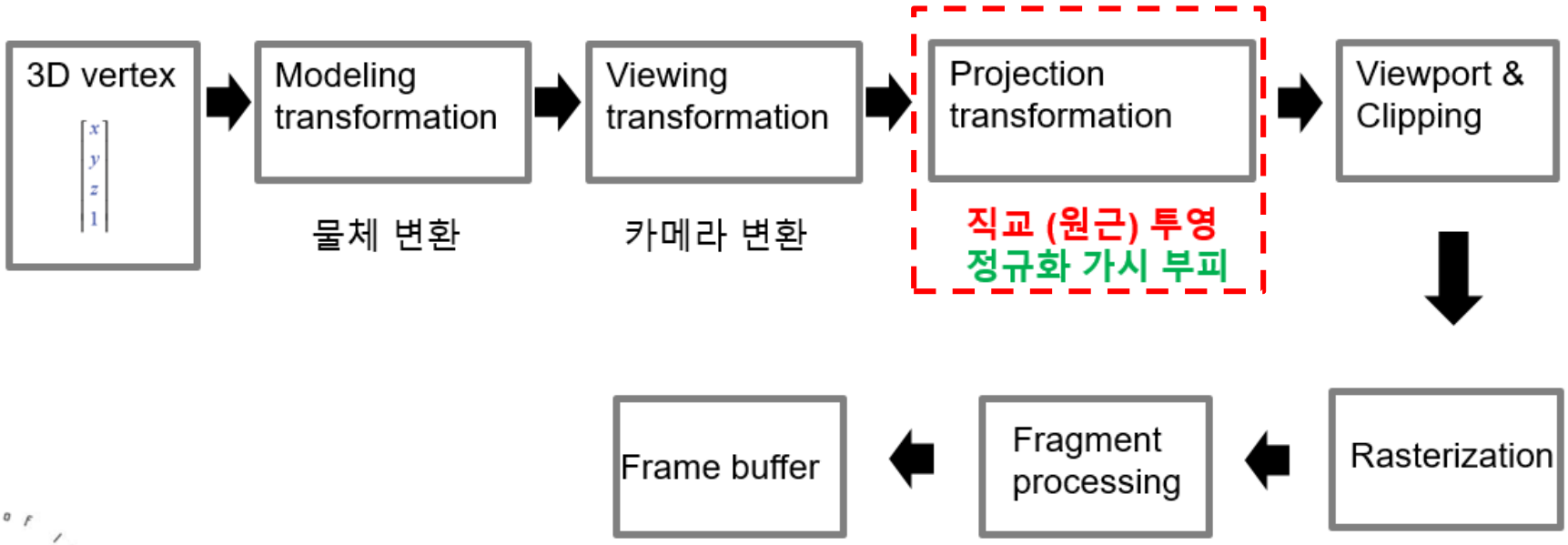
Computer Graphics

Prof. Jibum Kim

Department of Computer Science & Engineering

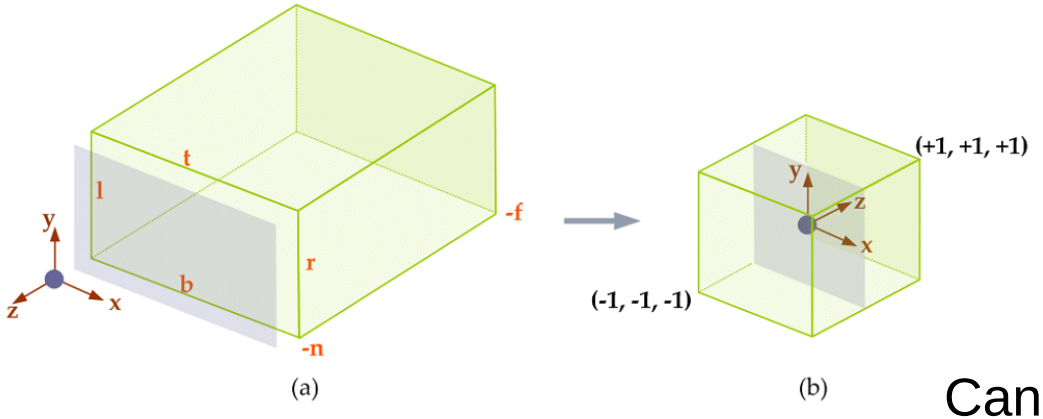
Incheon National University

- 정규화 가시 부피
- (canonical view volume, CVV)

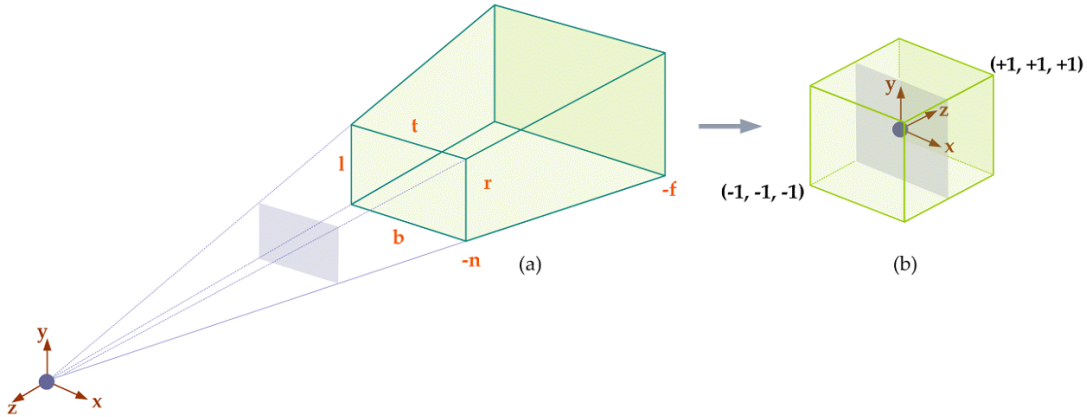


- OpenGL 내부에서는 투영 시 정규화 가시 부피 (canonical view volume)라는 단계를 내부적으로 한 단계 더 거친다
- 가시 부피 (viewing volume, viewing frustum)안의 각 vertex를 각 축에서 -1~1 사이의 standard cube (정육면체)로 변환 시킴
- 정규화 가시 부피 사용 이유
- 1. 직교 투영, 원근 투영을 동일한 모습의 정규화 가시 부피로 변형 (동일 파이프 라인 사용)
- 2. 정육면체를 기준으로 하기에 clipping 및 hidden surface revmovel이 쉽다 (viewing frustum을 사용시 clipping이 어렵다)
- 3. Canonical viewing volume후에 viewport 좌표 계로 mapping이 쉽다

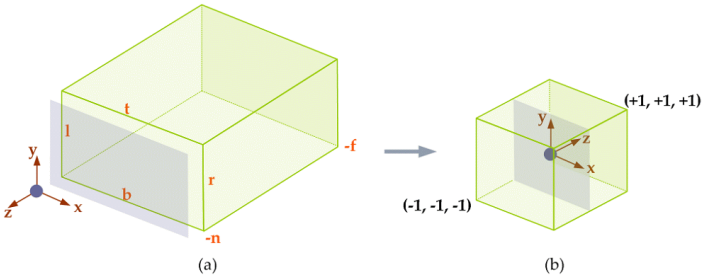
- **Canonical view volume으로의 변환 (주의: 좌표계 방향이 다르다)**



Canonical view volume



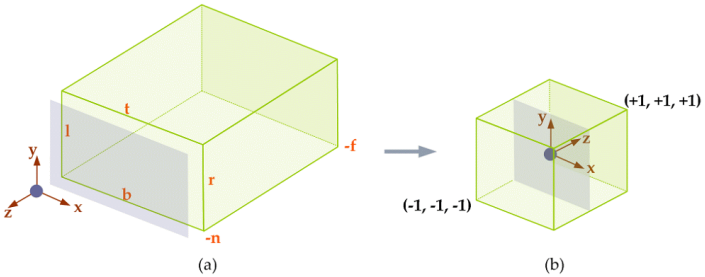
- 직교 투영 시 정규화 가시 부피로의 변환 행렬을 구해보자
- 왼쪽: 가시부피: glOrtho(l, r, b, t, n, f), 오른쪽: 정규화 가시 부피



- 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: glOrtho(l, r, b, t, n, f), 오른쪽: 정규화 가시 부피

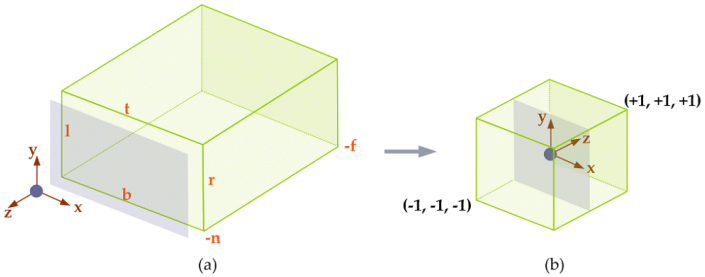


- 2. 가시 부피의 크기: x축 방향: $r - l$, y축 방향: $t - b$, z축 방향: $-n + f$

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: glOrtho(l, r, b, t, n, f), 오른쪽: 정규화 가시 부피



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

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

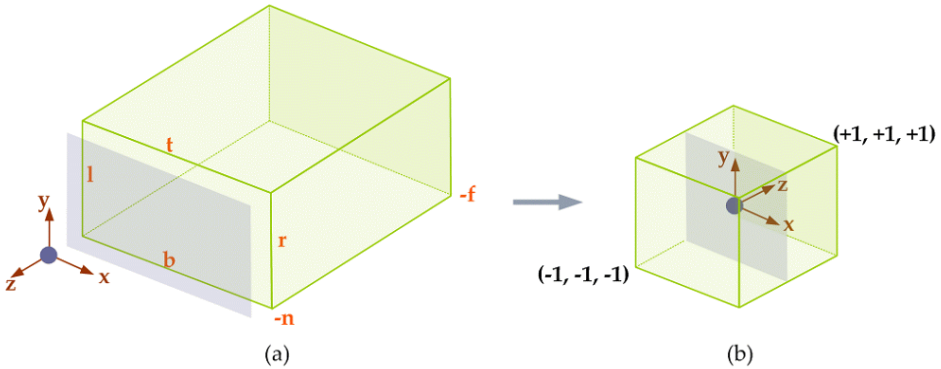
- 직교 투영 시 정규화 가시 부피로의 변환 행렬
- 정규화 변환 행렬 N , $N = R \cdot S \cdot T$
- 이 정규화 변환 행렬이 glOrtho(l, r, b, t, n, f) 사용시
- projection matrix로 올라감

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

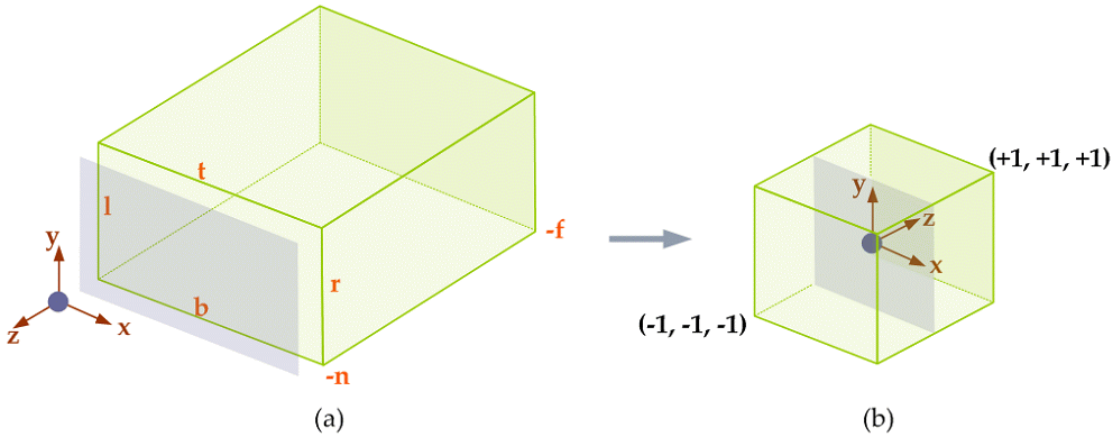
$$N = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- 정규화 가시 부피 후의 위치 (x_n, y_n, z_n), 가시 공간 위치 (x_e, y_e, z_e)
- (x_n, y_n, z_n)을 절단 좌표계 (clip coordinate system)이라고도 부른다

$$\begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$

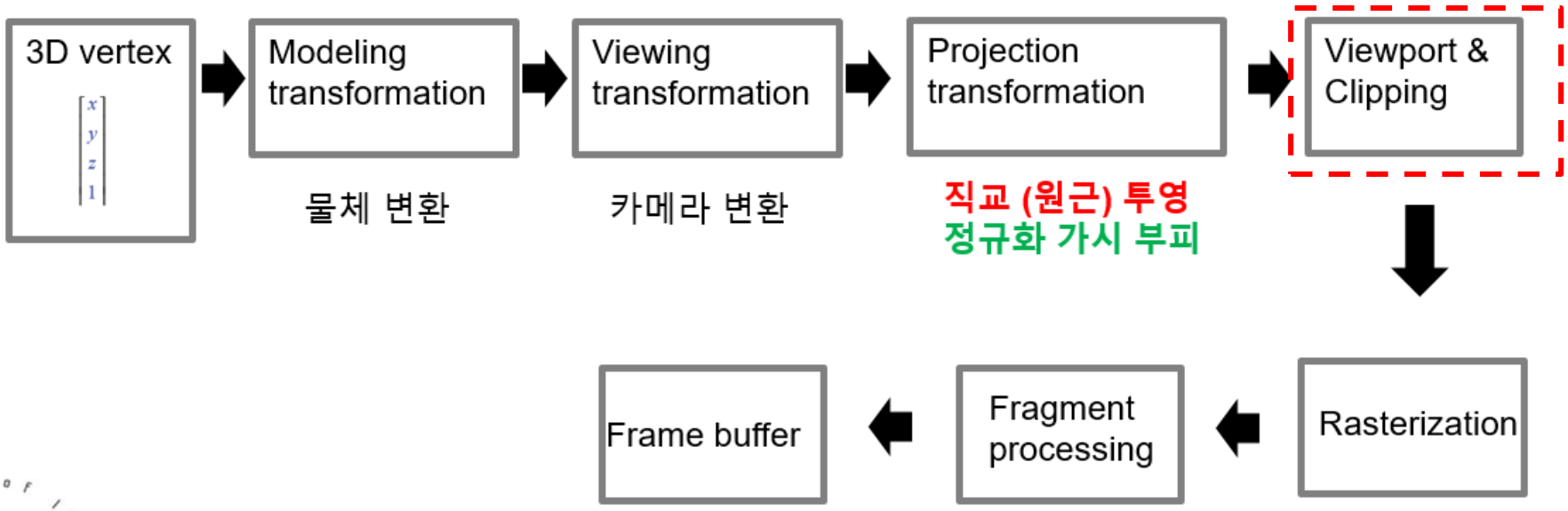


- OpenGL에서의 projection 세부 단계
- 1. 정규화 가시 부피 변환
- 2. Clipping
- 3. 2차원으로의 projection (z값 무시)



- **Clipping algorithm**

- **Clipping 알고리즘**
- **가시 부피 (viewing volume, viewing frustum)=> 정규화 가시 부피로 변환한 후에 CVV밖에 있는 선분, polygon에 대해 clipping 수행 (Line clipping, Polygon clipping)**



- **Bitwise operation**

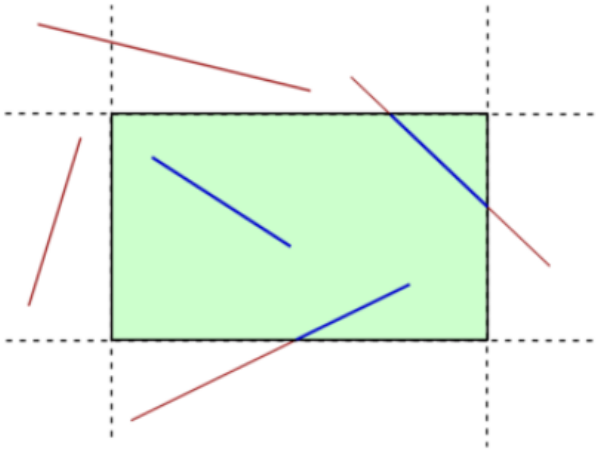
- **Bitwise AND 연산 (표기: &)**
- AND: 둘다 1일 때만 결과가 1
- 비트 열의 같은 위치의 이진수끼리 AND 연산
- 예: X=1000, Y=0000
- X&Y=0000, 혹은 0 으로 표시
- **Bitwise OR 연산 (표기: |)**
- 둘중에 하나라도 1이면 결과가 1
- 예: X=1000, Y=0000
- X|Y=1000

X	Y	X&Y	X|Y
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

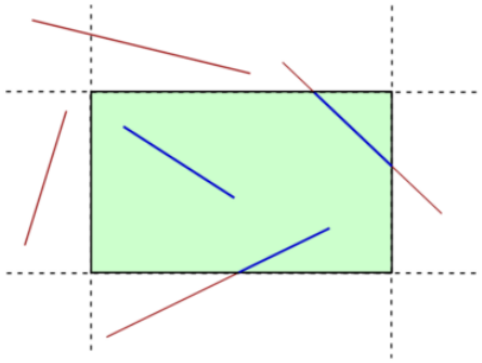
- **Cohen-Sutherland line clipping**

Line clipping algorithm

- **Cohen-Sutherland line clipping 알고리즘**
- **2차원 선분을 2차원 직사각형 기준으로 절단하는 Line clipping 알고리즘으로 divide and conquer 전략을 쓰며 recursive하게 동작하는 알고리즘으로 매우 중요한 알고리즘이다**
- **Viewing area (초록색): 예: 직사각형 모양의 가시 공간, CVV**
- **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)**

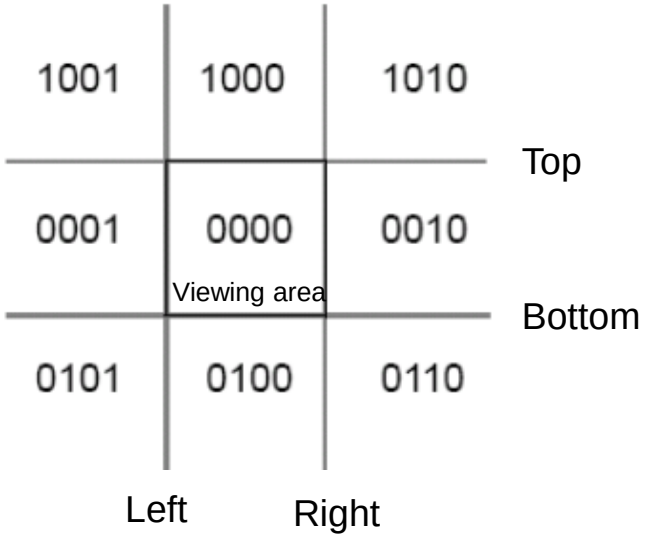
- **Basic idea**
- **선분의 양 끝점에 비트열 형식의 location code (outcode) 부여**
- **전체 영역을 두 개의 half-space로 나눈 후, inside인지 outside 인지 따져서 할당. 왜 half-space로 나눌까?**
- **4-bit의 outcode, $b_0 b_1 b_2 b_3$ (TBRL)**

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

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

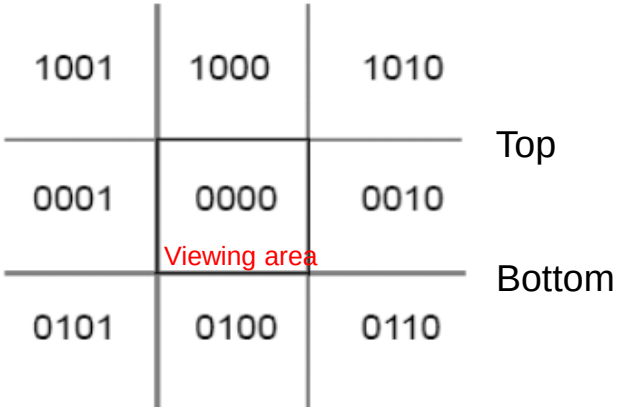
3) $b_2 = \begin{cases} 1, if\ x > right \\ 0 \end{cases}$, 점이 right오른쪽? 왼쪽? 판단

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

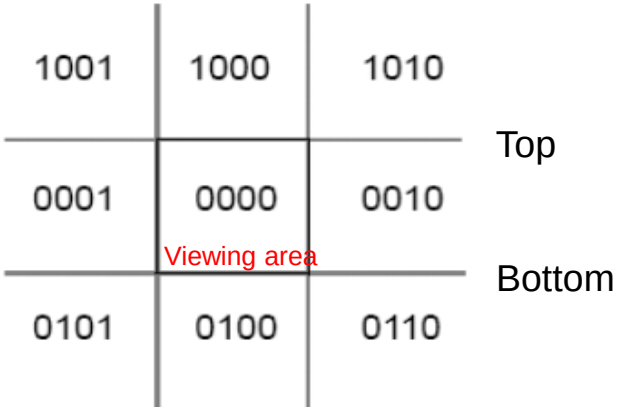


할당 후 양 끝점에 대해 비트 논리 연산 수행 (장점이 뭘까?)

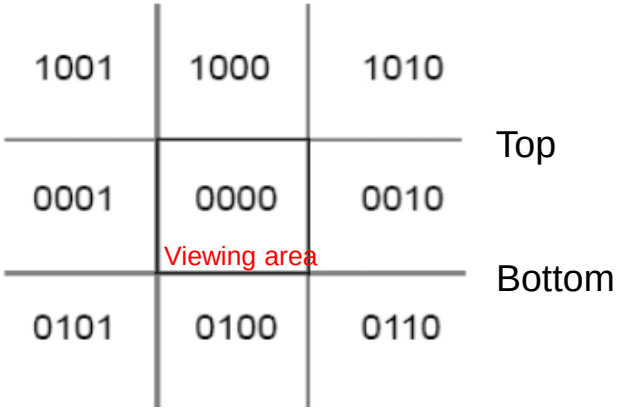
- Case 1: 선분의 양 끝점 : o1=outcode(x1, y1), o2=outcode(x2,y2)
- 예: o1=0000, o2=0000, 는 어떤 선분인가 그려보자.
- Q: o1과 o2가 어떤 bit 연산을 해야 두 선분의 양 끝점이 모두 viewing area 안에 있다는 걸 알 수 있을까?
- o1 | o2 = 0000
- 이를 알고리즘에서는 trivial accept 라고 한다



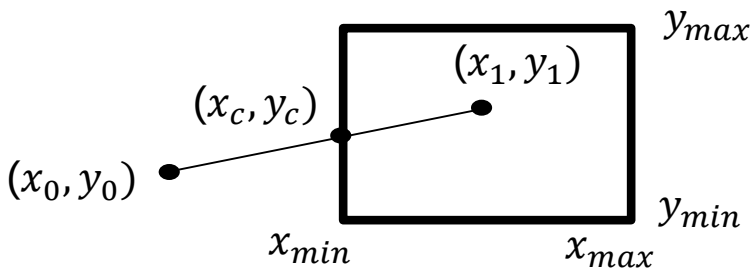
- Case 2: 선분의 양 끝점 : o1=outcode(x1, y1), o2=outcode(x2,y2)
- 예: o1=1001, o2=0101, 는 어떤 선분인가 그려보자.
- Q: o1과 o2가 어떤 bit 연산을 해야 이 선분은 완전히 clipping 되어야 하는 선분이란걸 알 수 있을까? Bitwise-AND
- o1 & o2≠0000 (half space의 특징을 이용)
- 이를 알고리즘에서는 trivial reject 라고 한다



- **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 교차점을 구한다**
- **이 경우: TBRL중에 L이 0이 아님**

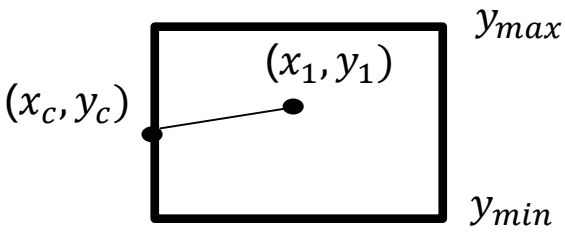


- **예: 선분과 viewing area와의 교차점 (x_c, y_c) 구하기**



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

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

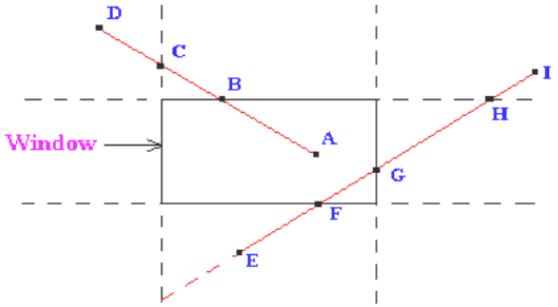


- 다시, 처음으로 돌아가서 양 끝점의 outcode가 trivial accept나 trivial reject인지 따지고 그렇지 않으면 다시 선분과 half space와의 경계 교차점을 구한다 (recursive하게)
- 이 경우에는 양 끝점의 outcode가 모두 0000이므로 trivial accept으로 종료

- 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

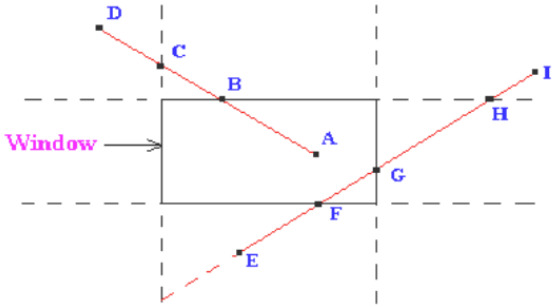
```
// 알고리즘에서 recursive한 부분의 Pseudo 코드
// TOP=1000, BOTTOM=0100, RIGHT=0010, LEFT=0001
While (true) {
  if ( ! (outcode0 | outcode1)) {  trivially accept; break; }
 else if (outcode0 & outcode1) { trivially reject; break; }
 else {
   pick an endpoint, which is outside, say outcodeOut
   if (outcodeOut & 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
}
```

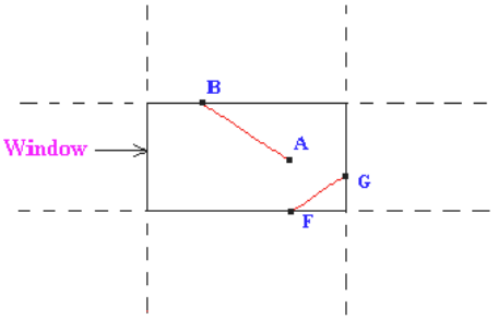
- 선분 AD



- 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 EI





- 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

- 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

- **참고: wikipedia**
- **https://en.wikipedia.org/wiki/Cohen%E2%80%93Sutherland_algorithm**

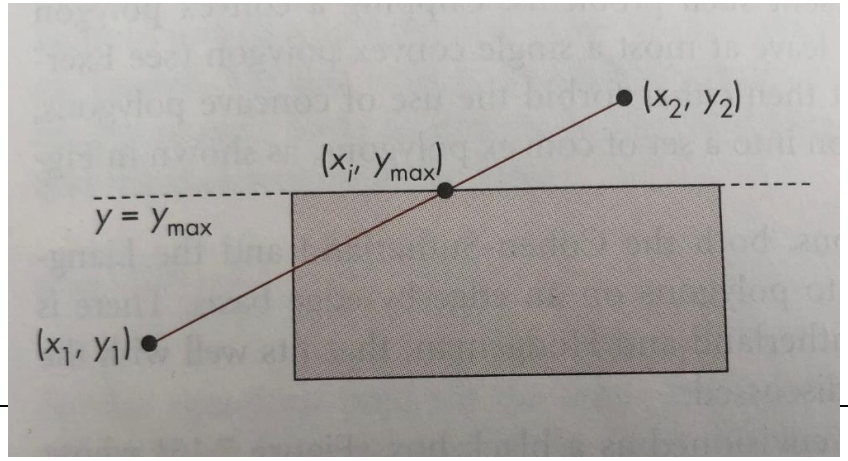
- **Github 코드 예**
- **https://gist.githubusercontent.com/vishal-wadhwa/71734603fcdd05c04131d341ed52f0ec/raw/e171fae352f88a1a29aa1c4da0eb84e0381a628d/cohen_sutherland.c**

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

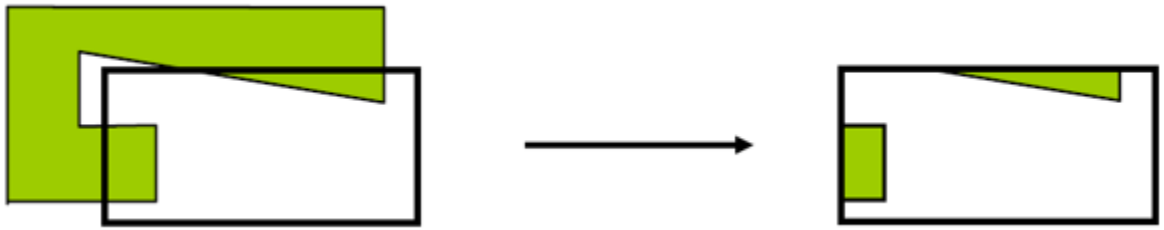
- $y - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$
- $y_{max} - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1)$
- $x_3 = x_1 + (y_{max} - y_1)\frac{x_2 - x_1}{y_2 - y_1}$, $y_3 = y_{max}$



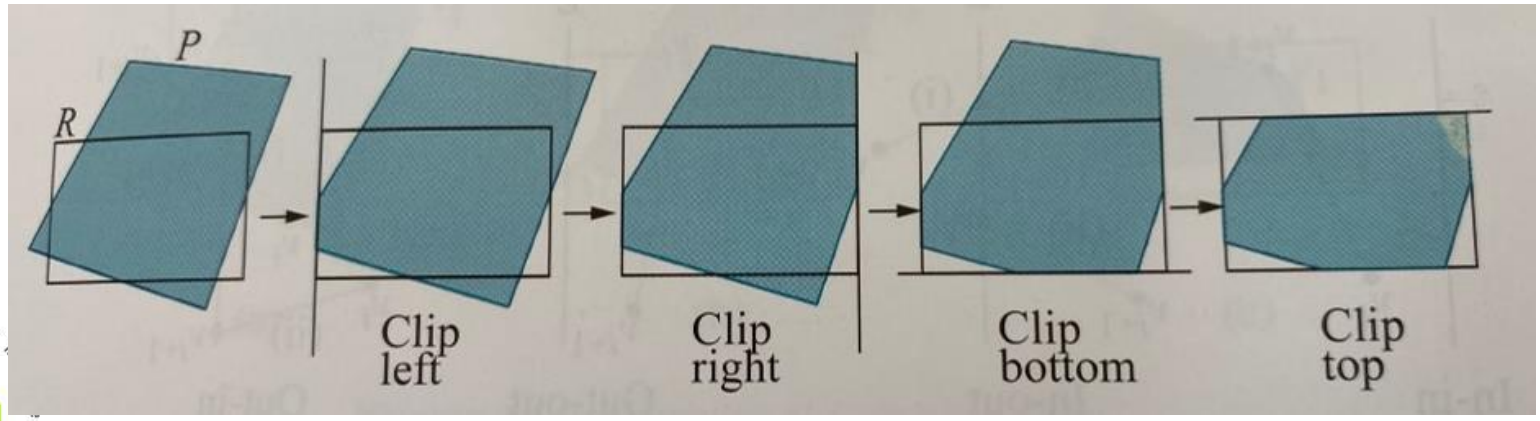
- 예: 직사각형 모양의 viewing area가 left=10, right=20, bottom=10, top=20으로 정의되어 있다. 선분: (8, 15), (15, 22)가 Top에 의해 clipping되었을 때의 교차점을 찾아보자

- **Polygon clipping algorithm**

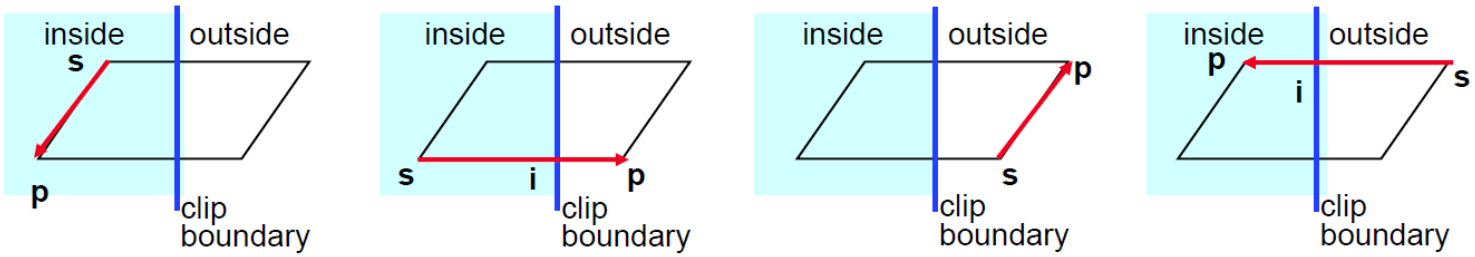
- Polygon clipping 알고리즘은 기본적으로 line clipping 알고리즘의 확장으로 직사각형 모양의 viewing area 의 바깥쪽에
- 기본적으로 polygon을 이루는 edge들을 순차적으로 방문하면서 polygon clipping이 이루어 진다
- Polygon clipping 알고리즘에서 가장 많이 사용되는 알고리즘 중의 하 나 인 Sutherland-Hodgman polygon clipping algorithm 에 대해서 배워보자



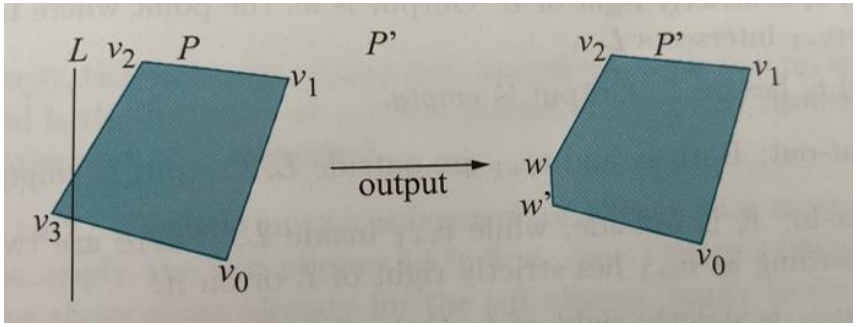
- **Sutherland-Hodgman polygon clipping algorithm**
- **2차원 직사각형 모양의 viewing area를 기준으로 2차원 convex polygon을 clip하기 위한 알고리즘**
- **Polygon을 viewing area 기준으로 clip시 4개의 clip boundary (left, right, bottom, top)기준으로 edge clip을 반복적으로 수행한다**
- **Left->right->bottom->top은 하나의 ordering이고 어떤 ordering도 가능**



- Polygon의 각각의 edge에 대하여 clipping시에 총 4가지의 경우가 생긴다
- Polygon의 4개의 Edge가 있을 때, 각각의 Edge sp에 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

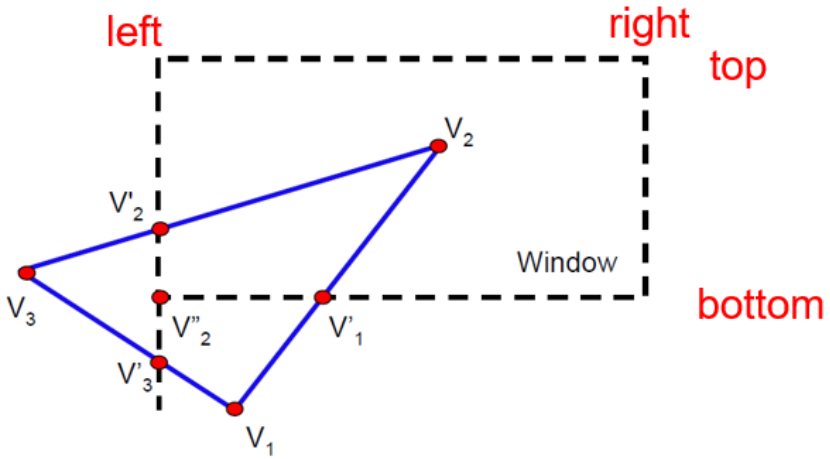


- Polygon clipping algorithm에서의 input은 ordered list {v_0, v_1,…,v_n} of the vertices of a convex polygon P and a vertical straight line L
- The output is an ordered list of vertices of the polygon P' resulting from clipping P off to the L



- 예: Clip left: input={vo, v1, v2, v3}, output={v0, v1, v2, w, w'}

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



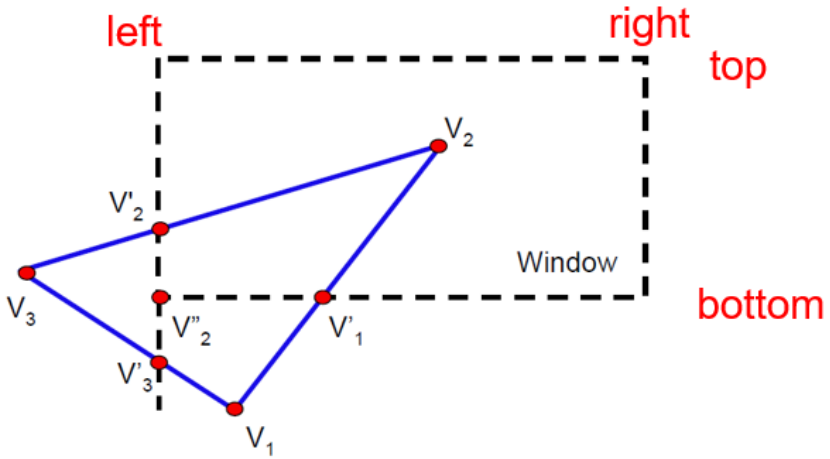
Step 1:
input: V1V2V3
Left clipping
1. **Edge: V1V2, output: V1V2**
2. **Edge: V2V3, output: V2V2'**
3. **Edge:V3V1, output: V3'V1**

Output: V1 V2 V2' V3'

Step 2:
input: V1V2 V2' V3'
Right clipping
1. **Edge: V1V2, output: V1V2**
2. **Edge: V2V2', output: V2V2'**
3. **Edge: V2'V3': output: V2'V3'**
4. **Edge:V3'V1, output: V3'V1**

Output: V1 V2 V2' V3'

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



Step 3:
Input: V1V2V2'V3'
Bottom clipping
1. **Edge: V1V2, output: V1'V2**
2. **Edge: V2V2', output: V2V2'**
3. **Edge: V2'V3', output: V2'V2"**
4. **Edge: V3'V1, output: none**

Output: V1'V2V2'V2"

Step 4:
input: V1'V2V2'V2"
Top clipping
….
Output: V1'V2V2'V2"

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