

# Computer Graphics

---

**Prof. Jibum Kim**

**Department of Computer Science & Engineering**

**Incheon National University**

---

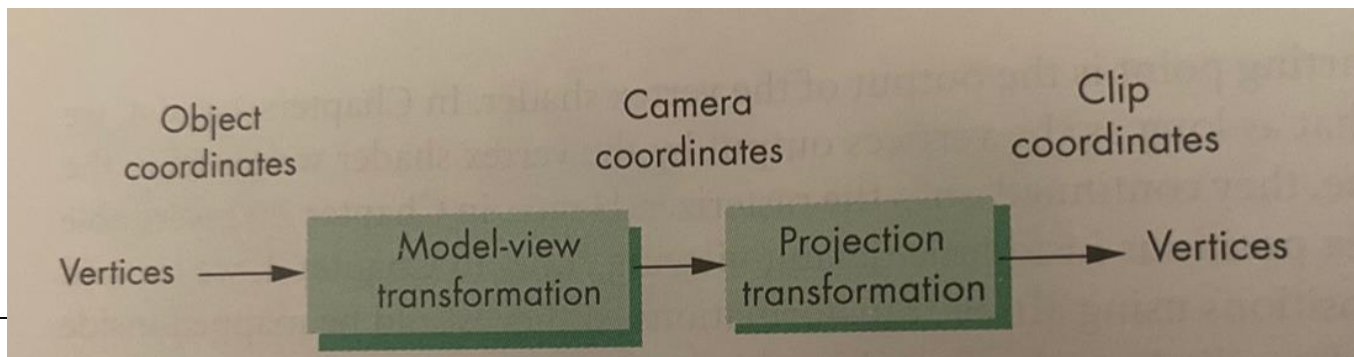
- **Clip coordinates and Canonical view volume (CVV)**

---

- OpenGL 좌표계 변환 순서

1. Local (model) coordinate (모델 좌표계)
2. World coordinate (세계 좌표계)
3. Eye coordinate (view, camera coordinate)
4. Clip coordinate (절단 좌표계)
5. Normalized device coordinate (NDC)
6. Screen coordinate (화면 좌표계)

- After vertices pass through the **model-view transformation**, they will be represented in **eye (camera) coordinates**
- Next step is the **projection transformation**
- The two most common types of projection are **perspective and orthographic**
- Vertices are transformed into **clip space** by applying a projection transformation

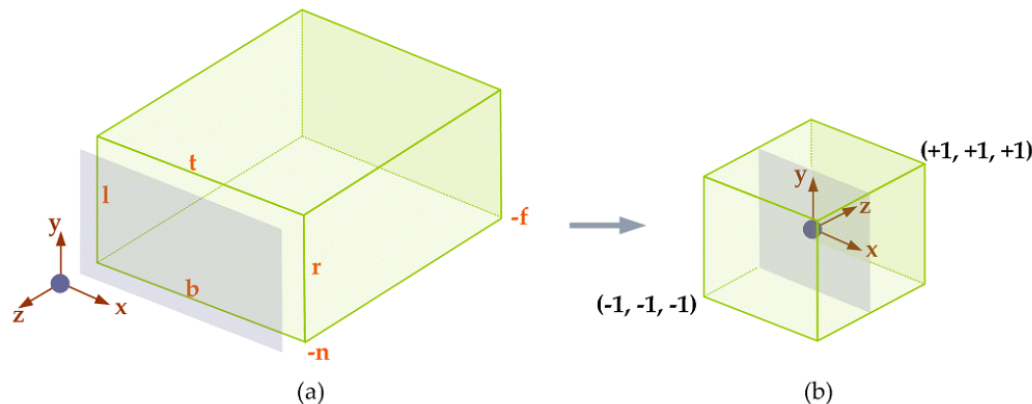


- 
- OpenGL expects the coordinates to be within a specific range and any coordinate that falls outside this range is **clipped**. This is also where **clip space** gets its name from
  - Clip coordinates are processed to the **-1.0 and 1.0 range** and determine which vertices will end up on the screen
  - Coordinates that are clipped are discarded, so the remaining coordinates will end up as **fragments** visible on your screen.

## 3D vertex $v$ 는

1. 가장 먼저 4x4 model transformation ( $M$ )행렬이 곱해짐  
: model coordinates  $\rightarrow$  world coordinates
2. 두번째로 4x4 viewing transformation ( $V$ )행렬이 곱해짐  
: world coordinates  $\rightarrow$  eye coordinates
3. 세번째로 4x4 projection transformation ( $P$ )행렬이 곱해짐
  - : eye coordinates  $\rightarrow$  clip coordinates
  - $v_{clip} = PVMv$ , 여전히  $v$ 는 동차좌표로 4행 1열로 표시

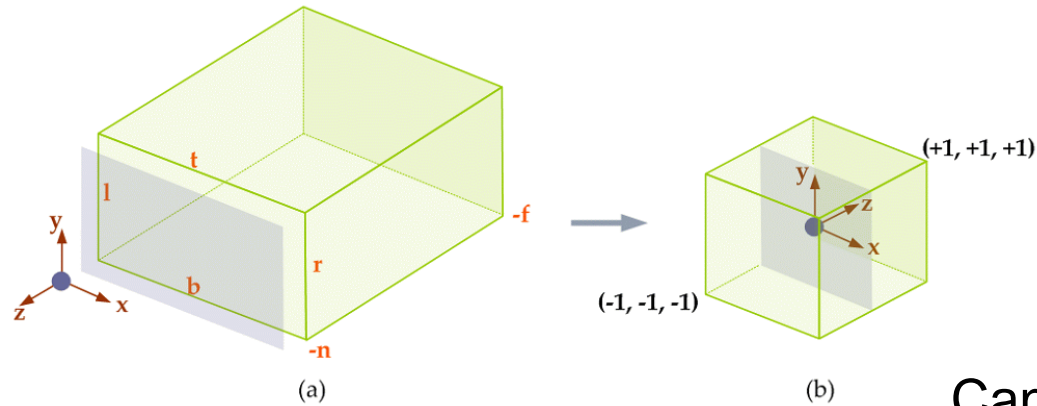
- Projection transformation시에 정규화 가시 부피 (canonical view volume, CVV)
- 가시 부피 (viewing volume, viewing frustum)안의 각 vertex를 각 축에서  $-1 \sim 1$  사이의 standard cube (정육면체)로 변환 (CVV)시킴
- Recall, default projection,  $\text{glOrtho}(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0)$
- The viewing volume is in fact the canonical view volume



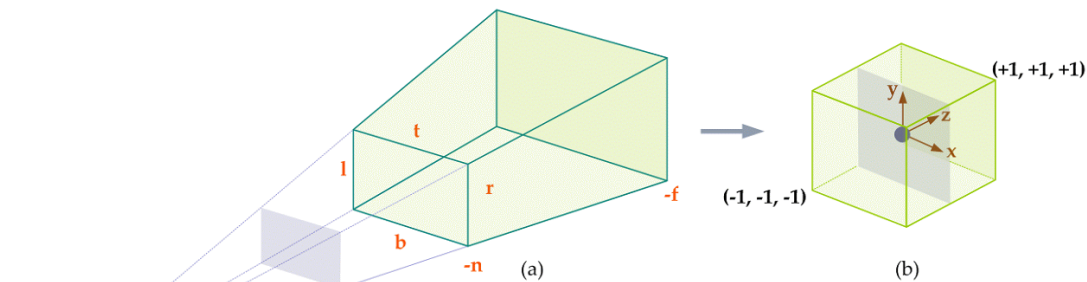
- 
- 정규화 가시 부피 (CVV) 사용 이유
  - 1. 직교 투영, 원근 투영을 동일한 모습의 정규화 가시 부피로 변형 (동일 파이프 라인 사용)
  - 2. The CVV simplifies the clipping process because the sides are aligned with the coordinate axes
  - 3. Canonical viewing volume 후에 viewport 좌표 계로 mapping이 쉽다



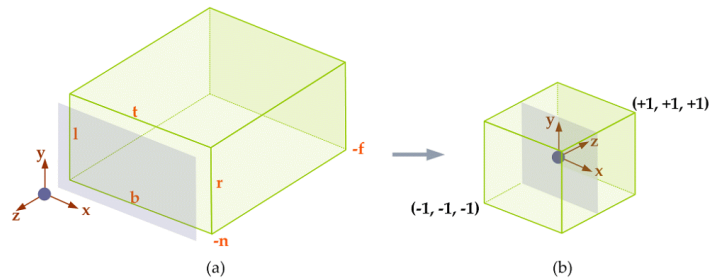
- Canonical view volume으로의 변환 (주의: z축 방향이 다르다)



Canonical view volume



- 직교 투영 시 정규화 가시 부피로의 변환 행렬을 구해보자
- 왼쪽: 가시부피:  $\text{glOrtho}(l, r, b, t, n, f)$ , 오른쪽: 정규화 가시 부피

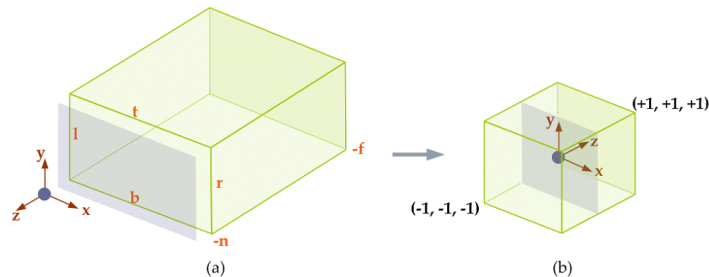


- 1. 가시 부피의 중심:  $(\frac{l+r}{2}, \frac{b+t}{2}, \frac{-n-f}{2})$ , 정규화 가시 부피의 중심:  $(0, 0, 0)$

■ 이동 변환 행렬  $T$ :

$$\begin{bmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- 직교 투영 시 정규화 가시 부피로의 변환 행렬을 구해보자
- 왼쪽: 가시부피:  $\text{glOrtho}(l, r, b, t, n, f)$ , 오른쪽: 정규화 가시 부피

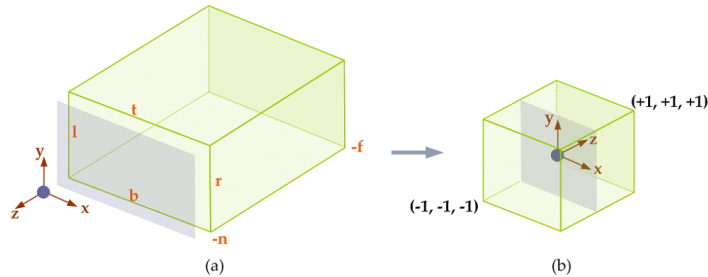


- 2. 가시 부피의 크기: x축 방향:  $r - l$ , y축 방향:  $t - b$ , z축 방향:  $-n + f$   
 정규화 가시 부피의 크기: x축 방향: 2, y축 방향: 2, z축 방향: 2

크기 변환 행렬  $S$ :

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- 직교 투영 시 정규화 가시 부피로의 변환 행렬을 구해보자
- 왼쪽: 가시부피:  $\text{glOrtho}(l, r, b, t, n, f)$ , 오른쪽: 정규화 가시 부피



- 3. 좌표계 변환: 오른손 좌표계에서 왼손 좌표계로 변경함에 따라서  $z$ 축의 방향이 바뀌는 반사 변환 (reflection)

- 반사 변환 행렬  $R$ : 
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- 직교 투영 시 정규화 가시 부피로의 변환 행렬
- 정규화 변환 행렬  $N, N = R \cdot S \cdot T$
- 이 정규화 변환 행렬이 glOrtho(l, r, b, t, n, f) 사용시
- projection matrix로 올라감

$$R = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad S = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad T = \begin{pmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & \frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$N = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

---

- **Normalized device coordinate (NDC)**

---

- OpenGL 좌표계 변환 순서

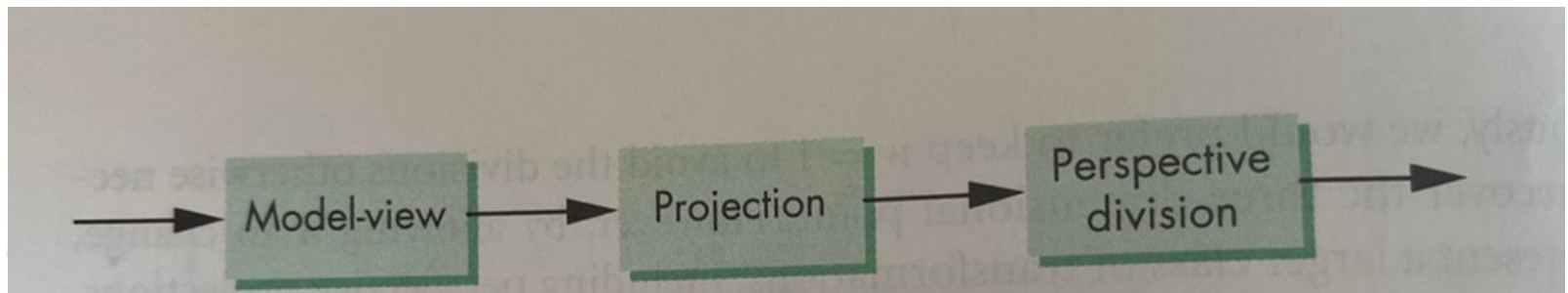
1. Local (model) coordinate (모델 좌표계)
2. World coordinate (세계 좌표계)
3. Eye coordinate (view, camera coordinate)
4. Clip coordinate (절단 좌표계)
5. Normalized device coordinate (NDC)
6. Screen coordinate (화면 좌표계)

- 앞에서 perspective projection의 경우에는 동차좌표를 사용시에 Projection matrix가 아래와 같이 표현되었다.
- 이 경우 동차좌표의 마지막 (x, y, z, w)의 w가 1이 아니기 때문에 내부적으로는 이를 w로 나누는 **perspective division**이라는 단계를 한번 더 거치고 이를 통해 4D homogeneous coordinate를 3D로 바꾼다. 이를 clip coordinate 다음 단계로 **normalized device coordinate (NDC)**으로 부른다

- $$P' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, \text{ 4x4 projection matrix}$$

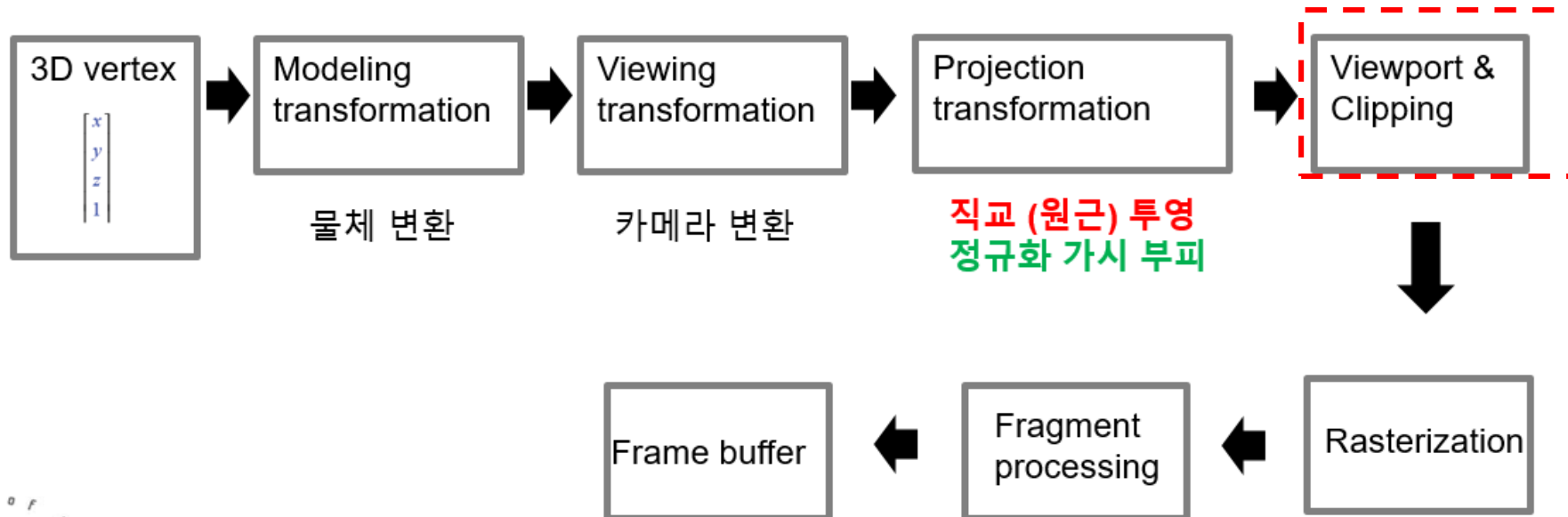


- Projection pipeline
- **Perspective division** can be made a part of the pipeline
- Perspective division이 끝나면 clip coordinate (4D, 동차좌표)이 normalized device coordinate (3D)이 된다



---

## ■ Clipping



- 
- We shall concentrate on **clipping of line segments and polygons** because they are the most common primitives to pass down the pipeline
  - Fortunately, many of the most efficient algorithms are almost identical in 2D and 3D
  - A **clipper** decides which primitives can possibly appear on the display and be passed on to the rasterizer

- 
- Primitives that fit within the specified view volume pass through the clipper, or are **accepted**
  - Primitives that cannot appear on the display are **eliminated (or culled)**
  - Primitive that are partially within the view volume must be **clipped** such that any part lying outside the volume is removed

---

## ■ Bitwise operation

- **Bitwise AND 연산 (표기: &)**
- AND: 둘다 1일 때만 결과가 1
- 비트 열의 같은 위치의 이진수끼리 AND 연산
- 예: X=1000, Y=0000
- $X \& Y = 0000$ , 혹은 0 으로 표시
- **Bitwise OR 연산 (표기: |)**
- 둘중에 하나라도 1이면 결과가 1
- 예: X=1000, Y=0000
- $X | Y = 1000$

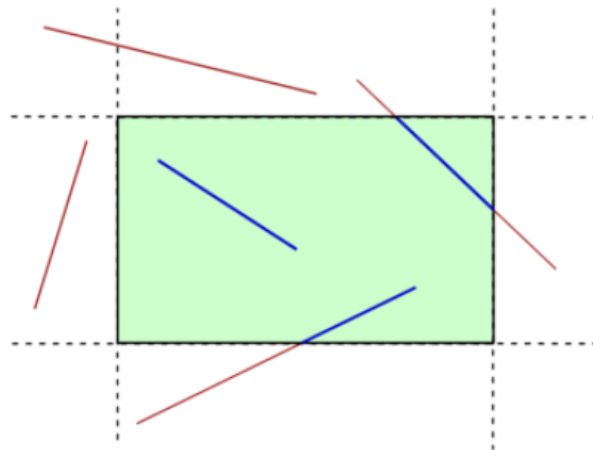
X	Y	$X \& Y$	$X   Y$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

---

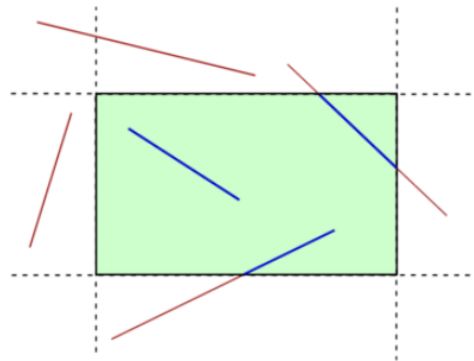
- **Cohen-Sutherland line clipping algorithm**



- **Cohen-Sutherland line clipping 알고리즘**
- 2차원 선분을 2차원 직사각형 기준으로 절단하는 line clipping 알고리즘으로 **divide and conquer 전략을 쓰며 recursive하게 동작**하는 알고리즘으로 매우 중요한 알고리즘이다
- Viewing area (초록색): 예: 직사각형 모양의 가시 공간
- 목적: viewing area 바깥에 위치한 선분 clipping



- **Cohen-Sutherland line clipping 알고리즘**
- **Basic idea**
- **First, do easy test (completely inside or outside the viewing area?)**
- **아래 예들 중에 어떤 선분?**



- **If no, we need a more complex test (또한 교차점 계산 필요)**

- **Basic idea**

- 선분의 양 끝점에 비트열 형식의 location code (**outcode**) 부여
- 전체 영역을 두 개의 **half-space**로 나눈 후, inside인지 outside 인지 따져서 할당. 왜 half-space로 나눌까?

- **4-bit의 outcode,  $b_0b_1b_2b_3$  (TBRL)**

1)  $b_0 = \begin{cases} 1, & \text{if } y > top \\ 0 & \end{cases}$  , 점이 top 위? 판단

2)  $b_1 = \begin{cases} 1, & \text{if } y < bottom \\ 0 & \end{cases}$  , 점이 bottom 아래? 판단

3)  $b_2 = \begin{cases} 1, & \text{if } x > right \\ 0 & \end{cases}$  , 점이 right오른쪽? 왼쪽? 판단

4)  $b_3 = \begin{cases} 1, & \text{if } x < left \\ 0 & \end{cases}$  , 점이 left 왼쪽? 오른쪽? 판단

1001	1000	1010	Top
0001	0000	0010	
Viewing area			Bottom
0101	0100	0110	
Left		Right	

할당 후 양 끝점에 대해 비트 논리 연산 수행

- Case 1: 선분의 양 끝점 :  $o1=outcode(x1, y1)$ ,  $o2=outcode(x2,y2)$
- 예:  $o1=0000$ ,  $o2=0000$ , 는 어떤 선분인가 그려보자.
- Q:  $o1$ 과  $o2$ 가 어떤 bit 연산을 해야 두 선분의 양 끝점이 모두 viewing area 안에 있다는 걸 알 수 있을까?
- $o1 \mid o2 = 0000$
- 이를 알고리즘에서는 **trivial accept** 라고 한다

1001	1000	1010	
0001	0000	0010	Top
0101	0100	0110	Bottom
	Left	Right	

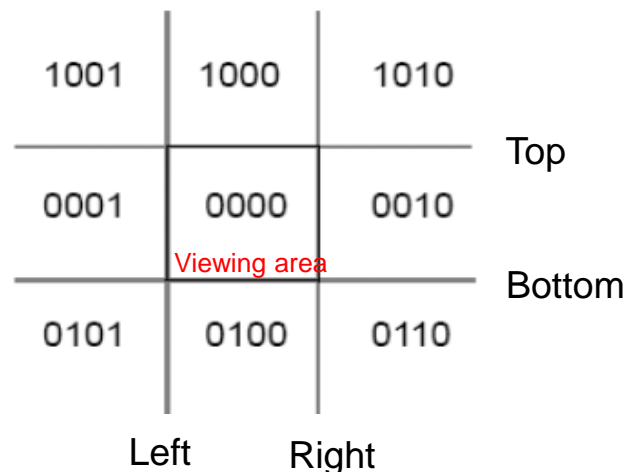
Viewing area

- Case 2: 선분의 양 끝점 :  $o1=outcode(x1, y1)$ ,  $o2=outcode(x2,y2)$
- 예:  $o1=1001$ ,  $o2=0101$ , 는 어떤 선분인가 그려보자.
- Q:  $o1$ 과  $o2$ 가 어떤 bit 연산을 해야 이 선분은 완전히 clipping 되어야 하는 선분이란걸 알 수 있을까? Bitwise-AND
- TBRL에서 두 끝점이 같은 위치에 있으면 동일한 위치에 1이 존재함
- $o1 \& o2 \neq 0000$  (half space의 특징을 이용)
- 이를 알고리즘에서는 **trivial reject** 라고 한다

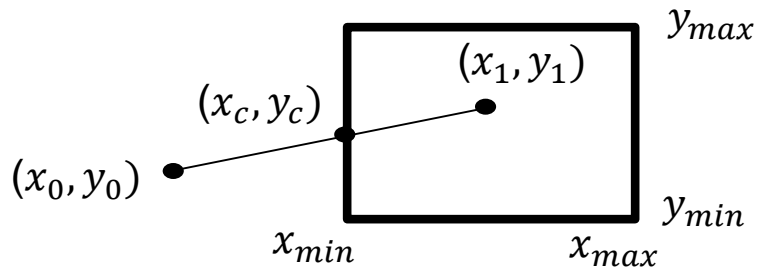
1001	1000	1010	Top
0001	0000	0010	
0101	0100	0110	Bottom
Left		Right	

Viewing area

- Case 3: 이외의 경우는 선분의 일부가 잘리는 경우로 recursive하게 동작
- 선분의 양 끝점의 outcode중에 0이 아닌 것이 있는 경우
- 예: o1=0001, o2=0000, 는 어떤 선분인가 그려보자.
- 이 경우에는 가장 왼쪽 bit 부터 어느 부분이 바깥에 있는지 따지고 선분과 half space와의 **경계 교차점**을 구한다
- 이 경우: TBRL중에 L이 0이 아님



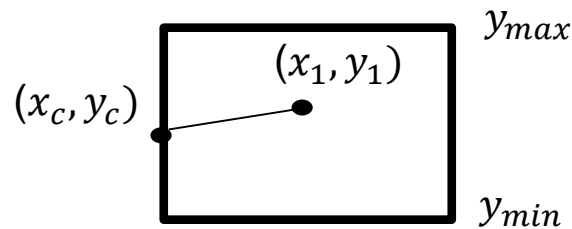
- 예: 선분과 viewing area와의 교차점  $(x_c, y_c)$  구하기



- $y - y_0 = \frac{y_1 - y_0}{x_1 - x_0} (x - x_0)$
- $(x_c, y_c) = (x_{min}, y_c)$  지남
- $y_c - y_0 = \frac{y_1 - y_0}{x_1 - x_0} (x_{min} - x_0),$

$$x_c = x_{min}, \quad y_c = y_0 + \frac{y_1 - y_0}{x_1 - x_0} (x_{min} - x_0)$$

- 그 다음  $(x_0, y_0)$ 를  $(x_c, y_c)$ 로 바꾸고  $(x_c, y_c)$ 의 outcode를 0000으로 바꾼다



- 다시, 처음으로 돌아가서 양 끝점의 outcode가 trivial accept나 trivial reject인지 따지고 그렇지 않으면 다시 선분과 half space와의 **경계 교차점**을 구한다 (recursive하게)
- 이 경우에는 양 끝점의 outcode가 모두 0000이므로 trivial accept으로 종료

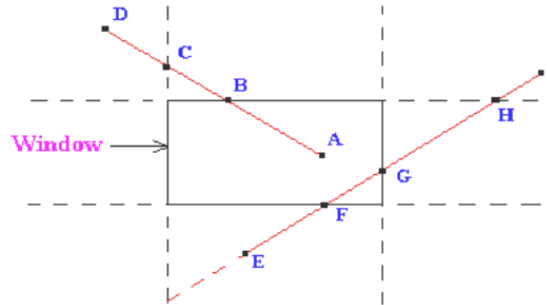


- **Cohen-Sutherland line clipping 알고리즘**
- 1. Given a line segment with endpoint  **$P1=(x1, y1)$ ,  $P2(x2, y2)$**
- 2. Compute the outcode for each endpoint
  - if both codes are 0000, **trivially accept (draw it)**
  - if both codes have a 1 in the same bit position, **trivially reject**
- 3. If a line cannot be trivially accepted or rejected, at least one of the two endpoints must lie outside the viewing area and the line segment crosses viewing area. The line must be clipped
- 4. Examine one of endpoints, say  **$P1=(x1, y1)$ . Read  $P1$ 's outcode in order. TBRL**
- 5. When a bit (1) is found, compute the **intersection I** of the corresponding viewing area's edge with the line from  $P1$  to  $P2$
- 6. **Replace  $P1$  with I and repeat the algorithm**

- // 알고리즘에서 recursive한 부분의 Pseudo 코드
- // TOP=1000, BOTTOM=0100, RIGHT=0010, LEFT=0001
- While (true) {
- if ( ! (outcode0 | outcode1)) { trivially accept; break; }
- else if (outcode0 & outcode1) { trivially reject; break; }
- else {
- pick an endpoint, which is outside, say outcodeOut
- if (outcodeOut & TOP) // point is above the viewing area
- find the intersection point
- else if (outcodeOut & BOTTOM) // point is below the viewing area
- find the intersection point
- else if (outcodeOut & RIGHT) // point is to the right of the viewing area
- find the intersection point
- else if (outcodeOut & LEFT) // point is to the left of the viewing area

Then, move outside point to intersection point

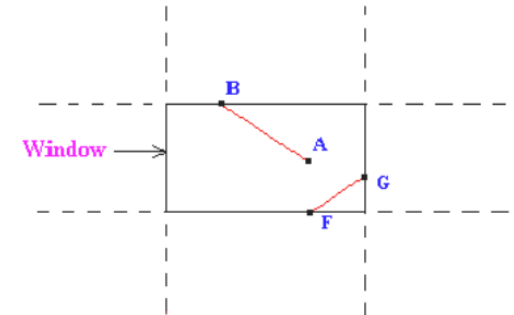
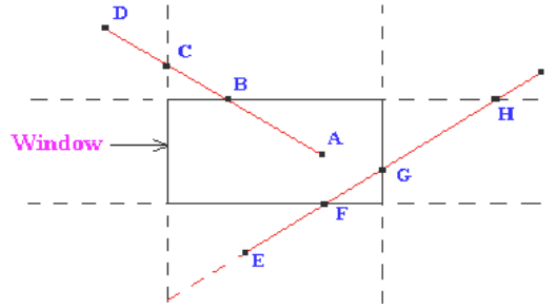
## ■ 선분 AD



### ■ A: outcode (0000), D: outcode (1001)

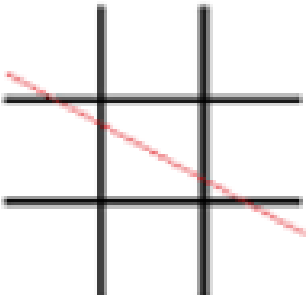
- 1. Trivially accept 되지 않음
- 2. Trivially reject 되지 않음
- 3. Choose D as the outside point
- 4. TBRL. Top edge에 의해 clip. 교차점 B 계산
- 5. B의 outcode 구함 (0000)
- 6. AB의 outcode 모두 0000 (bitwise OR 연산). Trivially accept 종료

## ■ 선분 TI



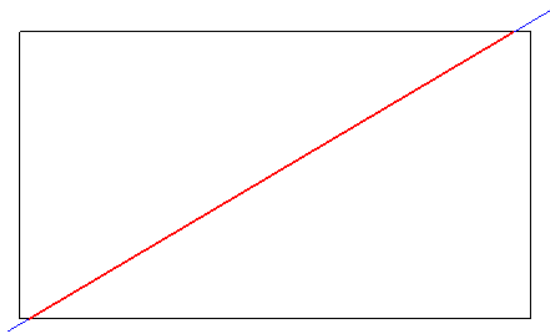
- E: outcode (0100), I: outcode (1010)
- 1. Trivially accept 되지 않음      2. Trivially reject 되지 않음
- 3. Choose E as the outside point
- 4. TBRL. Bottom edge에 의해 clip 됨. 교차점 F 계산
- 5. F의 outcode 구함 (0000)      6. F: 0000, I: 1010
- 7. Choose I as the outside point
- 8. TBRL. Top edge에 의해 clip 됨. 교차점 H 계산
- 9. H의 outcode 구함 (0010)      10. F: 0000 H: 0010
- 11. Choose H as the outside point
- 12. TBRL. Right edge에 의해 clip 됨. 교차점 G 계산
- 13. G의 outcode 구함 (0000)
- 14. FG의 outcode 모두 0000 (bitwise OR 연산). Trivially accept 종료

- 
- **Q: show that the maximum number of times Cohen-Sutherland can recursively call itself on a single input segment is four. Give an example of a segment.**



- 
- Cohen-Sutherland line clipping 알고리즘의 장점
  - bit 연산인 Boolean 연산으로 line clipping 판단
  - Viewing area와의 교차점도 필요한 경우에만 계산
  - 특히, 많은 line들이 존재하지만 대부분의 경우 clipping 되는 경우 매우 빠르게 동작한다
  - 3차원 line clipping으로 확장 가능
  - 한계: 오직 직사각형 모양의 viewing area 모양에서만 동작한다. 또한, it must be used recursively

- 
- 참고: wikipedia
  - [https://en.wikipedia.org/wiki/Cohen%E2%80%93Sutherland\\_algorithm](https://en.wikipedia.org/wiki/Cohen%E2%80%93Sutherland_algorithm)
  - Github 코드 예
  - [https://gist.github.com/vishal-wadhwa/71734603fcdd05c04131d341ed52f0ec/raw/e171fae352f88a1a29aa1c4da0eb84e0381a628d/cohen\\_sutherland.c](https://gist.github.com/vishal-wadhwa/71734603fcdd05c04131d341ed52f0ec/raw/e171fae352f88a1a29aa1c4da0eb84e0381a628d/cohen_sutherland.c)

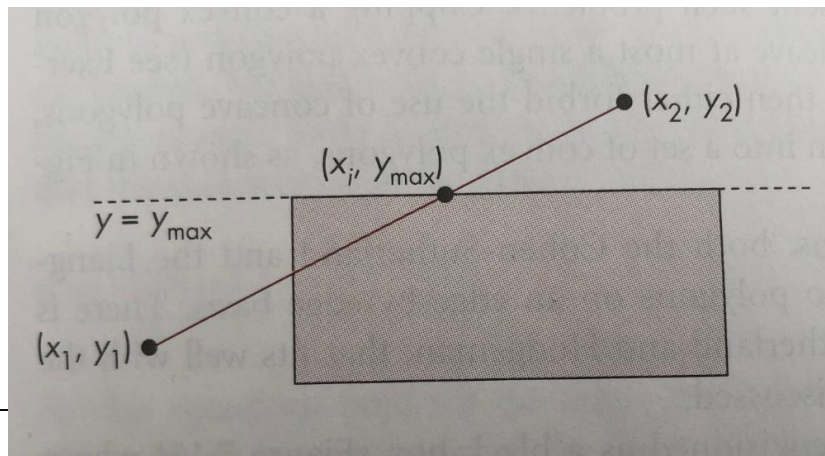


참고:

교차점 계산 예 (Top에 의한 line clipping), 다른 boundary도 유사함

선분:  $(x_1, y_1)$ ,  $(x_2, y_2)$ 의 Top ( $y=y_{\max}$ ), clip boundary와의 교차점  $(x_3, y_3)$  계산

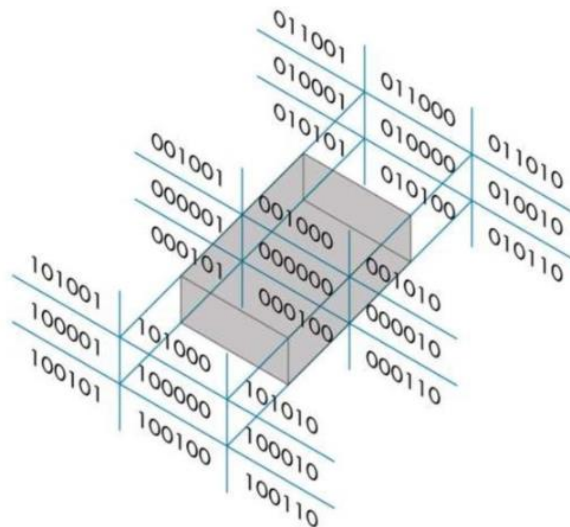
- $y - y_1 = \frac{y_2 - y_1}{x_2 - x_1} (x - x_1)$
- $y_{\max} - y_1 = \frac{y_2 - y_1}{x_2 - x_1} (x - x_1)$
- $x_3 = x_1 + (y_{\max} - y_1) \frac{x_2 - x_1}{y_2 - y_1}$ ,  $y_3 = y_{\max}$





- 
- 예: 직사각형 모양의 viewing area가  $\text{left}=10$ ,  $\text{right}=20$ ,  $\text{bottom}=10$ ,  $\text{top}=20$ 으로 정의되어 있다. 선분:  $(8, 15)$ ,  $(15, 22)$ 가 Top에 의해 clipping되었을 때의 교차점을 찾아보자

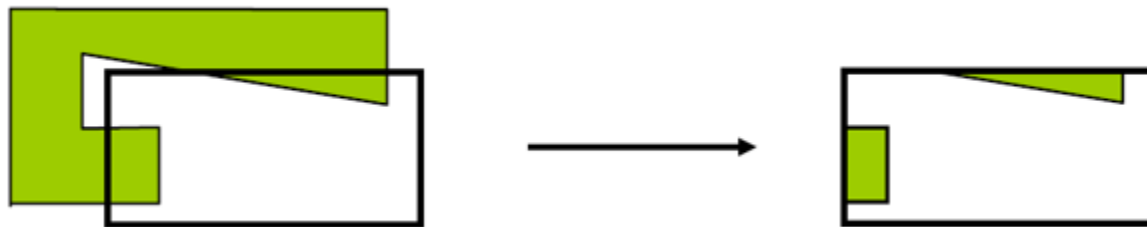
- **Cohen-Sutherland algorithm can be extended to 3D**
- We replace the 4-bit outcode with **a 6-bit outcode**
- The additional 2 bits are set of the point lies either in **front of or behind the clipping volume**
- The testing strategy is virtually identical for the 2D and 3D cases



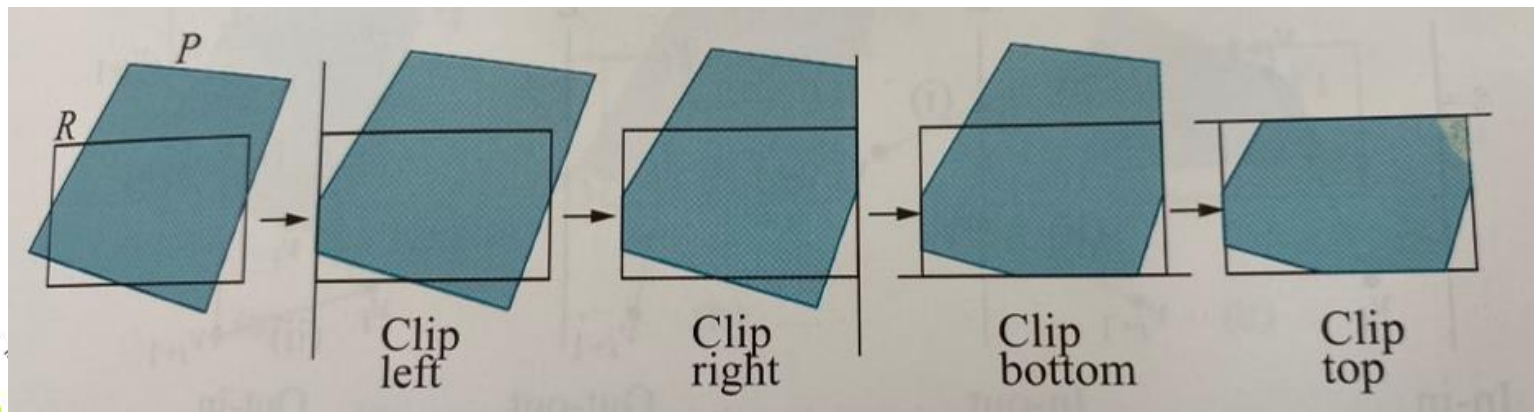
---

## ■ Polygon clipping algorithm

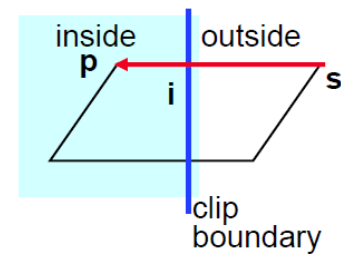
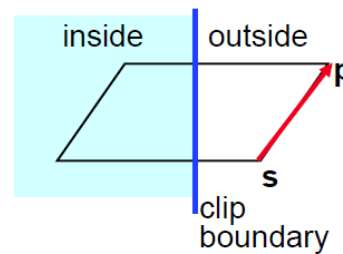
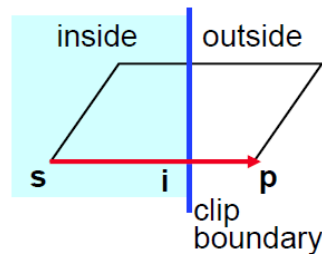
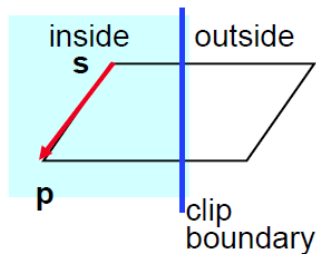
- Polygon clipping 알고리즘은 기본적으로 **line clipping 알고리즘의 확장**으로 직사각형 모양의 viewing area 의 바깥쪽에
- 기본적으로 polygon을 이루는 edge들을 순차적으로 방문하면서 polygon clipping이 이루어 진다
- Polygon clipping 알고리즘에서 가장 많이 사용되는 알고리즘 중의 하나 인 **Sutherland-Hodgman polygon clipping algorithm** 에 대해서 배워보자



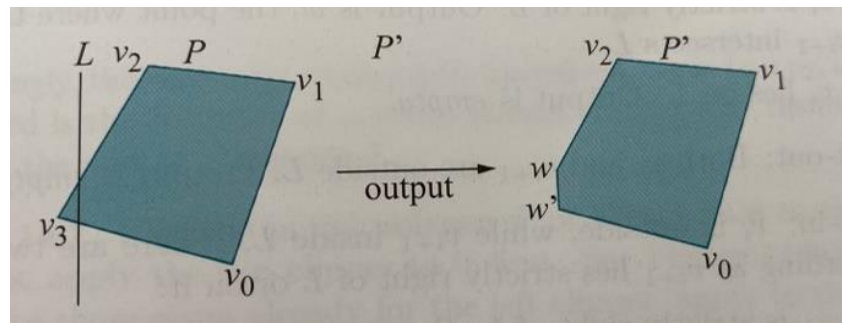
- **Sutherland-Hodgman polygon clipping algorithm**
- 2차원 직사각형 모양의 viewing area를 기준으로 2차원 convex polygon을 clip하기 위한 알고리즘
- Polygon을 viewing area 기준으로 clip시 4개의 clip boundary (left, right, bottom, top) 기준으로 edge clip을 반복적으로 수행한다
- Left->right->bottom->top은 하나의 ordering이고 어떤 ordering도 가능



- Polygon의 각각의 edge에 대하여 clipping시에 총 4가지의 경우가 생긴다
- Polygon의 4개의 Edge가 있을 때, 각각의 Edge **sp**에 대하여
  1. sp 모두 viewing area 내부 => **output: sp**
  2. s는 내부, p는 외부=> 내부점 s, 외부점 p대신 교차점 i, **output: si**
  3. sp 모두 viewing area 외부 => **no output**
  4. s는 외부, p는 내부=> 내부점 p, 외부점 s대신 교차점 i, **output: ip**

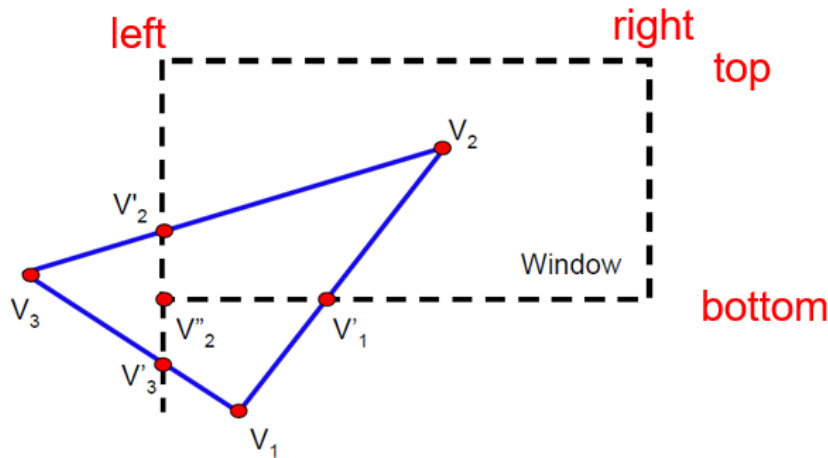


- Polygon clipping algorithm에서의 input은 ordered list  $\{v_0, v_1, \dots, v_n\}$  of the vertices of a convex polygon  $P$  and a vertical straight line  $L$
- The output is an ordered list of vertices of the polygon  $P'$  resulting from clipping  $P$  off to the  $L$



- 예: Clip left: input= $\{v_0, v_1, v_2, v_3\}$ , output= $\{v_0, v_1, v_2, w, w'\}$

## ■ 예: viewing area (window), polygon (triangle)



Step 1:

input:  $V_1V_2V_3$

**Left clipping**

1. Edge:  $V_1V_2$ , output:  $V_1V_2$
  2. Edge:  $V_2V_3$ , output:  $V_2V'_2$
  3. Edge:  $V_3V_1$ , output:  $V'_3V_1$
- Output:  $V_1 V_2 V'_2 V'_3$

Step 2:

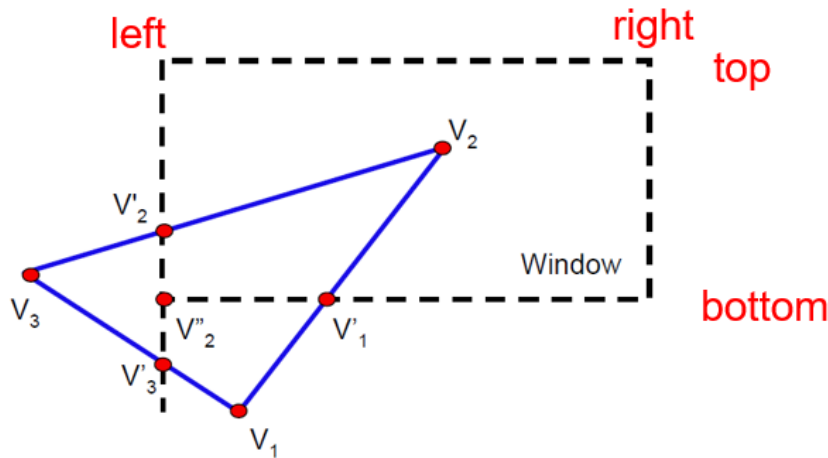
input:  $V_1V_2 V'_2 V'_3$

**Right clipping**

1. Edge:  $V_1V_2$ , output:  $V_1V_2$
  2. Edge:  $V_2V'_2$ , output:  $V_2V'_2$
  3. Edge:  $V'_2V'_3$ , output:  $V'_2V'_3$
  4. Edge:  $V'_3V_1$ , output:  $V'_3V_1$
- Output:  $V_1 V_2 V'_2 V'_3$



## ■ 예: viewing area (window), polygon (triangle)



Step 3:

Input:  $V_1V_2V_2'V_3'$

**Bottom clipping**

1. Edge:  $V_1V_2$ , output:  $V_1'V_2$
2. Edge:  $V_2V_2'$ , output:  $V_2V_2'$
3. Edge:  $V_2'V_3'$ , output:  $V_2'V_2''$
4. Edge:  $V_3'V_1$ , output: none

Output:  $V_1'V_2V_2'V_2''$

Step 4:

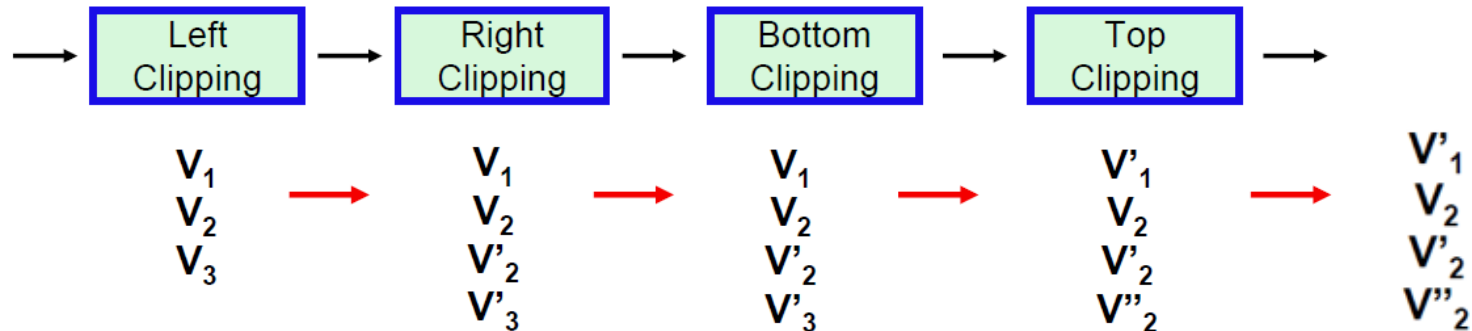
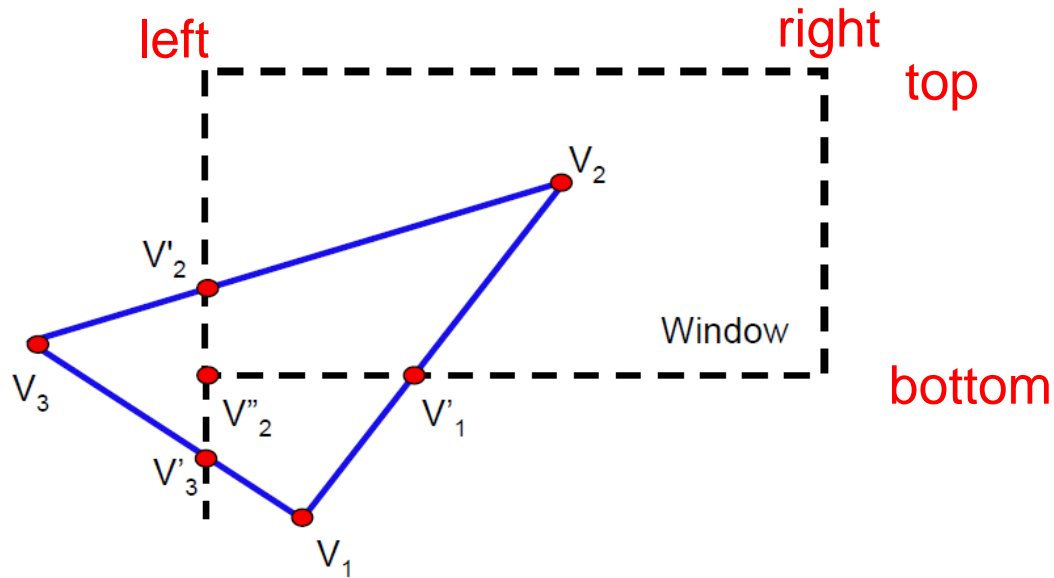
input:  $V_1'V_2V_2'V_2''$

**Top clipping**

....

**Output:  $V_1'V_2V_2'V_2''$**

- 예: viewing area (window), polygon (triangle)



- 
- **GitHub에 있는 Sutherland-Hodgman polygon clipping algorithm**
  - <https://www.dropbox.com/s/px0gheyw80giy1m/sutherland.txt?dl=0>
  - **window와 polygon (point 위치) 설정 후에 키보드의 'c' 키 누름**