Computer Graphics

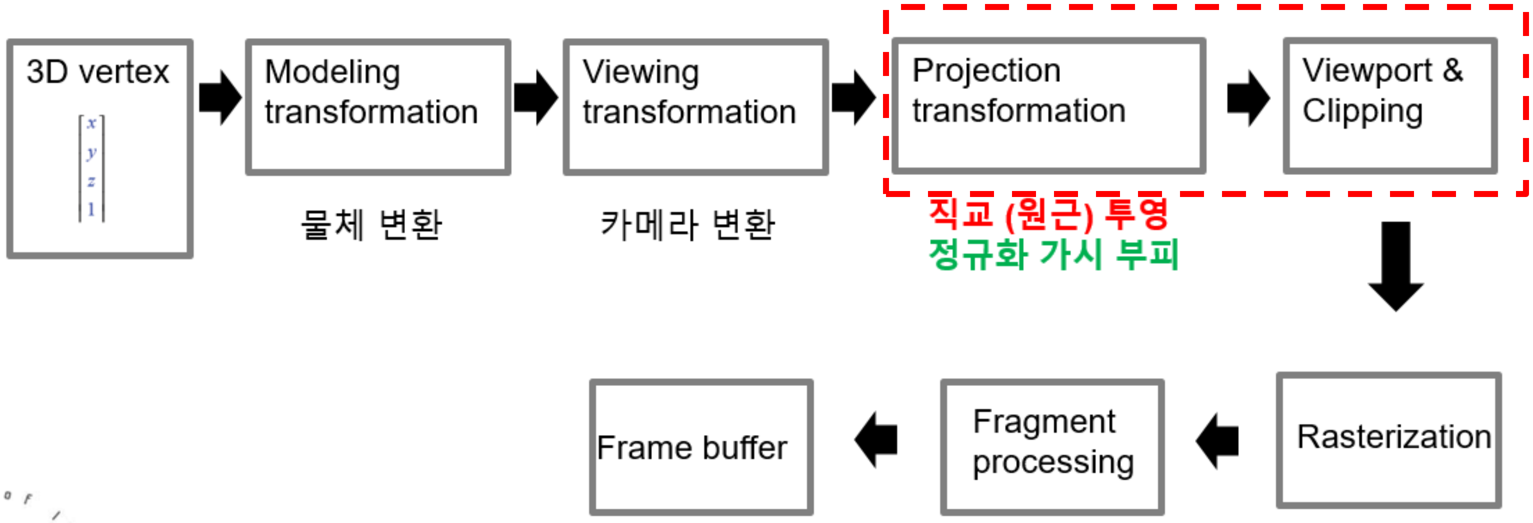
Prof. Jibum Kim

Department of Computer Science & Engineering

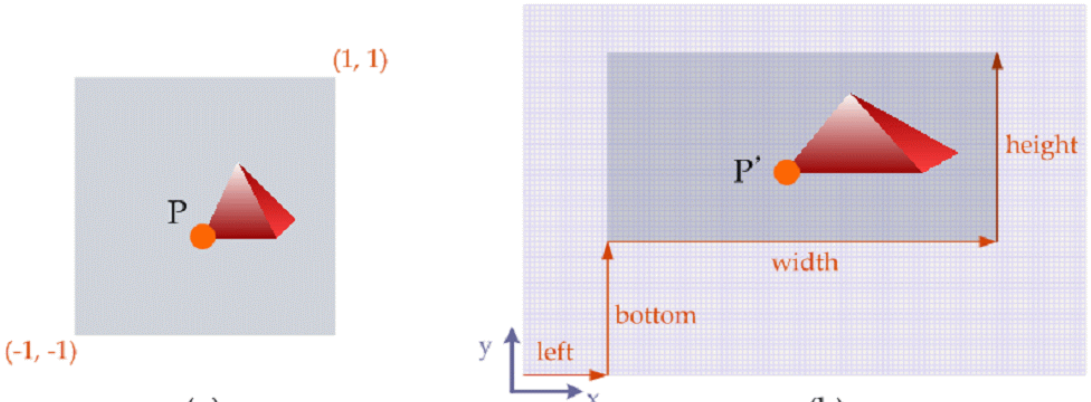
Incheon National University

- **정규화 가시 부피에서 Viewport 로의 mapping**

- 정규화 가시 부피에서 **Viewport** 로의 **mapping**
- 이번에는 앞에서의 정규화 가시 부피로의 **mapping**이 끝나고 **clipping**까지 끝난 후에 최종적으로 정규화 가시 부피에서 **viewport**로의 **mapping**에 대해서 살펴보자

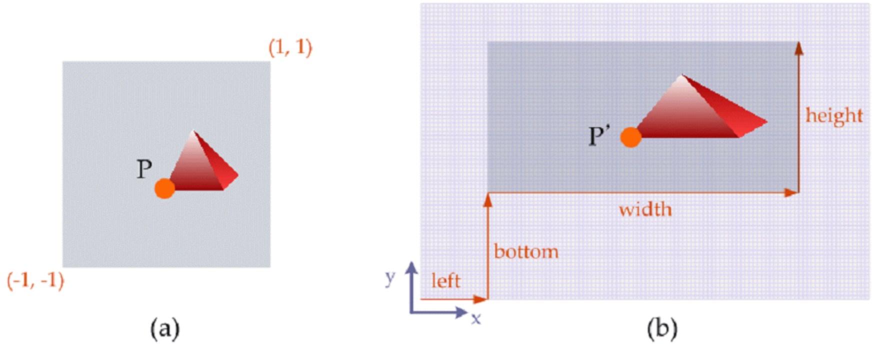


- 정규화 가시 부피 변환이 끝나면 **x, y, z** 모두 **[-1, 1]**사이의 값을 가졌다 이렇게 변환된 좌표가 절단 좌표계 **(혹은 normalized device coordinates, NDC)** 이다
- 이렇게 **NDC**로 변환된 좌표들은 최종적으로 **Viewport**로 **mapping (**사상**)**된다
- **(a) NDC**에서의 점 **P (x: [-1, 1], y:[-1, 1]**사이의 값 가짐**)**
- **(b) viewport (glViewport(left ,bottom, width, height))** 로 **mapping**된 점 **P'**
- 목적**: NDC**에서의 점 **P-> Viewport**에서의 점 **P'**로 **mapping** 시키는 함수 찾음



(a) (b)

- 목적: NDC에서의 점 P-> Viewport에서의 점 P’로 mapping 시키는 함수 찾음
- 단, 이 함수를 x축과 y축을 분리하여 생각하고자 한다



(a) NDC의 좌표 범위: x:[-1, 1], y:[-1, 1]

(b) viewport에서의 좌표 범위:x’:[left ~ left+width], y’:[bottom ~ bottom+height]

(glViewport(left ,bottom, width, height) 사용시)

1. x축 에서의 NDC에서의 viewport로의 mapping 함수 찾기

- (a) NDC : x:[-1, 1], y:[-1, 1]
- (b) viewport: x':[left, left+width], y':[bottom, bottom+height]
-
- $\boldsymbol{f}: \boldsymbol{x} \rightarrow \boldsymbol{x}',\ \boldsymbol{x}' = \frac{\boldsymbol{width}}{\boldsymbol{2}}(\boldsymbol{x} + \boldsymbol{1}) + \boldsymbol{left}$

2. y축 에서의 NDC에서의 viewport로의 mapping 함수 찾기

- (a) NDC : x:[-1 ,1], y:[-1 ,1]
- (b) viewport: x':[left, left+width], y':[bottom, bottom+height]
-
- $\boldsymbol{f}: \boldsymbol{y} \rightarrow \boldsymbol{y}', \ \boldsymbol{y}' = \frac{\boldsymbol{height}}{\mathbf{2}}(\boldsymbol{y}+\mathbf{1}) + \boldsymbol{bottom}$

인천대학교

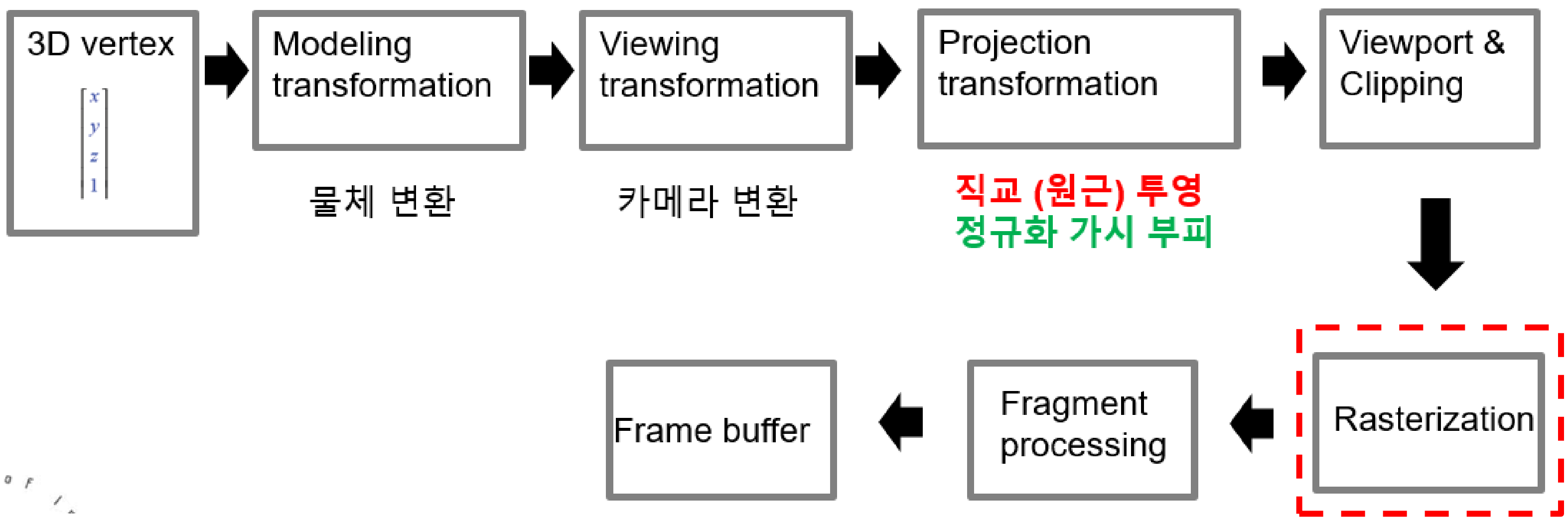
- 앞의 두 변환식을 적어보고 행렬로 바꿔보자
- $\boldsymbol{f}: (\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) \rightarrow (\boldsymbol{x}', \boldsymbol{y}', \boldsymbol{z}')$, **NDC->Viewport좌표**
- $\boldsymbol{f}: \boldsymbol{x} \rightarrow \boldsymbol{x}',\ \boldsymbol{x}' = \frac{\boldsymbol{width}}{\boldsymbol{2}}(\boldsymbol{x} + \boldsymbol{1}) + \boldsymbol{left}$
- $\boldsymbol{f}: \boldsymbol{y} \rightarrow \boldsymbol{y}',\ \boldsymbol{y}' = \frac{\boldsymbol{height}}{\boldsymbol{2}}(\boldsymbol{y} + \boldsymbol{1}) + \boldsymbol{bottom}$
- $\boldsymbol{f}: \boldsymbol{z} \rightarrow \boldsymbol{z}', \boldsymbol{z}' = \boldsymbol{z}$, 변화 없음

$$\begin{bmatrix} \boldsymbol{x}' \\ \boldsymbol{y}' \\ \boldsymbol{z}' \\ \boldsymbol{1} \end{bmatrix} = \begin{bmatrix} \frac{width}{2} & \boldsymbol{0} & \boldsymbol{0} & \frac{width}{2} + left \\ \boldsymbol{0} & \frac{height}{2} & \boldsymbol{0} & \frac{height}{2} + bottom \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{1} & 0 \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{1} \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{z} \\ \boldsymbol{1} \end{bmatrix}$$

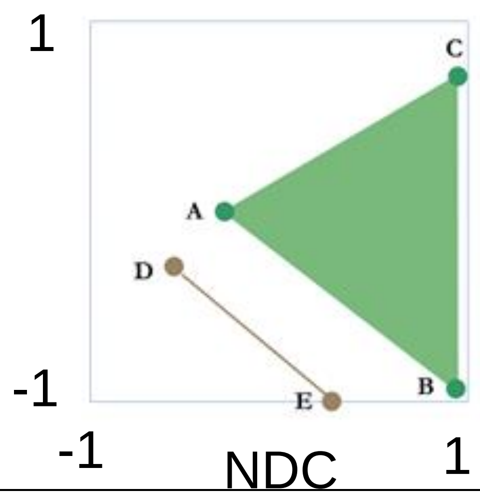
- 최종적으로 **viewport**에 표시될 때에는 **viewport**에 **mapping**된 좌표가 반드시 정수여야 한다. 픽셀을 나타낼 때에 정수로만 표시
- 예를 들어 **viewport**의 **mapping**된 좌표가 **(x', y')=(23.2, 15.8)**이라면 실제 위치는 **(x'=23, 16)**으로 표시되어야 한다 **(**예**: round()**연산**)**
- **Z**값에서는 **NDC->viewport**로의 **mapping**이 일어나지 않는다**.** 앞에서 설명했지만 이 **z**값은 물체의 깊이 **(depth)**측정 시 사용되는 **depth buffer**에 사용된다 **(**예**: hidden surface removal)**
- **OpenGL** 내부에서는 **z**값의 **re-normalization (**재 정규화**)** 과정을 한번 더 거친다 **NCD**에서 **z**값의 **[-1, +1]**를 **[0, 1]**로 **projection** 시킨다

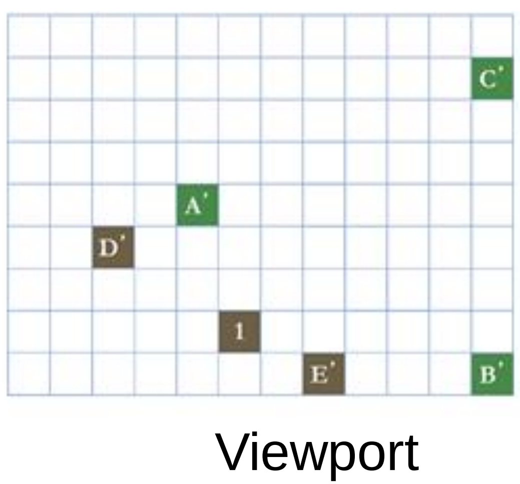
- **Rasterization (래스터 변환)**

Graphics pipeline



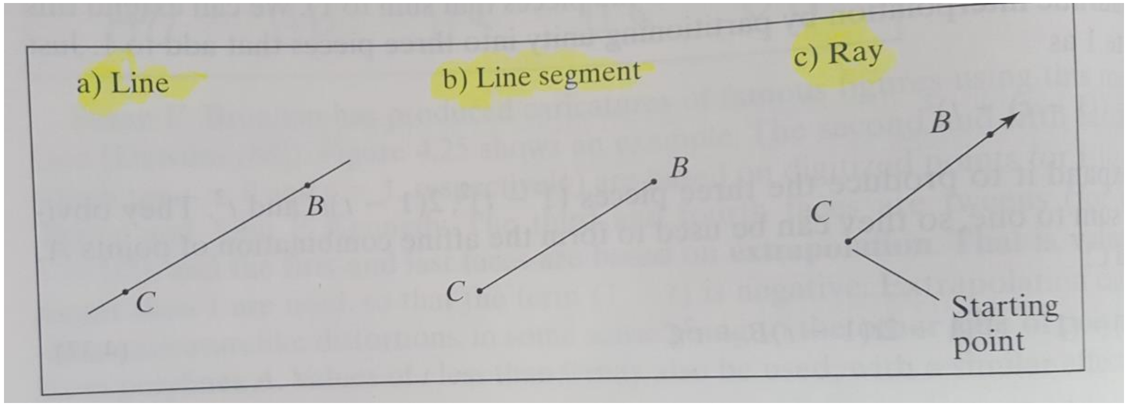
- 가시 공간 내에서 삼각형 **ABC**, 선분 **DE**가 먼저 정규화 가시 부피로의 변환 및 투영이 끝나고 아래 왼쪽 처럼 **5**개의 **vertex A, B, C, D, E**로 **mapping** 되었다
- 이를 앞에서 배운 정규화 가시 부피**=> viewport**로의 변환이 끝나고 각 **vertex**가 **A', B', C', D' ,E'**로 **mapping** 되었다고 하자**.**
- **Rasterization (Raster** 변환**)**이란 이렇게 **viewport**로 **mapping**된 후에 최종적으로 **viewport**내에서 어떠한 **pixel**들을 선택할지 정해서 가시 공간에서 그린 삼각형 **ABC,** 선분 **DE** 처럼 보이게 하는지 결정하는 작업을 말한다



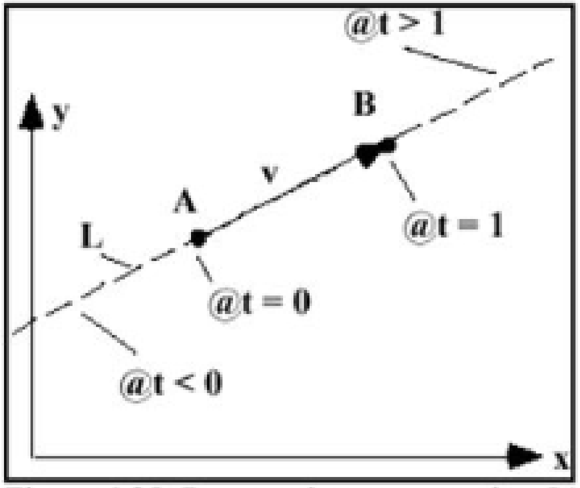


- **Line, Line, Ray**

- **Line (직선):** a line is defined by two points, say C and B. It is infinite in length, passing through the points and extending forever in both directions
- **Line segment (segment, 선분):** defined by two points (called endpoints) but extends only from one endpoint to the other
- **Ray (반직선):** it is specified by a point and a direction. A ray starts at a point and extends infinitely far in a given direction. A ray is semi-infinite. Ray tracying에 사용
- Ray tracing is a rendering technique that can produce incredibly realistic lighting effects



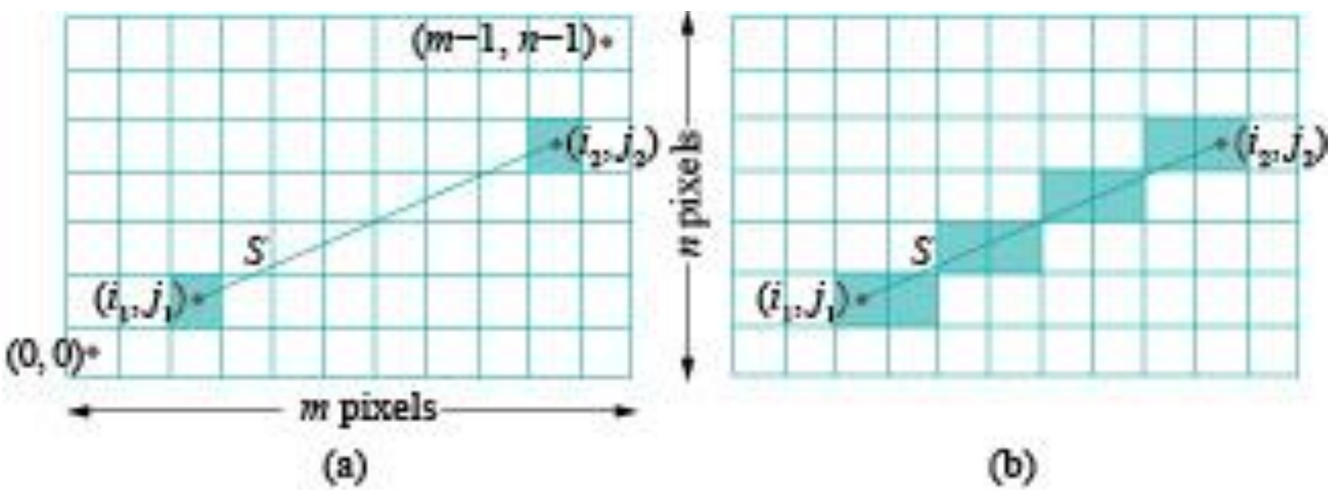
- Parametric representation of a line (using a parameter t)
- Call the line L and give the name L(t) to the position associated with t. Using *v* (vector)=B-A, we have
- L(t)=A+*v*t
- L(0)=A, L(1)=B
- Segment: $0 \le t \le 1$
- Ray: $0 \le t < \infty$
- Line: $-\infty < t < \infty$



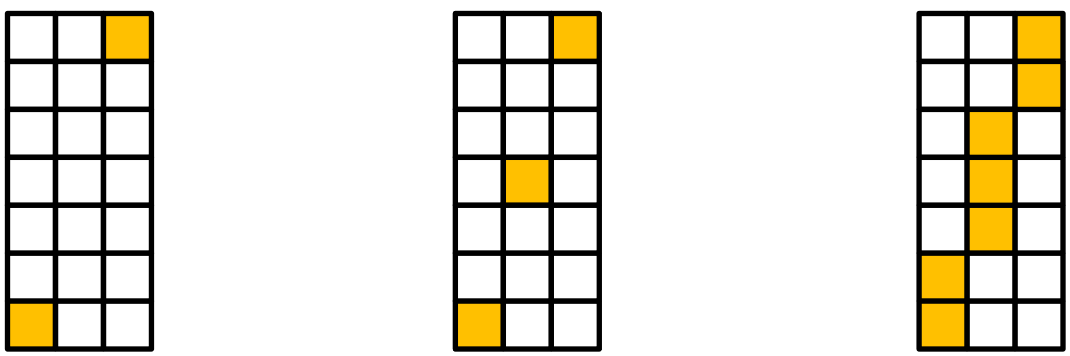
- **Find a parametric form for the line that passes through C=(3, 5) and B=(2, 7)**

- 선분의 Rasterization

- Line segment (선분) 의 raster 변환 문제
- 가로 **m**개의 픽셀, 세로 **n**개의 픽셀이 주어져 있고 시작점 **(i_1,j_1)**과 끝점 **(i_2,j_2)**가 주어져 있을 때 이 둘을 연결하는 **line segment** (밑의 오른쪽 그림 **S**)를 잘 표현하는 픽셀들을 선택하는 문제



- 선분의 시작 **pixel**가 끝 **pixel**이 주어져 있을 때 **선분의 rasterization** 문제
- 이 경우에는 선분의 기울기가 **1**보다 크다 **(기울기=3). Pixel**은 정수 단위이므로 아래의 **(a) (b)**중에 한가지 방법을 선택해서 선분의 **rasterization**을 수행해 보고자 한다
- **(a) x**좌표를 **1**씩 증가시키면서 선분과 교차하는 화소를 선택**, y=3x**
- **1) x=0, y=0 2) x=1, y=3 3) x=2, y=6**
- **(b) y**좌표를 **1**씩 증가시키면서 선분과 교차하는 화소를 선택**, y=3x**
- **1) x=0, y=0 2) y=1, x=1/3 (rounding, x=0) 3) y=2, x=2/3 (x=1) 4) y=3, x=1 …..**
- **Q) (a)**와 **(b)**중에 어떠한 래스터 변환이 실제 선분에 가까운가**? Why?**



- **Observation**

1. Line의 slope (기울기)가 1보다 크면 (앞 페이지와 같이) y좌표를 하나씩 증가시키는 것이 좋다

2. Line의 slope가 1보다 작으면 x좌표를 하나씩 증가시키는 것이 좋다

즉, line을 rasterization을 할 때에는 line의 slope를 고려해야 한다

- **Digital Differential Analyzer Algorithm (DDA algorithm)**

Digital Differential Analyzer Algorithm (DDA algorithm): Line segment를 래스터 변환하는 간단한 알고리즘

- 가정: 선분의 기울기가 **1**보다 작다**.** 즉**, x**를 하나씩 증가
- **Line**의 **slope (m)=** $\frac{y\text{의 증가량}(dy)}{x\text{의 증가량}(dx)} = y\text{의 변화량}$
- **y**의 변화량 **(**증가량**)= m (**기울기**)** 이다

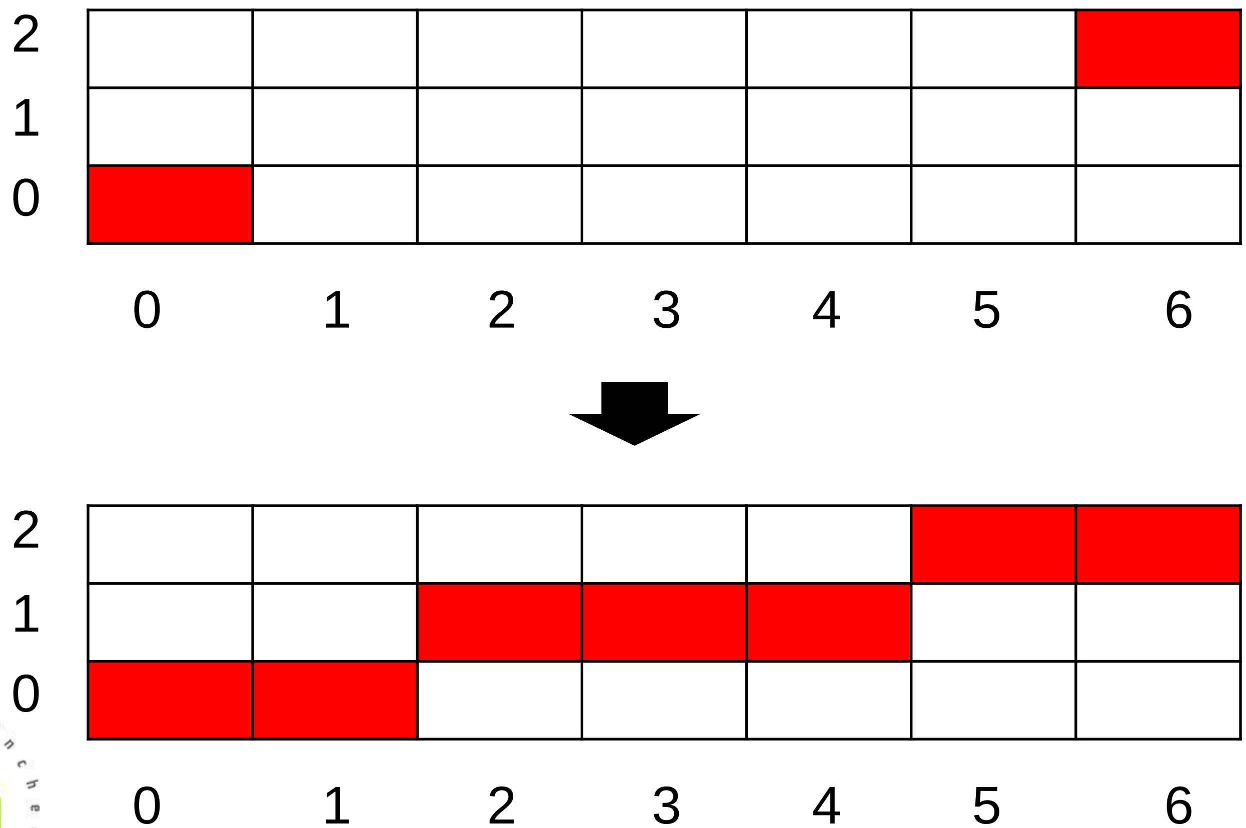
- // DDA algorithm
- // 선분: (x1,y1) - (x2, y2) 단, // x2> x1, // y2> y1 , m (기울기)<1

```
void LineDraw(int x1, int y1, int x2, int y2){
    float m, y;   int dx, dy;
    dx = x2 - x1;  //x 변화량
    dy = y2 - y1;  //y 변화량
    m = dy / dx;    //  slope (기울기)
    y = y1;
    for (int x = x1; x <= x2; x++) {
        DrawPixel(x, round(y));        // rounding (반올림)
        y += m;                 // y값에 기울기 만큼 더해줌
    }
}
```

- // Line rasterization using DDA algorithm
- // (x1,y1)=(0,0),   (x2,y2)=(6,2), m(slope)=2/6≈0.33

x	y	DDA rasterization
x=0	y=0	(0, 0)
x=1	y=0.33	(1, 0)
x=2	y=0.66	(2, 1)
x=3	y=1	(3, 1)
x=4	y=1.33	(4, 1)
x=5	y=1.66	(5, 2)
x=6	y=2	(6, 2)

- // DDA algorithm을 이용한 선분 래스터 변환
- // (x1,y1)=(0,0),   (x2,y2)=(6,2), m(slope)=2/6≈0.33, 총 7개의 픽셀 선택



- **https://www.dropbox.com/s/w5vd1ieul2thvof/DDA.txt?dl=0**

- 지금까지는 기울기가 **1**보다 작은 경우의 **line rasterization**을 행하는 **DDA** 알고리즘을 살펴보았다
- 만일**,** 기울기가 **1**보다 크면 앞의 코드를 어떻게 바뀌어야 할지 생각해 보자
- **https://en.wikipedia.org/wiki/Digital_differential_analyzer_(graphics_algorithm)**

- **DDA algorithm의 문제점**
- **1.Rounding (반올림) 연산**
  - round( ) 함수 연산을 매번 수행해야 함
  - round() 함수 실행에 걸리는 시간

- **2.** 앞에서 기울기 **1/3**을 유효숫자로 인하여 **0.33**으로
- 근사화 하였다**.** 연속적인 덧셈시 오류가 누적될 수 있다

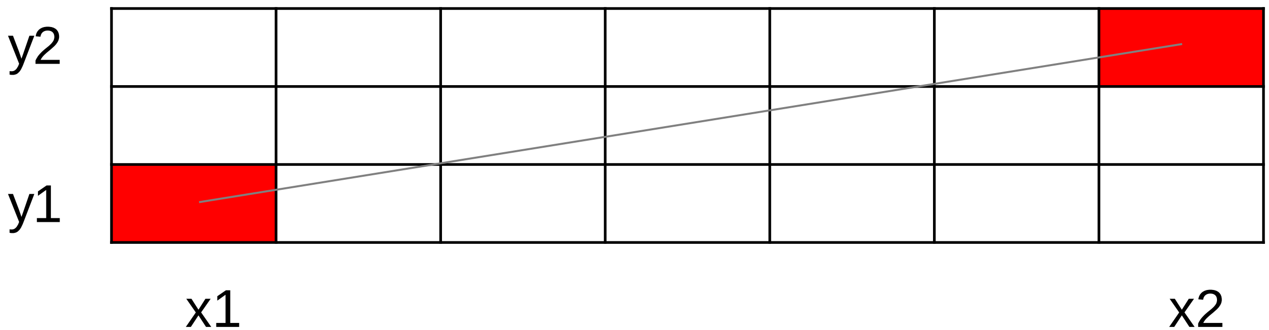
- **3.** 부동 소수점 연산은 정수 연산에 비해서 느리다

- **Bresenham Algorithm**
- **(브레스넘 알고리즘)**

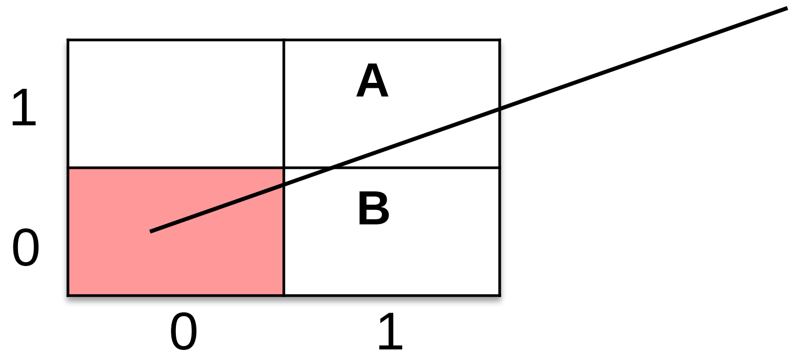
인천대학교

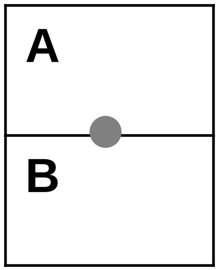
- 브레스넘 알고리즘은 **DDA** 알고리즘 처럼 선분을 래스터화 하는 알고리즘이다
- 브레스넘 알고리즘의 동기
- 곱셈이 나눗셈보다는 빠르다
- **1. DDA** 알고리즘의 나눗셈 연산
- **=>** 브레스넘 알고리즘에서는 곱셈연산으로 바꿈
- 정수 연산이 **float** 연산보다는 빠르다
- **2. DDA** 알고리즘 **float** 연산 **(**예**:** 기울기**)** 및 **round** 연산
- **=>** 브레스넘 알고리즘에서는 정수 연산으로 바꿈

- 가정: 선분의 양 끝점 **(x1, y1), (x2, y2)**가 주어져 있을 때 이 선분을 래스터 변환하려고 한다
- 편의상**, x1<x2** 이고 선분의 기울기는 **0**에서 **1**사이라고 가정하자
- 예:

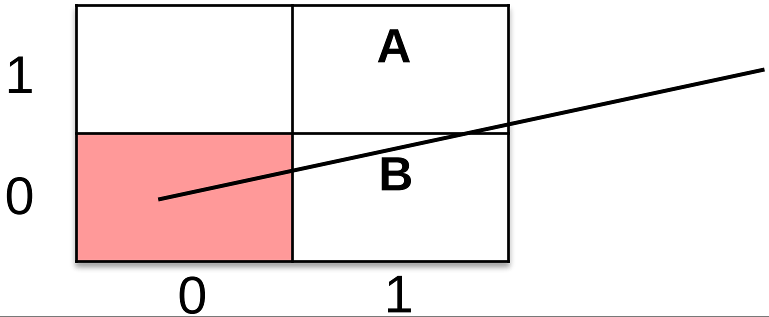


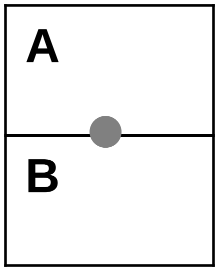
- 기울기가 **[0, 1]** 사이인 선분에 대하여 **(0,0) pixel**이 칠해져 있으면 다음 **pixel**을 칠할 후보는 **pixel A (1, 1)** 혹은 **B (1,0)** 이다.
- **A, B**중에 어떤 **pixel**을 칠하는 것이 실제 **line**에 더 가깝게 보일까?
- 브레스넘 알고리즘: 다음 후보 **A, B** 중간 점 **(1, ½)**을 이용하자
- **Case 1:**



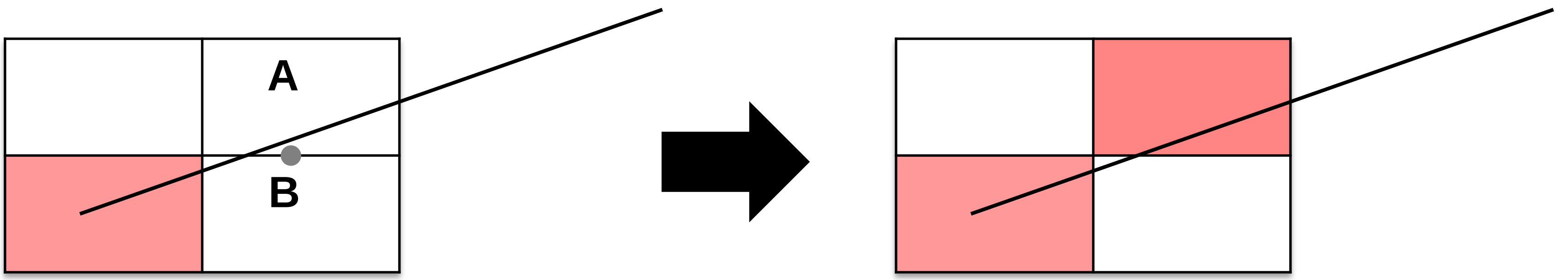


- **Case 2:**

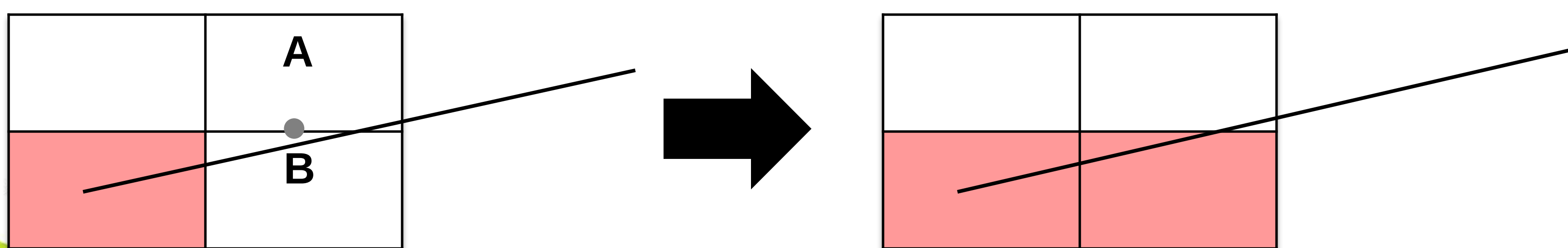




- 브레스넘 알고리즘(**Bresenham Algorithm**)
- **Case 1:** 다음 **pixel** 후보들의 중간점 이 선분 아래에 있으면 **Pixel A** 칠함
-



- **Case 2:** 다음 **pixel** 후보들의 중간점 이 선분 위쪽에 있으면 **Pixel B** 칠함

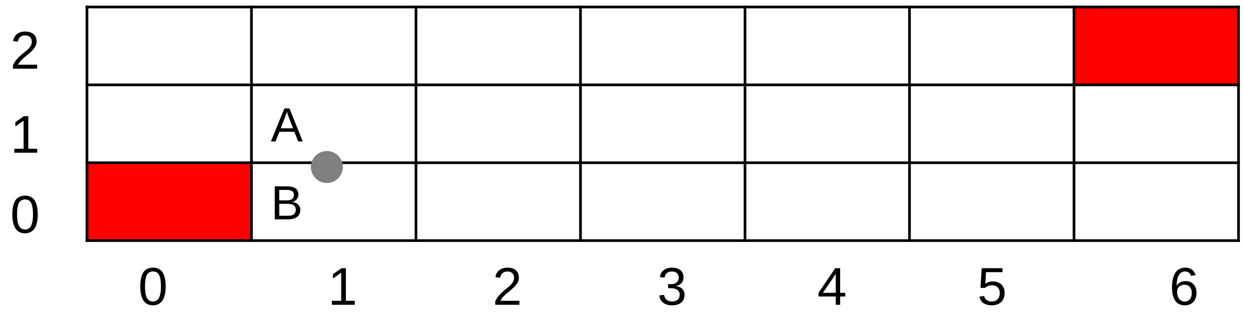


- 일반적인 직선 (선분)의 Implicit representation
- dx: x의 변화량, dy: y의 변화량, 기울기=$\frac{dy}{dx}$
- $y = \frac{dy}{dx}x + b$
- $(dx)y = (dy)x + (dx)b$
- $f(x, y) = Ax + By + C$, 단, $A = dy, B = -dx, C = (dx)b$
- 이런 implicit 표현의 장점은 뭘까?
- 1. 어떤 점이 이 선분 위쪽 에 있다면 $f(x, y) < 0$
- 2. 어떤 점이 이 선분 아래쪽에 있다면 $f(x, y) > 0$
- 또 다른 장점은? 정수 연산 (DDA 알고리즘과의 차이)

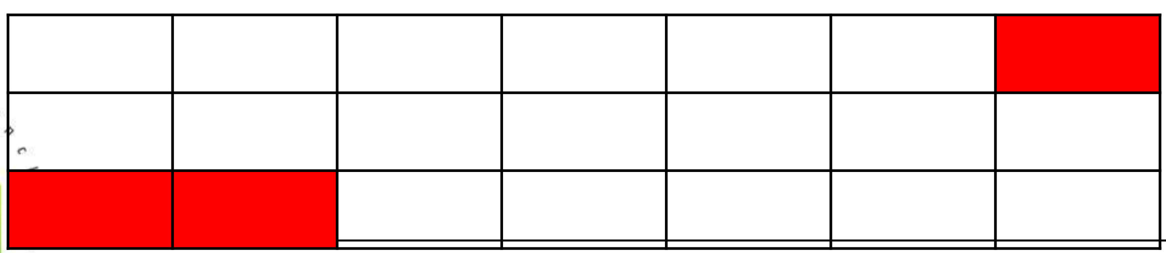
- 예: 선분의 양 끝점 **(1, 1), (6, 2)**일때
- **1.** 이 선분의 **implicit form**인 $\boldsymbol{f(x, y)}$를 구해보자
- $\boldsymbol{f(x, y) = x - 5y + 4}$

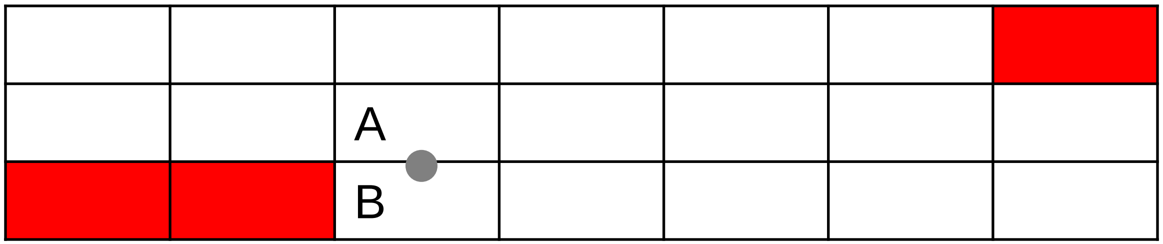
- **2.** $\mathbf{(3, 10)}$이 선분의 위쪽에 있는지 아래쪽에 있는지 $\boldsymbol{f(x, y)}$의 부호로 판단해 보자

- // 양 끝점이 주어진 선분에 대하여 브레스넘 알고리즘으로 선분 래스터 변환
- // $(x_1,y_1)=(0,0)$, $(x_2,y_2)=(6,2)$, **Implicit equation f(x, y)=2x-6y,** 선분 기울기**?**



- 두 후보 **PIXEL (A, B)**의 중점 **M=(1,1/2)**
- **f(x,y)=2x-6y**
- 중점 **(1, 1/2)** 대입, **f(x,y)<0,** 중점이 선분 위쪽에 있음 **=>** 동쪽 픽셀 **(B)** 칠함

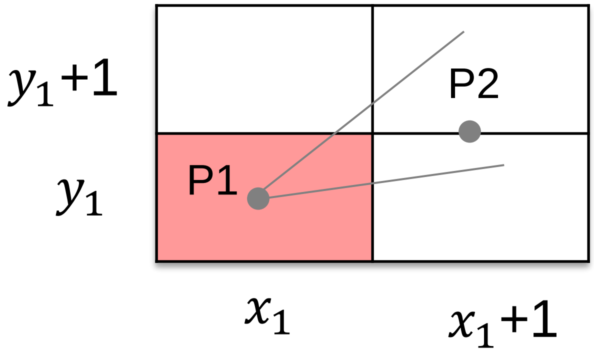


- 다음 두 후보 **PIXEL (A, B)**의 중점 **M=(2, 0.5)** 선택
- 

- **f(x, y)=2x-6y**
- 중점 **(2, 0.5)** 대입, **f(x,y)>0**, 중점이 선분 아래에 있음 **=> A (**동북쪽 픽셀**)** 칠함
- 계속 반복한다 **(x=6**인 **pixel**까지**)**

- 브레스넘 알고리즘의 코드 작성

- 직선 (선분)의 Implicit representation
- dx: x의 변화량, dy: y의 변화량 , 기울기=$\frac{dy}{dx}$, 기울기 는 0에서 1사이
- $f(x,y) = Ax + By + C = 0$, 단, $A = dy, B = -dx, C = (dx)b$ ....(1)
- P1: 최초 점 위치 (x_1 ,y_1), P2:다음 후보 픽셀들의 중간 위치 ($x_1 + 1$, y_1+1/2)



1. P1 지남, (1)에 대입

$$f(x_1, y_1) = dy \cdot x_1 - dx \cdot y_1 + (dx)b = 0 \ldots(2)$$

2. P2를 (1)에 대입 후 부호로, P2가 선분 위,아래인지 판별

$$f\left(x_1 + 1, y_1 + \frac{1}{2}\right) = dy(x_1 + 1) - dx\left(y_1 + \frac{1}{2}\right) + (dx)b \ldots(3)$$

(2)와 (3)을 합치면, $dy - dx \cdot \frac{1}{2}$ 의 부호로 P2가 선분 위, 아래에 있다 판별

인천대학교

- $dy - dx \cdot \frac{1}{2}$ 의 부호로 판단가능
- 어차피 부호로 판단하므로, 위의 식에 **2**배 **(float** 계산을 피하려고**)**
- 이를 결정 변수 **(decision variable, D)**라고 한다
- $D = 2 \cdot dy - dx$

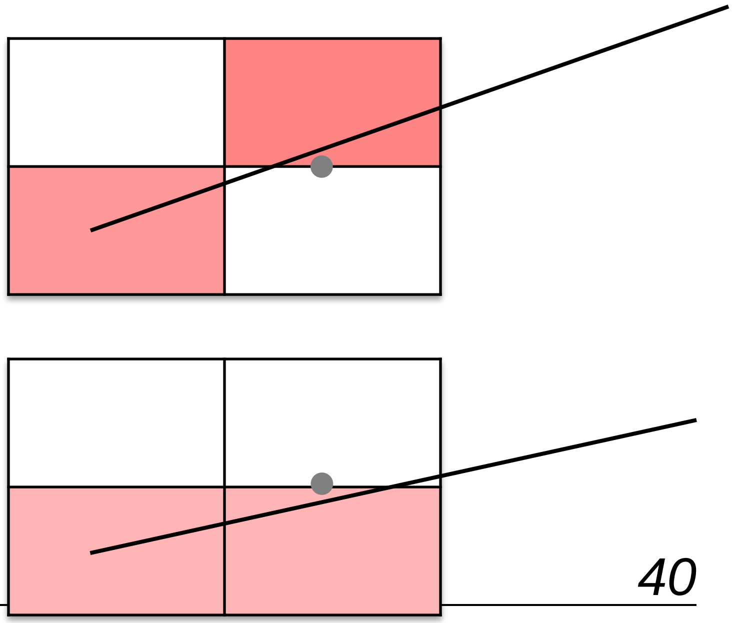
- **D**값 계산

if D > 0 **//** 중점이 선분 아래 있다

동북쪽 픽셀 선택

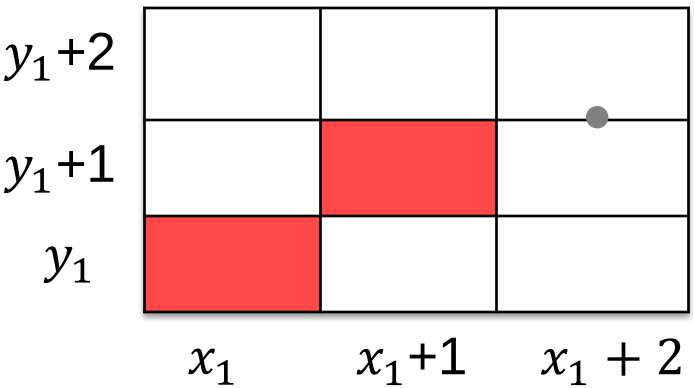
else **//** 중점이 선분 위에 있다

동쪽 픽셀 선택



- 다음 후보 픽셀들의 위치는 계속 변한다. 따라서, 결정 변수 **D** 값도 **update** 해주어야 함

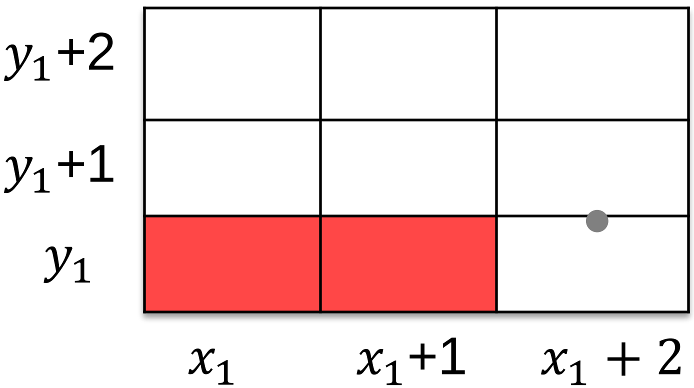
Case 1: 최초 위치 (x_1 ,y_1)의 동북쪽 픽셀 선택 시, 다음 후보픽셀들의 중점 (x_1+2 ,$y_1+3/2$)



- **1. (x_1 ,y_1) 지남, $f(x,y) = dy \cdot x_1 - dx \cdot y_1 + (dx)b = 0$**
- **2. $f(x_1+2, y_1+3/2) = dy(x_1+2) - dx\left(y_1+\frac{3}{2}\right) + (dx)b$ 의 부호로 판별**
- **1을 2에 대입하면, $2dy - \frac{3}{2}dx$ 부호로 판별, 최초: $dy - \frac{1}{2}dx$ 의 부호로 판별**
- **차이 $= dy - dx$, D값 update: $incNE = (2 * dy - 2 * dx)$ <= 차이에 2배 해줌**

- 다음 후보 픽셀들의 위치는 계속 변하므로 결정 변수 값도 **update** 해주어야 함

Case 2: 최초 위치 (x_1, y_1)의 동쪽 픽셀 선택 시, 다음 후보픽셀들의 중점 $(x_1+2, y_1+1/2)$



- **1.** (x_1, y_1) 지남, $\boldsymbol{f(x,y) = dy \cdot x_1 - dx \cdot y_1 + (dx)b = 0}$
- **2.** $\boldsymbol{f(x,y) = dy(x_1+2) - dx\left(y_1 + \frac{1}{2}\right) + (dx)b}$ 의 부호로 판별
- **1**을 **2**에 대입하면, $\boldsymbol{f(x,y) = 2dy - \frac{1}{2}dx}$ 부호로 판별, 최초: $\boldsymbol{dy - \frac{1}{2}dx}$ 부호로 판별
- 차이 $=\boldsymbol{dy}$, $\boldsymbol{D}$값 업데이트: $\boldsymbol{incE = (2 * dy)}$ **<= 차이에 2배 해줌**

- 브레스넘 알고리즘

1. 처음 화소 위치에서 $f(x, y) = D = 2 \cdot dy - dx$ 계산한다

2. if D > 0 // 중점이 선분 아래 있다

동북쪽 픽셀 선택

D=D+incNE // D값 업데이트

else // 중점이 선분 위에 있다

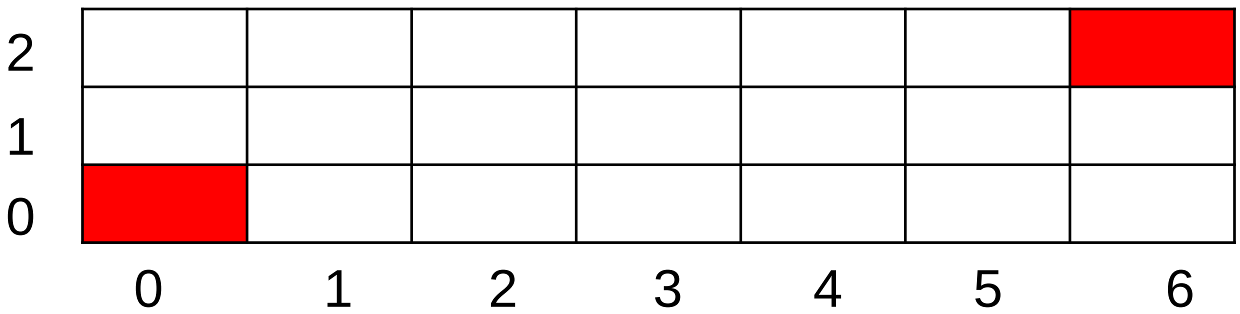
동쪽 픽셀 선택

D=D+incE // D값 업데이트

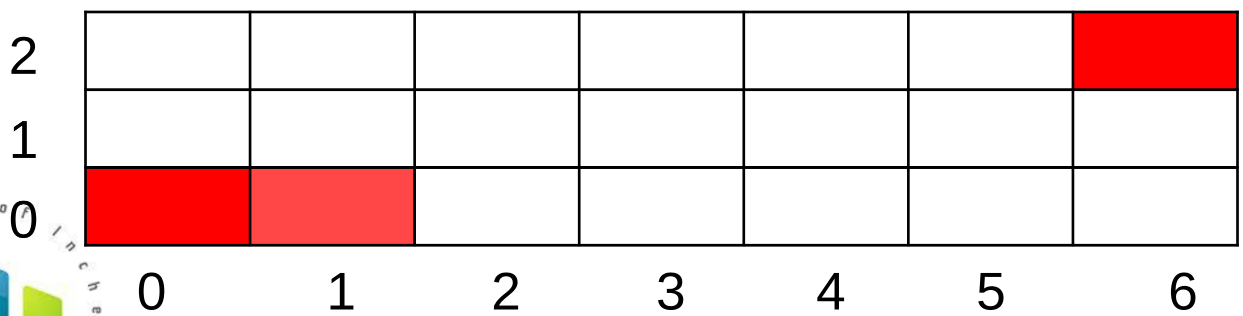
```
// 브레스넘 알고리즘 코드
void MidpointLine(int x1, int y1, int x2, int y2){
   int dx, dy, incrE, incrNE, D, x, y;
   dx = x2 - x1; dy = y2 - y1;
   D = 2*dy - dx;                  //결정변수 값을 초기화
   incrE = 2*dy;                   //동쪽 화소 선택시 증가분
   incrNE = 2*dy - 2*dx;           //동북쪽 화소 선택시 증가분
   x = x1;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

- 예: 앞의 브레스넘 알고리즘의 코드를 이용하여

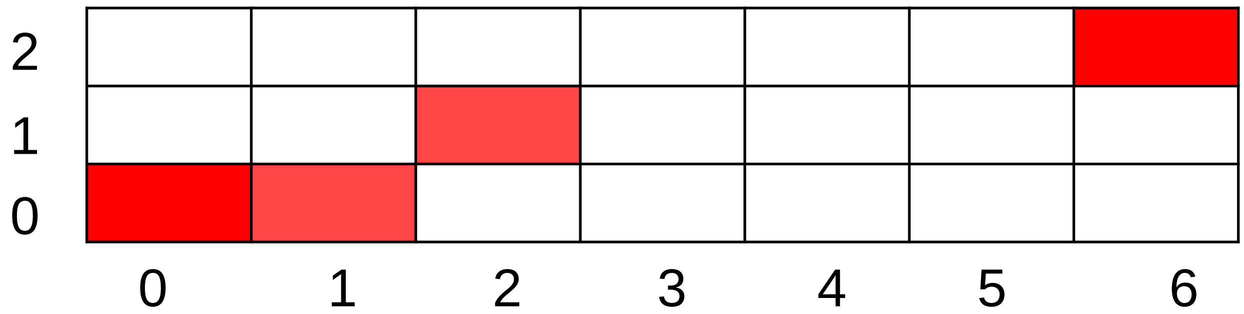
(x1,y1)=(0,0), (x2,y2)=(6,2) 의 래스터 변환을 수행하여 보자



1. dx=6, dy=2, D=2*dy-dx=-2<0 (동쪽), incrE=2*dy=4, D=D+incrE=2



- **2. D=2,   D>0 (동북쪽 화소선택), incrNE=2*dy-2*dx=-8, D=D+incrNE=-6**



- 계속 반복 언제까지
- **x=x2 (즉, x=6일 때 까지)**

- 브레스넘 알고리즘의 장점
- **1.** 선분의 **implicit** 표현을 사용하여 정수 사이의 연산으로만 되어 있다
- 정수 연산이 부동 소수점 연산보다 빠르다
- **2. round** 연산이 없다