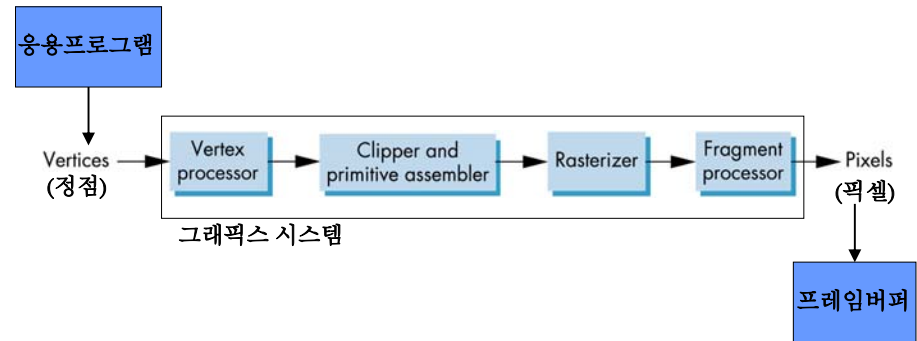


# From Vertices to Fragments

321190  
2008년 봄학기  
6/9/2008  
박경신

## Geometric Pipeline

- 기하 파이프라인 (geometric pipeline)
  - 정점 처리 (vertex processing)
  - 클리핑과 기본요소로 조립 (clipping and primitive assembly)
  - 래스터화 (rasterization)
  - 단편 처리 (fragment processing)

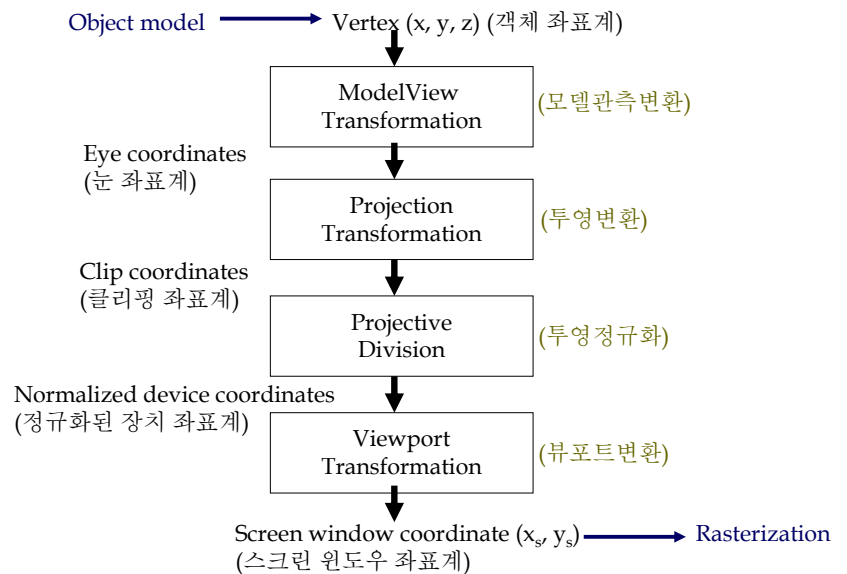


## Four Major Tasks

- 모델링 (modeling)
  - 기하학적 객체들의 집합 생성
- 기하학적 처리 (geometric processing)
  - 변환 (transformation)
  - 음영 (shading)
  - 기본요소조립 (primitive assembly), 클리핑 (clipping)
  - 은면처리 (hidden surface removal)
- 래스터화 (rasterization)
  - 주사변환 (scan conversion) 알고리즘
- 단편 처리 (fragment processing)
  - 엔티앨리어싱 (anti-aliasing)

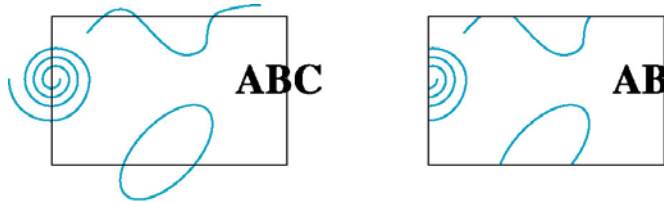


## Geometric Processing



## Clipping

- 2차원 클리핑 윈도우 (clipping window)
- 3차원 클리핑 경계 입체 (clipping volume)
- 곡선과 텍스트는 먼저 선과 다각형으로 바꿔줄 것

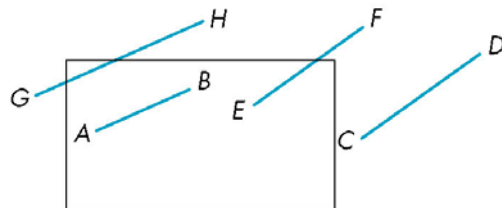


## Clipping

- 선분 클리핑 알고리즘
  - Cohen-Sutherland 클리핑 알고리즘
    - 선분의 끝점에 외곽부호 (outcode)를 사용
    - 많은 수의 선분이 있지만 실제로 몇 개만 디스플레이되는 경우에 가장 잘 작동함
    - 알고리즘이 간단하며, 3차원 클리핑 알고리즘으로 확장이 쉬움
  - Liang-Barsky 클리핑 알고리즘
    - 직선의 매개변수 형식을 사용
    - Cohen-Sutherland 방식과 비교하여 선분 출입(즉, 교차점의 계산)의 여러 번 반복을 줄여줌
    - 3차원 클리핑 알고리즘으로 확장이 쉬움
- 다각형 클리핑 알고리즘
  - Sutherland-Hodgeman 클리핑 알고리즘
    - 다각형의 정점을 클리핑 윈도우의 각 경계면에 대해 차례로 처리함

## 2D Line-Segment Clipping

- 선분 클리핑 (clipping 2D line segments)
  - 절단기(clipper)는 어떤 기본 요소 또는 그 일부가 화면에 나타나야 되고 래스터화기로 보내져야 하는 지를 결정
  - 수용 (accepted): 지정한 관측 공간에 들어온 기본 요소는 수용
  - 거부 (rejected) 또는 선별 (culled): 화면에 나타날 수 없는 기본 요소는 제거됨



## Cohen-Sutherland Algorithm

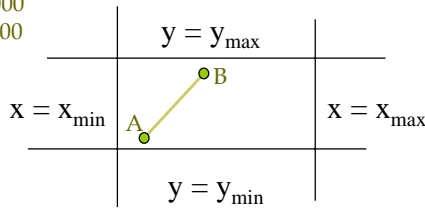
- Cohen-Sutherland 클리핑 알고리즘
  - 클리핑 윈도우의 4개의 변 무한대로 확장하고, 공간을 9개의 영역으로 분할
 

|               |                       |               |
|---------------|-----------------------|---------------|
| 1001          | 1000<br>$y = y_{max}$ | 1010          |
| 0001          | 0000                  | 0010          |
| $x = x_{min}$ |                       | $x = x_{max}$ |
| 0101          | 0100<br>$y = y_{min}$ | 0110          |
  - 각 영역에 고유의 외곽부호 (outcode),  $b_0b_1b_2b_3$ 를 다음과 같이 할당
 
$$b_0 = \begin{cases} 1 & \text{if } y > y_{max} \\ 0 & \text{otherwise} \end{cases} \quad b_1 = \begin{cases} 1 & \text{if } y < y_{min} \\ 0 & \text{otherwise} \end{cases} \quad b_2 = \begin{cases} 1 & \text{if } x > x_{max} \\ 0 & \text{otherwise} \end{cases} \quad b_3 = \begin{cases} 1 & \text{if } x < x_{min} \\ 0 & \text{otherwise} \end{cases}$$
  - 외곽부호에 기초하여 4가지 경우를 판단

## Cohen-Sutherland Algorithm

- 선분 AB의 경우: A's outcode = B's outcode = 0
  - 선분의 양 끝점이 클리핑 윈도우 내부에 있는 경우, **accepted**
- 선분 CD의 경우: C's outcode AND D's outcode ≠ 0
  - 선분의 양 끝점이 클리핑 윈도우의 **같은 변의 외부**에 있는 경우, **rejected**

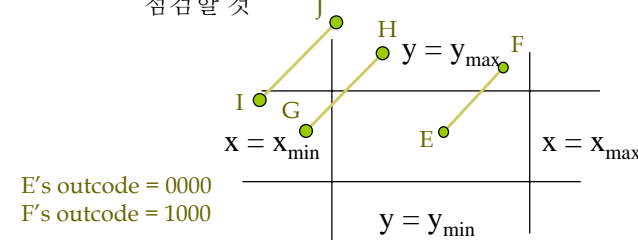
A's outcode = 0000  
B's outcode = 0000



C's outcode = 0010  
D's outcode = 1010  
C AND D = 0010 ≠ 0

## Cohen-Sutherland Algorithm

- 선분 EF의 경우: E's outcode ≠ 0, F's outcode = 0
  - 선분의 한 끝점은 클리핑 윈도우 내부에 있고, 다른 하나는 외부에 있는 경우, **subdivide**
  - 1개 교차점 (intersection)을 찾아야 함
- 선분 GH, 선분 IJ의 경우: G's outcode AND H's outcode = 0
  - 선분의 양 끝점이 모두 외부에 있는 경우, **subdivide**. 선분 GH 경우 선분의 일부가 클리핑 윈도우 내부에 있음
  - 적어도 윈도우 한 변과 교차 계산하고 그 결과 점의 외곽부호를 점검할 것



E's outcode = 0000  
F's outcode = 1000

G's outcode = 0001  
H's outcode = 1000  
G AND H = 0000

I's outcode = 0001  
J's outcode = 1000  
I AND J = 0000

## Liang-Barsky Algorithm

- Liang-Barsky 클리핑 알고리즘

1. 매개변수형 직선 공식

$$P(\alpha) = (1-\alpha)P_1 + \alpha P_2, 0 \leq \alpha \leq 1$$

$$x(\alpha) = (1-\alpha)x_1 + \alpha x_2$$

$$y(\alpha) = (1-\alpha)y_1 + \alpha y_2$$

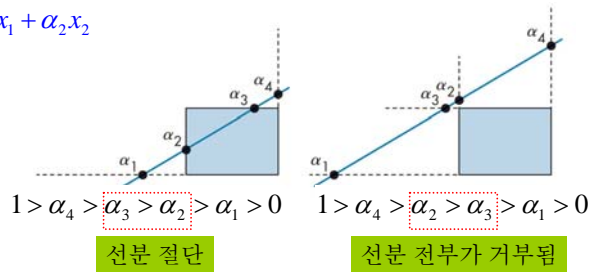
2. 선분이 클리핑 윈도우의 확장된 변과 교차하는 4점을 계산

$$y_{\max} = (1-\alpha_3)y_1 + \alpha_3 y_2$$

$$x_{\min} = (1-\alpha_2)x_1 + \alpha_2 x_2$$

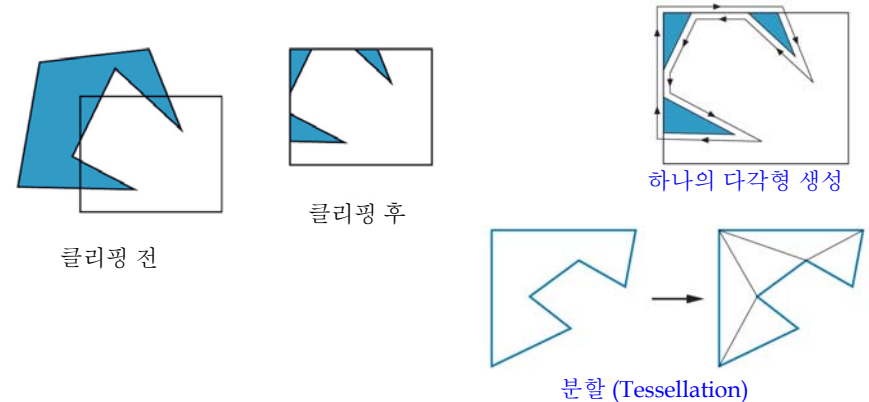
$$\alpha_3 = \frac{y_{\max} - y_1}{y_2 - y_1}$$

$$\alpha_2 = \frac{x_{\min} - x_1}{x_2 - x_1}$$



## Polygon Clipping

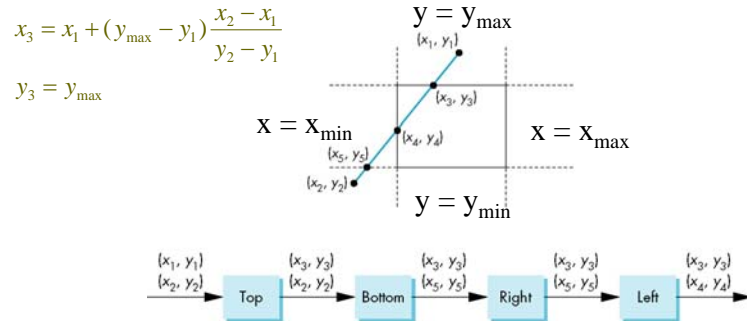
- 오목한 다각형 (concave polygon)의 클리핑
  - 방법1: 클리핑 후 하나의 다각형으로 묶는 방법
  - 방법2: 오목한 다각형의 집합으로 나누고 (tessellate), 클리핑



## Pipeline Clipping of Line Segments

### □ Sutherland-Hodgeman 알고리즘

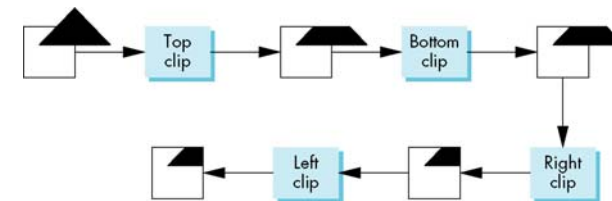
- 절단기를 각각 윈도우의 한 변에 대해서 클리핑하는 더 간단한 절단기의 파이프라인으로 세분함



## Pipeline Clipping of Polygons

### □ Sutherland-Hodgeman 알고리즘

- Input: 다각형 (정점 리스트)과 클리핑 면
- Output: 새로운 클리핑이 된 다각형 (정점 리스트)
- 2차원 다각형에 대한 연속적인 절단기(pipeline clipping of polygons)
- 3차원의 경우, 전면 (front)과 후면 (back) 클리핑을 추가함

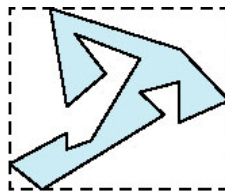


## Bounding Boxes

### □ 다각형의 축정렬 경계상자 (axis-aligned bounding box)

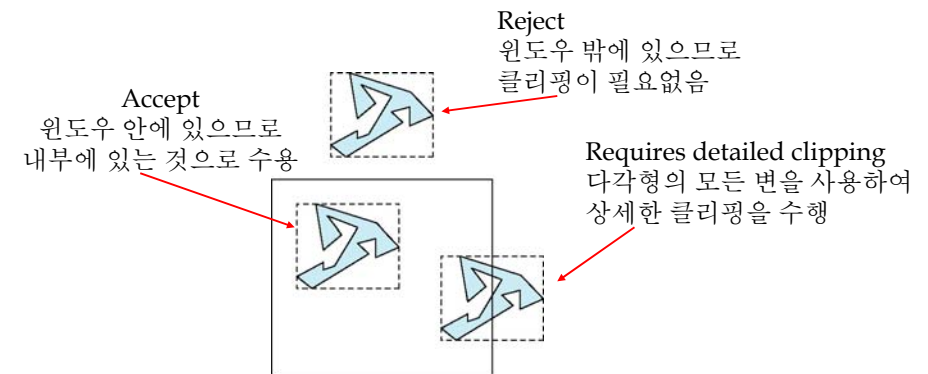
또는 범위 (extent)를 클리핑에 사용

- 많은 변을 가진 복잡한 다각형의 경우
- 경계상자는 다각형을 포함하는 윈도우에 정렬된 가장 작은 사각형
- 경계상자는 다각형 정점 x와 y값의 최소값 (min)과 최대값 (max)를 계산하여 얻어짐



## Bounding boxes

### □ 경계상자를 사용하여 간단한 클리핑 수행

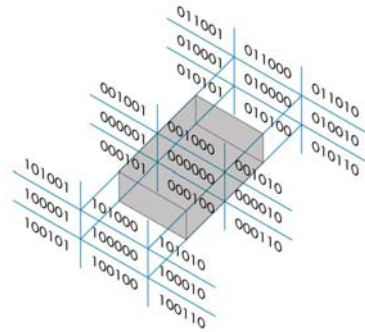


## Cohen-Sutherland Algorithm in 3D

- 3차원에서는 평면에서의 경계영역이 아닌 경계 공간 (bounding volume)에 대하여 클리핑
- Cohen-Sutherland 클리핑 알고리즘
  - 클리핑 공간에 대한 4비트 외곽부호를 6비트 외곽 부호로 대체하고 2차원의 경우와 같이 계산함

$$b_4 = \begin{cases} 1 & \text{if } z > z_{max} \\ 0 & \text{otherwise} \end{cases}$$

$$b_5 = \begin{cases} 1 & \text{if } z < z_{min} \\ 0 & \text{otherwise} \end{cases}$$



## Liang-Barsky Algorithm in 3D

- Liang-Barsky 클리핑 알고리즘

- 선분의 3차원 매개변수 표현

$$P(\alpha) = (1-\alpha)P_1 + \alpha P_2, 0 \leq \alpha \leq 1$$

$$x(\alpha) = (1-\alpha)x_1 + \alpha x_2$$

$$y(\alpha) = (1-\alpha)y_1 + \alpha y_2$$

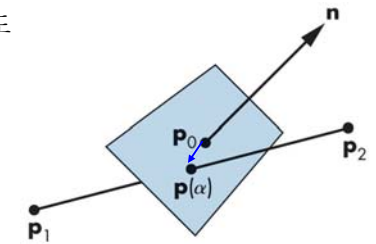
$$z(\alpha) = (1-\alpha)z_1 + \alpha z_2$$

- 평면 ( $P_0, n$ )의 공식으로부터  $\alpha$ 유도

$$P(\alpha) = (1-\alpha)P_1 + \alpha P_2$$

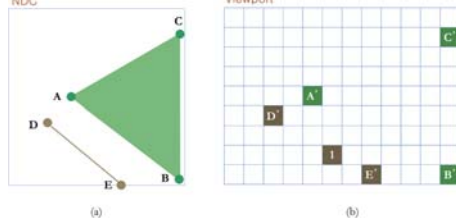
$$n \cdot (P(\alpha) - P_0) = 0$$

$$\alpha = \frac{n \cdot (P_0 - P_1)}{n \cdot (P_2 - P_1)}$$



## Rasterization

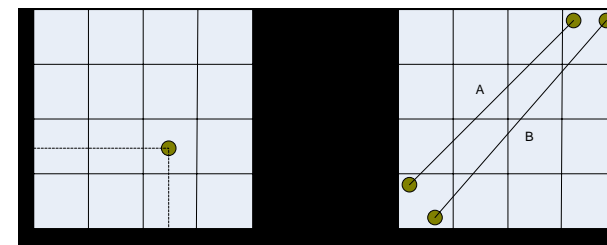
- 래스터화 (rasterization) 또는 스캔 변환 (scan conversion)
  - 프레임버퍼에서 단편의 형성에 이르는 과정의 마지막 단계
  - 물체를 표현하기 위해 어떤 화소를 밝힐 것인지를 결정하는 작업
  - 정규화 가시부피 (normalized device coordinates)에서 뷰포트 (viewport)로의 사상
  - 정점좌표를 화면좌표로 변환한 결과를 기준으로
    - 선분을 화면좌표로 변환
    - 내부면을 화면좌표로 변환
    - 아래 그림에서 A', B', C'으로 둘러싸인 곳에서 어떤 화소를 칠해야 삼각형 ABC를 가장 잘 표현할 수 있는가?



## Rasterization

- 실수(float) 좌표를 정수(int) 좌표로 변환

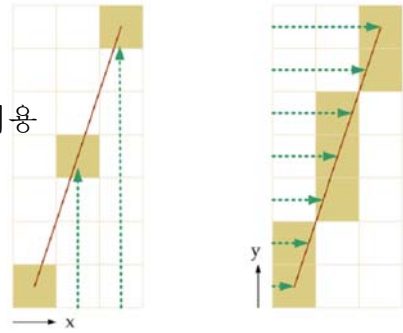
- 때로는 반올림이 필요
- 예를 들어, 정점의 뷰포트 좌표가 (1.95, 1.4) → 화소 (2, 1)로 변환됨
- 화소 경계선 내부 (1.5 ≤ x < 2.5)이고 (0.5 ≤ y < 1.5)인 모든 정점은 (2, 1)로 사상됨



A, B는 모두 같은 선분으로 사상됨

## Line Scan-Conversion

- 래스터 변환 알고리즘이 적용되는 가장 기본적인 객체가 선분 (line segment)임
- 일단 선분 양 끝 정점이 화면의 어떤 화소로 사상되는 지를 결정한 후에 나머지 화소 부분을 처리
- 기울기를 기준으로 샘플링
  - 1보다 크면 y 좌표를 증가
  - 1보다 작으면 x 좌표를 증가
- 기울기가 음수라면 절댓값 이용

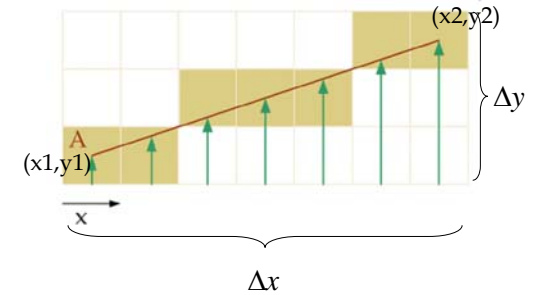


## Line Scan-Conversion

- 교차점 계산에 의한 변환은 부동소수점셈으로 인해 속도저하

```
void LineDraw(int x1, int y1, int x2, int y2)
```

```
{
    float y, m;
    int dx, dy;
    dx = x2 - x1;
    dy = y2 - y1;
    m = dy / dx;
    for (x = x1; x <= x2; x++) {
        y = m*(x - x1) + y1;
        DrawPixel(x, round(y));
    }
}
```



두 점  $(x_1, y_1)(x_2, y_2)$ 을 지나는 직선방정식

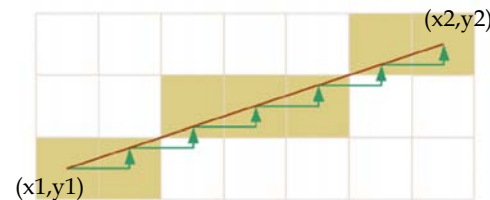
$$y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1$$

## DDA (Digital Differential Analyzer)

- 부동소수 곱셈을 부동소수 덧셈으로 변환

```
void LineDraw(int x1, int y1, int x2, int y2)
```

```
{
    float m, y;
    int dx, dy;
    dx = x2 - x1;
    dy = y2 - y1;
    m = dy / dx;
    y = y1;
    for (int x = x1; x <= x2; x++) {
        y += m;
        DrawPixel(x, round(y));
    }
}
```



$$y = mx + h \text{ where } m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$$

$$\Rightarrow \Delta y = m \Delta x$$

$$\Rightarrow \Delta y = m \text{ (x가 1씩 증가할 때)}$$

## DDA (Digital Differential Analyzer)

- DDA 알고리즘에 의한 연산

| x     | (x, y)    | 반올림 결과 |
|-------|-----------|--------|
| x = 0 | (0, 0.00) | (0, 0) |
| x = 1 | (1, 0.33) | (1, 0) |
| x = 2 | (2, 0.66) | (2, 1) |
| x = 3 | (3, 0.99) | (3, 1) |
| x = 4 | (4, 1.32) | (4, 1) |
| x = 5 | (5, 1.65) | (5, 2) |
| x = 6 | (6, 1.98) | (6, 2) |

## DDA (Digital Differential Analyzer)

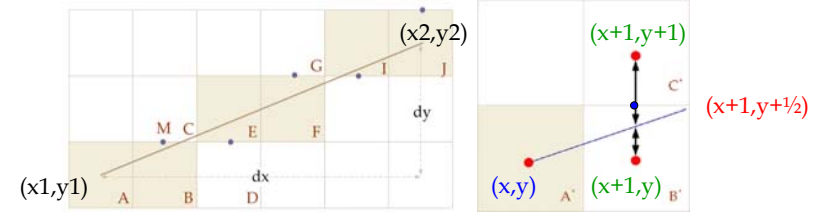
### □ DDA 단점

- 부동소수 연산
  - 부동소수 덧셈이 정수 연산에 비해 느림
- 반올림 연산
  - round() 함수 실행에 걸리는 시간
- 연산 결과의 정확도
  - 부동소수의 경우 뒷 자리가 잘려나감
  - 연속적인 덧셈에 의한 오류 누적
  - 선택된 화소가 실제 선분에서 점차 멀어져서 표류(Drift)

## Bresenham's Line Algorithm

### □ 중점 알고리즘(Midpoint Algorithm) 라고도 불림

- 모든 부동소수점 계산을 피하고 정수 계산만 이용
- 현재 래스터기의 표준 알고리즘이 된 직선 래스터화 알고리즘



### □ A (x, y) 선택

- 다음 화소는 B (x+1, y), C (x+1, y+1) 중 하나
- 화소 중심과 선분간의 수직 거리에 의해 판단
- 선분이 중점 M의 아래에 있으면 화소 B, 위에 있으면 화소 C를 선택

## Bresenham's Line Algorithm

- 화소 A=(x1, y1)이라 하면 화소 B, C의 중점 M의 좌표는 (x1 + 1, y1 + 1/2)이 되는 데, 이를 F에 대입해보면

$$y = mx + h, m = \frac{dy}{dx}$$

$$y = \frac{dy}{dx}x + h$$

$$ydx = xdy + hdx$$

$$0 = xdy - ydx + hdx$$

$$F(x, y) = 2xdy - 2ydx + 2hdx$$

$$F(x_1, y_1) = 2x_1dy - 2y_1dx + 2hdx = 0$$

$$F(x, y) = 2dy - dx$$

$$F(x, y) = F\left(x_1 + 1, y_1 + \frac{1}{2}\right)$$

$$= 2(x_1 + 1)dy - 2\left(y_1 + \frac{1}{2}\right)dx + 2hdx$$

$$= 2x_1dy - 2y_1dx + 2hdx + 2dy - dx$$

$$= F(x_1, y_1) + 2dy - dx$$

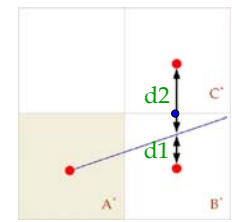
## Bresenham's Line Algorithm

- 결정변수 (decision variable) F(x, y)에 의해 중점이 선분의 위인지 아래인지를 판단
  - 만약 F(x, y) < 0이라면, 중점이 선분 위에 있고 따라서 동쪽 화소를 선택
  - 만약 F(x, y) > 0이라면, 동북쪽 화소를 선택

$$F(x, y) = 2dy - dx$$

if (F(x, y) < 0) select E // 동쪽 화소 선택

else select NE // 동북쪽 화소 선택



$$d2 > d1 \Rightarrow F(x, y) < 0$$



## Bresenham's Line Algorithm

- 현재 고려되고 있는 화소의 좌표를  $(x, y)$ 라고 하고 만약 동쪽 화소가 선택되었다면 다음 단계의 중점 위치는  $(x + 1, y)$ , 동북쪽 화소가 선택되었다면 다음 단계의 중점 위치는  $(x + 1, y + 1)$ 이 된다
- 다음 단계의 결정변수와 현 단계의 결정변수의 차이는 다음과 같이 계산

$$\begin{aligned} \text{incrE} &= F(x+1, y) - F(x, y) \\ &= (2(x+1)dy - 2ydx + 2hdx) - (2xdy - 2ydx + 2hdx) \\ &= 2dy \end{aligned}$$

$$\begin{aligned} \text{incrNE} &= F(x+1, y+1) - F(x, y) \\ &= (2(x+1)dy - 2(y+1)dx + 2hdx) - (2xdy - 2ydx + 2hdx) \\ &= 2dy - 2dx \end{aligned}$$

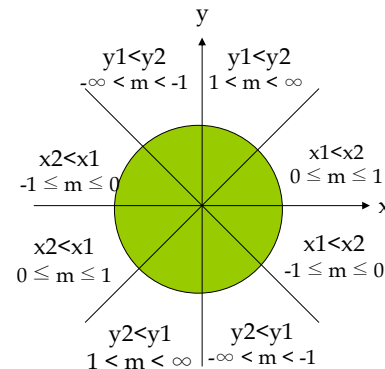
## Bresenham's Line Algorithm

```
void MidpointLineDraw(int x1, int y1, int x2, int y2)
{
    int dx, dy, incrE, incrNE, D, x, y=y1;
    dx = x2 - x1; dy = y2 - y1;
    D = 2*dy - dx; // 결정변수 값을 초기화
    incrE = 2*dy; // 동쪽 화소 선택시 증가분
    incrNE = 2*dy - 2*dx; // 동북쪽 화소 선택시 증가분
    for (x=x1; x <= x2; x++) {
        if (D <= 0) { // 결정변수가 음수. 동쪽화소 선택
            D += incrE; // 결정변수 증가
        }
        else { // 결정변수가 양수. 동북쪽 화소 선택
            D += incrNE; // 결정변수 증가
            y++; // 다음 화소는 동북쪽
        }
        DrawPixel (x, y); // 화소 그리기
    }
}
```

$0 \leq m \leq 1$

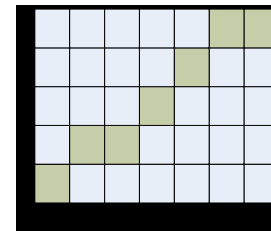
## Bresenham's Line Algorithm

- $|m| > 1.0$ 
  - $x$ 와  $y$ 를 바꿔서 계산함
  - $y$ 방향으로 증가시키면서,  $x$ 값을 결정함
- 그 외에, 특수한 경우는 따로 처리함
  - $\Delta y = 0$  (horizontal line)
  - $\Delta x = 0$  (vertical line)
  - $|\Delta x| = |\Delta y|$  (diagonal lines)



## Bresenham's Line Algorithm

- 예를 들어  $(0, 0)$ 과  $(6, 4)$ 를 연결하는 선분

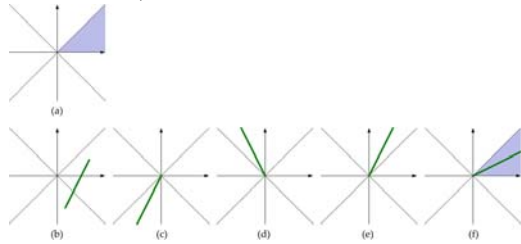


|          |         |
|----------|---------|
| $(0, 0)$ | $D > 0$ |
| $(1, 1)$ | $D < 0$ |
| $(2, 1)$ | ...     |
| ...      | ...     |
| $(6, 4)$ |         |

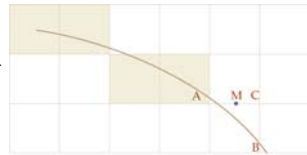


## Bresenham's Line Algorithm

- 정수연산에 의한 속도증가 + 하드웨어로 구현
- 첫 8분면에서만 정의
  - 다른 선분은 이동, 반사하여 적용

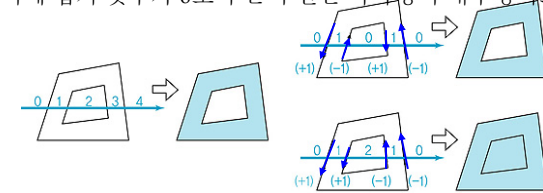


- 원 생성 알고리즘
  - 선분생성 알고리즘과 유사



## Polygon Scan-Conversion

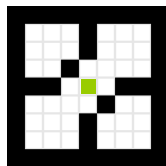
- 다각형의 래스터화 = 다각형 채우기 (polygon filling)
  - 점이 다각형의 내부에 있다면 그것을 내부색으로 칠함
- 다각형 내부의 판단규칙
  - 홀짝 규칙 (even-odd rule)
    - 주사선 별로 경계가 홀수(odd) 번째 교차하면 내부, 짝수(even) 번째 교차하면 외부가 시작된다고 판단
  - 접기횟수 규칙 (non-zero winding rule)
    - 주사선 별로 아래쪽 경계와 교차하면 접기 횟수를 1 증가, 위쪽방향의 경계와 교차하면 1감소
    - 이때 접기 횟수가 0보다 큰 구간은 다각형의 내부영역으로 판단



## Flood Fill

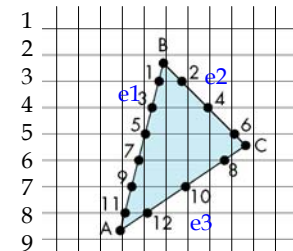
- 범람 채우기 (flood fill)
  - 내부로 정의된 영역 채우기
  - 다각형 내부의 시작점 (seed point)에서 시작하여, 순환적으로 이웃을 살펴보고, 만약 이들이 변의 점이 아니라면 채우기색으로 칠함

```
void flood_fill(int x, int y) { // 다각형 내부 초기점 (x, y)에서 시작
    if(read_pixel(x,y) != WHITE) { // 현재 픽셀이 배경색(white)이면
        write_pixel(x,y,BLACK); // 채우기색 (black)으로 칠함
        flood_fill(x+1, y); // 오른쪽으로 반복
        flood_fill(x-1, y); // 왼쪽으로 반복
        flood_fill(x, y+1); // 아래로 반복
        flood_fill(x, y-1); // 위로 반복
    }
}
```



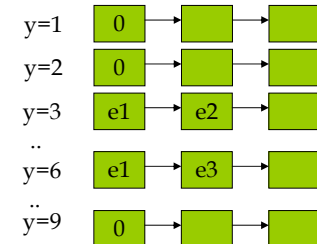
## Scan Line Fill

- 주사선 채우기 (scan line fill)
  - Y-X 다각형 주사선 알고리즘:
    - 전체 edge를 Y값 순서로 정렬하여 Edge List (EL)를 구성
    - 매 주사선이 새로 교차하는 에지를 EL에서 꺼내어 Active Edge List (AEL)로 이동
    - 해당 주사선과 각 edge와 교차점을 2개씩 짝을 지어 사이를 채움



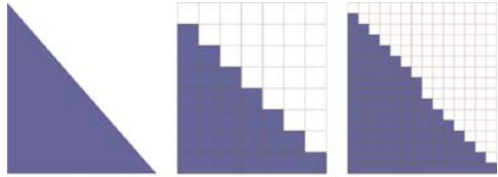
Edge List e1 → e2 → e3

Active Edge List



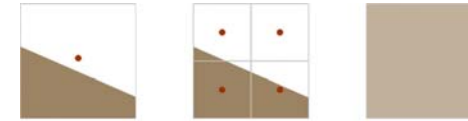
## Aliasing

- 계단(Stair-step, Jaggies) 모양의 거친 경계선
  - 비트맵 표현에서는 화소 단위로 근사화 할 수 밖에 없기 때문
  - 무한 해상도를 지닌 물체를 유한 해상도를 지닌 화소 면적 단위로 근사화 할 때 필연적으로 일어나는 현상

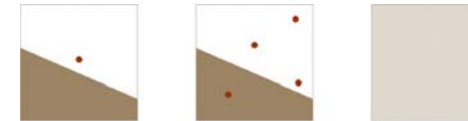


## Anti-Aliasing

- 슈퍼 샘플링(Super-Sampling)
  - 부분 화소에서 샘플링. 사후 필터링
  - 부분 화소의 평균값을 반영

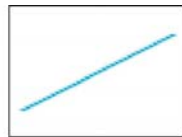
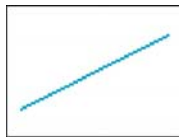


- 지터(jitter)에 의한 슈퍼 샘플링
  - 물체 자체가 불규칙이라면 불규칙 샘플링이 유리

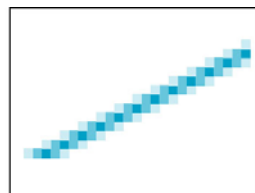
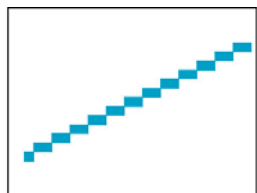


## Anti-Aliasing

Aliasing



Anti-aliased



Magnified