Chapter 7

Visible surface detection methods

7.1 Overview

Generally, any procedure that eliminates those portions of a picture that are either inside or outside of a specified region of space is referred to as a **clipping algorithm** or simply **clipping**. Usually a clipping region is a rectangle, although we could use any shape for a clipping application.

The most common application of clipping is in the viewing pipeline, where clipping is applied to extract a designated portion of a scene (either two-dimensional or three-dimensional) for display on an output device. Clipping methods are also used to anti-alias object boundaries, to construct objects using solid-modeling methods, to manage multi-window environments, etc.

7.1 Overview

Clipping algorithms are applied in two-dimensional viewing procedures to identify those parts of a picture that are within the clipping window (i.e. viewport). Everything outside the clipping window is then eliminated from the scene description that is transferred to the output device for display. An efficient implementation of clipping in the viewing pipeline is to apply the algorithms to the normalized boundaries of the clipping window. This reduces calculations, because all geometric and viewing transformation matrices can be concatenated and applied to a scene description before clipping is carried out. The clipped scene can then be transferred to screen coordinates for final processing.

7.1 Overview

Pipelines

Graphics hardware uses a pipelined approach to process vertices and convert primitives into the final image. The pipeline basically involves the following steps:

- Modeling
- Geometry Processing
- Rasterization
- Fragment Processing

7.1 Overview

Modeling

The conversion of analog (real world) objects into discrete data

- i.e. creating vertices and connectivity via range scanning

The design of a complex structure from simpler primitives

- i.e. architecture and engineering designs

Done Offline

- We will ignore this step for now

7.1 Overview

Application programmer pipes modeling output into…

Geometry Processing

- Animate objects
- Move objects into camera space
- Project objects into device coordinates
- Clip objects external to viewing window

7.1 Overview

Rasterization

- Conversion of geometry in device coordinates into fragments (or pixels) in screen coordinates
- After this step there is no notion of a “polygon”, just fragments

7.1 Overview

Fragment Processing

- Texture lookups
- Coloring
- Programmable GPU steps

7.1 Overview

These last 3 steps need to be *FAST*

- Developed 20-40 years ago… but little has changed
- Efficient memory use speeds things up
  - Cache, cache, cache
- Integers and bit ops over floating point
- Fewer bits usually faster
  - float over double, half over float
- Parallel processing

7.1 Overview

Rasterization is very expensive

- More or less linear w/ number of fragments created
- Consists of adds, rounding and logic branches *per pixel*
- Only rasterize objects that are in viewable region

A few operations now needed to remove invisible onjects saves many later.

7.1 Overview

Geometry Processing

- Apply modelview and projection matrix.
- Not all primitives map to inside window
  - *Cull* those that are completely outside
  - *Clip* those that are partially inside
- 2D vs. 3D
  - Projection plane v. projection cube
  - Clipping can occur in either space
  - Choice of visible surface algorithm used forces one or the other

7.1 Overview

Clipping algorithms are available for basic primitives used in computer graphics, such as

- Point clipping
- Line clipping (straight-line segments)
- Fill-area clipping (polygons)
- Curve clipping
- Text clipping

In the following, we will assume that the clipping region is a rectangular window with boundary edges at x_{min}, x_{max}, y_{min}, and y_{max}.

7.2 Point clipping

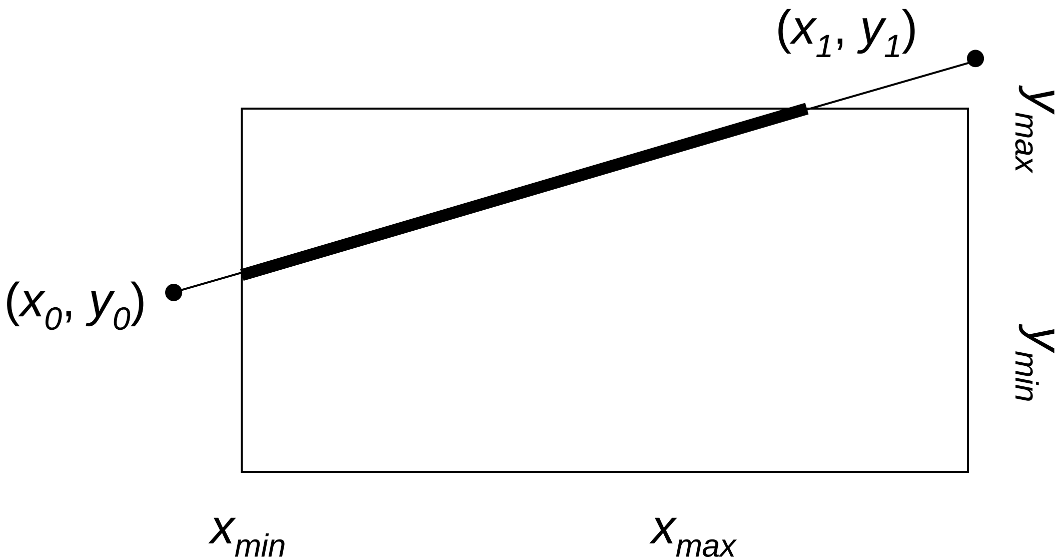
Since the projection of a point P results in coordinates (x, y), we can easily identify the necessary equations for a clipping algorithm:

$$x_{min} < x < x_{max}, \; y_{min} < y < y_{max}$$

Point clipping can be useful for particle systems, such as smoke simulation or cloud modeling.

7.3 Line clipping

Line clipping against rectangles



The problem: Given a set of 2D lines or polygons and a window, clip the lines or polygons to their regions that are *inside* the window.

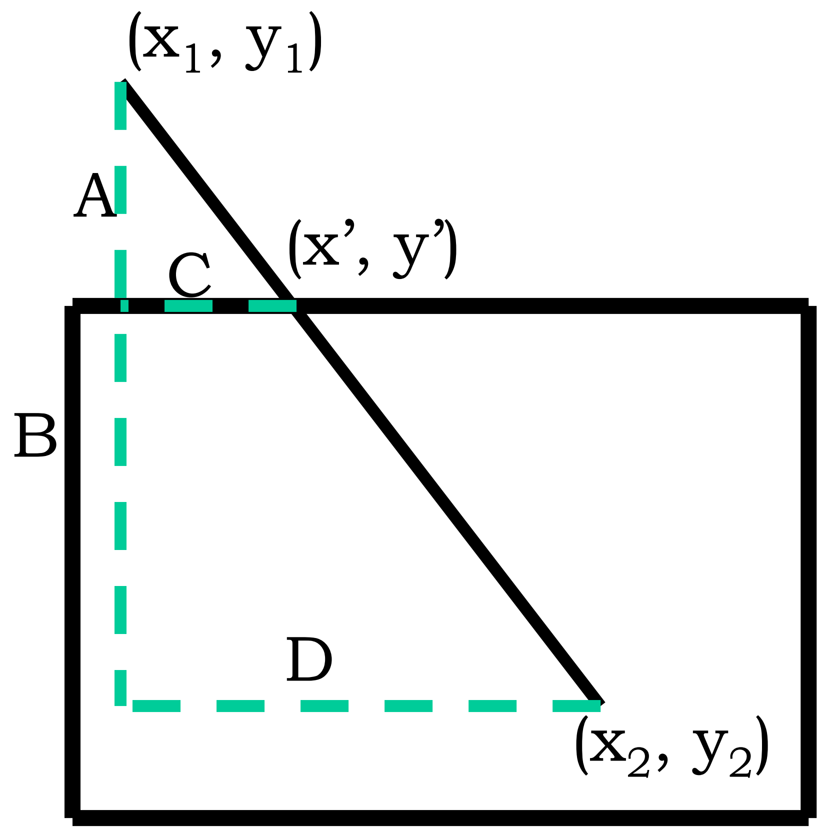
7.3 Line clipping

Direct approach

Clip a line against 1 edge of a square

Similar Triangles

- A/B = C/D
- Which do we know?
- B = $(y_1 - y_2)$
- D = $(x_1 - x_2)$
- A = $(y_1 - y_{max})$
- C = AD/B
- $(x', y') = (x_1 + C, y_{max})$



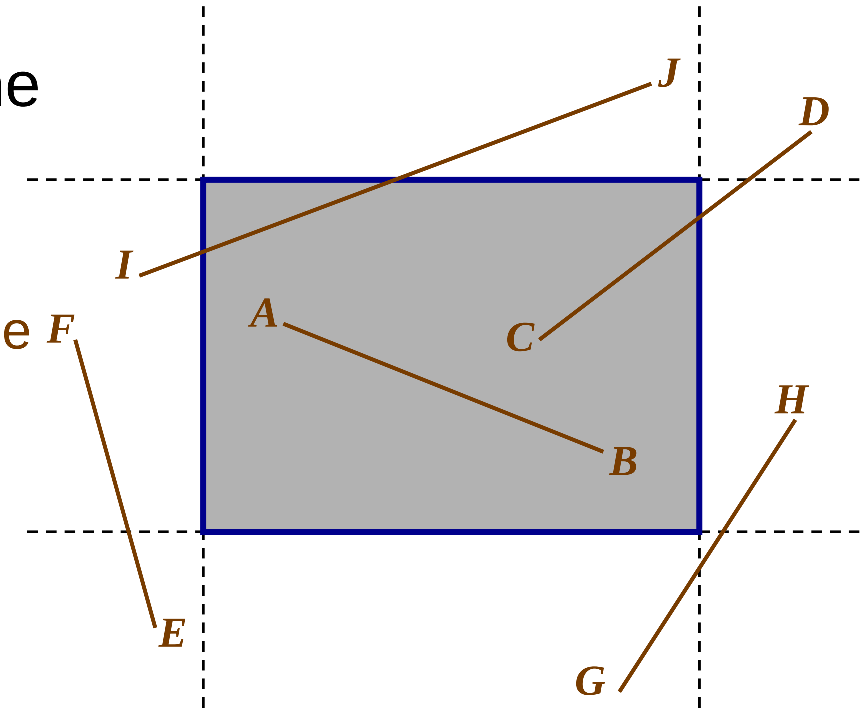
7.3 Line clipping

- Similarly handled for the other cases
- Extends easily to 3D
- EXPENSIVE! (below for 2D)
  - 4 floating point additions/subtractions
  - 2 floating point multiplications
  - 1 floating point div
  - 4 times (for each edge!)
- We need to save ourselves some operations

7.3 Line clipping

Possible Configurations

- Both endpoints are inside the region (line ***AB***)
  - No clipping necessary
- One endpoint in, one out (line ***CD***)
  - Clip at intersection point
- Both endpoints outside the region:
  - No intersection (lines ***EF***, ***GH***)
  - Line intersects the region (line ***IJ***)
    - Clip line at both intersection points

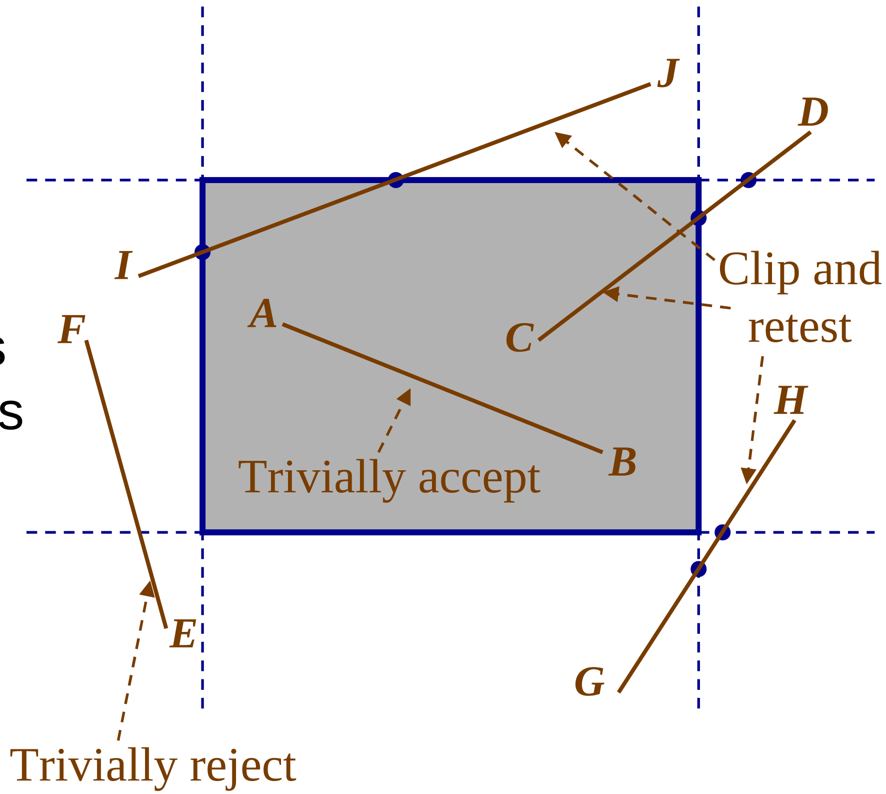


7.3 Line clipping

Cohen-Sutherland

Basic algorithm:

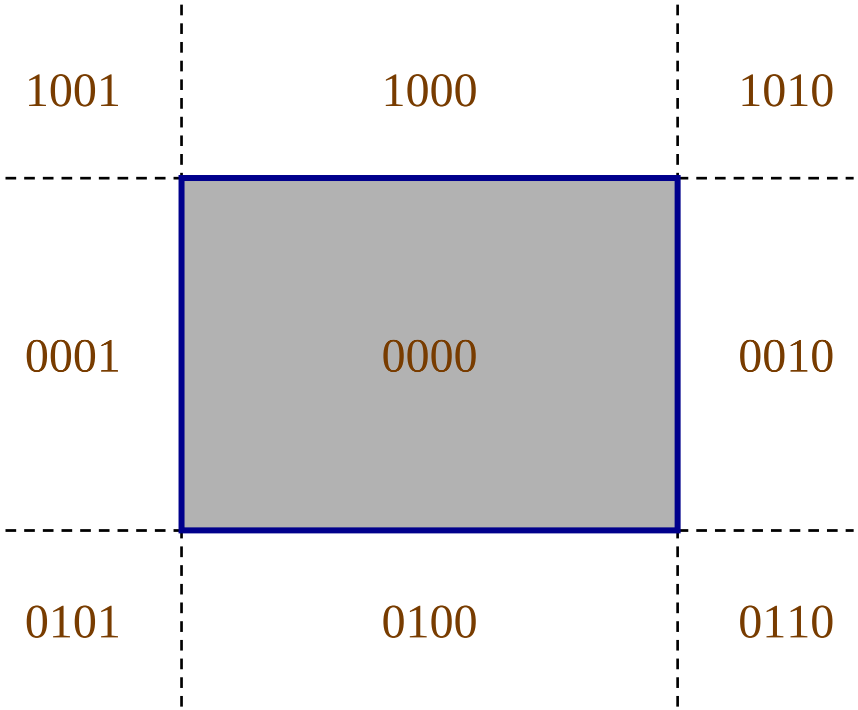
- Accept (and draw) lines that have both endpoints inside the region
- Reject (and don't draw) lines that have both endpoints less than x_{min} or y_{min} or greater than x_{max} or y_{max}
- Clip the remaining lines at a region boundary and repeat steps 1 and 2 on the clipped line segments



7.3 Line clipping

Assign 4-bit code to each endpoint corresponding to its position relative to region:

First	bit (1000): if $y > y_{max}$
Second	bit (0100): if $y < y_{min}$
Third	bit (0010): if $x > x_{max}$
Fourth	bit (0001): if $x < x_{min}$



Test:

if $code_0$ OR $code_1$ = 0000
accept (draw)
else if $code_0$ AND $code_1 \neq 0000$
reject (don't draw)
else clip and retest

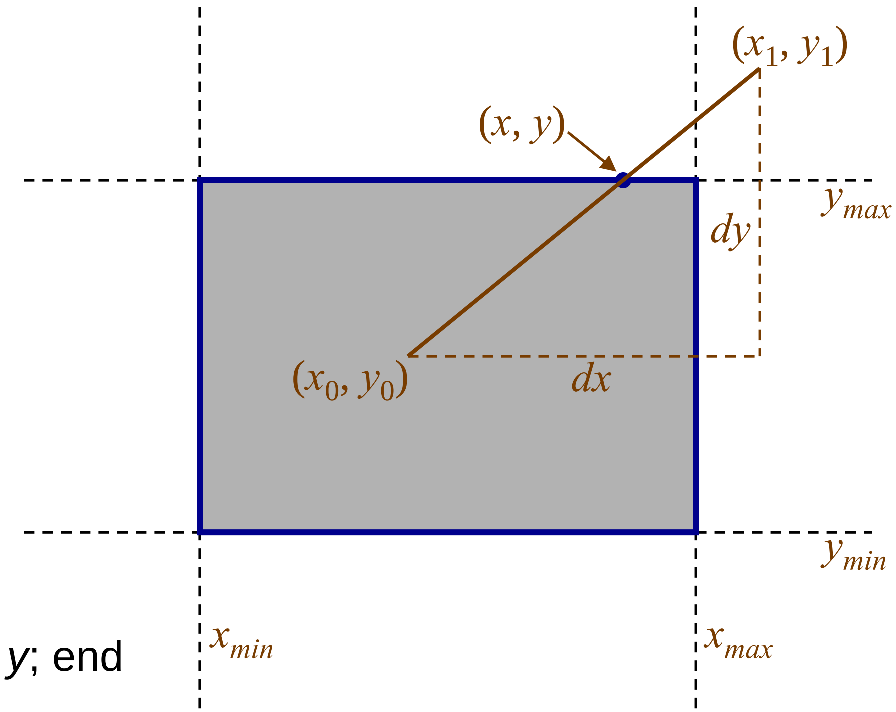
7.3 Line clipping

Intersection algorithm:

if $code_0 \neq 0000$ then $code = code_0$
else $code = code_1$

$dx = x_1 - x_0$; $dy = y_1 - y_0$
if *code* AND 1000 then begin // y_{max}
    $x = x_0 + dx * (y_{max} - y_0) / dy$; $y = y_{max}$
end
else if *code* AND 0100 then begin // y_{min}
    $x = x_0 + dx * (y_{min} - y_0) / dy$; $y = y_{min}$
end
else if *code* AND 0010 then begin // x_{max}
    $y = y_0 + dy * (x_{max} - x_0) / dx$; $x = x_{max}$
end
else begin // x_{min}
    $y = y_0 + dy * (x_{min} - x_0) / dx$; $x = x_{min}$
end

if $code = code_0$ then begin $x_0 = x$; $y_0 = y$; end
else begin $x_1 = x$; $y_1 = y$; end



7.3 Line clipping

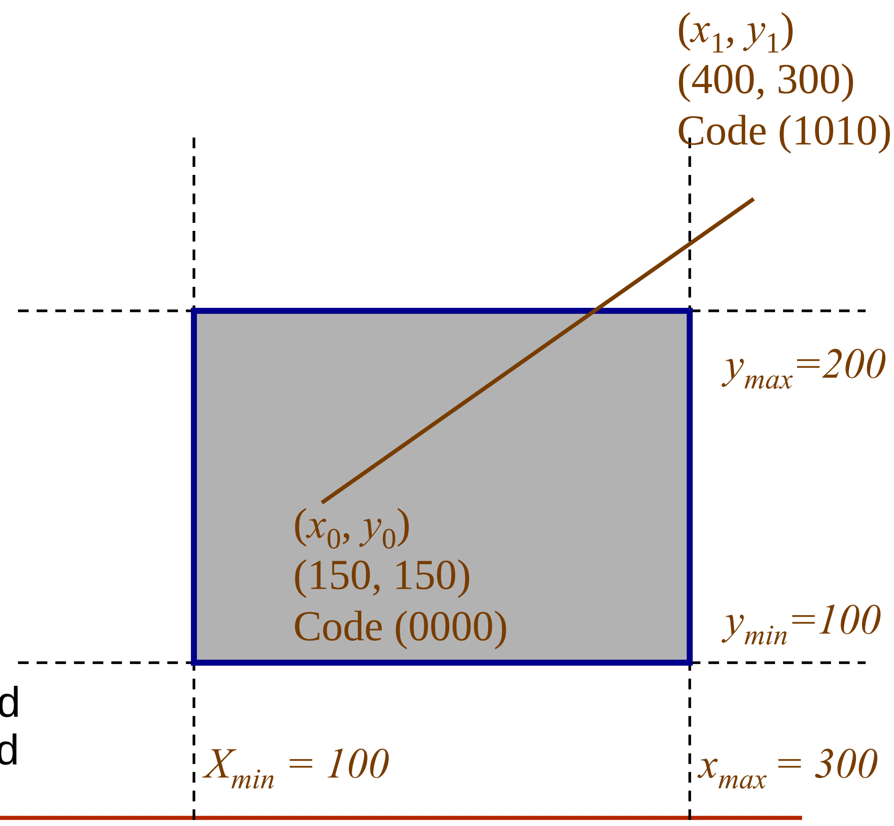
Intersection algorithm:

if $code_0 \neq 0000$ then $code = code_0$
else $code = code_1$

$dx = x_1 - x_0$; $dy = y_1 - y_0$
if *code* AND 1000 then begin // y_{max}
    $x = x_0 + dx * (y_{max} - y_0) / dy$; $y = y_{max}$
end
else if *code* AND 0100 then begin // y_{min}
    $x = x_0 + dx * (y_{min} - y_0) / dy$; $y = y_{min}$
end
else if *code* AND 0010 then begin // x_{max}
    $y = y_0 + dy * (x_{max} - x_0) / dx$; $x = x_{max}$
end
else begin // x_{min}
    $y = y_0 + dy * (x_{min} - x_0) / dx$; $x = x_{min}$
end

if $code = code_0$ then begin $x_0 = x$; $y_0 = y$; end
else begin $x_1 = x$; $y_1 = y$; end

Code	dx	dy	x	y



7.3 Line clipping

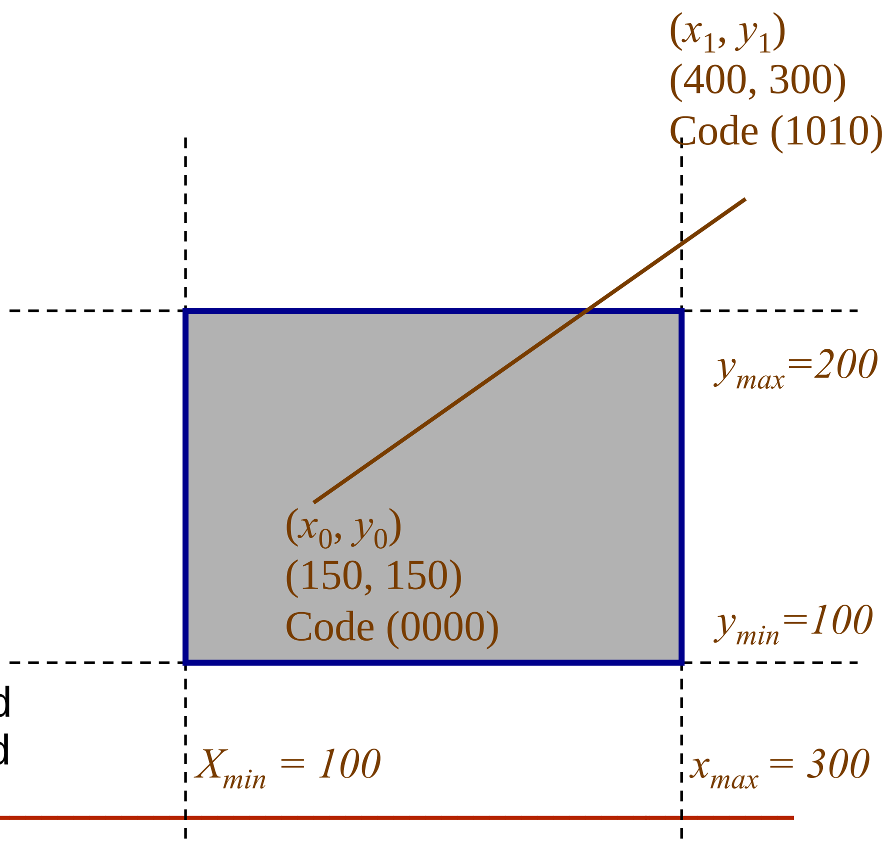
Intersection algorithm:

Code	dx	dy	x	y

if $code_0 \neq 0000$ then $code = code_0$
else $code = code_1$

$dx = x_1 - x_0$; $dy = y_1 - y_0$
if *code* AND 1000 then begin // y_{max}
  $x = x_0 + dx * (y_{max} - y_0) / dy$; $y = y_{max}$
end
else if *code* AND 0100 then begin // y_{min}
  $x = x_0 + dx * (y_{min} - y_0) / dy$; $y = y_{min}$
end
else if *code* AND 0010 then begin // x_{max}
  $y = y_0 + dy * (x_{max} - x_0) / dx$; $x = x_{max}$
end
else begin // x_{min}
  $y = y_0 + dy * (x_{min} - x_0) / dx$; $x = x_{min}$
end

if $code = code_0$ then begin $x_0 = x$; $y_0 = y$; end
else begin $x_1 = x$; $y_1 = y$; end



7.3 Line clipping

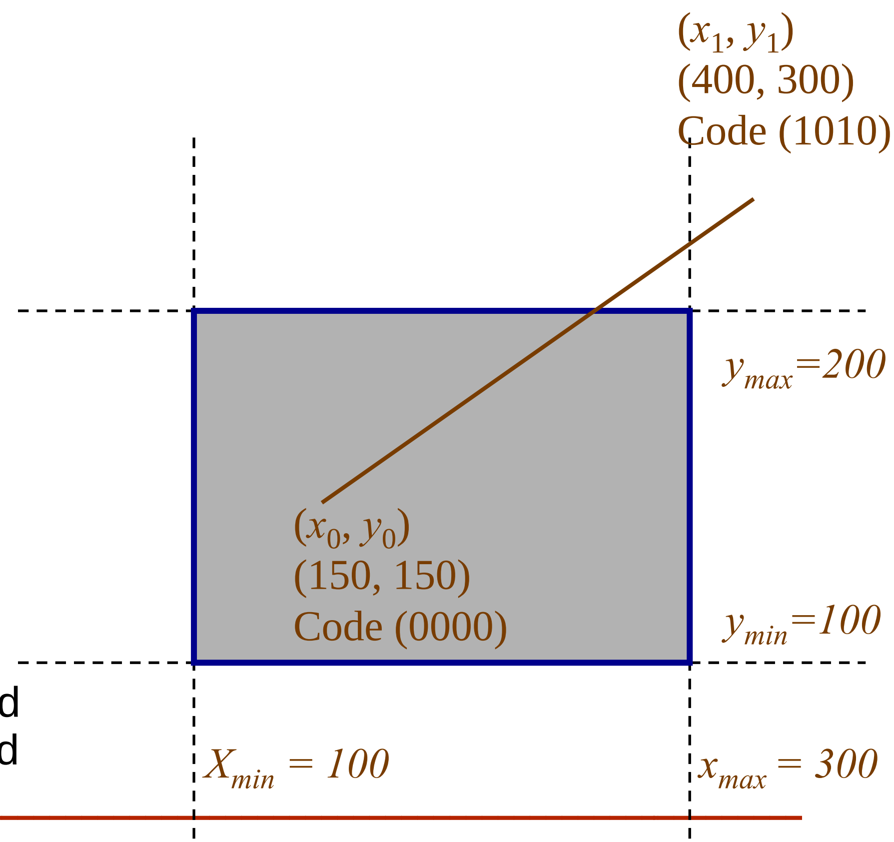
Intersection algorithm:

Code	dx	dy	x	y
1010				

if $code_0 \neq 0000$ then $code = code_0$
else $code = code_1$

$dx = x_1 - x_0$; $dy = y_1 - y_0$
if *code* AND 1000 then begin // y_{max}
 $x = x_0 + dx * (y_{max} - y_0) / dy$; $y = y_{max}$
end
else if *code* AND 0100 then begin // y_{min}
 $x = x_0 + dx * (y_{min} - y_0) / dy$; $y = y_{min}$
end
else if *code* AND 0010 then begin // x_{max}
 $y = y_0 + dy * (x_{max} - x_0) / dx$; $x = x_{max}$
end
else begin // x_{min}
 $y = y_0 + dy * (x_{min} - x_0) / dx$; $x = x_{min}$
end

if $code = code_0$ then begin $x_0 = x$; $y_0 = y$; end
else begin $x_1 = x$; $y_1 = y$; end



7.3 Line clipping

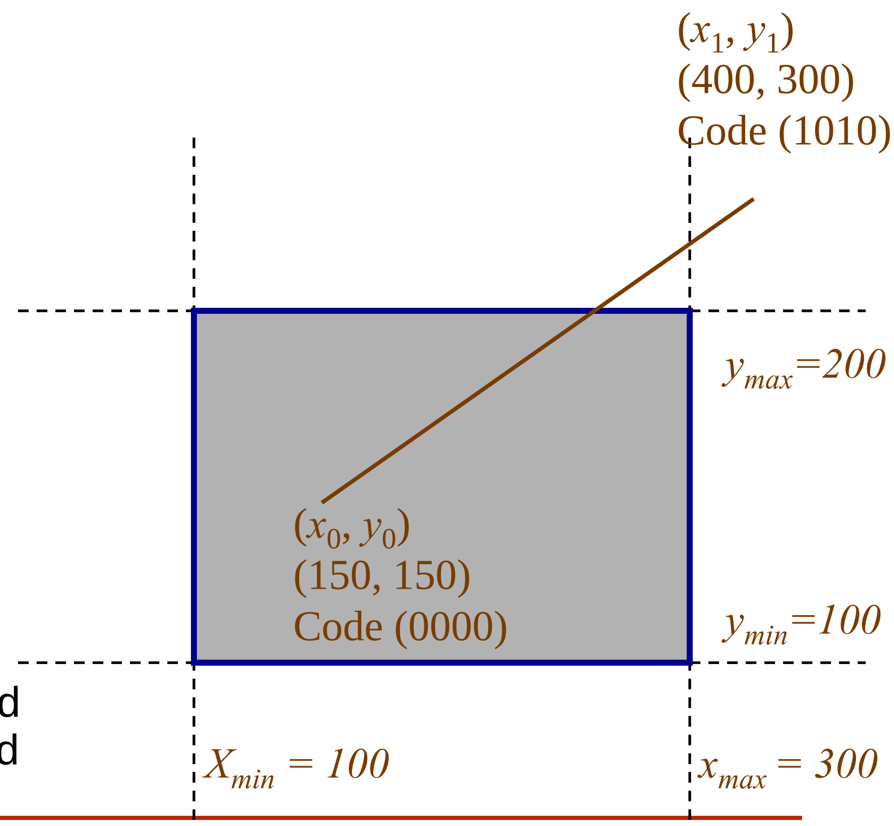
Intersection algorithm:

Code	dx	dy	x	y
1010	250	150		

if $code_0 \neq 0000$ then $code = code_0$
else $code = code_1$

$dx = x_1 - x_0$; $dy = y_1 - y_0$
if *code* AND 1000 then begin // y_{max}
$x = x_0 + dx * (y_{max} - y_0) / dy$; $y = y_{max}$
end
else if *code* AND 0100 then begin // y_{min}
$x = x_0 + dx * (y_{min} - y_0) / dy$; $y = y_{min}$
end
else if *code* AND 0010 then begin // x_{max}
$y = y_0 + dy * (x_{max} - x_0) / dx$; $x = x_{max}$
end
else begin // x_{min}
$y = y_0 + dy * (x_{min} - x_0) / dx$; $x = x_{min}$
end

if $code = code_0$ then begin $x_0 = x$; $y_0 = y$; end
else begin $x_1 = x$; $y_1 = y$; end



7.3 Line clipping

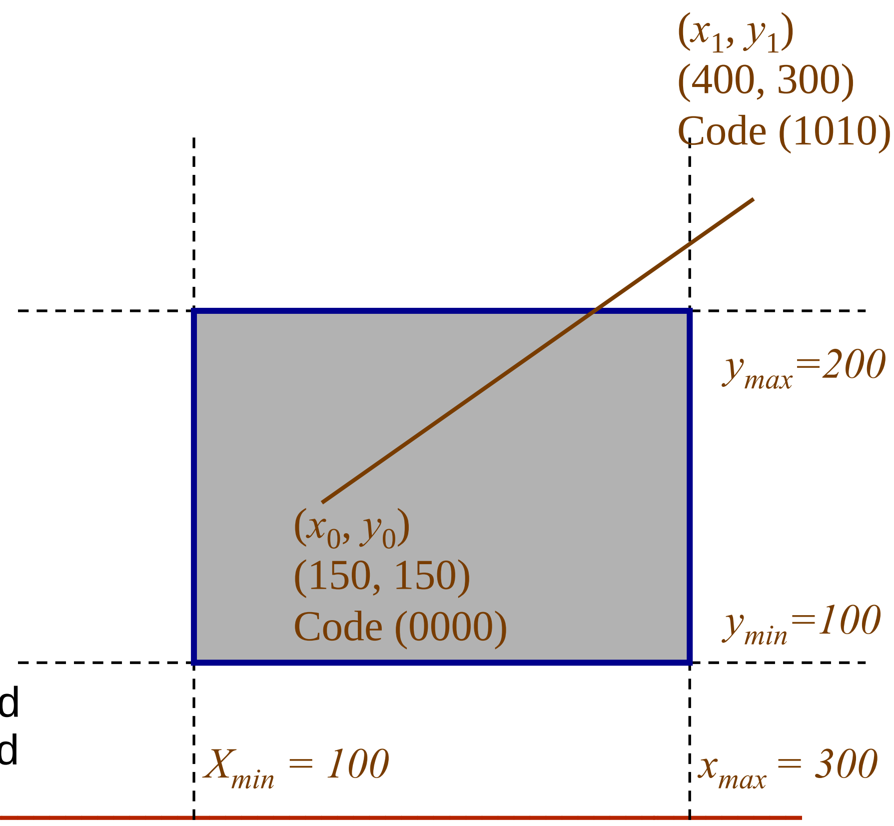
Intersection algorithm:

Code	dx	dy	x	y
1010	250	150		

if $code_0 \neq 0000$ then $code = code_0$
else $code = code_1$

$dx = x_1 - x_0$; $dy = y_1 - y_0$
if *code* AND 1000 then begin // y_{max}
    $x = x_0 + dx * (y_{max} - y_0) / dy$; $y = y_{max}$
end
else if *code* AND 0100 then begin // y_{min}
    $x = x_0 + dx * (y_{min} - y_0) / dy$; $y = y_{min}$
end
else if *code* AND 0010 then begin // x_{max}
    $y = y_0 + dy * (x_{max} - x_0) / dx$; $x = x_{max}$
end
else begin // x_{min}
    $y = y_0 + dy * (x_{min} - x_0) / dx$; $x = x_{min}$
end

if $code = code_0$ then begin $x_0 = x$; $y_0 = y$; end
else begin $x_1 = x$; $y_1 = y$; end



7.3 Line clipping

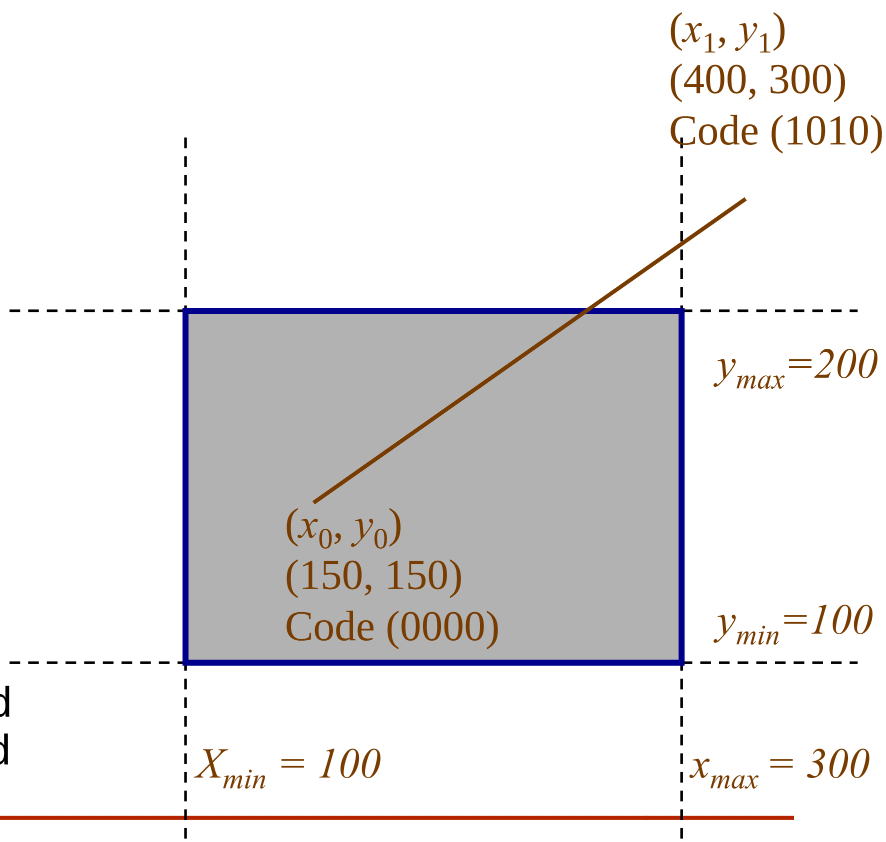
Intersection algorithm:

Code	dx	dy	x	y
1010	250	150	233	200

if $code_0 \neq 0000$ then $code = code_0$
else $code = code_1$

$dx = x_1 - x_0$; $dy = y_1 - y_0$
if *code* AND 1000 then begin // y_{max}
$x = x_0 + dx * (y_{max} - y_0) / dy$; $y = y_{max}$
end
else if *code* AND 0100 then begin // y_{min}
$x = x_0 + dx * (y_{min} - y_0) / dy$; $y = y_{min}$
end
else if *code* AND 0010 then begin // x_{max}
$y = y_0 + dy * (x_{max} - x_0) / dx$; $x = x_{max}$
end
else begin // x_{min}
$y = y_0 + dy * (x_{min} - x_0) / dx$; $x = x_{min}$
end

if $code = code_0$ then begin $x_0 = x$; $y_0 = y$; end
else begin $x_1 = x$; $y_1 = y$; end



7.3 Line clipping

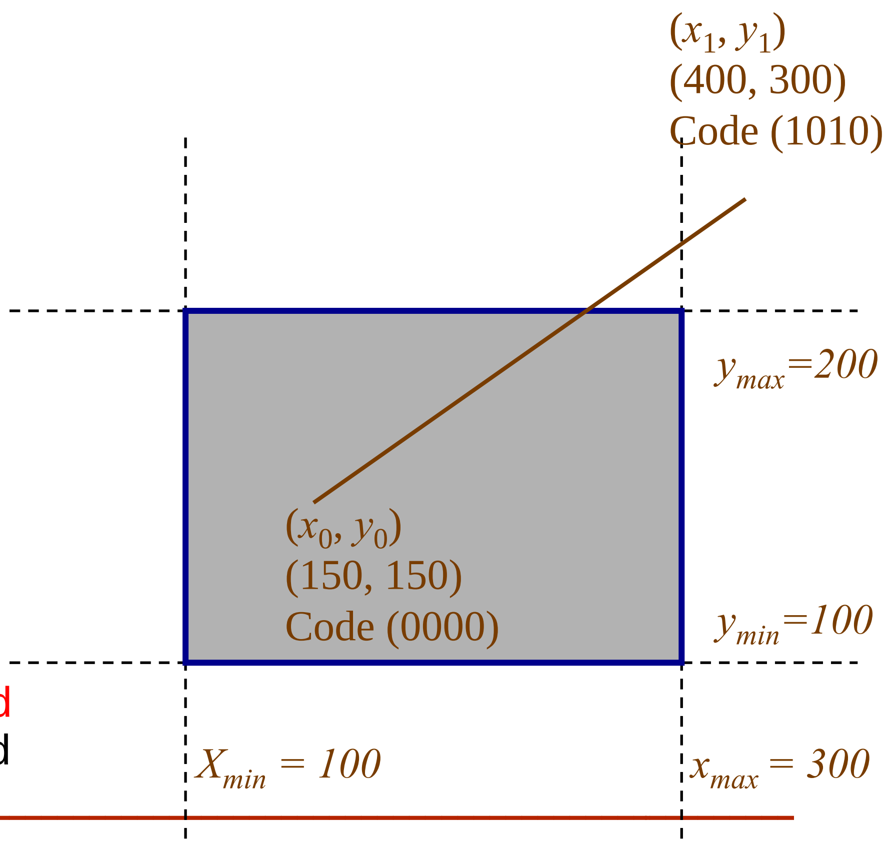
Intersection algorithm:

Code	dx	dy	x	y
1010	250	150	233	200

if $code_0 \neq 0000$ then $code = code_0$
else $code = code_1$

$dx = x_1 - x_0$; $dy = y_1 - y_0$
if *code* AND 1000 then begin // y_{max}
    $x = x_0 + dx * (y_{max} - y_0) / dy$; $y = y_{max}$
end
else if *code* AND 0100 then begin // y_{min}
    $x = x_0 + dx * (y_{min} - y_0) / dy$; $y = y_{min}$
end
else if *code* AND 0010 then begin // x_{max}
    $y = y_0 + dy * (x_{max} - x_0) / dx$; $x = x_{max}$
end
else begin // x_{min}
    $y = y_0 + dy * (x_{min} - x_0) / dx$; $x = x_{min}$
end

if $code = code_0$ then begin $x_0 = x$; $y_0 = y$; end
else begin $x_1 = x$; $y_1 = y$; end



7.3 Line clipping

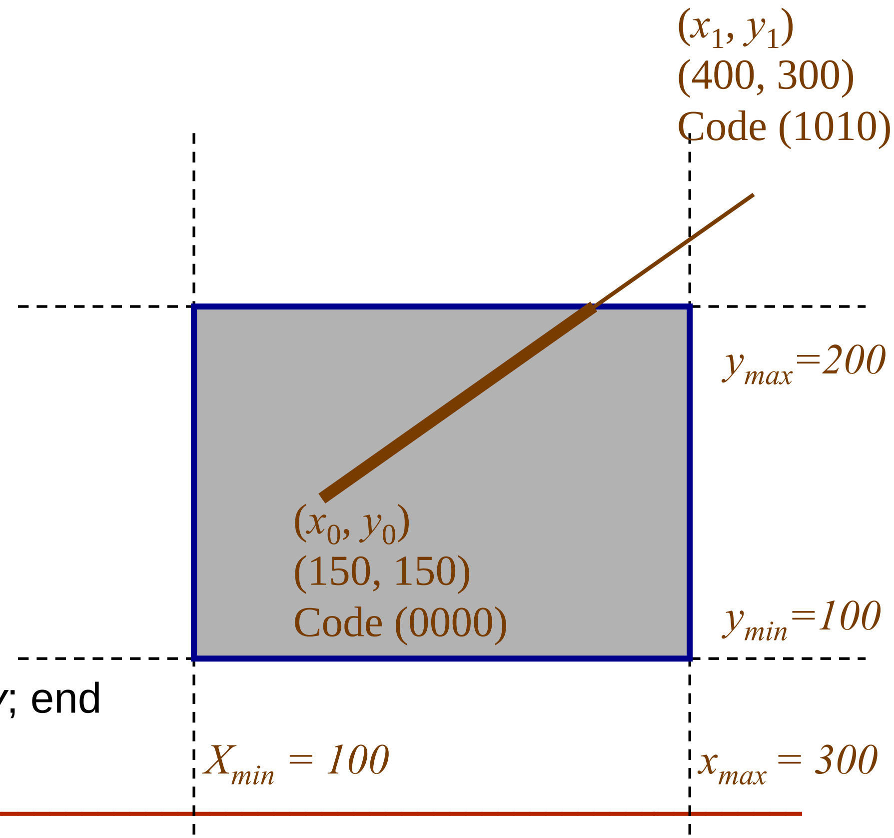
Intersection algorithm:

Code	dx	dy	x	y
1010	250	150	233	200

if $code_0 \neq 0000$ then $code = code_0$
else $code = code_1$

$dx = x_1 - x_0$; $dy = y_1 - y_0$
if *code* AND 1000 then begin // y_{max}
    $x = x_0 + dx * (y_{max} - y_0) / dy$; $y = y_{max}$
end
else if *code* AND 0100 then begin // y_{min}
    $x = x_0 + dx * (y_{min} - y_0) / dy$; $y = y_{min}$
end
else if *code* AND 0010 then begin // x_{max}
    $y = y_0 + dy * (x_{max} - x_0) / dx$; $x = x_{max}$
end
else begin // x_{min}
    $y = y_0 + dy * (x_{min} - x_0) / dx$; $x = x_{min}$
end

if $code = code_0$ then begin $x_0 = x$; $y_0 = y$; end
else begin $x_1 = x$; $y_1 = y$; end

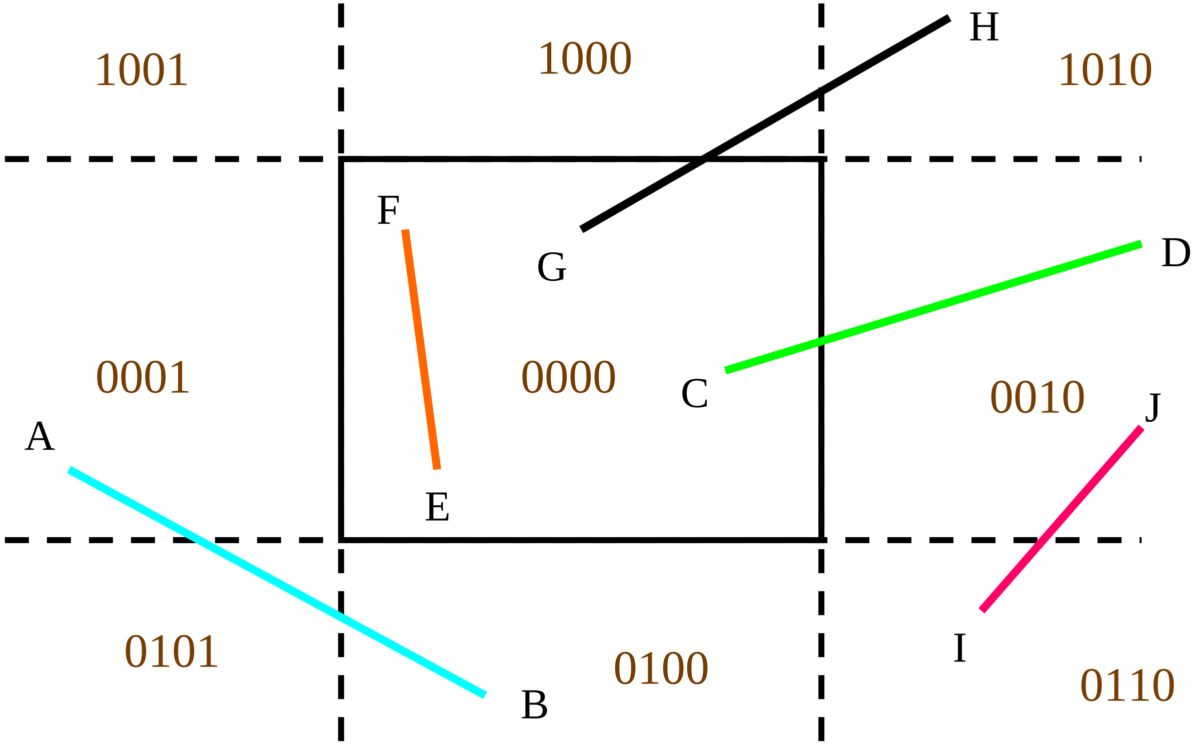


7.3 Line clipping

Cohen-Sutherland algorithm: summary

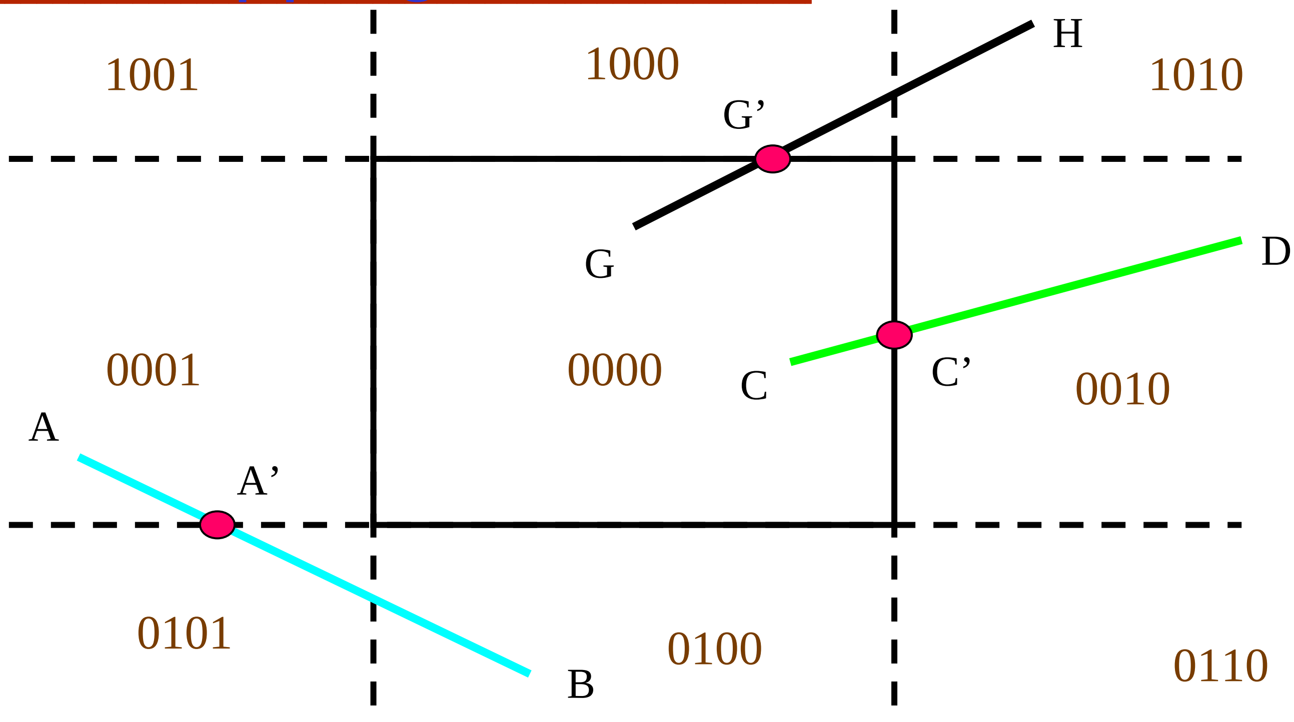
- Choose an endpoint outside the clipping region
- Using a consistent ordering (top to bottom, left to right) find a clipping border the line intersects
- Discard the portion of the line from the endpoint to the intersection point
- Set the new line to have as endpoints the new intersection point and the other original endpoint
- You may need to run this several times on a single line (e.g., a line that crosses multiple clip boundaries)

7.3 Line clipping



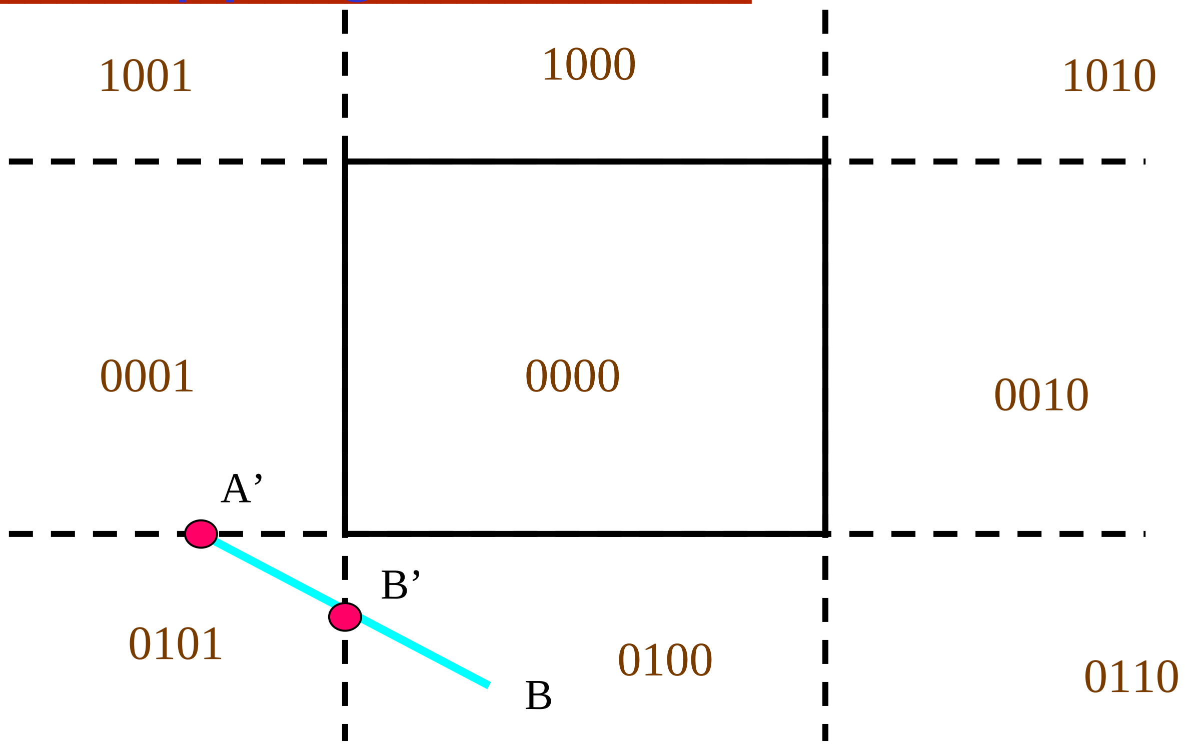
A 0001	C 0000	E 0000	G 0000	I 0110
B 0100	D 0010	F 0000	H 1010	J 0010
OR 0101	OR 0010	OR 0000	OR 1010	OR 0110
AND 0000	AND 0000	AND 0000	AND 0000	AND 0010
subdivide	**subdivide**	**accept**	**subdivide**	**reject**

7.3 Line clipping



A 0001	A' 0001	C 0000	C' 0000	G 0000	G' 0000
A' 0001	B 0100	C' 0000	D 1010	G' 0000	H 1010
	OR 0101	OR 0000		OR 0000	
remove	AND 0000	AND 0000	**remove**	AND 0000	**remove**
	subdivide	**accept**		**accept**	

7.3 Line clipping



A' 0001	B' 0100
B' 0100	B 0100
	OR 0100
remove	AND 0100
	reject

7.3 Line clipping

Liang-Barsky line clipping

To achieve faster clipping, we should do a little more testing before we actually compute the intersection.

Parametric definition of a line:

- $x = x_1 + u\Delta x$
- $y = y_1 + u\Delta y$
- $\Delta x = (x_2 - x_1)$, $\Delta y = (y_2 - y_1)$, $0 \leq u \leq 1$

Goal: find range of u for which x and y *both* inside the viewing window

7.3 Line clipping

Liang-Barsky line clipping (continued)

Mathematically, we need to find values for u that fulfill the following inequalities:

$$x_{min} \leq x_1 + u\Delta x \leq x_{max}$$
$$y_{min} \leq y_1 + u\Delta y \leq y_{max}$$

This can be rearranged to:

1: $u \cdot (-\Delta x) \leq (x_1 - x_{min})$

2: $u \cdot (\Delta x) \leq (x_{max} - x_1)$

3: $u \cdot (-\Delta y) \leq (y_1 - y_{min})$

4: $u \cdot (\Delta y) \leq (y_{max} - y_1)$

Or in general: $u \cdot (p_k) \leq (q_k)$

7.3 Line clipping

Liang-Barsky line clipping (continued)

Rules:

$p_k = 0$: the line is parallel to boundaries

If for that same k, $q_k < 0$, it's outside

Otherwise it's inside

$p_k < 0$: the line starts outside this boundary

$r_k = q_k/p_k$

$u_1 = \max(0, r_k, u_1)$

$p_k > 0$: the line starts inside the boundary

$r_k = q_k/p_k$

$u_2 = \min(1, r_k, u_2)$

If $u_1 > u_2$, the line is completely outside

7.3 Line clipping

Liang-Barsky line clipping (continued)

The algorithm also extends to 3D

- Add $z = z_1 + u\Delta z$ to the parametric description of a line
- Add 2 more p's and q's
- Still only 2 u's (since the line is still a 2-D primitive)

7.3 Line clipping

Liang-Barsky v. Cohen-Sutherland

- Generally, Liang-Barsky is more efficient
  - Requires only one division
  - Find intersection values for (x,y) only at end
- This depends, however, on the application
- Cohen-Sutherland may be easier to implement

7.3 Line clipping

Nicholl-Lee-Nicholl line clipping

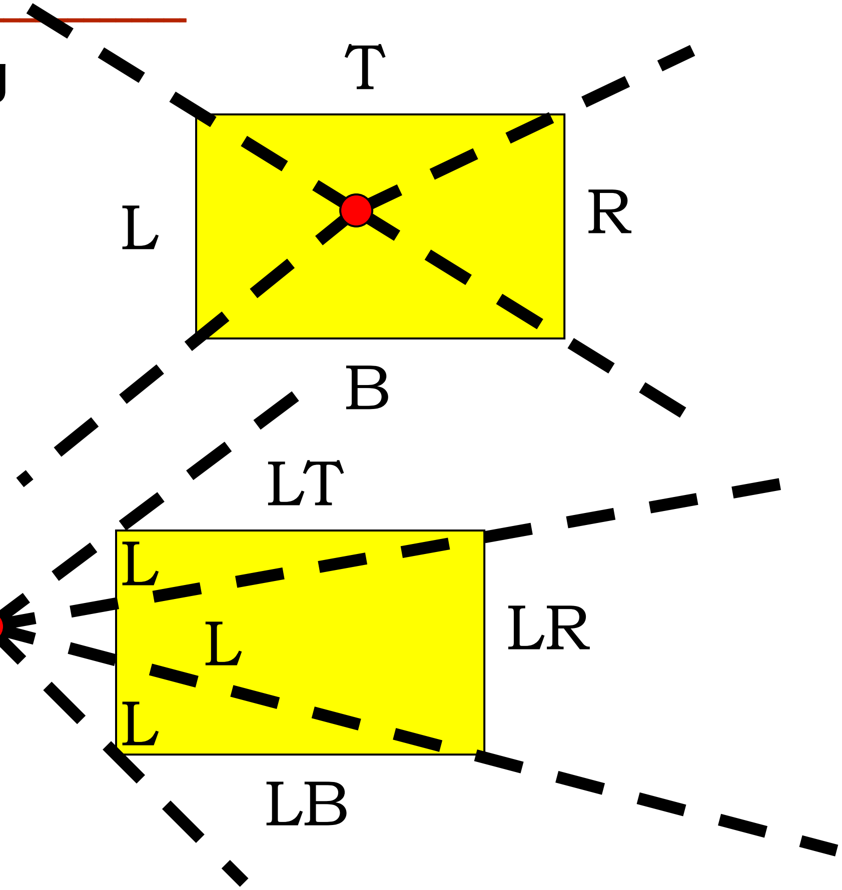
- This test is most complicated
- Also the fastest
- Only works well for 2D
- Quick overview here

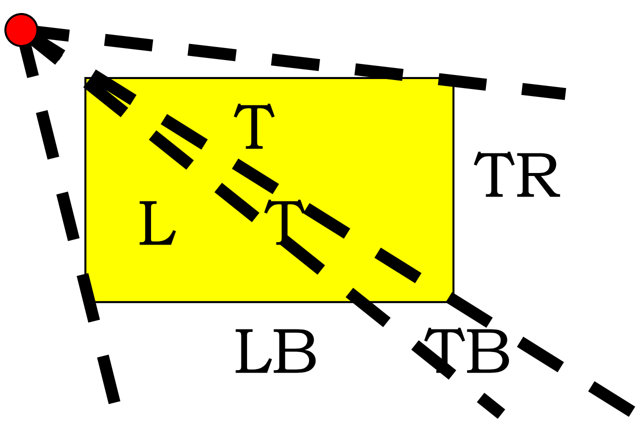
7.3 Line clipping

Nicholl-Lee-Nicholl line clipping

Divide the region based on the location of the first point p_1

- Case 1: p_1 inside
- Case 2: p_1 across edge
- Case 3: p_1 across corner





7.3 Line clipping

Nicholl-Lee-Nicholl Line Clipping

- Symmetry handles other cases
- Find slopes of the line and 4 region bounding lines
- Find which region P_2 is in
  - If not in any labeled, the line is discarded
- Subtractions, multiplies and divisions can be carefully used to minimum

7.3 Line clipping

A note on redundancy

Why present multiple forms of clipping?

- Why do you learn multiple sorts?
- Not always easy to do the fastest
- The fastest for the *general* case isn't always the fastest for every *specific* case
  - Mostly sorted list → bubble sort
- History repeats itself
  - You may need something similar in a different area. Grab the one that maps best.

7.4 Polygon clipping

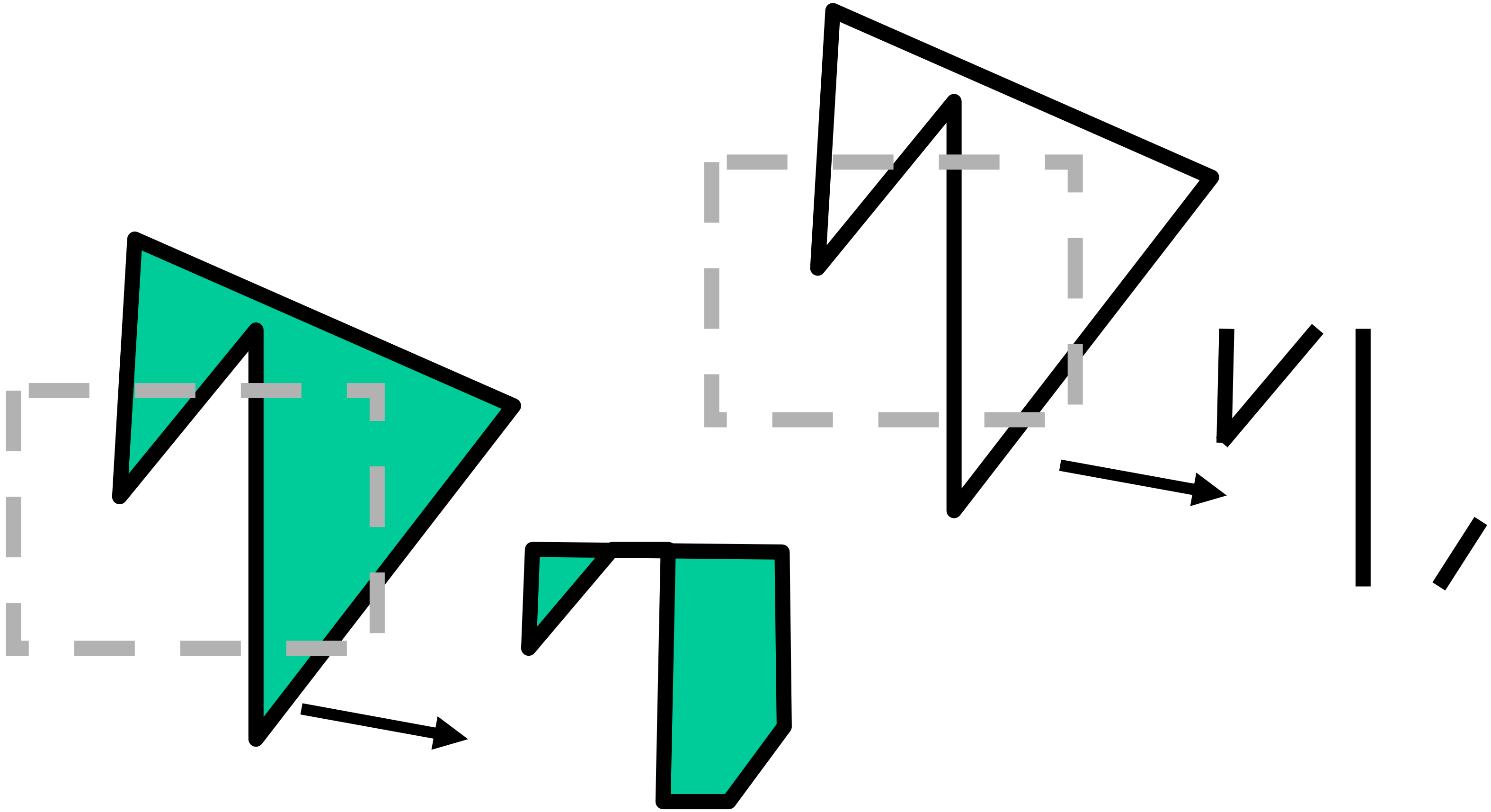
Clipping polygons is more complex than clipping the individual lines

- Input: polygon
- Output: original polygon, new polygon, or nothing

Since polygons are bounded by line segments, can we just use line clipping?

7.4 Polygon clipping

Why can't we just clip the lines of a polygon?

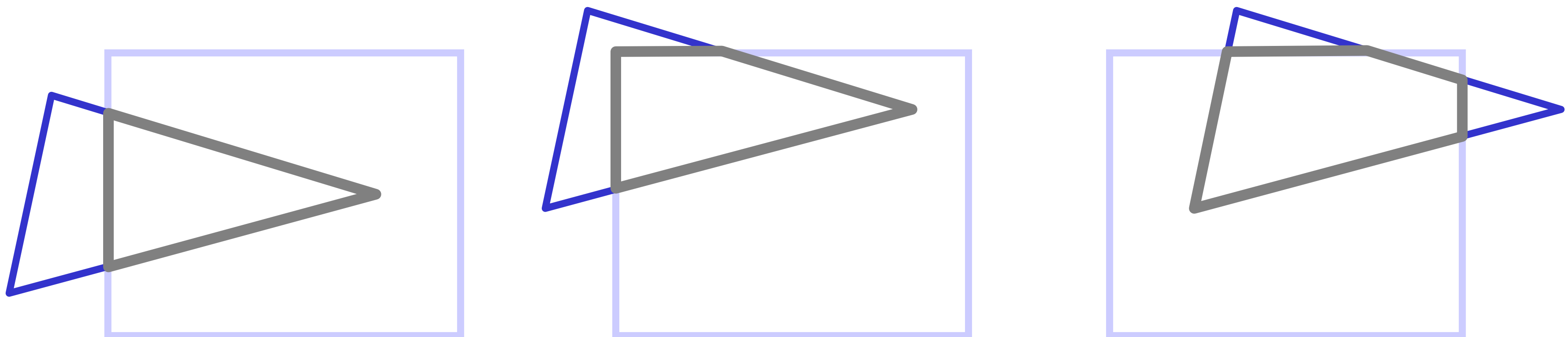


7.4 Polygon clipping

Why Is Clipping Hard?

What happens to a triangle during clipping?

Possible outcomes:



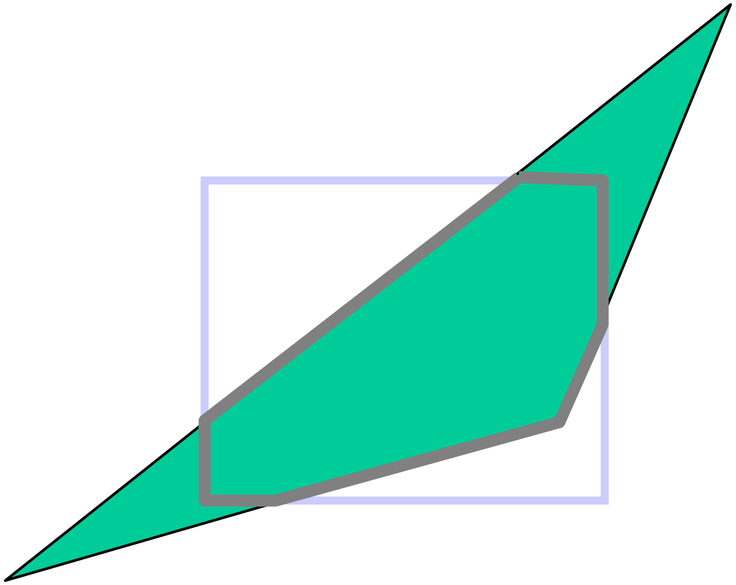
triangle ⇨ triangle   triangle ⇨ quad   triangle ⇨ 5-gon

How many edges can a clipped triangle have?

7.4 Polygon clipping

How many edges?

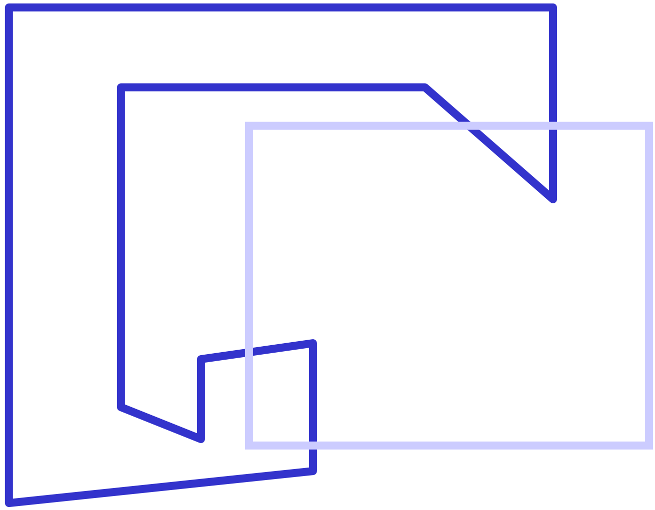
Seven…



7.4 Polygon clipping

Why Is Clipping Hard?

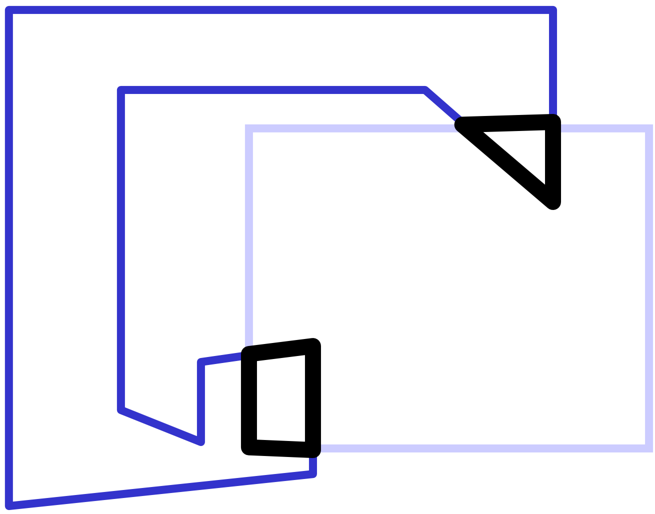
A really tough case:



7.4 Polygon clipping

Why Is Clipping Hard?

A really tough case:



concave polygon ⇨ multiple polygons

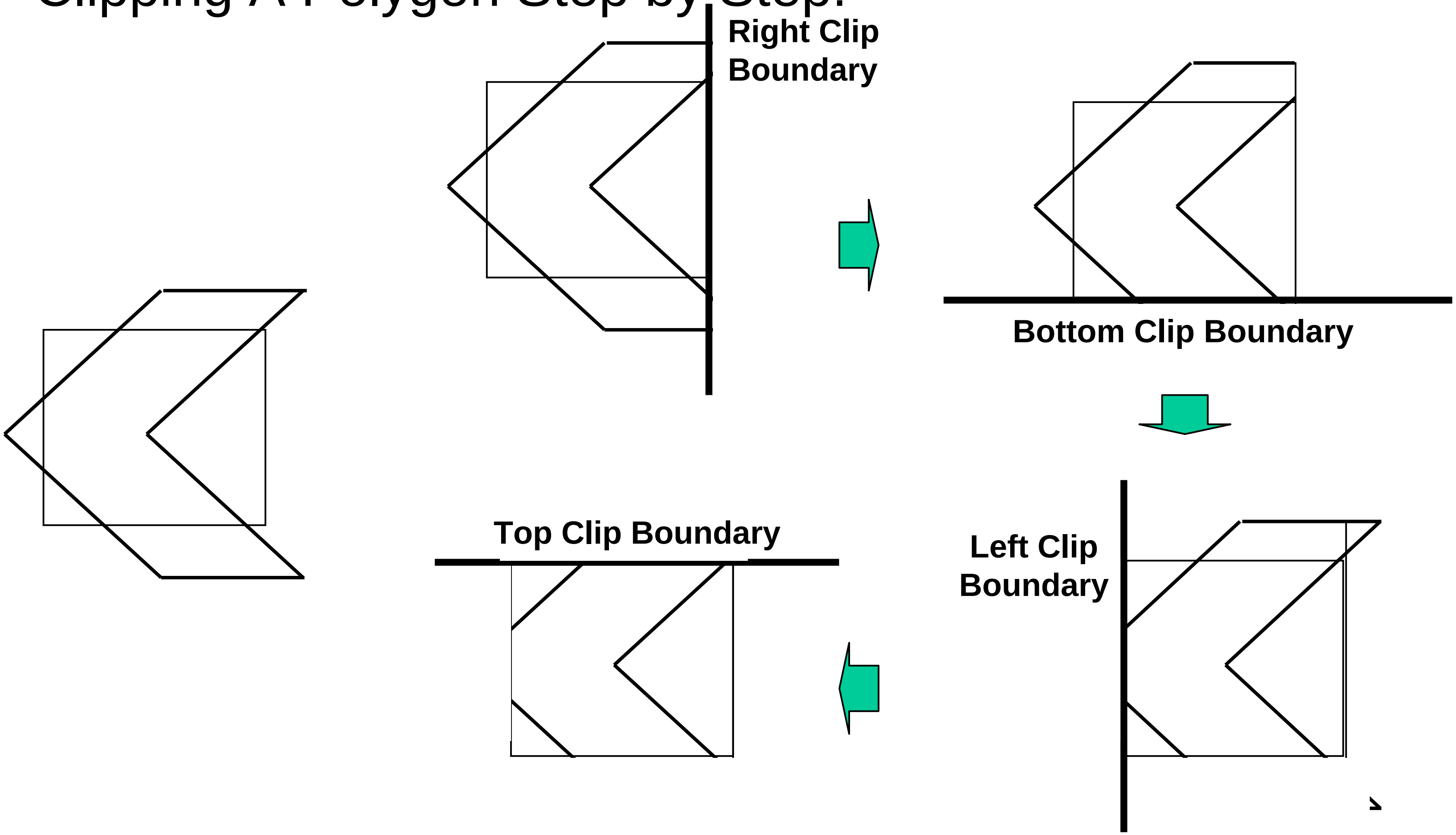
7.4 Polygon clipping

Sutherland-Hodgeman algorithm (A divide-and-conquer strategy)

- Polygons can be clipped against each edge of the window one at a time. 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.
- Note that the number of vertices usually changes and will often increases.

7.4 Polygon clipping

Clipping A Polygon Step by Step:



7.4 Polygon clipping

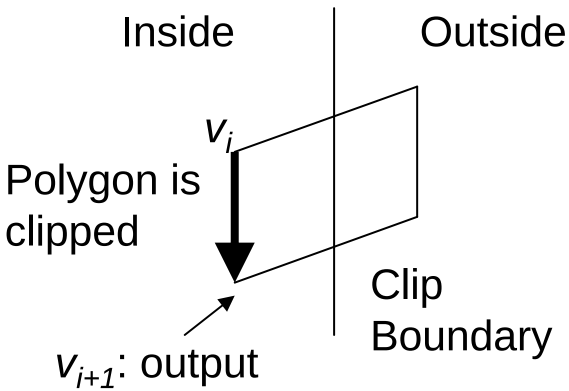
Sutherland-Hodgeman Algorithm

Note the difference between this strategy and the Cohen-Sutherland algorithm for clipping a line: the polygon clipper clips against each window edge in succession, whereas the line clipper is a recursive algorithm.

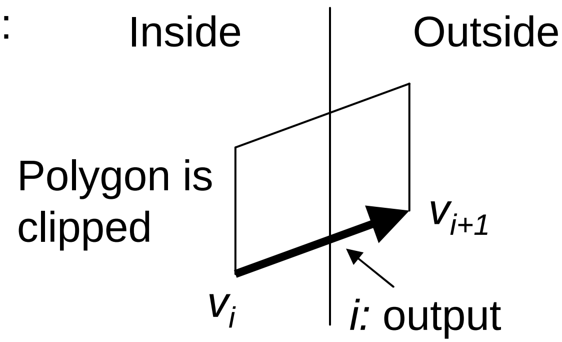
Given a polygon with n vertices, $v_1, v_2, \ldots, v_n$, the algorithm clips the polygon against a single, infinite clip edge and outputs another series of vertices defining the clipped polygon. In the next pass, the partially clipped polygon is then clipped against the second clip edge, and so on. Let us consider the polygon edge from vertex v_i to vertex v_{i+1}. Assuming that the start point v_i has been dealt with in the previous iteration, four cases will appear.

7.4 Polygon clipping

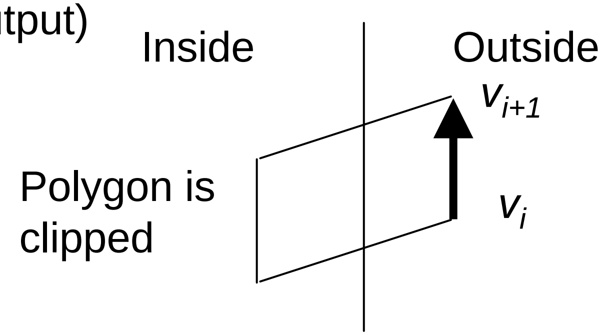
Case 1:



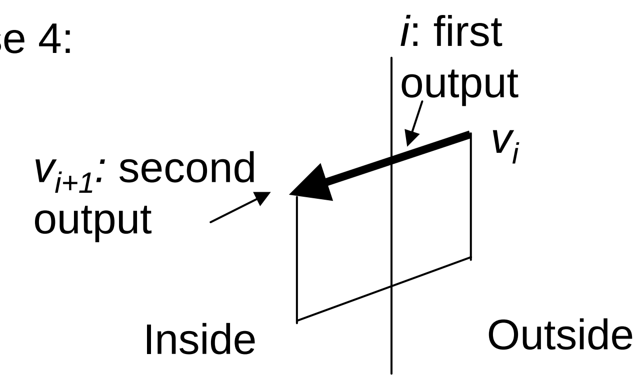
Case 2:



Case 3:
(no output)



Case 4:



7.4 Polygon clipping

Sutherland-Hodgeman Clipping

Four cases:

- *s* inside plane and *p* inside plane
  - Add *p* to output
  - Note: *s* has already been added
- *s* inside plane and *p* outside plane
  - Find intersection point *i*
  - Add *i* to output
- *s* outside plane and *p* outside plane
  - Add nothing
- *s* outside plane and *p* inside plane
  - Find intersection point *i*
  - Add *i* to output, followed by *p*

Point-to-Plane test

Point-to-Plane test

A very general test to determine if a point p is “inside” a plane P, defined by q and n:

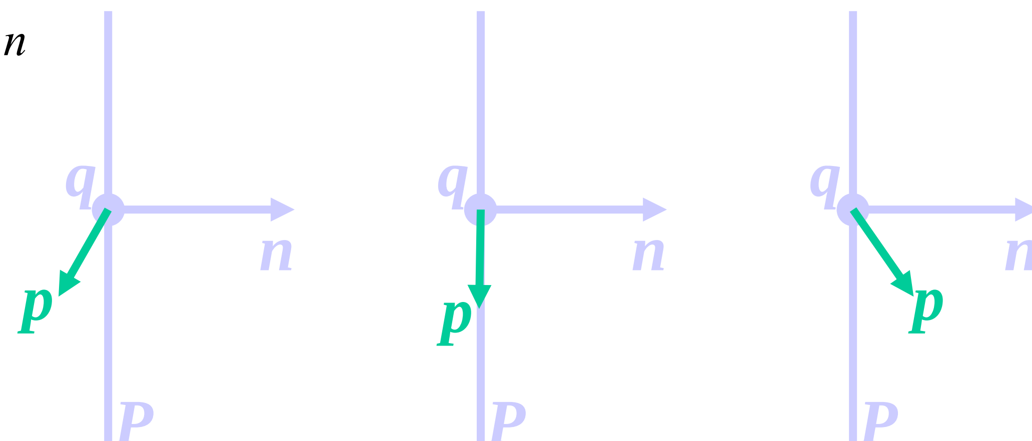
$(p - q) \bullet n < 0$: p inside P

$(p - q) \bullet n = 0$: p on P

$(p - q) \bullet n > 0$: p outside P

Remember: $\mathbf{p} \bullet n = |p|\ |n|\ cos\ (q)$

θ = angle between p and n



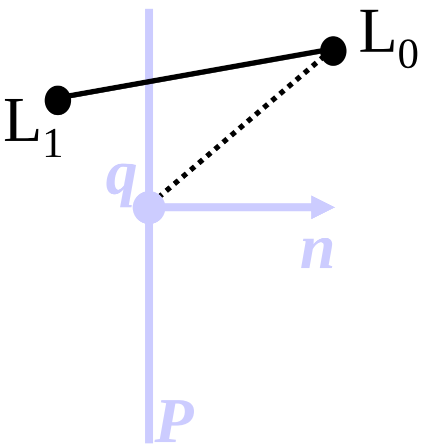
Finding Line-Plane Intersections

Edge intersects plane P where $E(t)$ is on P

- q is a point on P
- n is normal to P

$$(\mathbf{L}(t) - \mathbf{q}) \bullet \mathbf{n} = 0$$

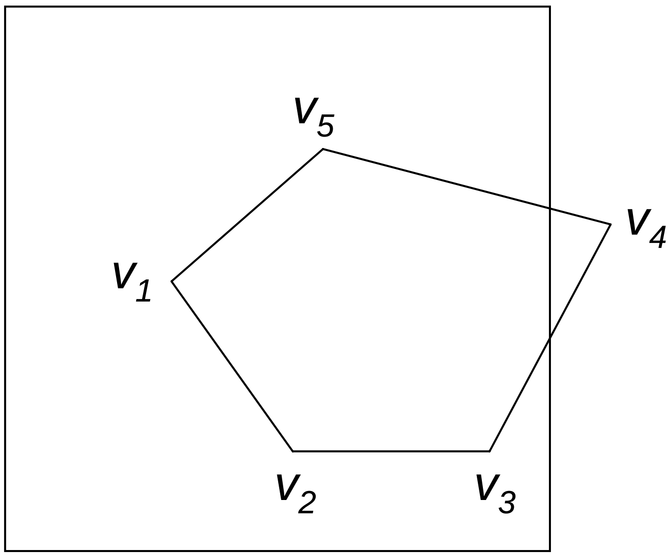
$$t = [(\mathbf{q} - \mathbf{L}_0) \bullet \mathbf{n}] / [(\mathbf{L}_1 - \mathbf{L}_0) \bullet \mathbf{n}]$$



The intersection point $\mathbf{i} = \mathbf{L}(t)$ for this value of t.

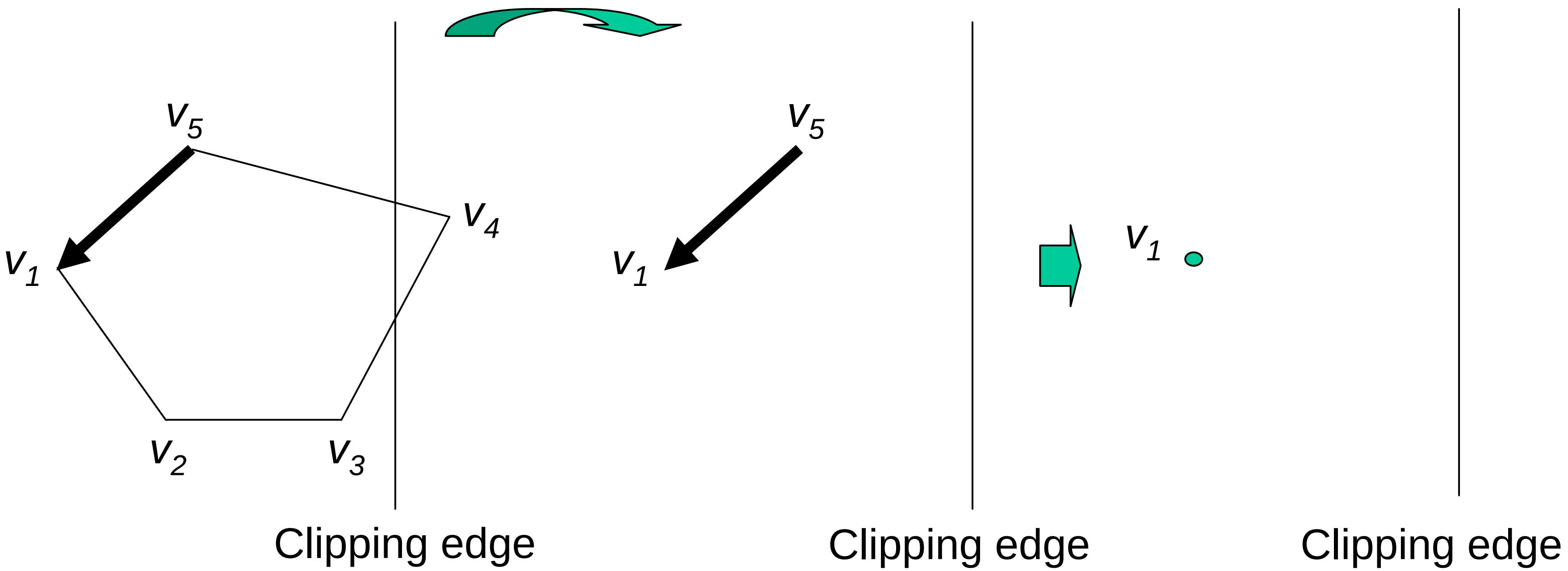
7.4 Polygon clipping

An example for the polygon clipping



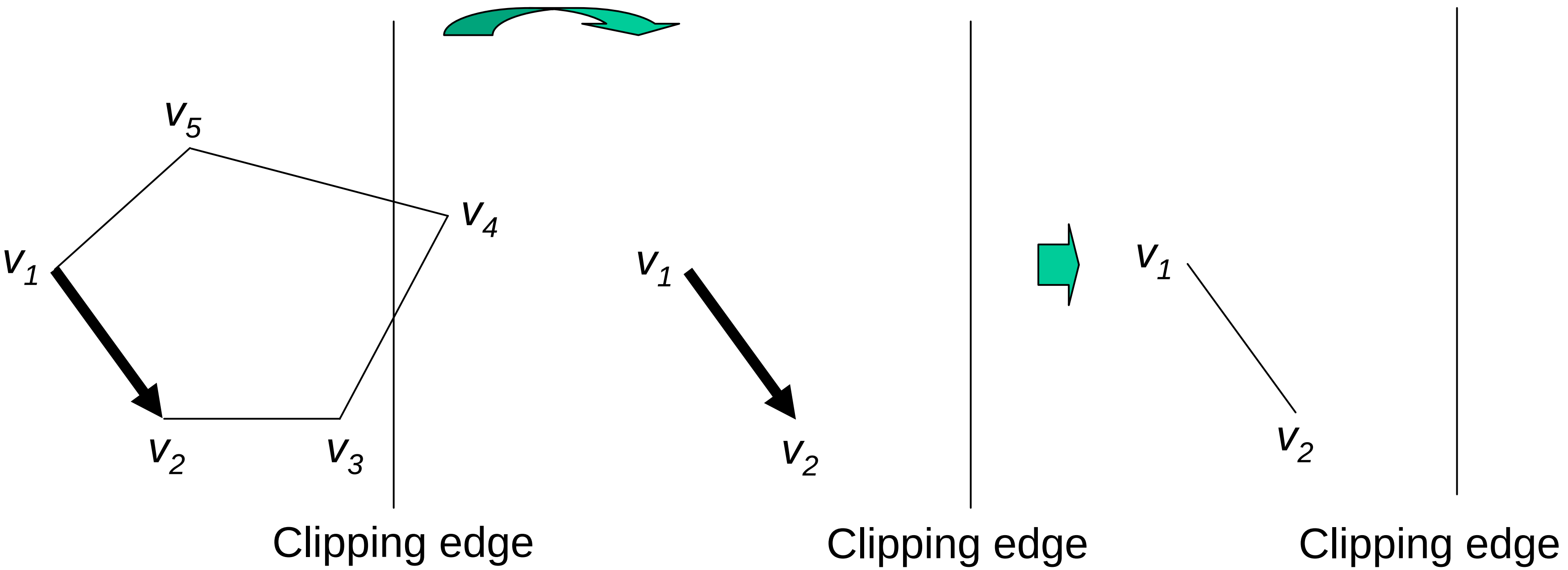
7.4 Polygon clipping

As we said, the Sutherland-Hodgeman algorithm clips the polygon against one clipping edge at a time. We start with the right edge of the clip rectangle. In order to clip the polygon against the line, each edge of the polygon have to be considered. Starting with the edge, represented by a pair of vertices, v_5v_1:



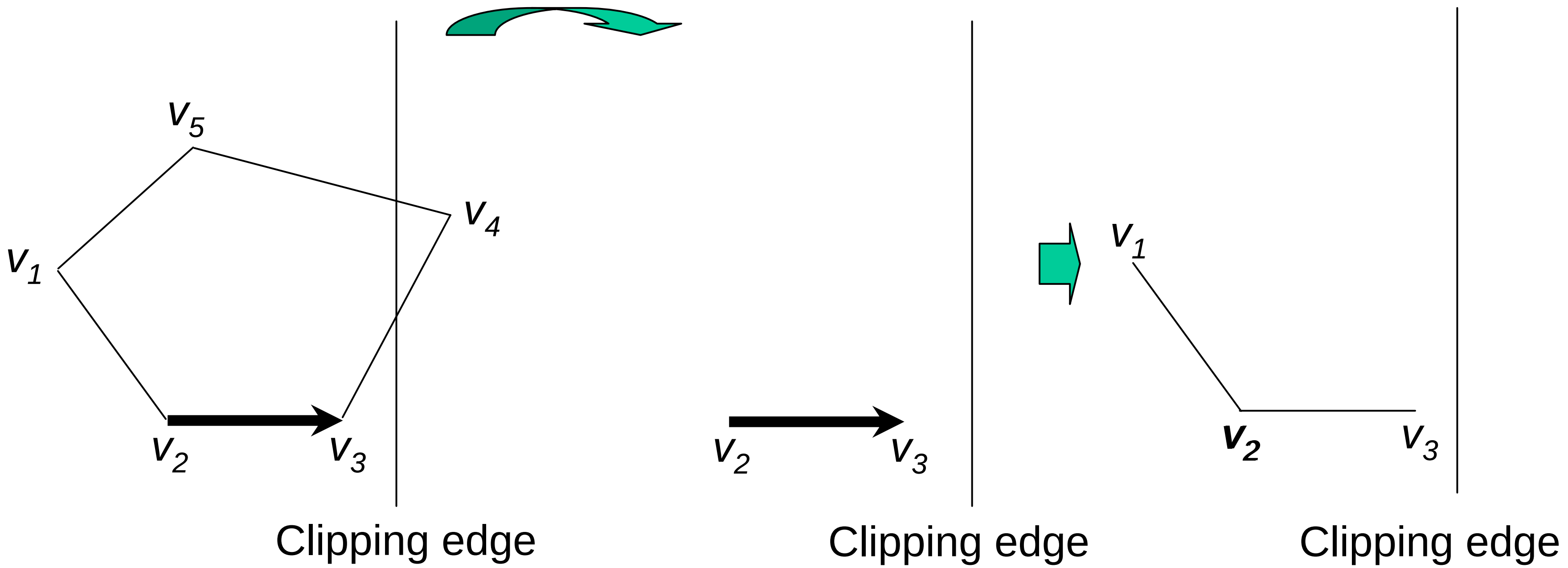
7.4 Polygon clipping

Now v_1v_2:



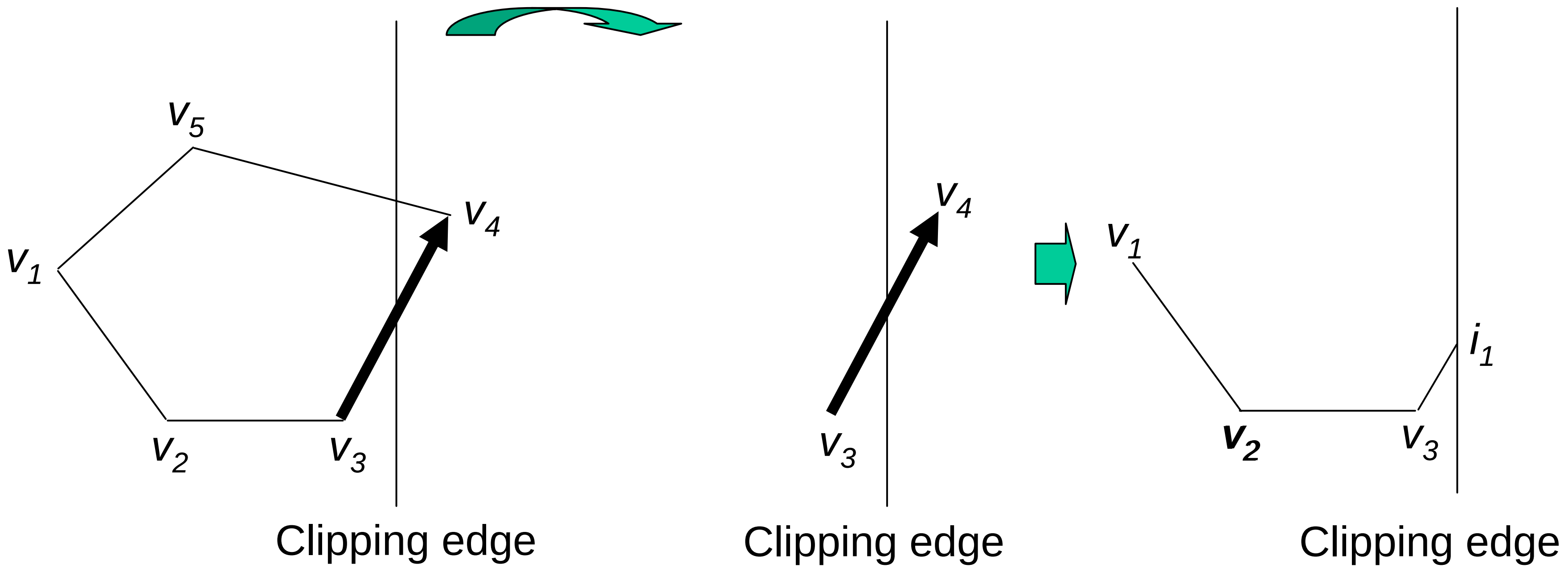
7.4 Polygon clipping

Now v_2v_3:



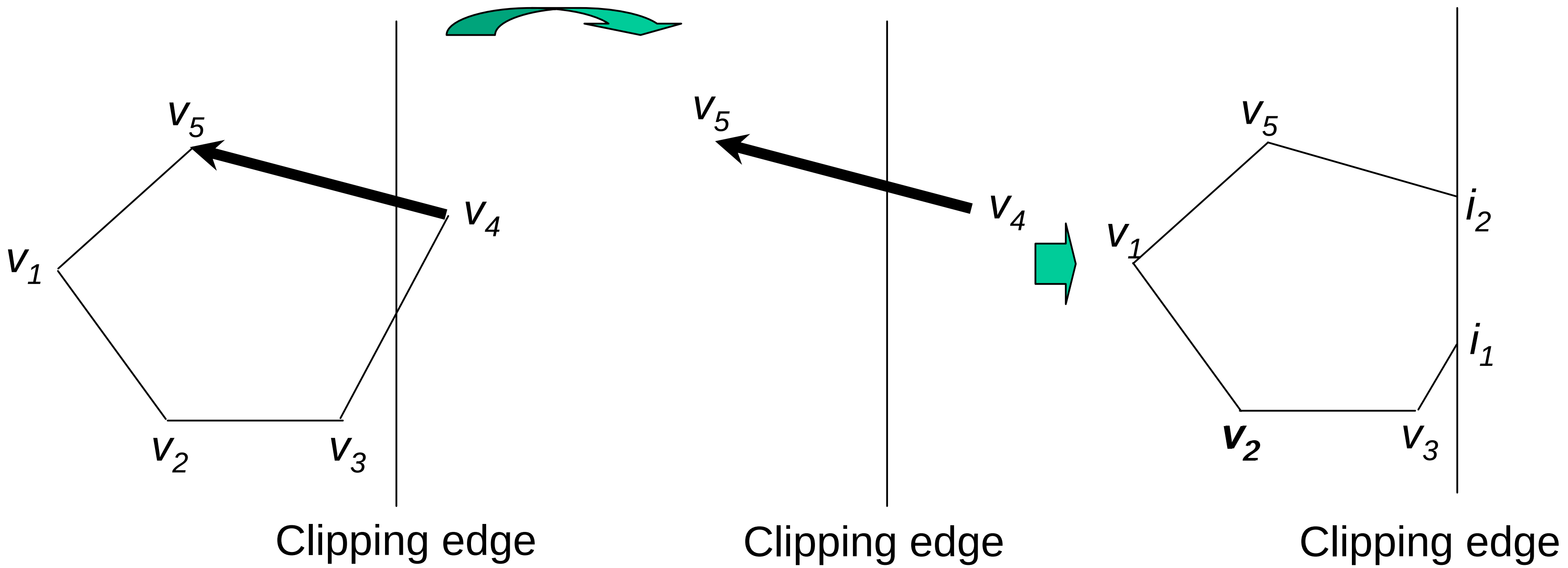
7.4 Polygon clipping

Now v_3v_4:



7.4 Polygon clipping

Now v_4v_5:

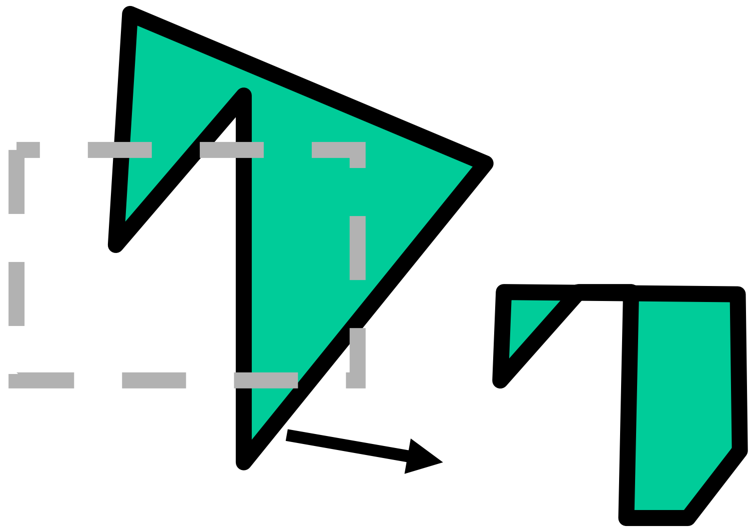


After these, we have to clip the polygon against the other three edges of the window in a similar way.

7.4 Polygon clipping

Problem with Sutherland-Hodgeman

Concavities can end up linked:



Weiler-Atherton creates separate polygons in cases like this.

7.4 Polygon clipping

Weiler-Atherton Polygon Clipping

To find the edges for a clipped polygon, we follow a path (either clockwise or counterclockwise) around the fill area that detours along a clipping-window boundary whenever a polygon edge crosses to the outside of that boundary. The direction of a detour at a clipping-window border is the same as the processing direction for the polygon edges.

For a counterclockwise traversal of the polygon vertices, we apply the following **Weiler-Atherton** procedures:

7.4 Polygon clipping

Weiler-Atherton Polygon Clipping (continued)

1. Process the edges of the polygon in a counterclockwise order until an inside-outside pair of vertices is encountered for one of the clipping boundaries; that is, the first vertex of the polygon edge is inside the clip region and the second vertex is outside the clip region.
2. Follow the window boundaries in a counterclockwise direction from the exit-intersection point to another intersection point with the polygon. If this is a previously processed point, proceed to the next step. If this is a new intersection point, continue processing polygon edges in a counterclockwise order until a previously processed vertex is encountered.

7.4 Polygon clipping

Weiler-Atherton Polygon Clipping (continued)

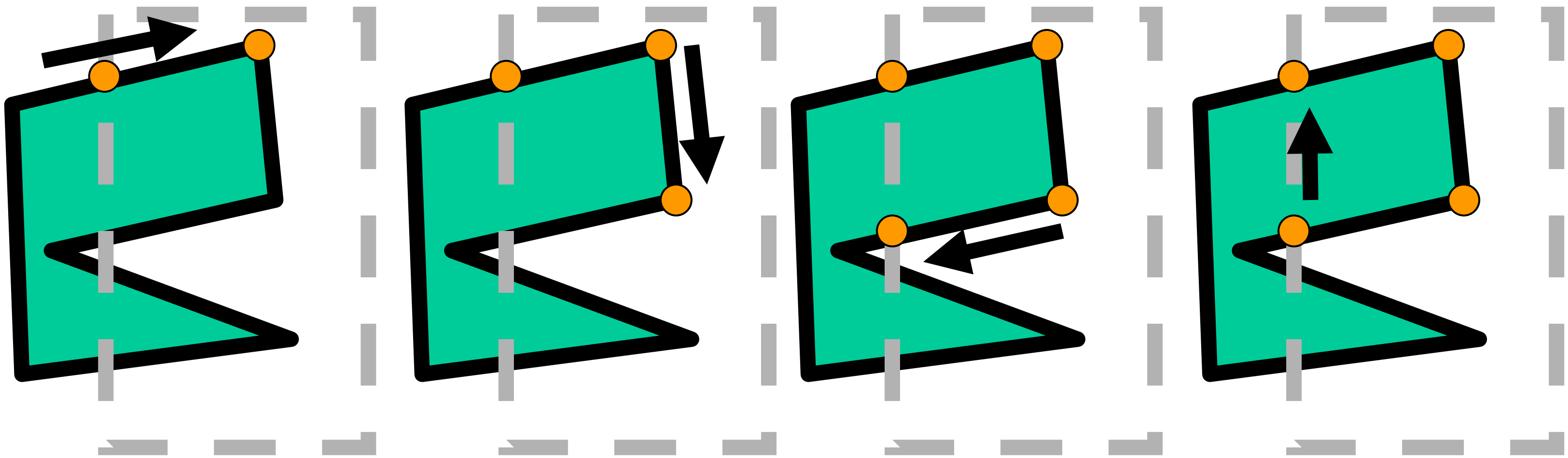
3. Form the vertex list for this section of the clipped polygon.
4. Return to the exit-intersection point and continue processing the polygon edges in a counterclockwise order.

Note: this may generate more than one polygon!

7.4 Polygon clipping

Weiler-Atherton Polygon Clipping (continued)

Example:

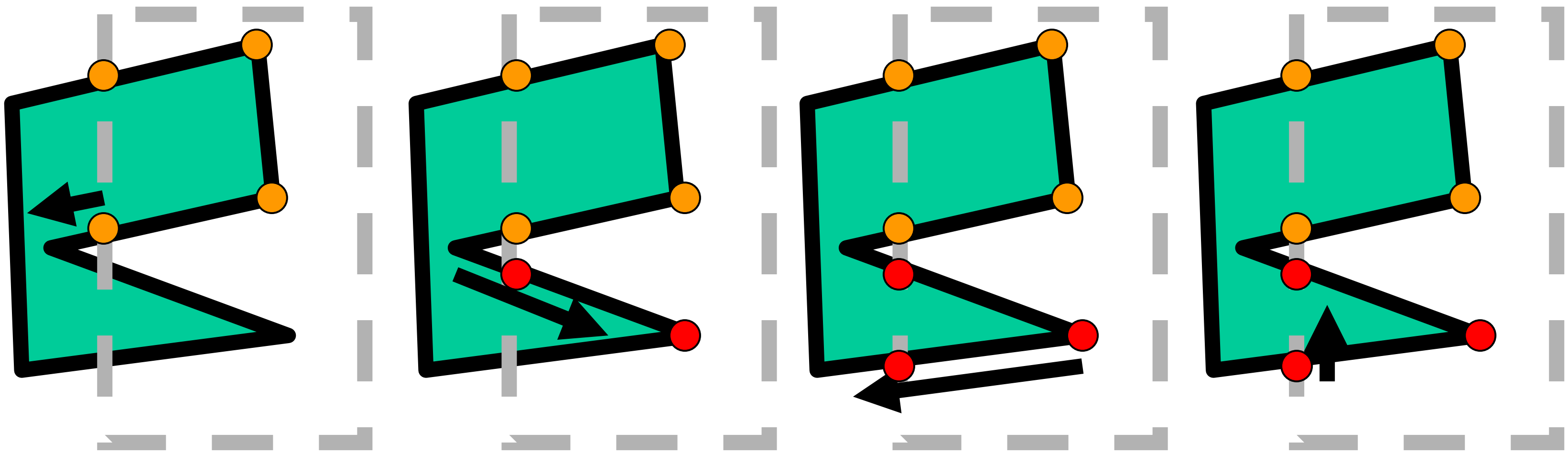


add clip pt. and end pt.	add end pt.	add clip pt. cache old dir.	follow clip edge until a) new crossing found b) reach pt. already added

7.4 Polygon clipping

Weiler-Atherton Polygon Clipping (continued)

Example (continued)

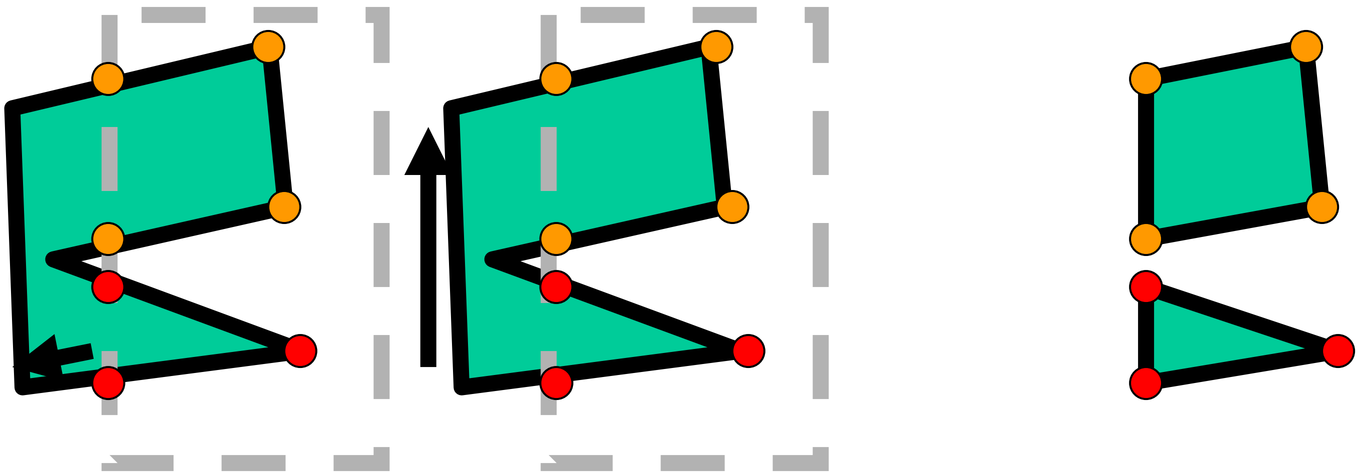


continue from cached location	add clip pt. and end pt.	add clip pt. cache dir.	follow clip edge until a) new crossing found b) reach pt. already added

7.4 Polygon clipping

Weiler-Atherton Polygon Clipping (continued)

Example (continued)



continue from cached location

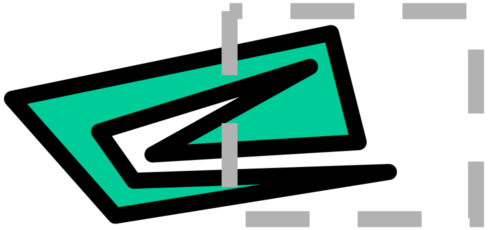
nothing added finished

Final result: Two *unconnected* polygons

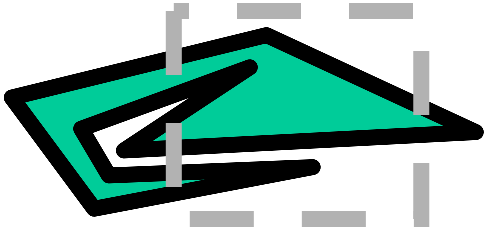
7.4 Polygon clipping

Difficulties with Weiler-Atherton polygon clipping

What if the polygon re-crosses edge?



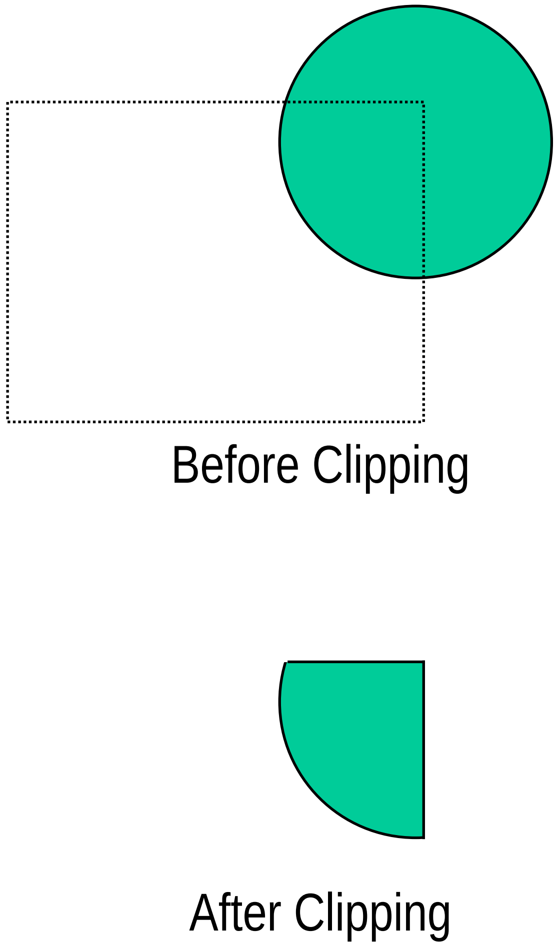
How many "cached" crossings?



Your geometry step must be able to *create* new polygons instead of 1-in-1-out

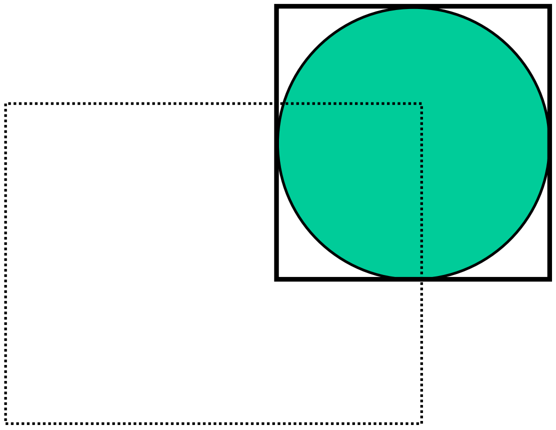
7.5 Curve clipping

Areas with curved boundaries can be clipped with methods similar to those discussed in the previous sections. If the objects are approximated with straight-line segments, we use a polygon-clipping method. Otherwise, the clipping procedures involve nonlinear equations, and this requires more processing than for objects with linear boundaries.

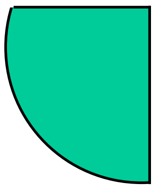


7.5 Curve clipping

For a simple accept/reject test, the bounding box can be used. This box (in the 2-D case just a square) describes the maximal extent of the curved object parallel to the coordinate axes. If the bounding box does not intersect with the clipping region, no part of the object is inside. Otherwise, if the bounding box is completely contained by the clipping region, the entire object is going to be inside.



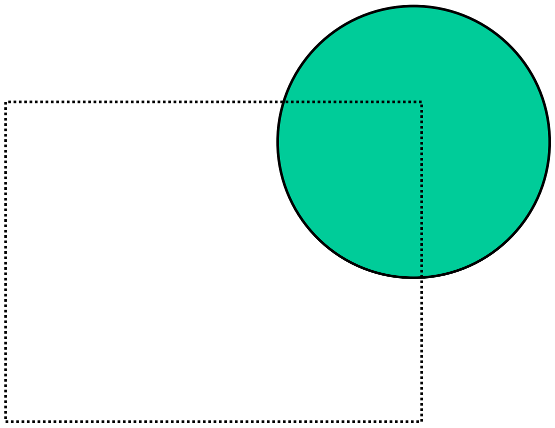
Before Clipping



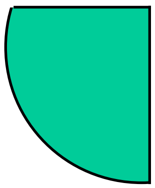
After Clipping

7.5 Curve clipping

If the bounding box is partly inside the clipping area we have to do further testing. Similar to polygon clipping, the intersections with the boundaries of the clipping region need to be computed. An intersection calculation involves substituting a clipping-boundary position (x_{min}, x_{max}, y_{min}, and y_{max}) in the nonlinear equation for the object boundary and solving for the other coordinate value.



Before Clipping



After Clipping

7.6 Text Clipping

There are several techniques that can be used to provide text clipping. The simplest method for processing character strings relative to the clipping window is to use the **all-or-none string clipping** strategy. This procedure is implemented by examining the coordinate extent of the text string (bounding box). If the coordinate limits of this bounding box are not entirely within the clipping window, the string is rejected.

Sometimes, only the lower left corner is used for clipping: only if this point is within the clipping region the string is drawn. This, for example, is how OpenGL clips the Bitmap Characters (based on the current raster position).

7.6 Text Clipping

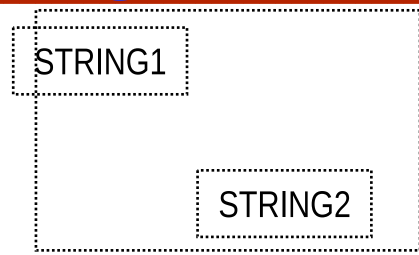
An alternative is to use the **all-or-none character clipping** strategy. Here we eliminate only those characters that are not completely inside the clipping region. In this case, the coordinate extents of individual characters are compared to the clipping boundaries. Any character that is not completely within the clipping-window boundary is eliminated.

7.6 Text Clipping

A third approach to text clipping is to clip the components of individual characters. This provides the most accurate display of clipped character strings, but it requires the most processing. If an individual character overlaps a clipping boundary, we clip off only the parts of the character that are outside the clipping region. Outline character fonts defined with line segments are processed in this way using polygon-clipping algorithms. Characters defined with bitmaps are clipped by comparing the relative position of the individual pixels in the character grid patterns to the borders of the clipping region.

7.6 Text Clipping

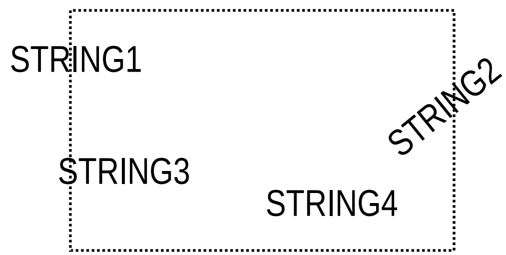
All or none text clipping



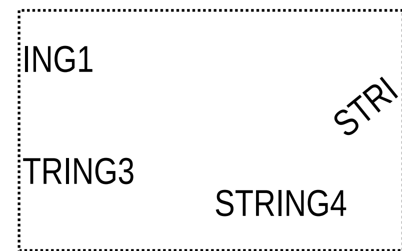
Before Clipping

After Clipping

All or none character clipping

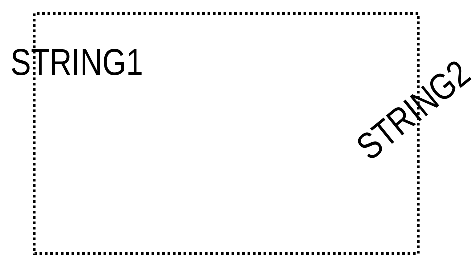


Before Clipping

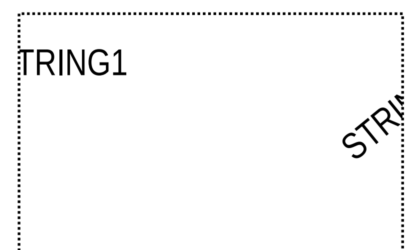


After Clipping

Clipping individual character



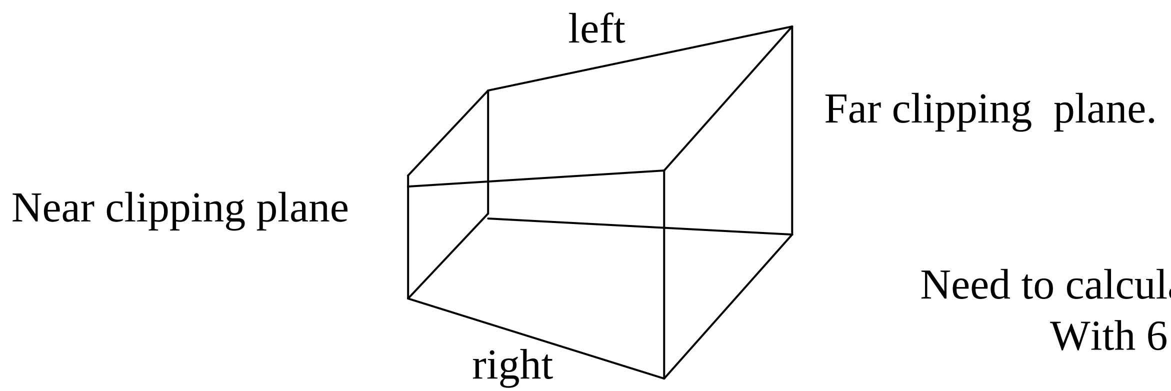
Before Clipping



After Clipping

7.7 3-D Clipping

- For orthographic projection, view volume is a box.
- For perspective projection, view volume is a *frustrum*.



Need to calculate intersection
With 6 planes.

7.7 3D Clipping

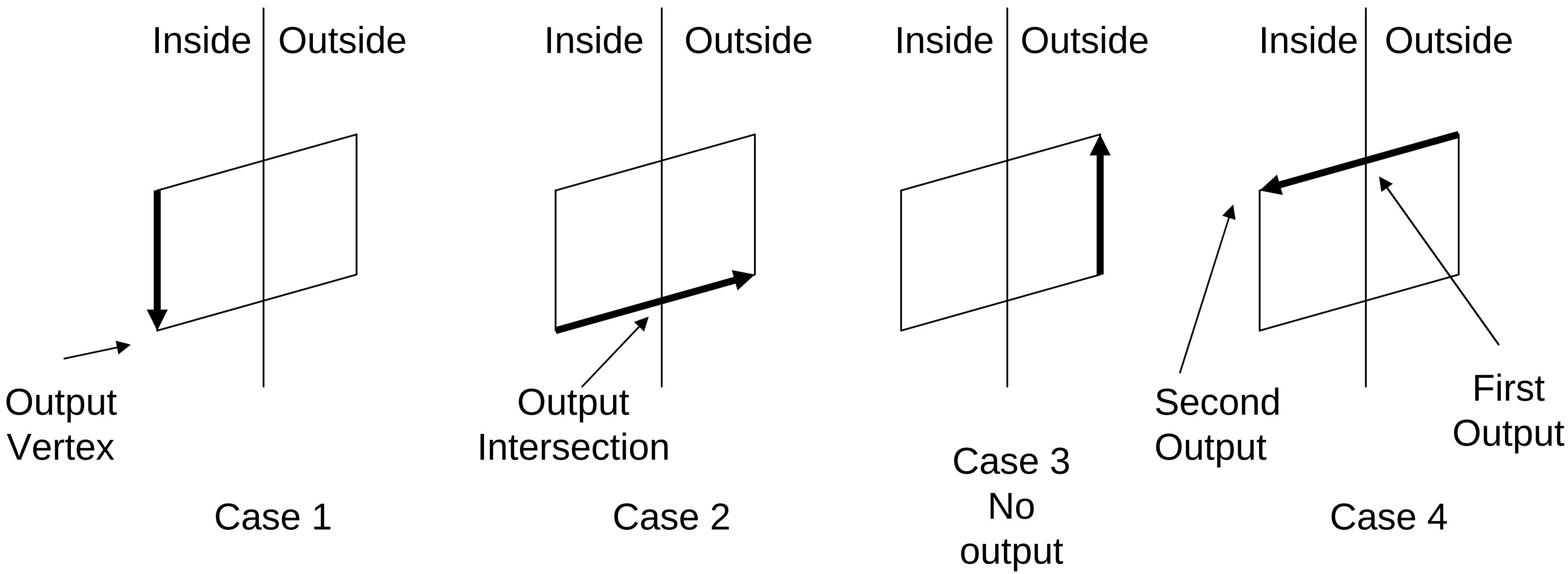
We extend the Cohen-Sutherland algorithm.

- Now 6-bit code instead of 4 bits.
- Trivial acceptance where both endpoint codes are all zero.
- Perform logical AND, reject if non-zero.
- Find intersect with a bounding plane and add the two new lines to the line queue.
- Line-primitive algorithm.

7.7 3D Clipping

Sutherland-Hodgman Algorithm

Four cases of polygon clipping :



7.7 3D Clipping

- Sutherland-Hodgman extends easily to 3D
- Call ‘CLIP’ procedure 6 times rather than 4
- Polygon-primitive algorithm

7.8 Hidden Surface Removal

Visibility

- Given a set of polygons, which is visible at each pixel? (in front, etc.). Also called hidden surface removal
- Very large number of different algorithms known. Two main classes:
  - Object precision: computations that operate on primitives
  - Image precision: computations at the pixel level
- All the spaces in the viewing pipeline maintain depth, so we can work in any space
  - World, View and Canonical Screen spaces might be used
  - Depth can be updated on a per-pixel basis as we scan convert polygons or lines

7.8 Hidden Surface Removal

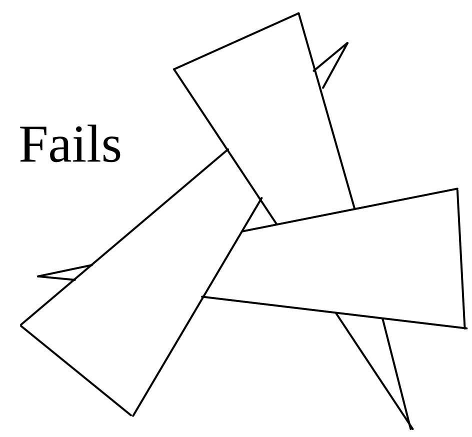
Visibility Issues

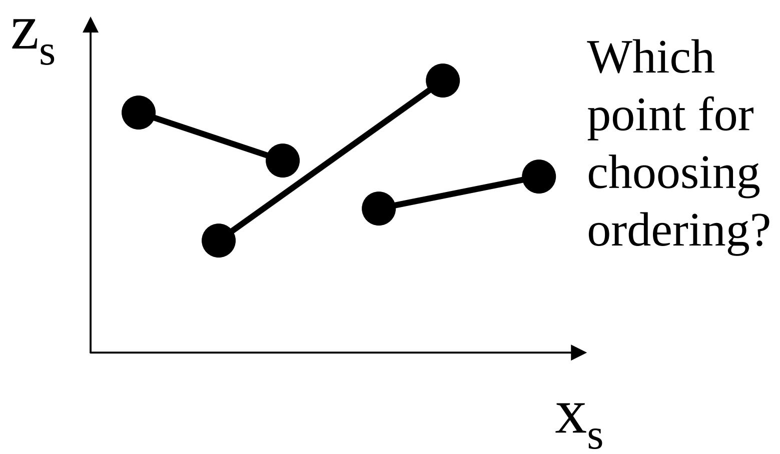
- Efficiency – it is slow to overwrite pixels, or scan convert things that cannot be seen
- Accuracy - answer should be right, and behave well when the viewpoint moves
- Must have technology that handles large, complex rendering databases
- In many complex worlds, few things are visible
  - How much of the real world can you see at any moment?
- Complexity - object precision visibility may generate many small pieces of polygon

7.8 Hidden Surface Removal

Painters Algorithm (Image Precision)

- Algorithm:
  - Choose an order for the polygons based on some choice (e.g. depth to a point on the polygon)
  - Render the polygons in that order, deepest one first
- This renders nearer polygons over further
- Difficulty:
  - works for some important geometries (2.5D - e.g. VLSI)
  - doesn't work in this form for most geometries - need at least better ways of determining ordering





7.8 Hidden Surface Removal

Depth Sorting (Object Precision, in view space)

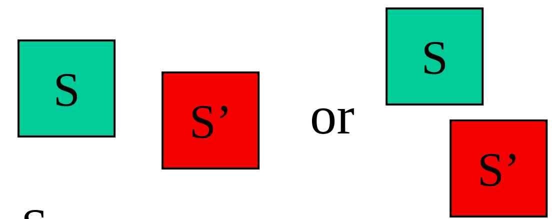
- An example of a *list-priority* algorithm
- Sort polygons on depth of some point
- Render from back to front (modifying order on the fly)
- Rendering: For surface S with greatest depth
  - If no overlap in depth with other polygons, scan convert
  - Else, for overlaps in depth, test for overlaps in the image plane
    - If none, scan convert and go to next polygon
  - If S, S' overlap in depth, swap order and try again
  - If S, S' have been swapped already, split and reinsert

7.8 Hidden Surface Removal

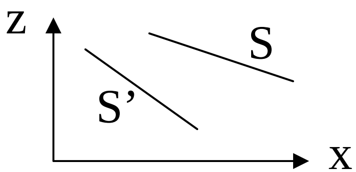
Depth Sorting (continued)

Testing for overlaps: Start drawing when first condition is met:

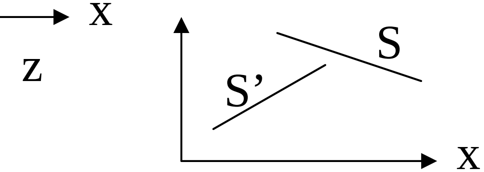
- x-extents or y-extents do not overlap



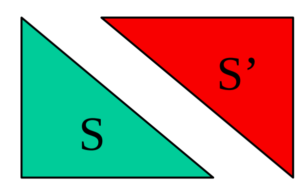
- S is behind the plane of S'



- S' is in front of the plane of S



- S and S' do not intersect in the image plane



7.8 Hidden Surface Removal

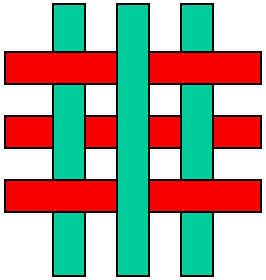
Depth Sorting (continued)

Advantages:

- Filter anti-aliasing works fine
- Composite in back to front order
- No depth quantization error
- Depth comparisons carried out in high-precision view space

Disadvantages:

- Over-rendering
- Potentially very large number of splits - $\Omega(n^2)$ fragments from n polygons



7.8 Hidden Surface Removal

Area Subdivision

- Exploits *area coherence*: Small areas of an image are likely to be covered by only one polygon
- Three easy cases for determining what's in front in a given region:
  - a polygon is completely in front of everything else in that region
  - no surfaces project to the region
  - only one surface is completely inside the region, overlaps the region, or surrounds the region

7.8 Hidden Surface Removal

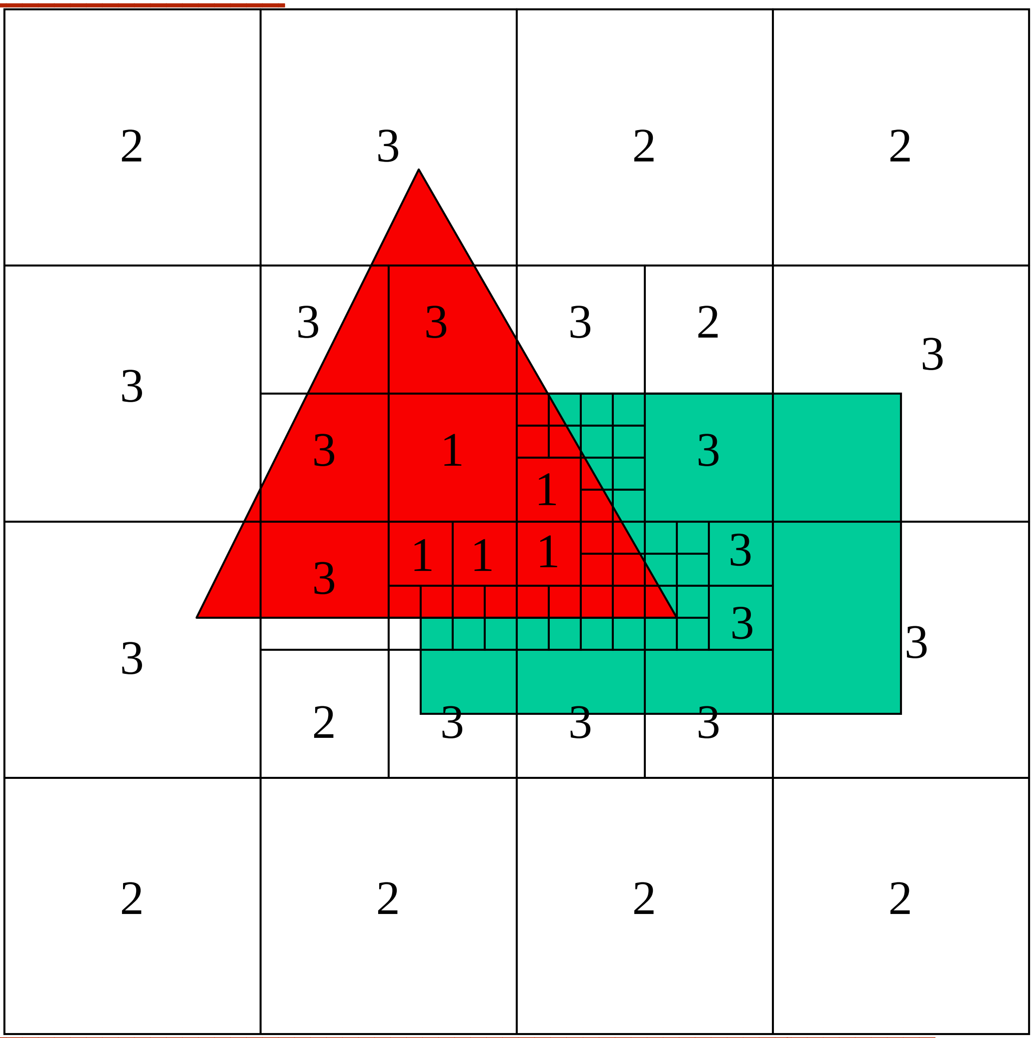
Warnock’s Area Subdivision (Image Precision)

- Start with whole image
- If one of the easy cases is satisfied (previous slide), draw what’s in front
- Otherwise, subdivide the region and recurse
- If region is single pixel, choose surface with smallest depth
- Advantages:
  - No over-rendering
  - Anti-aliases well - just recurse deeper to get sub-pixel information
- Disadvantage:
  - Tests are quite complex and slow

7.8 Hidden Surface Removal

Warnock's Algorithm

- Regions labeled with case used to classify them:
  1) One polygon in front
  2) Empty
  3) One polygon inside, surrounding or intersecting
- Small regions not labeled
- Note it's a rendering algorithm and a HSR algorithm at the same time
  - Assuming you can draw squares



7.8 Hidden Surface Removal

BSP-Trees (Object Precision)

Construct a *binary space partition* tree

- Tree gives a rendering order
- A list-priority algorithm

Tree splits 3D world with planes

- The world is broken into convex cells
- Each cell is the intersection of all the half-spaces of splitting planes on tree path to the cell

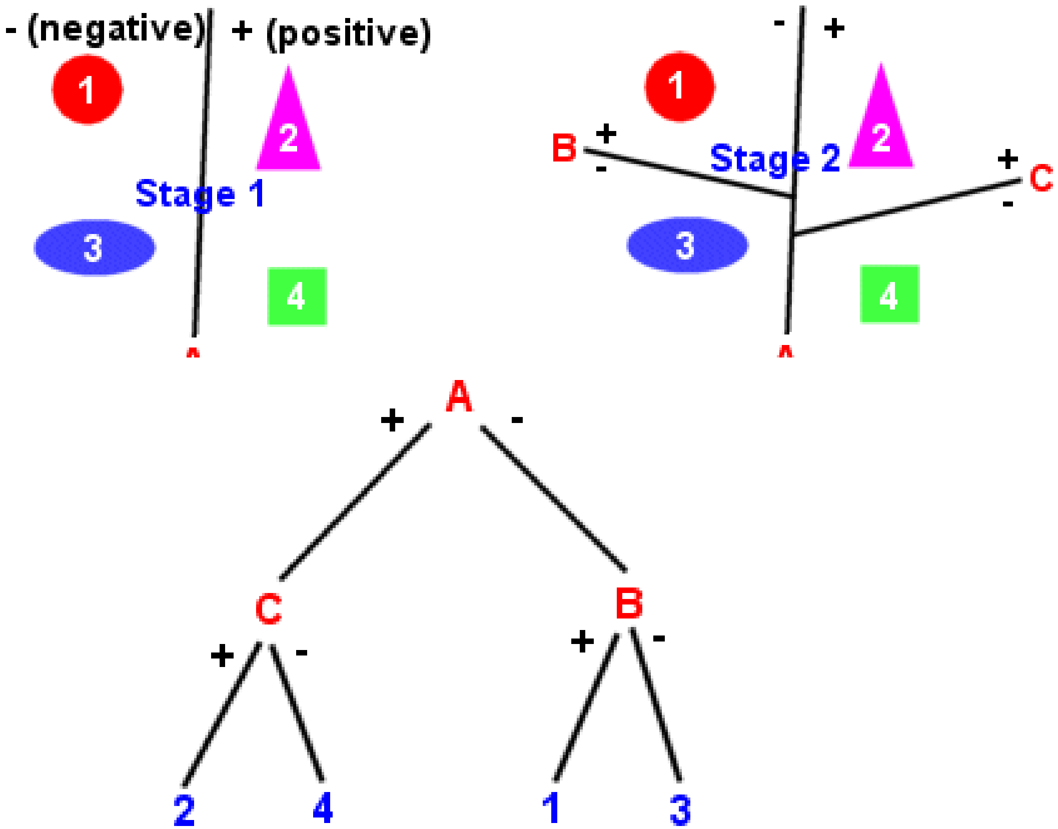
Also used to model the shape of objects, and in other visibility algorithms

- BSP visibility in games does **not** necessarily refer to this algorithm

7.8 Hidden Surface Removal

BSP-Trees (continued)

Example:



7.8 Hidden Surface Removal

BSP-Trees (continued)

If a cutting plane intersects an object the object needs to be split.

To render the scene, we process that part of the tree which is further away from the view point with respect to the cutting plane. This way, the objects are drawn in a back to front order. Thus, the foreground objects are painted over the background objects.

Note: if the viewpoint changes we can still use the same BSP tree (assuming the objects did not change); only the front and back side with respect to the cutting planes may switch.