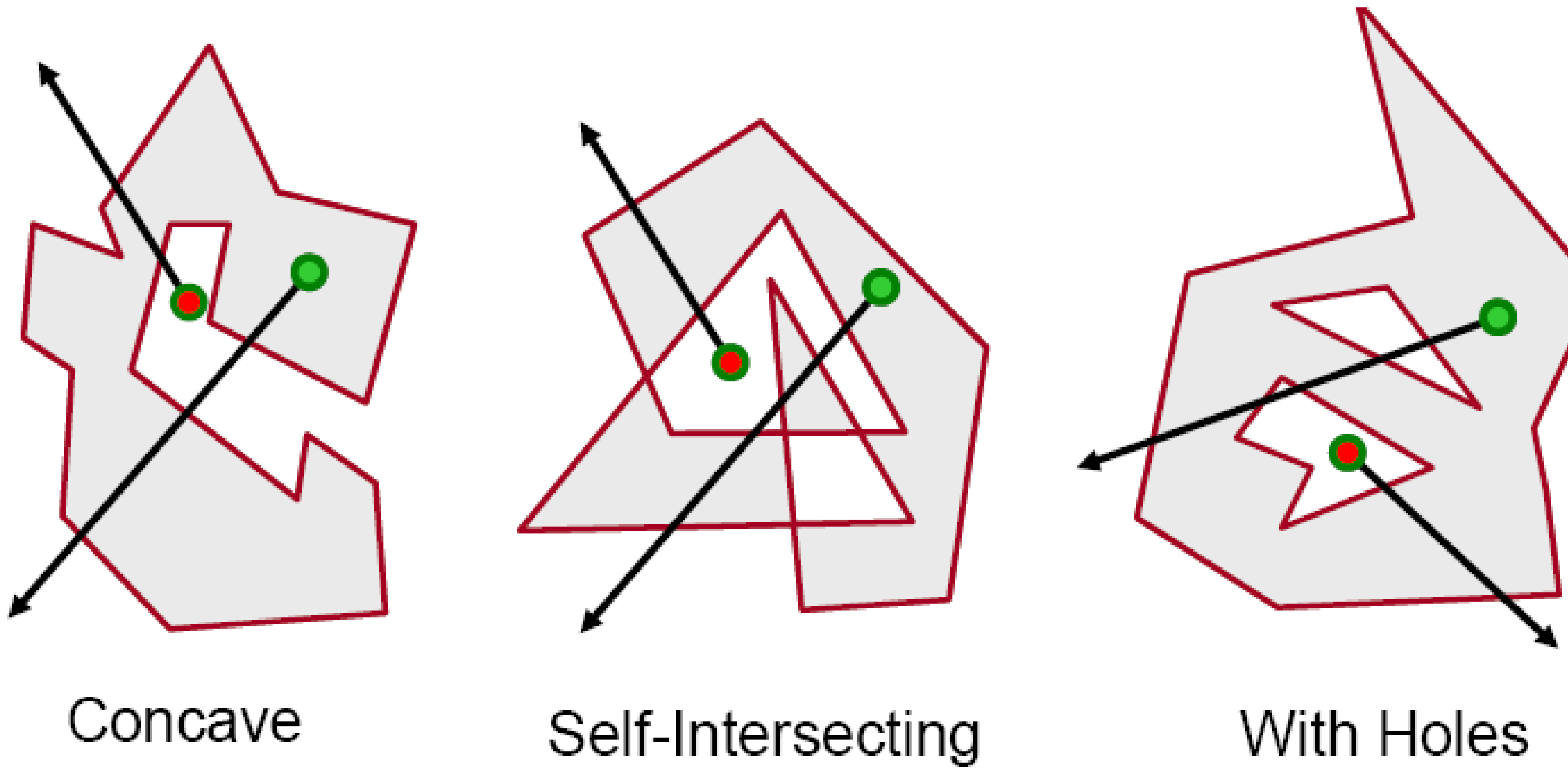
- How to tell inside from outside
  - Convex easy, but..
  - What is a good rule for which pixels are inside?



Odd-even test

Any ray from P to infinity crosses a number of edges

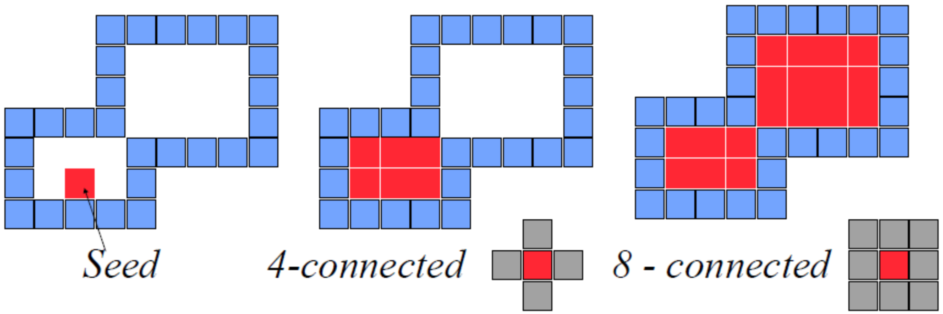
- Odd = inside polygon
- Even = outside polygon



Flood Fill Method

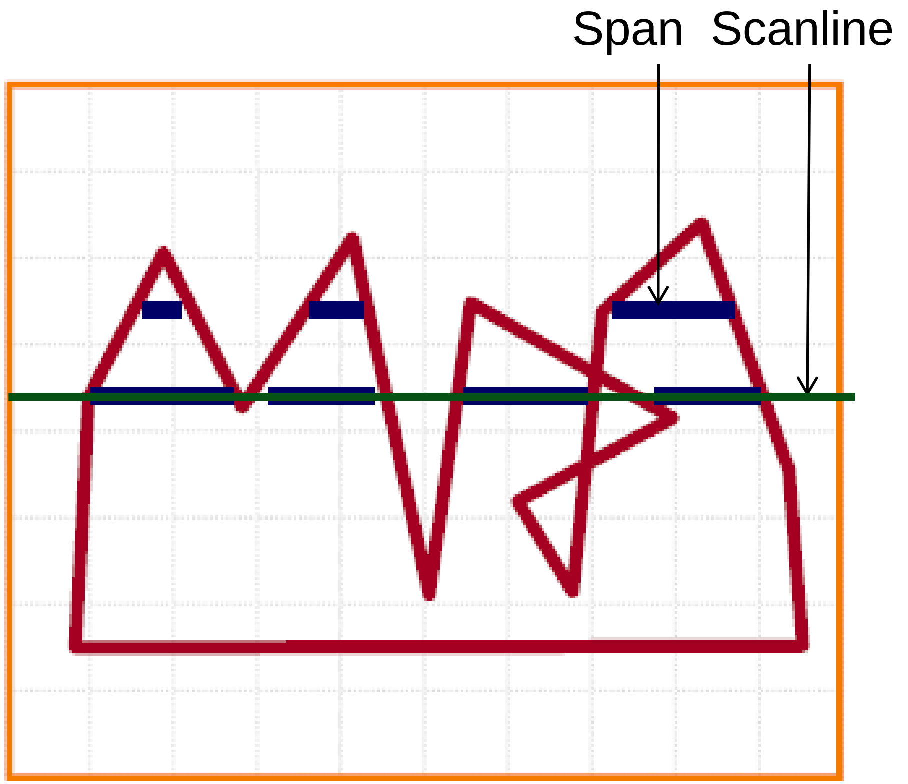
- Given an initial point(x,y) inside the polygon- a seed point - then we can look at its neighbors recursively, coloring them with the foreground color if they are not edge point.
- use black for rasterize edge
- Scan convert edges into buffer in edge/inside color (RED)

```
flood_fill(int x, int y) {
    if(read_pixel(x,y)== WHITE) {
       write_pixel(x,y,RED);
       flood_fill(x-1, y);
       flood_fill(x+1, y);
       flood_fill(x, y+1);
       flood_fill(x, y-1);
}   }
```



Scan Line Fill Method

Incremental algorithm to find spans **for each scanline (top-to-bottom)**, and determine "insideness"



Polygon

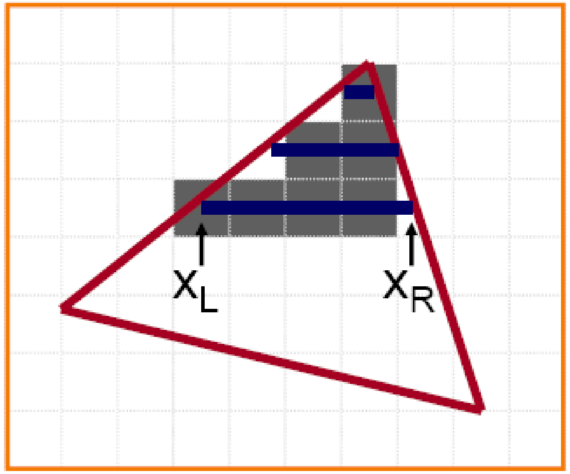
Each span can be processed independently (parallel span processor)

Scan Line Fill Method

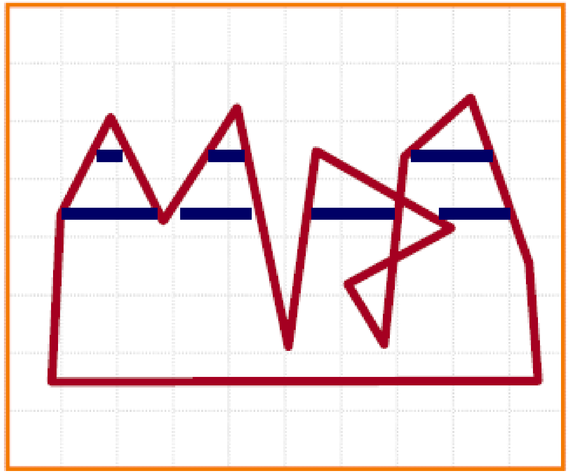
Proceeding from **top** to **bottom**, from **left** to **right** the intersections are paired and intervening pixels are set to the specified intensity

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

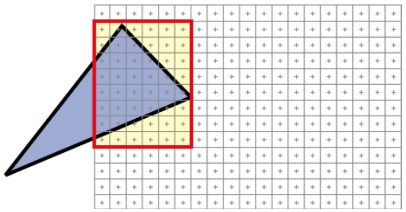


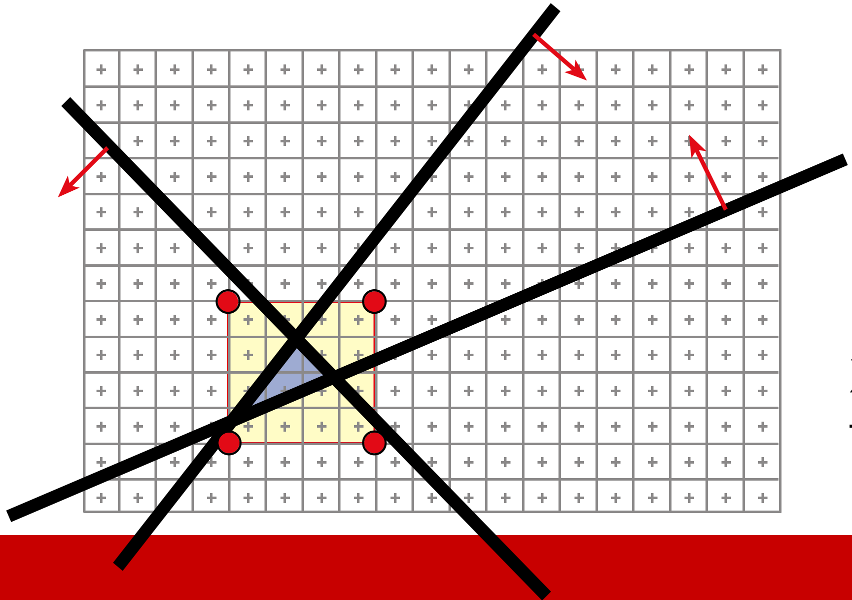
Triangle


Polygon

Scan Line Fill (for triangle)

```
For every triangle
      Compute projection for vertices, compute the Ei
      Compute bbox, clip bbox to screen limits
      For all pixels x,y in bbox
            Evaluate edge functions Ei
            If all Ei < 0
                  Framebuffer[x,y ] = triangleColor
```





How do we get such a bounding box?
Xmin, Xmax, Ymin, Ymax of the projected triangle vertices

Can we do better?

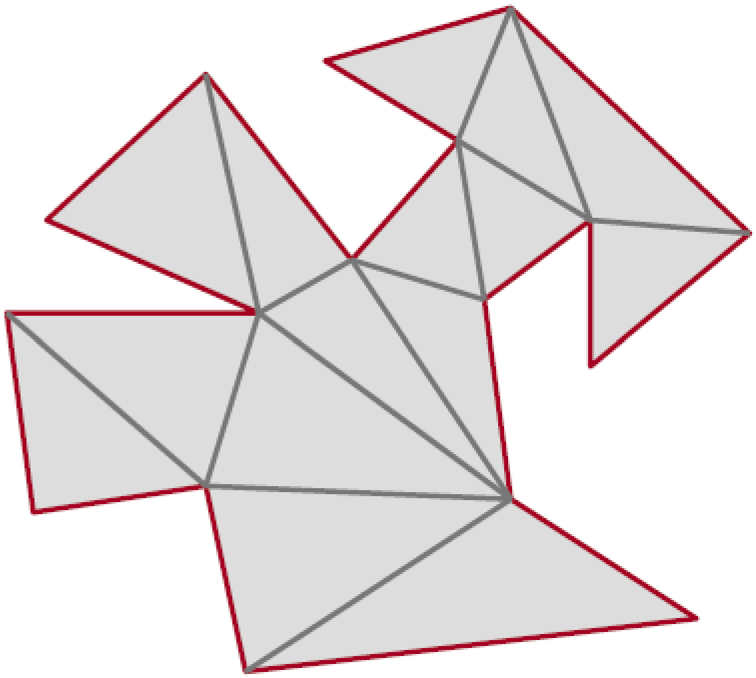
```
For every triangle
  Compute projection for vertices
  Compute bbox, clip bbox to screen limits
  For all scanlines y in bbox
     Evaluate all Ei's at (x0,y): Ei = aix0 + biy + ci
     For all pixels x in bbox
          If all Ei<0
                Framebuffer[x,y ] = triangleColor
          Increment line equations: Ei += ai
```

$E_i(x+1,y) = E_i(x,y) + a_i$

We save ~two multiplications and two additions per pixel when the triangle is large

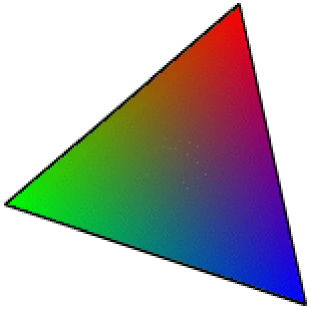
Scanline for concave polygons: tessellator

Convert everything into triangles
then scan convert the triangles



Framebuffer[x,y] = Color(?)

- We store data (such as color, etc.) on the vertices of triangles, and subsequently use interpolation to compute values of this data on the interior of the triangle



For example:

•Specify an (R,G,B) color on each vertex of a triangle

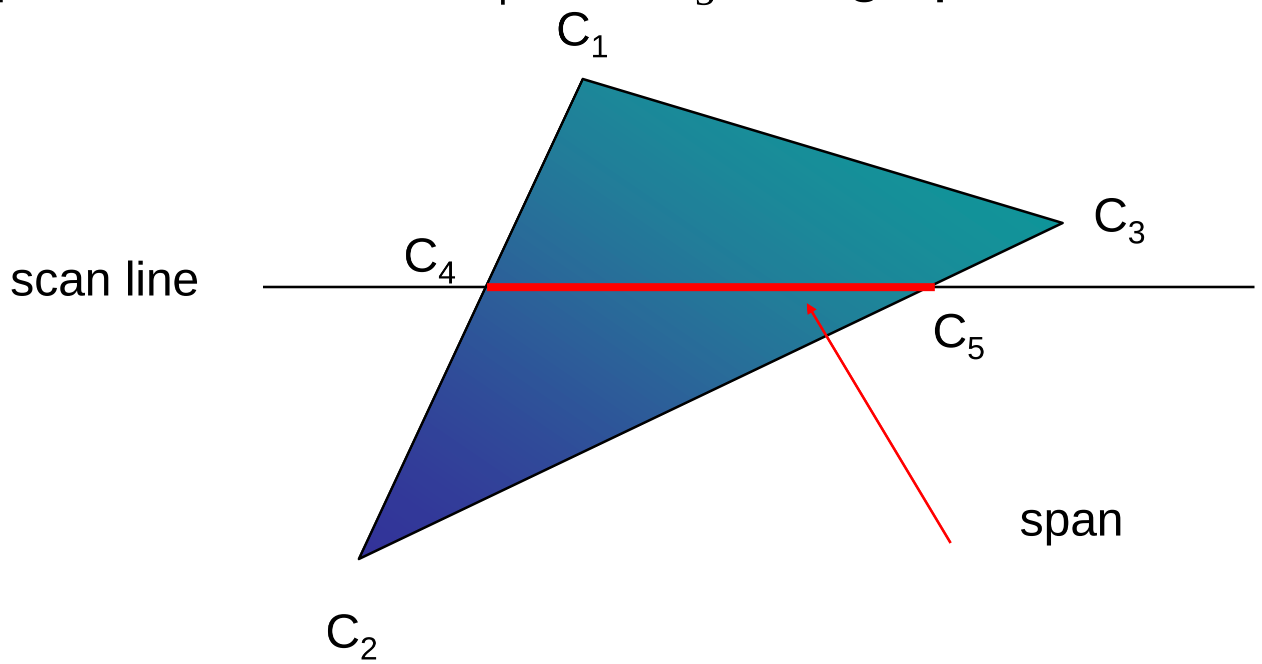
•For each pixel inside the triangle, compute the interpolation coordinates for that pixel

•Then use these interpolation coordinates to compute an interpolated (R,G,B) color value for that pixel

Per-pixel color: linear Interpolation

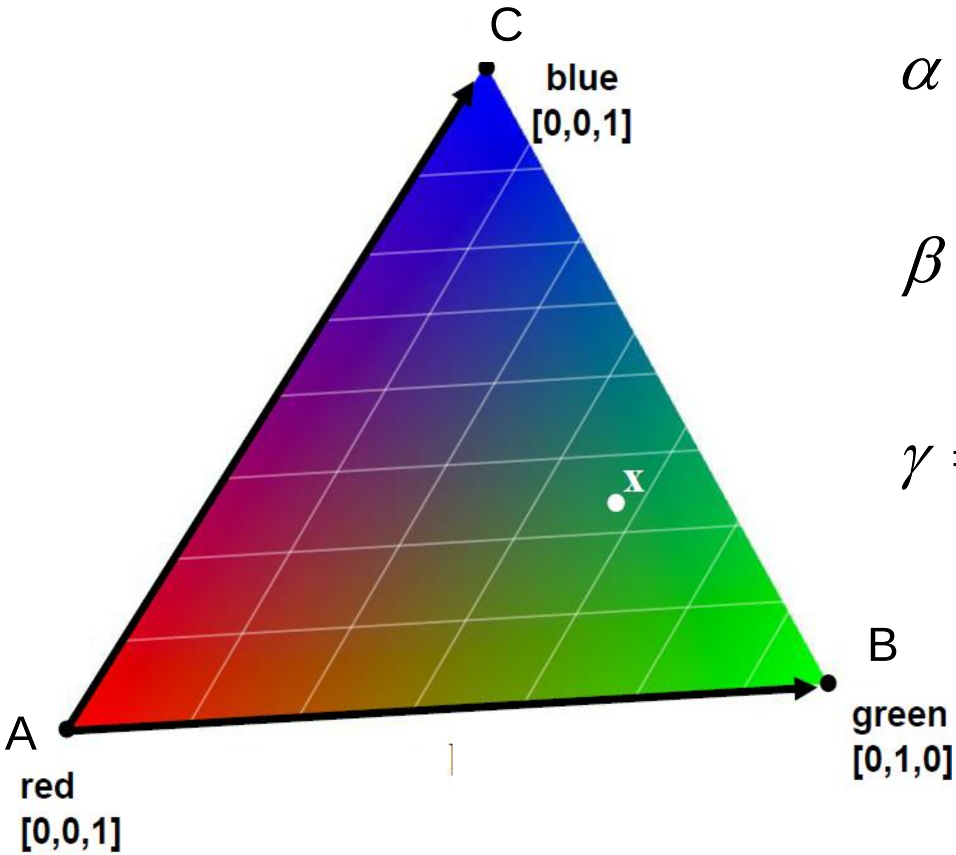
C_1 C_2 C_3 specify `Color` or by vertex shading

C_4 determined by interpolating between C_1 and C_2
C_5 determined by interpolating between C_2 and C_3
interpolate between C_4 and C_5 along span



Per-Pixel color: barycentric interpolation

Triangle's color at point x?



$$\alpha = \frac{area_{xBC}}{area_{ABC}} = \frac{\frac{1}{2} E_{BC}(\mathrm{x})}{\frac{1}{2} E_{BC}(\mathrm{A})} = \frac{E_{BC}(\mathrm{x})}{E_{BC}(\mathrm{A})}$$

$$\beta = \frac{E_{CA}(\mathrm{x})}{E_{CA}(\mathrm{B})}$$

$$\gamma = \frac{E_{AB}(\mathrm{x})}{E_{AB}(\mathrm{C})}$$

Color at **x** is **affine** combination of color at three triangle vertices

$$\boxed{\mathrm{x}_{color} = \alpha A_{color} + \beta B_{color} + \gamma C_{color}}$$

Per-pixel attributes

Interpolate colors

- $R = \alpha_0 R_0 + \alpha_1 R_1 + \alpha_2 R_2$
- $G = \alpha_0 G_0 + \alpha_1 G_1 + \alpha_2 G_2$
- $B = \alpha_0 B_0 + \alpha_1 B_1 + \alpha_2 B_2$

Interpolate normal vectors

- $N = \alpha_0 N_0 + \alpha_1 N_1 + \alpha_2 N_2$

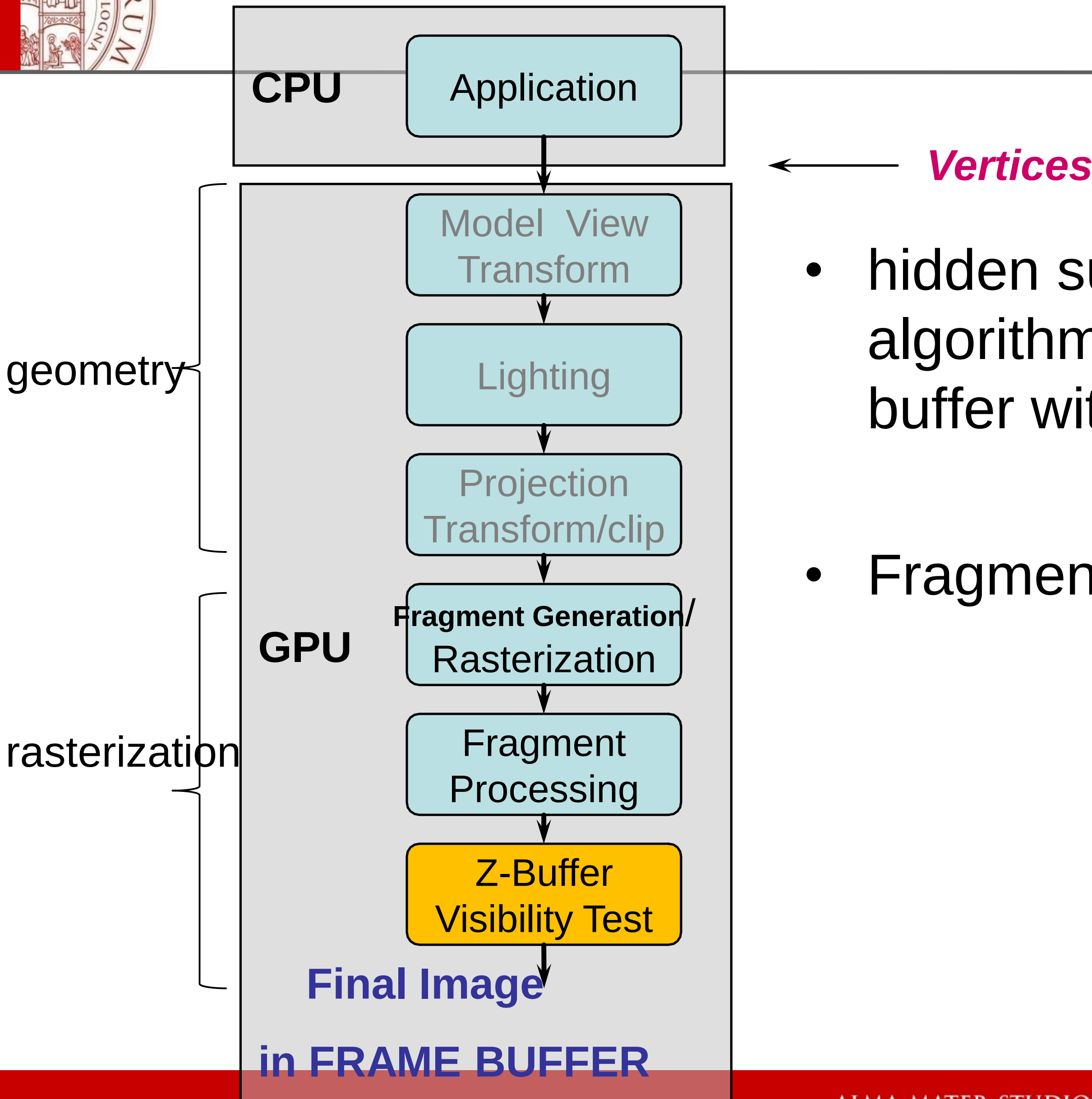
Interpolate z-buffer depth values

- $z = \alpha_0 z_0 + \alpha_1 z_1 + \alpha_2 z_2$

Interpolate texture coordinates

- $u = \alpha_0 u_0 + \alpha_1 u_1 + \alpha_2 u_2$
- $v = \alpha_0 v_0 + \alpha_1 v_1 + \alpha_2 v_2$

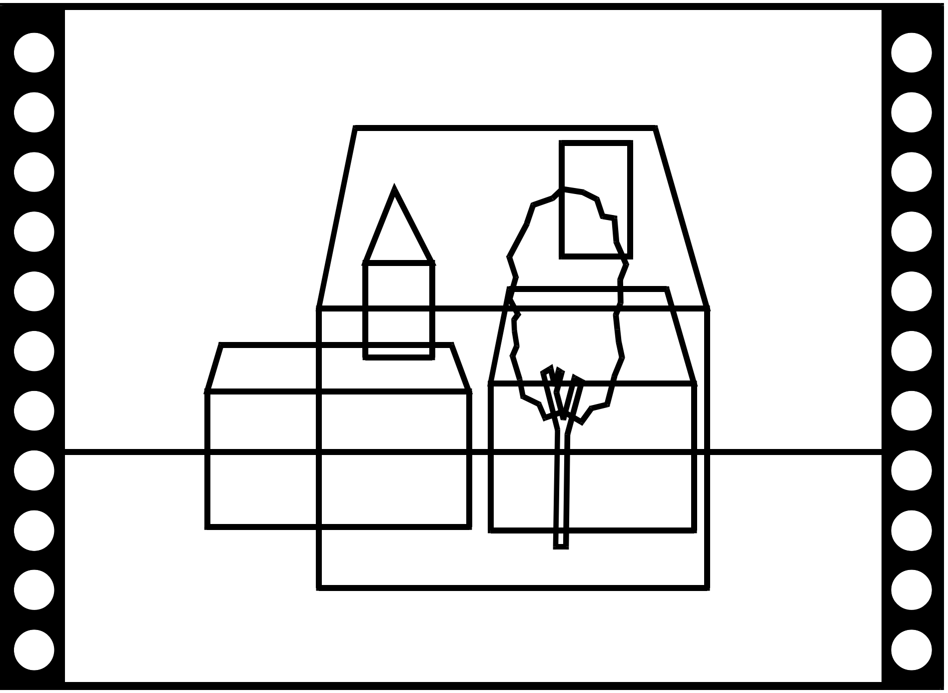
Visibility

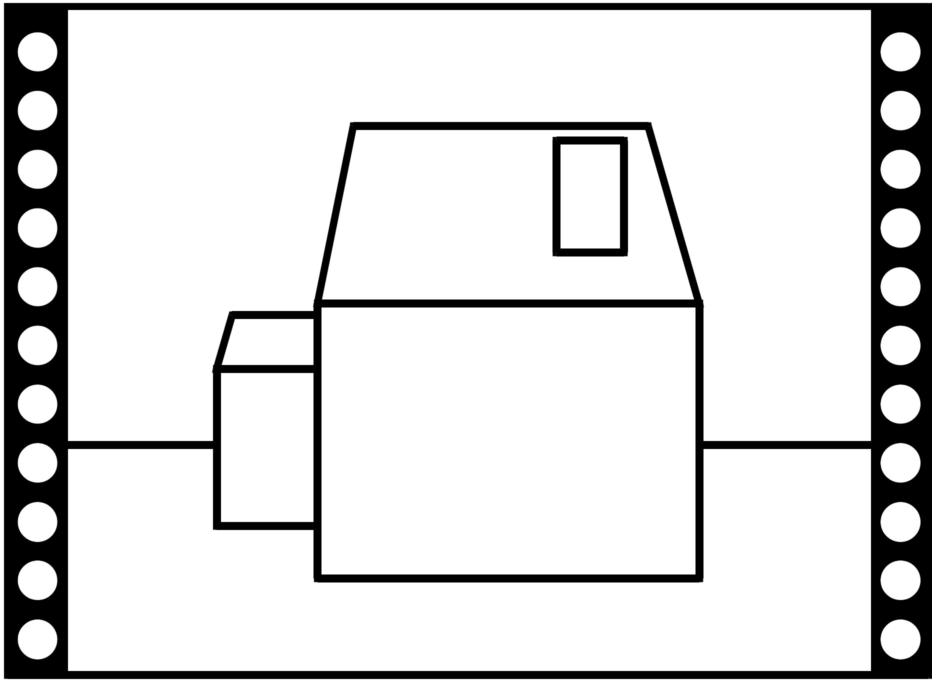


- hidden surface removal algorithms: update the frame buffer with the closest object
- Fragment processing

Visibility

How do we know which parts are visible/in front?





Given a set of 3-D objects and a view specification (camera), determine which lines or surfaces of the object are visible

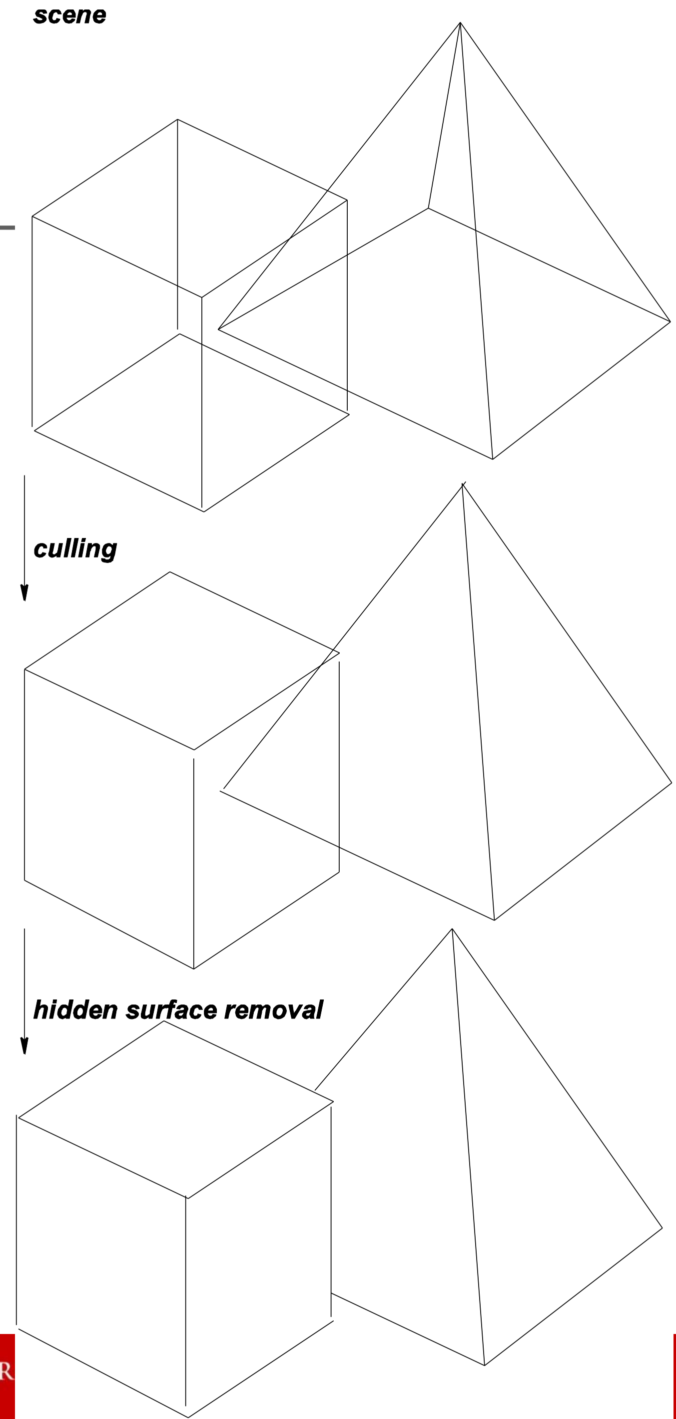
Culling vs HSR

Culling

refers to the process of determining and culling polygons which are *not* visible

Before HSR

HSR (hidden surface removal) refers to the process of determining which parts of the polygons are *not* visible

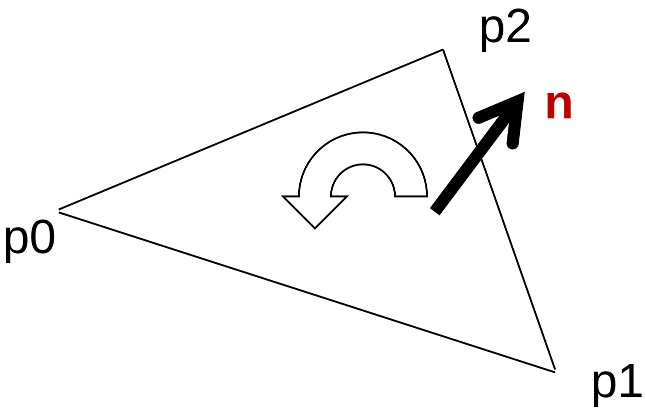


Culling

- There are three common reasons to cull a particular triangle
  - If it doesn't lie within the view volume (**view frustum culling**)
  - If it is facing 'away' from the viewer (**back-face culling**)
  - If it is *degenerate* (area=0)
- The first case is built automatically into the clipping algorithm which we already covered
- In the third case normal **n** will be [0 0 0]

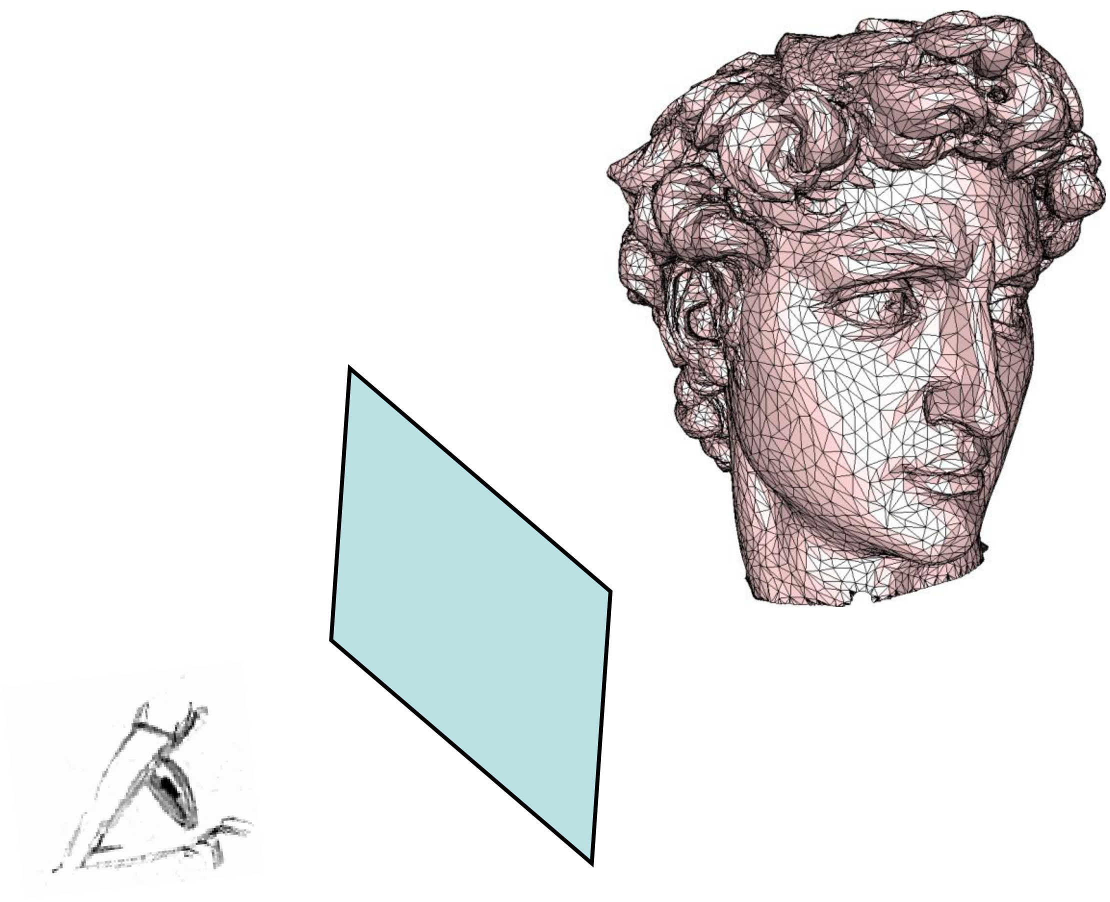
Back-face Culling

- By convention, the **front side of the triangle** is defined as the side where the vertices are arranged in a counterclockwise fashion



- Most renderers allow triangles to be defined as one or two sided. Only one-sided triangles need to be backface culled.

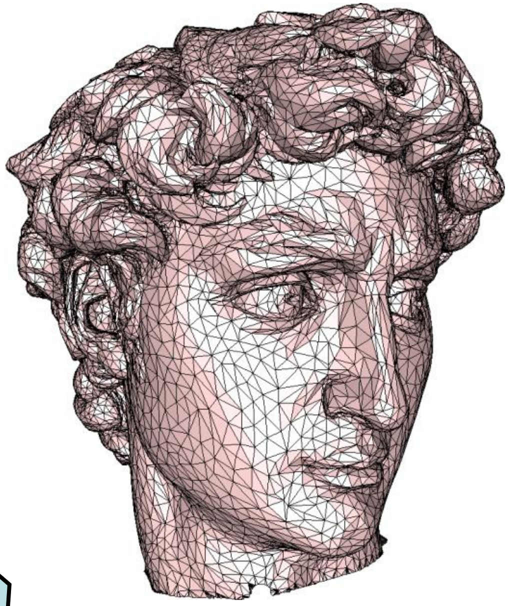
Backface Culling



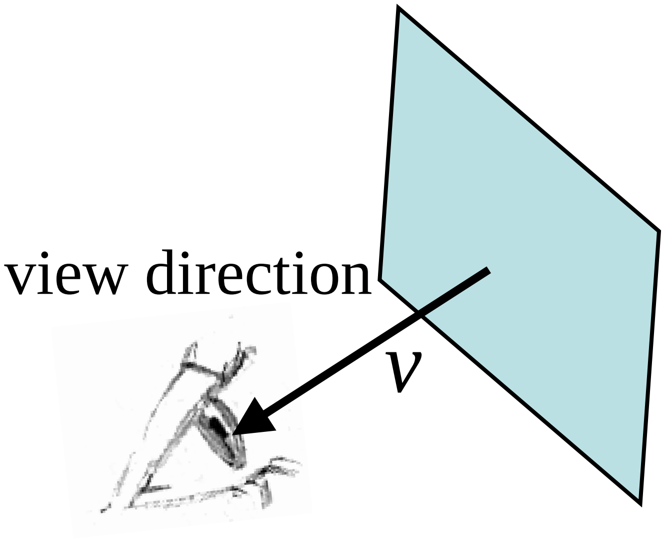
Any back facing triangles should be culled as early as possible, as it would be expected that up to 50% of the triangles in a scene would be back facing

Usually, back-face culling is done before clipping, as it is a very quick operation and will affect a much larger percentage of triangles than clipping

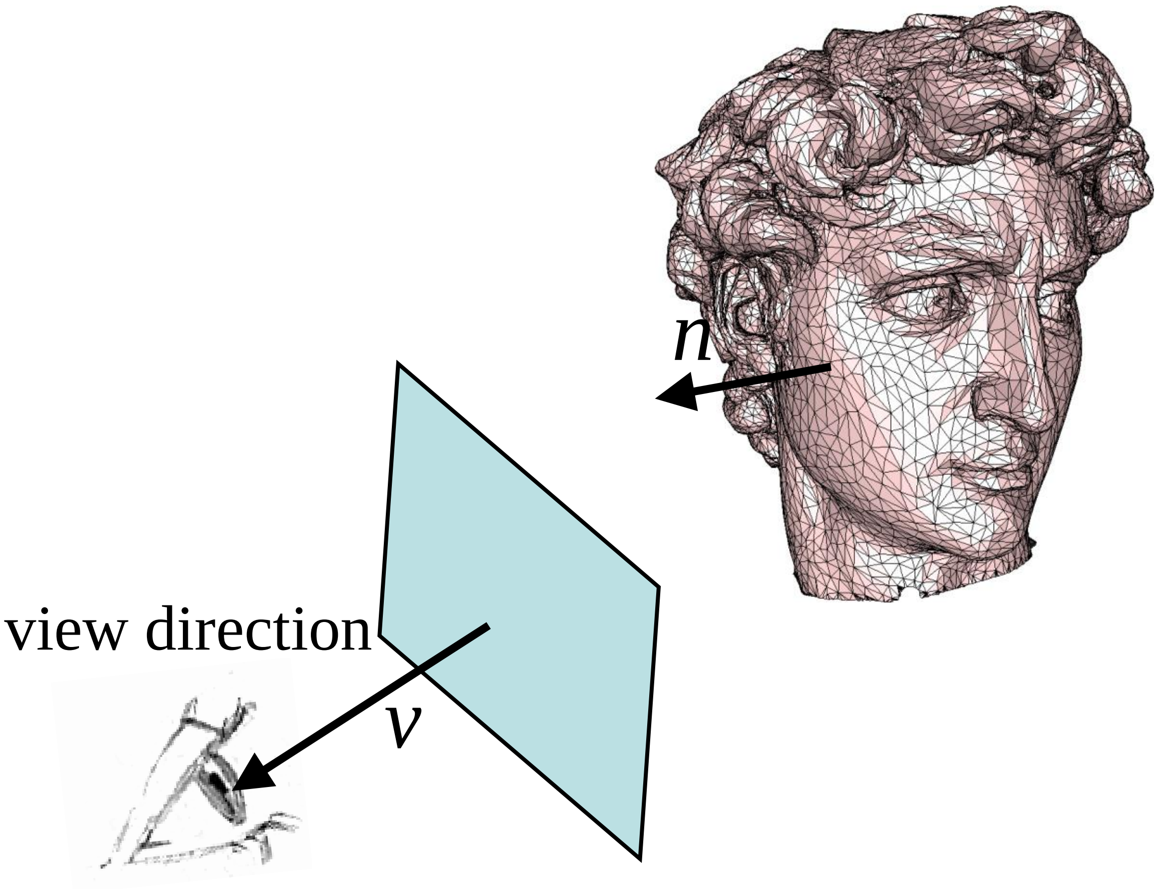
Backface Culling



the projector from any point P on polygon to the center of view



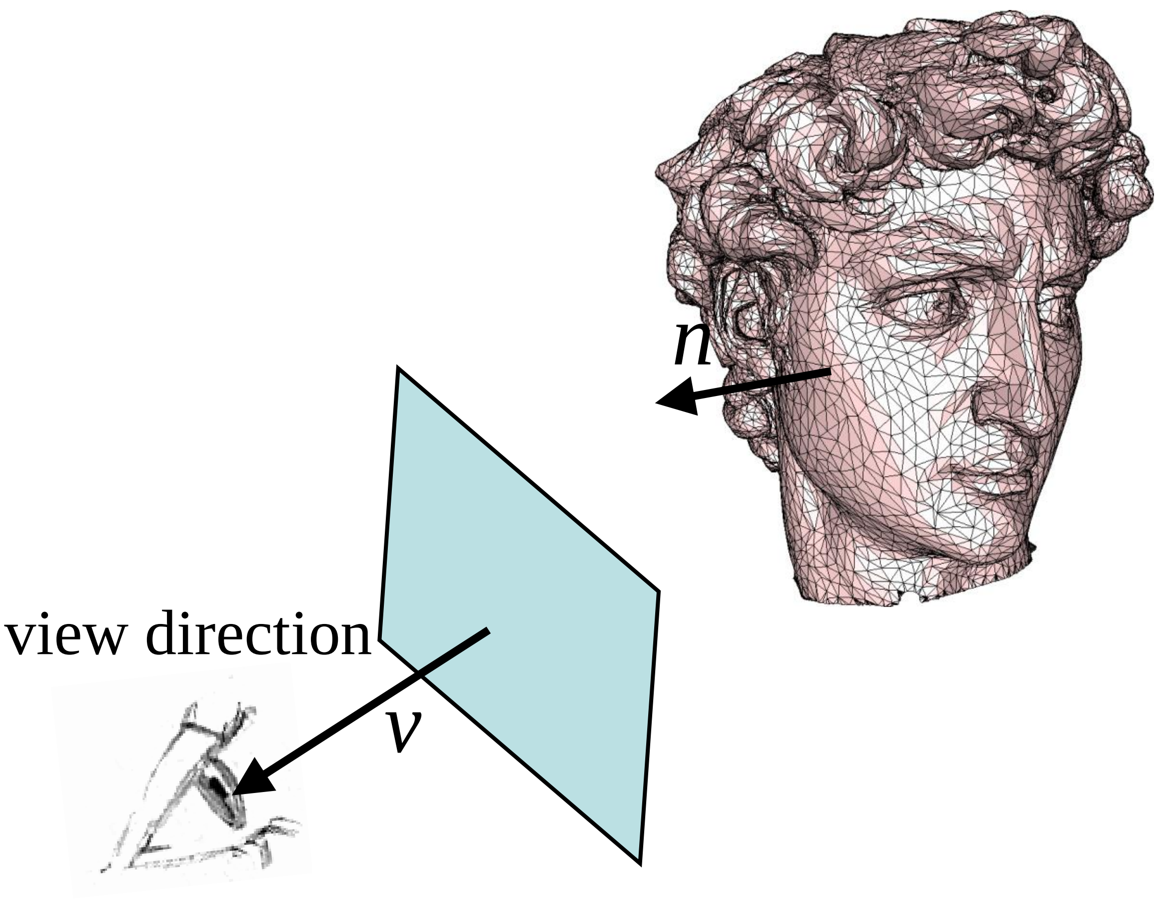
Backface Culling



Backface Culling

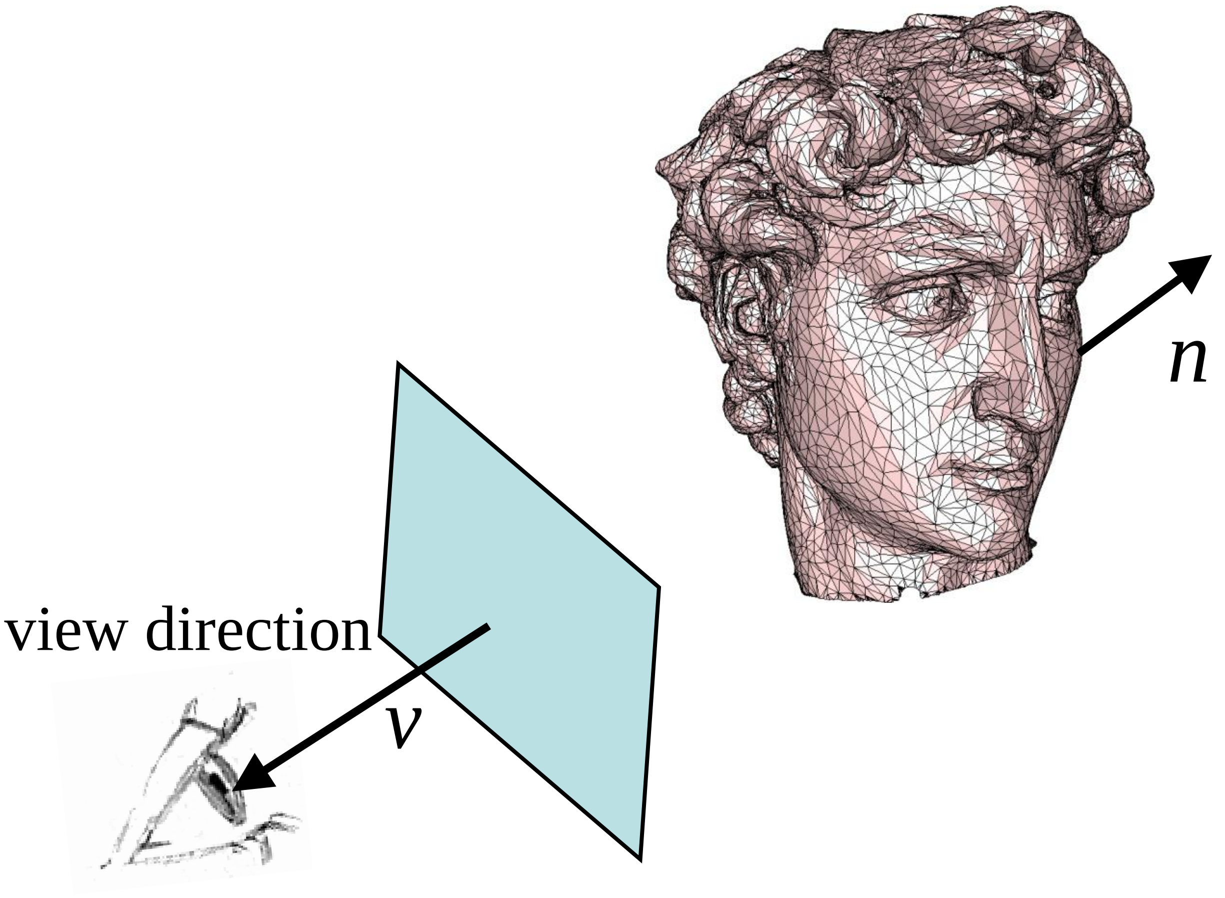
$$n \cdot v = \cos(\theta) \geq 0 \text{ , draw polygon}$$

$$-90 \leq \theta \leq 90$$



Backface Culling

$n \cdot v < 0$, cull polygon



Visible Surface determination

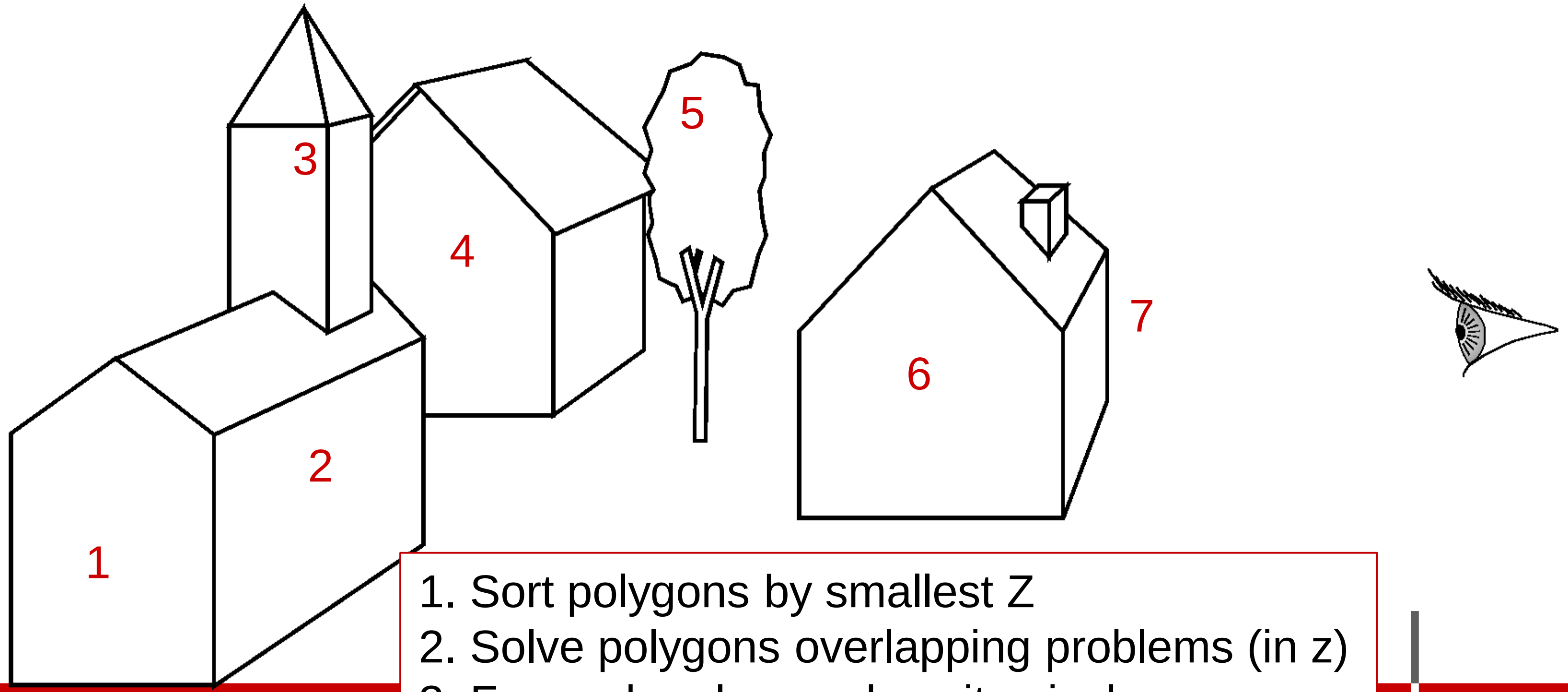
Two approaches:
Object space vs. image space

Object space: algorithms which work in the view coordinate system.
Complexity:
for k objects $O(k^2)$ since each object must be compared with all the others

Image Space: algorithms which work in the screen coordinate system;
Complexity:
for image nxm and k objects O(k)

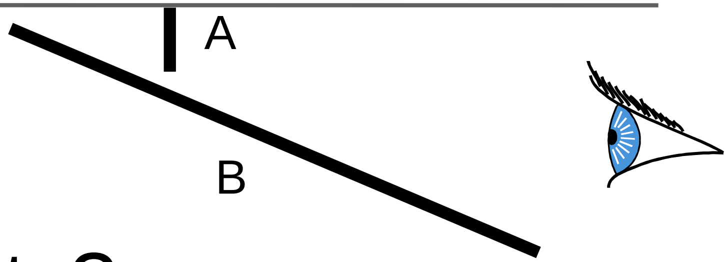
Painter's algorithm

- Draw back-to-front
- How do we sort objects?
- Can we always sort objects?

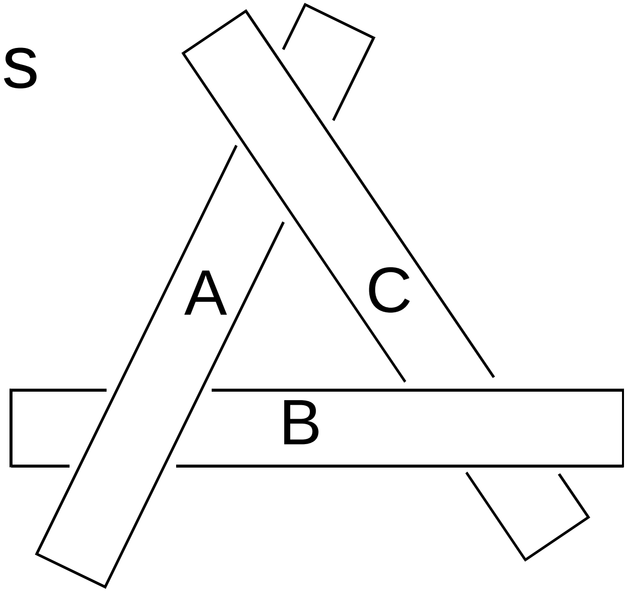


1. Sort polygons by smallest Z
2. Solve polygons overlapping problems (in z)
3. For each polygon, draw its pixels

Painter's algorithm (Depth-Sort)

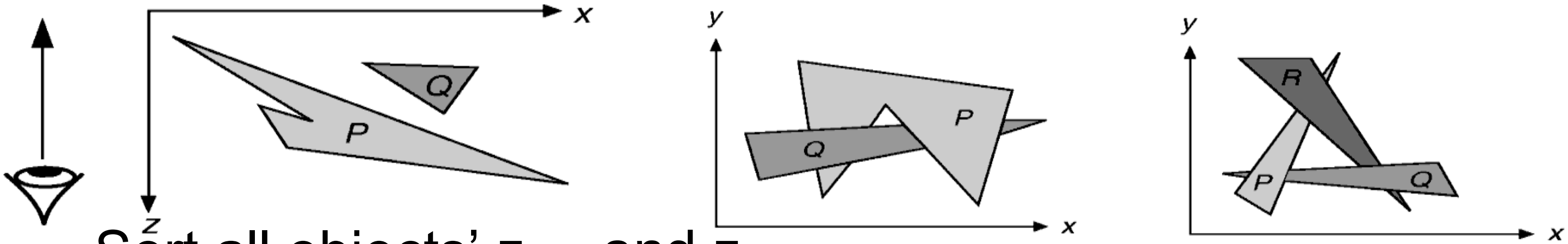


- Can we always sort objects?
  - No, there can be cycles
  - Requires to split polygons



3D Depth-Sort Algorithm

– Handles errors/ambiguities of Z-sort:



1. Sort all objects' z_{min} and z_{max}
2. If an object is uninterrupted (its z_{min} and z_{max} are adjacent in the sorted list), it is fine
3. If 2 objects DO overlap

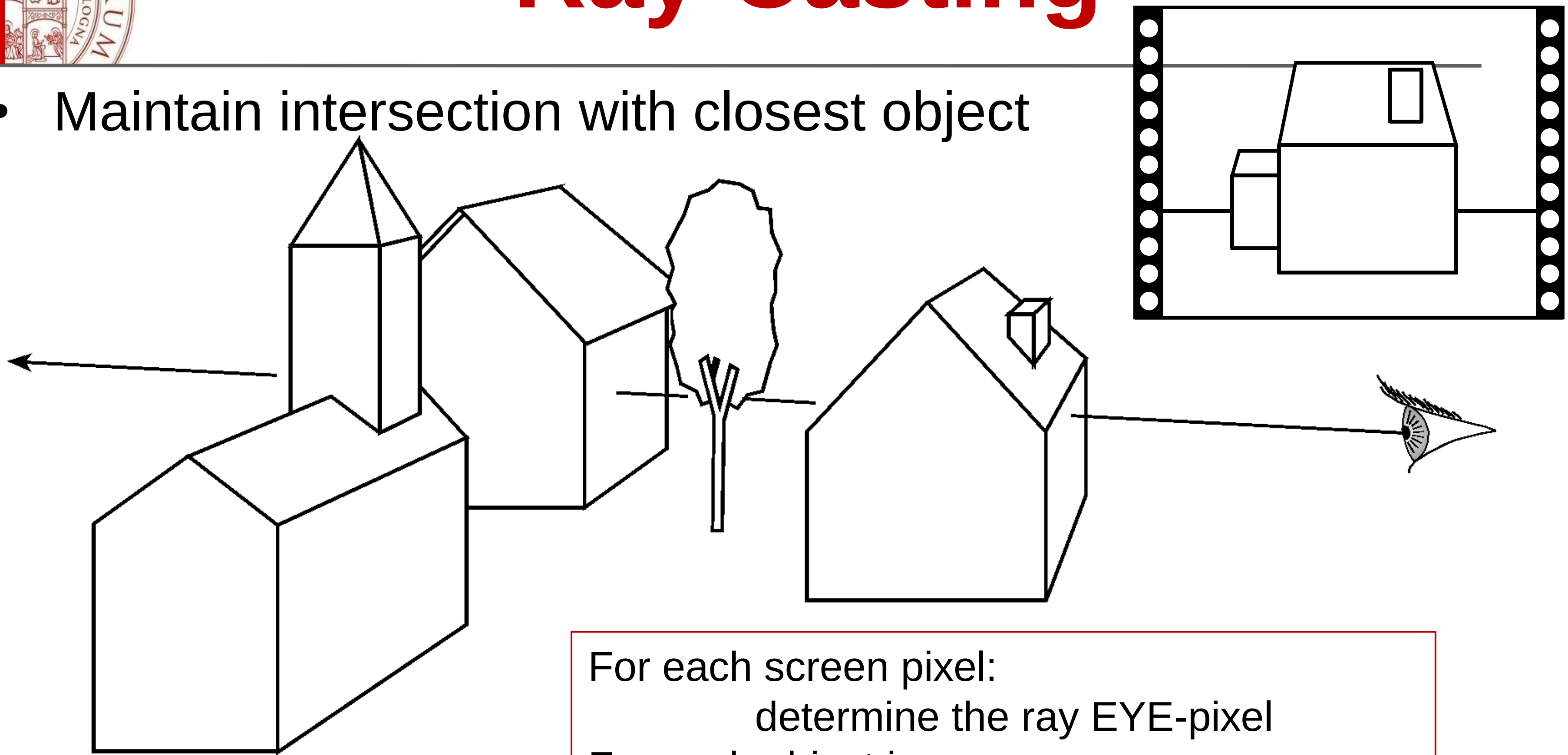
   3.1 Check if they overlap in x

   - If not, they are fine

   3.2 Check if they overlap in y

   - If not, they are fine
   - If yes, need to split one

Ray Casting



- Maintain intersection with closest object

For each screen pixel:
determine the ray EYE-pixel
For each object in scene:
compute intersection ray/object
If (inters. Is closest)
Then the object is visible
store the pixel object;
color the pixel;

Z-Buffer Algorithm (Catmull 1975)

– Requires two “buffers” of the same sizes

Color Buffer —RGB pixel buffer
—initialized to background color

Depth (“Z”) Buffer —depth of scene at each pixel
—initialized to far depth = 255

- Polygons are scan-converted in arbitrary order. When pixels overlap, use Z-buffer to decide which polygon “gets” that pixel

255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255

+

127	127	127	127	127	127	127
127	127	127	127	127	127	
127	127	127	127	127		
127	127	127	127			
127	127	127				
127	127					
127						

=

127	127	127	127	127	127	127	255
127	127	127	127	127	127	255	255
127	127	127	127	127	255	255	255
127	127	127	127	255	255	255	255
127	127	127	255	255	255	255	255
127	127	255	255	255	255	255	255
127	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255

127	127	127	127	127	127	127	255
127	127	127	127	127	127	255	255
127	127	127	127	127	255	255	255
127	127	127	127	255	255	255	255
127	127	127	255	255	255	255	255
127	127	255	255	255	255	255	255
127	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255

+

63				
63	63			
63	63	63		
63	63	63	63	
63	63	63	63	63

=

127	127	127	127	127	127	127	255
127	127	127	127	127	127	255	255
127	127	127	127	127	255	255	255
63	127	127	127	255	255	255	255
63	63	127	255	255	255	255	255
63	63	63	255	255	255	255	255
63	63	63	63	255	255	255	255
63	63	63	63	63	255	255	255

Example using integer Z-buffer with near = 0, far = 255

Usually, every pixel stores a depth (or z) value, in 32 bit fixed point format

Z-Buffer Algorithm

– draw every polygon that we can't reject trivially

– If we find a pixel of a polygon that is closer to the front, we paint over whatever was behind it

– Inizialize the FrameBuffer/zBuffer

```
void Init_zBuffer() {
  int x, y;
  for (y = 0; y < YMAX; y++)
  for (x = 0; x < XMAX; x++) {
    FrameBuffer (x, y, BACKGROUND_VALUE);
    zBuffer (x, y, 1);
  }
 }
```

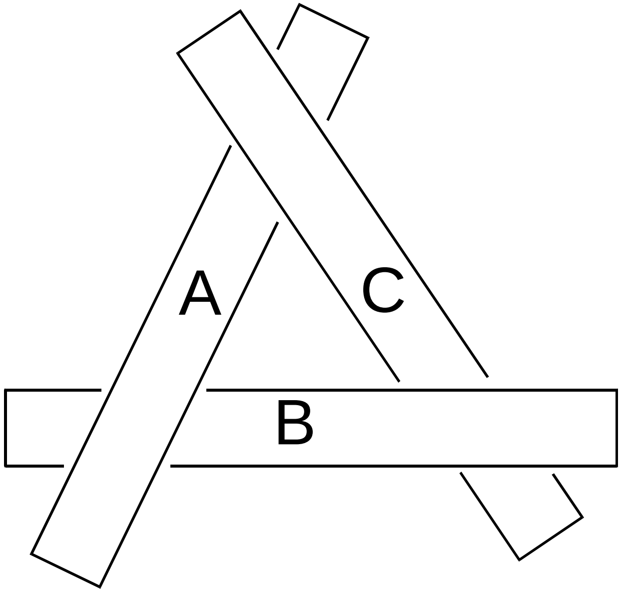
Rasterizer with Z-buffer pseudo code

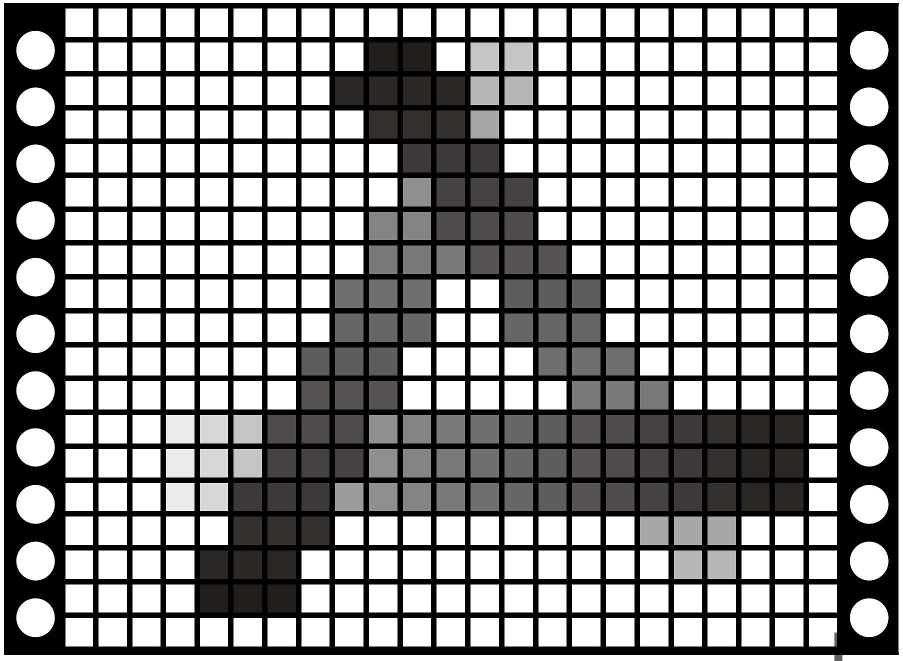
```
For every triangle
   Compute Projection of vertices
   Compute bbox, clip bbox to screen limits
   Setup 3 line equations
   For all pixels in bbox
        Increment line equations
        If all line equations<0 //pixel [x,y] in triangle
               Compute barycentric coordinates
               Compute currentZ
               Compute currentColor
               If currentZ < zBuffer[x,y] //pixel is visible
                      Framebuffer[x,y]=currentColor
                      zBuffer[x,y]=currentZ
```

Z-Buffer Pros

- Simplicity lends itself well to hardware implementations: FAST -- used by all graphics cards
- Polygons do not have to be compared in any particular order: no presorting in *z* necessary
- Only consider one polygon at a time
- Z-buffer can be stored w/ an image; allows you to correctly composite multiple images w/o having to merge models
  - great for incremental addition to a complex scene
- Can be used for non-polygonal surfaces, CSGs (intersect, union, difference), and any $z = f(x,y)$

Works for hard cases!





MIT EECS 6.837, Cutler and Durand

Serena Morigi

Dipartimento di Matematica

serena.morigi@unibo.it

http://www.dm.unibo.it/~morigi