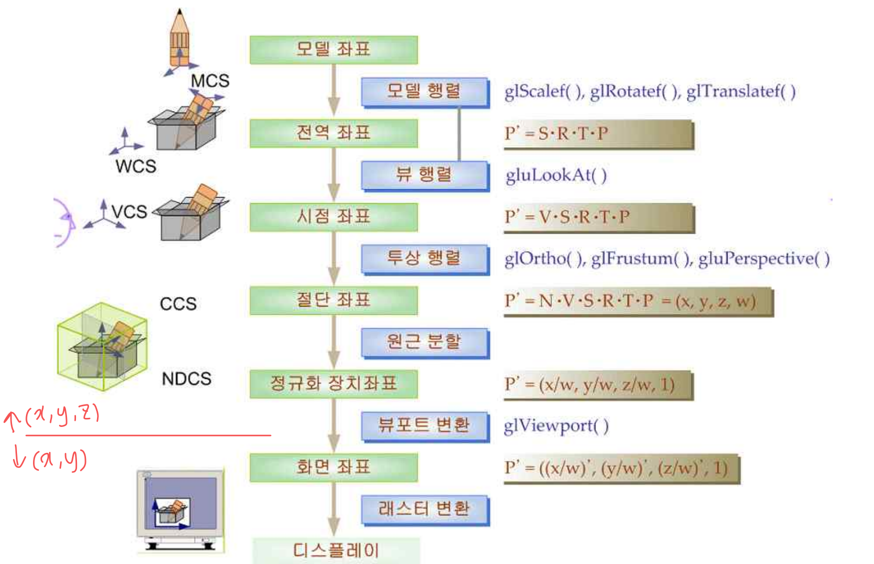
OpenGL의 파이프라인



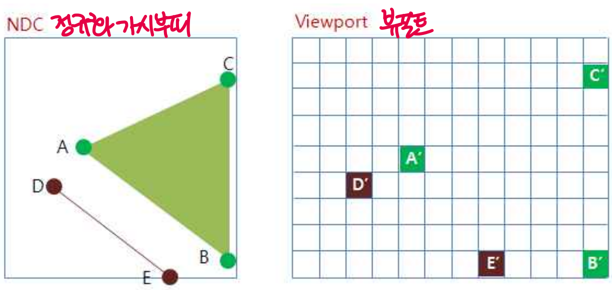
2

9장. 학습 내용

- 래스터 변환
- 선분의 래스터 변환 방법
- Bresenham 선분 그리기 알고리즘
- Midpoint 원 그리기 알고리즘
- Polygon Filling Algorithm
- 점과 다각형의 내부외부 판정
- 문자의 표현
- 앤티에일리어싱 (Anti-aliasing)

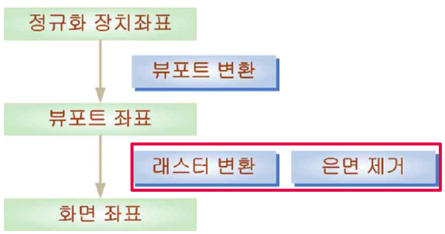
9.1 래스터 변환

- 래스터 변환 ~~또는~~ 스캔 변환(Scan Conversion)
  - Raster = 화소
  - 물체를 표현하기 위해 어떤 화소를 밝힐 것인지를 결정
  - 정규화 가시부피에서 뷰포트로의 사상
  - 정점좌표를 화면좌표로 변환한 결과를 기준으로
    - 선분을 화면좌표로 변환
    - 내부면을 화면좌표로 변환



OpenGL의 래스터변환

- 은면제거와 동시에 진행
  - 깊이와 색을 보간
  - 정점의 z 값으로부터 선분 및 내부면의 깊이를 보간
  - 정점의 색으로부터 선분 및 내부면의 색을 보간

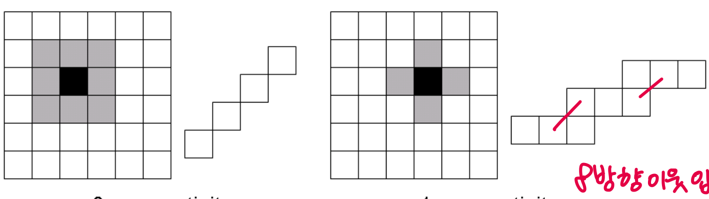


- 화면에 보이는 모든 것은 래스터 변환 결과
  - 최대의 연산속도, 최대의 정확성이 요구됨

5

9.2 선분의 래스터 변환

- 고려 사항
  - Smooth, even, and continuous as possible !!!
  - Simple and fast !!!
- 화소의 연결성
  - 4방향 이웃(4-connective neighbor)
  - 8방향 이웃(8-connective neighbor)



8-connectivity

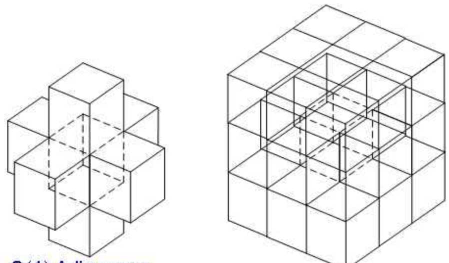
4방향 입장에서는 연결되어있지 X

4-connectivity

8방향 이웃 입장에서도 연결되어 있음

6

- 공간에서의 복셀의 연결성 Voxel
  - O(1)-Adjacency 면으로만 붙어있음
  - O(2)-Adjacency 하나의 edge까지 이웃가능
  - O(3)-Adjacency 하나의 정점까지 이웃가능 (?)



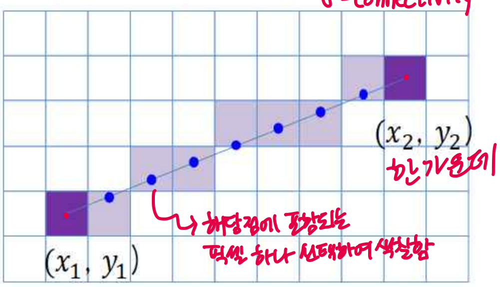
O(1) Adjacency

O(3) Adjacency

기본적인 선분 그리기 알고리즘

- 직선의 방정식을 이용

8-connectivity



$$y = mx + b$$

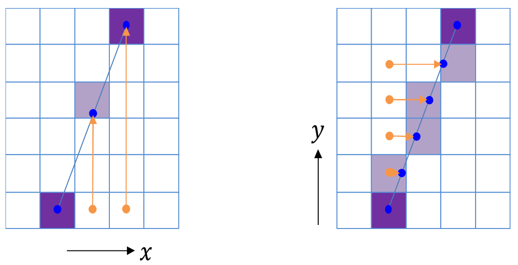
$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

$$b = y_1 - mx_1$$

y절편 구하는것 다시보기

선분의 기울기에 따른 고려

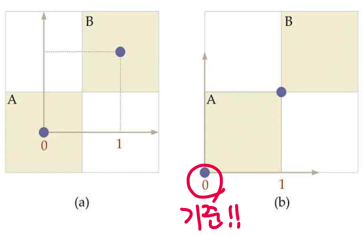
- 기울기를 기준으로 샘플링 축의 변경 필요
  - ① 1보다 크면 y 좌표를 증가 y++ → 이때의 x값
  - ② 1보다 작으면 x 좌표를 증가 x++ → 이때의 y값

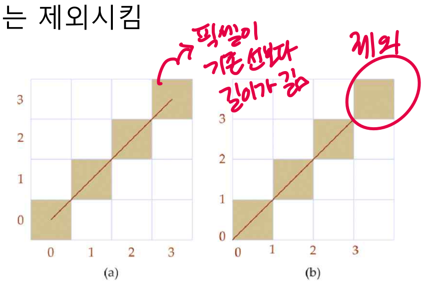


②X 기울기 m > 1 ①O

화소 좌표에 따른 고려

- 화소의 좌하단을 기준으로 부여하는 것이 일반적
  - 선분길이 조정을 위해 마지막 화소는 제외시킴
  - 면적 조정을 위해 외곽 화소는 제외시킴





알고리즘

```
void lineBasic(GLint x1, GLint y1, GLint x2, GLint y2)
{
    double m = (double)(y2-y1) / (x2-x1);  기울기
    for (int x = x1; x <= x2; x++) {  x증가시 y값구함
        double y = m * (x - x1) + y1;  곱셈사용
        displayPixel(x, ROUND(y));
    }
}
```

- 특징
  - 간단함
  - 부동 소수점 곱셈이 반복 → 속도가 느림

11

개선

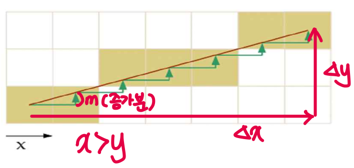
DDA 알고리즘

- DDA(Digital Differential Analyzer)
  - 증가분 m을 계산
  - 증가분을 반복적으로 더함

i) x>y

$$x_{i+1} = x_i + 1$$

$$y_{i+1} = y_i + m$$



ii) x<y $y_{i+1} = y_i + 1$, $x_{i+1} = x_i + m$

- 특징
  - 곱셈을 반복적으로 사용하지 않음
  - 여전히 부동 소수점 연산이 필요 3+4 (x), 3.0+4.0 (O)
  - 이상적인 m과 부동소수점으로 표현하는 m의 차이 (에러)
  - 긴 선분을 그리는 경우 에러가 점점 쌓이게 된다는 문제

12

DDA 알고리즘

더 큰쪽을 1씩 증가

```
void lineDDA(GLint x1, GLint y1, GLint x2, GLint y2)
{
    int dX = abs(x2 - x1); Δx
    int dY = abs(y2 - y1); Δy
    int steps = max(dX, dY);      //  (dX > dY) ? dX : dY;
    double incX = (double)(x2 - x1) / steps;
    double incY = (double)(y2 - y1) / steps;
    double x = x1;
    double y = y1;

    for (int i = 0; i <  steps; i++, x+=incX, y+=incY)
        displayPixel(ROUND(x), ROUND(y));
}
```

모든 방향 기울기 다 처리해줌 (abs 사용x)

덧셈 사용 → 반복사용x , 부동소수점 연산 필요

부동소수점, 반올림

- [Lab 9-1] 선분 그리기 알고리즘을 테스트할 수 있도록 OpenGL로 프로그램을 구현하라. 화면에 그림과 같이 정사각형 그리드가 그려지도록 하고, 마우스 이벤트를 처리하여 선분을 위한 두 점 p1, p2를 지정할 수 있도록 하라.

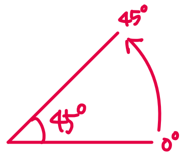
- [Lab 9-2] 앞에서 설명한 선분 그리기 알고리즘들을 구현하라. 특히 모든 상황에 대해 선분을 그릴 수 있도록 구현하라.

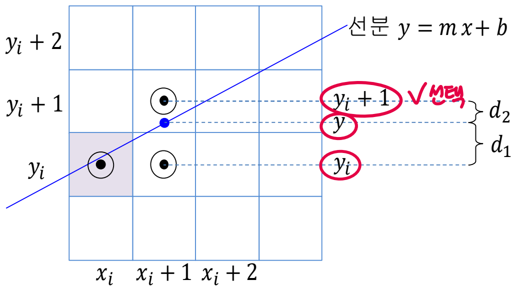
9.3 Bresenham 선분 그리기 알고리즘

- 보다 효율적인 선분 그리기 알고리즘
  - 브레스넘 알고리즘(Bresenham Algorithm)
  - 중점 알고리즘(中點, Midpoint Algorithm)

기말고사 설명출제
(시험에 그림+설명)

- 조건
  - 선분의 기울기는 0~1 사이임(0~45도)





√선택

$d_2 < d_1$

y에 더가까움

Bresenham 알고리즘 유도

그림에서 x_{i+1}일 때 $y = mx_{i+1} + b = m(x_i + 1) + b$ 이므로 d_1과 d_2는 다음과 같이 구해진다.

$$\begin{cases} d_1 = y - y_i = m(x_i + 1) + b - y_i \\ d_2 = (y_i + 1) - y = y_i + 1 - m(x_i + 1) - b \end{cases}$$

$d = d_1 - d_2$라 하면 d는 다음과 같이 정리된다.

$$d = d_1 - d_2 = 2m(x_i + 1) - 2y_i + 2b - 1$$

$d < 0$: $d_1 < d_2 \Rightarrow y_i$ 선택
$d > 0$: $d_1 > d_2 \Rightarrow y_{i+1}$ 선택

$m = \Delta y / \Delta x$ 이므로 d에 Δx를 곱한 값을 p_i라 하면 다음과 같이 정리된다.

$$\begin{aligned} p_i = \Delta x d &= 2\Delta y(x_i + 1) - 2\Delta x y_i + \Delta x(2b - 1) \\ &= 2\Delta y x_i - 2\Delta x y_i + \underbrace{(2\Delta y + \Delta x(2b - 1))}_{= c} \\ &= 2\Delta y x_i - 2\Delta x y_i + c \end{aligned}$$

$d_1 < d_2 : y_i$ 선택

만약 $p_i < 0$이면 d_1이 d_2보다 작은 상황이므로 다음 화소는 (x_i+1, y_i)가 되어야 할 것이고, 그렇지 않으면 다음 화소는 (x_i+1, y_i+1)가 되어야 한다. 이제 p_i와 함께 p_{i+1}도 계산해 보자.

$P_i \geq 0 : d_1 > d_2 : y_i+1$ 선택

$$p_i = 2\triangle y x_i - 2\triangle x y_i + c$$
$$p_{i+1} = 2\triangle y x_{i+1} - 2\triangle x y_{i+1} + c$$
$$p_{i+1} - p_i = 2\triangle y \underbrace{(x_{i+1} - x_i)}_{=1} - 2\triangle x (y_{i+1} - y_i)$$
$$\therefore p_{i+1} = p_i + 2\triangle y - 2\triangle x (y_{i+1} - y_i)$$
$$\boxed{y_{i+1} = \begin{cases} y_i & if\ p_i < 0 \\ y_i + 1 & otherwise \end{cases}} \rightarrow P_i \geq 0$$

맨 처음의 p_1는 다음과 같이 계산된다.

$$\begin{aligned} p_1 &= 2\triangle y x_1 - 2\triangle x y_1 + c \\ &= 2\triangle y x_1 - 2\triangle x y_1 + 2\triangle y + \triangle x (2b - 1) \\ &= 2\triangle y x_1 - 2\triangle x y_1 + 2\triangle y + \triangle x (2(y_1 - (\triangle y / \triangle x) x_1) - 1) \\ &= \cancel{2\triangle y x_1} - \cancel{2\triangle x y_1} + 2\triangle y + \cancel{2\triangle x y_1} - \cancel{2\triangle y x_1} - \triangle x \end{aligned}$$
$$\therefore p_1 = 2\triangle y - \triangle x$$

결국 이 알고리즘은 다음과 같이 정리된다.

$$\boxed{\begin{aligned} &\therefore p_{i+1} = \begin{cases} p_i + 2\triangle y & if\ p_i < 0 \\ p_i + 2\triangle y - 2\triangle x & otherwise \end{cases} \\ &\therefore p_1 = 2\triangle y - \triangle x \end{aligned}}$$

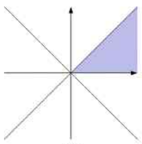
알고리즘(기본 상황만 처리)

```
void lineBresenham(GLint x1, GLint y1, GLint x2, GLint y2)
{
    int dX = x2 - x1;
    int dY = y2 - y1;
    int const1 = 2 * dY;
    int const2 = 2 * (dY - dX);
    int p = 2 * dY - dX;
    int y = y1;
    for (int x = x1 + 1; x < x2; x++) {
        if (p<0) p += const1;   y는 그대로 〃 →.
        else {
            p += const2;
            y++;
        }
        displayPixel(x, y);
    }
    displayPixel(x1, y1);
    displayPixel(x2, y2);
}
```

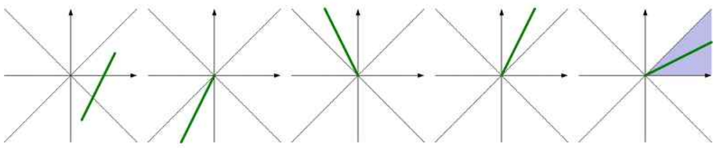
19

알고리즘 특징

- 정수 연산, 덧셈에 의한 속도 증가 + 하드웨어로 구현
- 첫 8분면에서만 정의
  - 다른 선분은 이동, 반사하여 적용



(a)

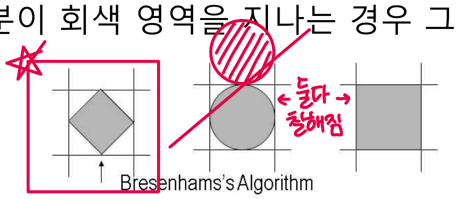


(b) (c) (d) (e) (f)

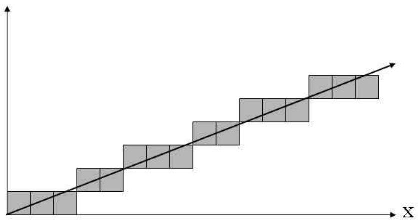
20

기하학적 이해

- 선분이 회색 영역을 지나는 경우 그 화소를 그림

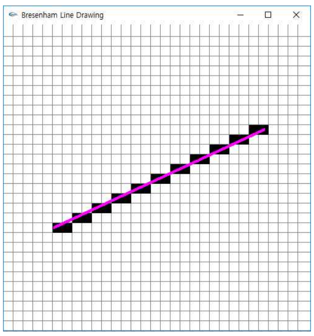


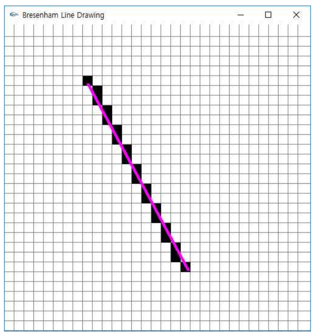
- 8-connectivity의 연속성을 기준



21

- [Lab 9-3] 프로그램 9.3의 Bresenham 알고리즘을 모든 입력 상황에 대해 처리할 수 있도록 확장하라. 다음과 같은 실행결과를 참고하라.

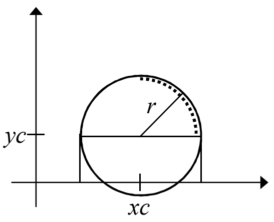




22

9.4 Midpoint 원 그리기 알고리즘

- 원의 다양한 표현 방법

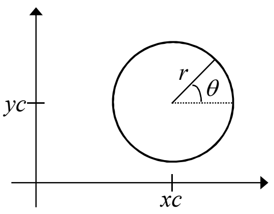


Pythagorean Theorem

$$(x-xc)^2+(y-yc)^2=r^2$$

중심:(xc, yc)
반지름: r

$$y=yc\pm\sqrt{r^2-(x-xc)^2}$$



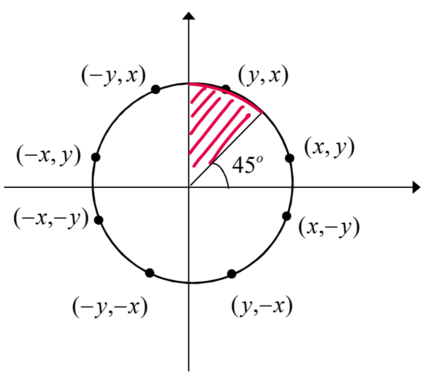
Polar Form

$$x=xc+r\cos\theta$$
$$y=yc+r\sin\theta$$

23

시험

원 그리기의 대칭성

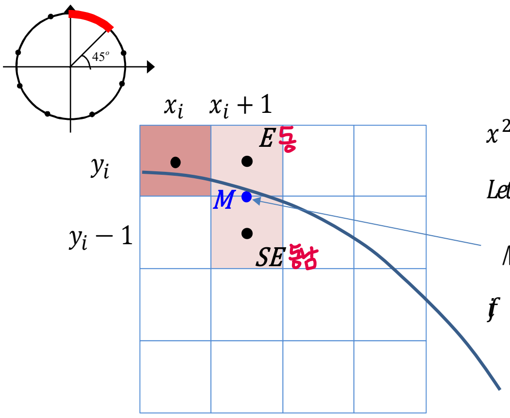


원의 1/8인 45°만
그리면 원완성됨

using symmetry

24

Midpoint Circle Algorithm



$x^2 + y^2 = R^2$

Let $F(x, y) = x^2 + y^2 - R^2$

= 원 위에
< 원내부
> 원외부

Midpoint M: $(x_i + 1,\ y_i - \frac{1}{2})$ $x^2+y^2<R^2$ ↳ 원의내부

중점

if $F(M) < 0$ → E (M이 원의 내부 ⇒ 선을 E에 그림)
otherwise → SE (M이 원의 외부 ⇒ 선을 SE에 그림)

25

Midpoint Circle 알고리즘 유도

중점 M의 좌표 $\underbrace{(x_i+1, y_i-\frac{1}{2})}_{M}$을 $F(x,y)$에 넣고 이를 p_i라 하자.

$$p_i = F(x_i+1, y_i-\frac{1}{2}) = (x_i+1)^2 + (y_i-\frac{1}{2})^2 - R^2$$

만약 $p_i < 0$이면 M이 원의 내부에 있으므로 다음 화소는 (x_i+1, y_i)가 되어야 하고 M이 원의 외부이면 다음 화소는 (x_i+1, y_i-1)이 되어야 한다.

$$(x_{i+1}, y_{i+1}) = \begin{cases} (x_i+1, y_i) & if\ p_i < 0 \\ (x_i+1, y_i-1) & otherwise \end{cases}$$

26

$p_i = F(x_i+1, y_i-\frac{1}{2}) = (x_i+1)^2+(y_i-\frac{1}{2})^2-R^2$

$p_{i+1} = F(x_{i+1}+1, y_{i+1}-\frac{1}{2}) = (x_i+2)^2+(y_{i+1}-\frac{1}{2})^2-R^2$ $\quad (y_i-\frac{3}{2})^2$

$p_i = x_i^2+2x_i+1+y_i^2-y_i+\frac{1}{4}-R^2$

if $p_i<0$: $p_{i+1} = x_i^2+4x_i+4+y_i^2-y_i+\frac{1}{4}-R^2$

otherwise : $p_{i+1} = x_i^2+4x_i+4+y_i^2-3y_i+\frac{9}{4}-R^2$

같은 방법으로 p_{i+1}를 구하면 다음과 같다.

$$p_{i+1} = F(x_{i+1}+1, y_{i+1}-\frac{1}{2}) = (x_i+2)^2+(y_{i+1}-\frac{1}{2})^2-R^2$$

이때, $x_{i+1}=x_i+1$이고, $y_{i+1} = \begin{cases} y_i & if\ p_i<0 \\ y_i-1 & otherwise \end{cases}$ 이므로 이를 이용해 $p_{i+1}-p_i$를 구하고 p_{i+1}를 중심으로 정리하면 다음과 같다.

$$p_{i+1} = \begin{cases} p_i+2x_i+3 & if\ p_i<0 \\ p_i+2(x_i-y_i)+5 & otherwise \end{cases}$$

초기조건 p_0을 구해보자. 첫 좌표는 $(x_0, y_0)=(0, R)$이므로 이를 넣어 정리하면 p_0은 다음과 같이 구해진다.

(0,R)

$$p_0 = F(x_0+1, y_0-\frac{1}{2}) = (0+1)^2+(R-\frac{1}{2})^2-R^2 = \frac{5}{4}-R$$

27

결론적으로 Midpoint 원그리기 알고리즘은 다음과 같이 정리된다.

$$\boxed{\begin{aligned} (x_{i+1}, y_{i+1}) &= \begin{cases} (x_i+1, y_i) & if\ p_i<0 \\ (x_i+1, y_i-1) & otherwise \end{cases} \\ p_{i+1} &= \begin{cases} p_i+\boxed{(2x_i+3)} & if\ p_i<0 \\ p_i+\boxed{(2x_i-2y_i+5)} & otherwise \end{cases} \\ p_0 &= \frac{5}{4}-R \end{aligned}}$$

결과를 보면 p_{i+1}는 p_i에 정수를 계속 더하는 방법으로 계산된다. 그런데 우리는 p_i의 부호만을 사용할 것이고, p_i에는 정수만이 더해질 것이므로 알고리즘 구현해서 부동 소수점 계산을 피하기 위해, 약간 변형할 수 있다. 즉 $d_i = p_i - 1/4$라 하면 위의 식은 다음과 같이 정수 형태로 정리된다.

부호변하지X

$$\boxed{\begin{aligned} (x_{i+1}, y_{i+1}) &= \begin{cases} (x_i+1, y_i) & if\ d_i<0 \\ (x_i+1, y_i-1) & otherwise \end{cases} \\ d_{i+1} &= \begin{cases} d_i+(2x_i+3) & if\ d_i<0 \\ d_i+(2x_i-2y_i+5) & otherwise \end{cases} \\ d_0 &= 1-R \end{aligned}}$$

28

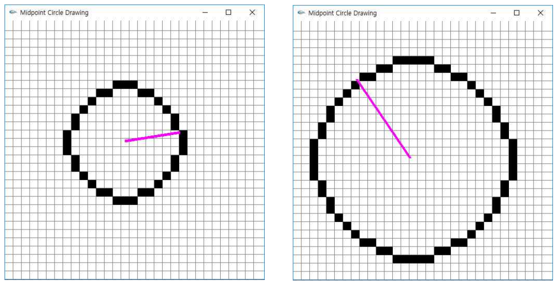
알고리즘(기본 상황만 처리)

```
void circleMidPoint(GLint radius) // 중심이 (0,0)이고 반지름이 R인 원
{
    int x = 0;
    int y = radius;
    int d = 1 - radius;

    drawCirclePoint(x, y);
    for( ; y > x ; x++) {
        if (d < 0) y는 그대로
            d = d + 2 * x + 3;
        else {
            d = d + 2 * (x-y) + 5;
            y = y - 1;
        }
        drawCirclePoint(x, y);
    }
}
```


알고리즘 특징

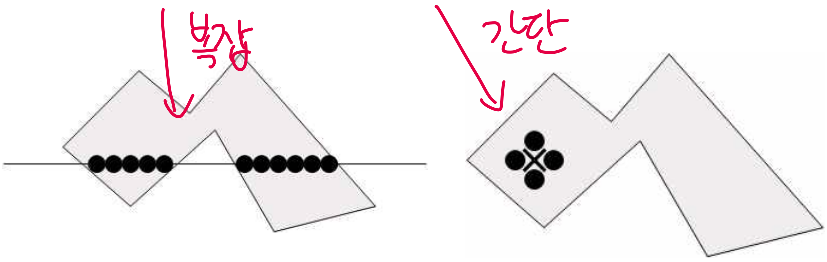
- 부동 소수점 연산은 필요 없음
- 곱셈이 사용 → 2의 배수 → 시프트(shift) 연산
- 전체 원의 1/8을 그림 $2x_i$, $2y_i$
  - 대칭을 이용하여 나머지 원의 좌표가 계산된다.

- [Lab 9-4] 프로그램 9.4의 Bresenham 알고리즘을 모든 입력 상황에 대해 처리할 수 있도록 확장하라. 특히 drawCirclePoint(x,y) 함수에서는 8 화소에 대해 그려야 할 것이다. 다음과 같은 실행결과를 참고하라.



9.5 다각형 채움 알고리즘

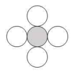
- 스캔라인 방식 / 씨앗 채움 방식



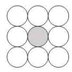
코드는 간단 but 효율적 X
영역이 넓으면 상당히 느려짐 ∴ 일반적으로는 스캔라인 씀

Boundary Fill / Seed Fill Algorithm

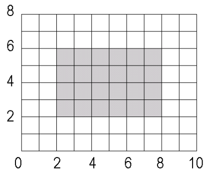
- 내부에 씨앗이 되는 화소를 정하고 이 화소에서부터 인접한 화소들을 채워 나가는 방식
  - 재귀함수로 구현하거나 stack을 사용하여 구현
  - 화소의 연결성에 대한 정의가 중요
- 내부가 정의되어 있으면: flood fill
- 경계가 정의되어 있으면: boundary fill



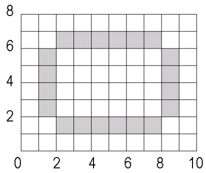
4-connected



8-connected



Interior-defined
↓
flood fill algorithm



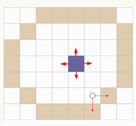
boundary-defined
↓
boundary fill algorithm

33

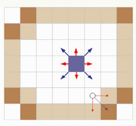
Boundary Fill Algorithm

```
boundaryFill ( int x, int y, int fill, int boundary)
{
   int  current;
   current = getPixel (x, y);
   if (current != boundary && current != fill) {
      setPixel (x, y, fill);
      boundaryFill (x+1, y, fill, boundary);
      boundaryFill (x-1, y, fill, boundary);
      boundaryFill (x, y+1, fill, boundary);
      boundaryFill (x, y-1, fill, boundary);
   }
}
```

순환호출



(a)

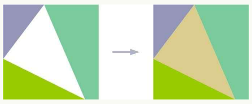


(b)

34

Flood Fill Algorithm

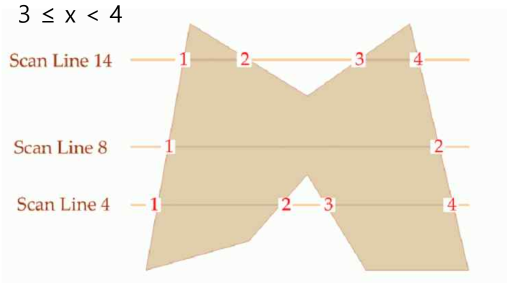
```
floodFill4 ( int x, int y, int fillColor, int oldColor)
{
  if (getPixel(x, y) == oldColor) {
     setPixel (x, y, fillColor);
     floodFill4 (x+1, y, fillColor, oldColor);
     floodFill4 (x-1, y, fillColor, oldColor);
     floodFill4 (x, y+1, fillColor, oldColor);
     floodFill4 (x, y-1, fillColor, oldColor);
  }
}
```



35

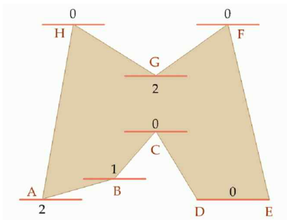
Scanline Fill Algorithm

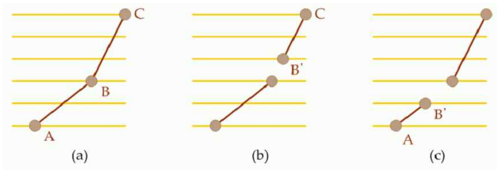
- Jordan Theorem으로 불리는 Odd-Even Rule 적용
- 홀수 규칙(Odd Parity Rule, Even-Odd Rule)
  - 홀수번째 교차화소부터 짝수번째 교차화소 직전까지 채움
  - 짝수번째를 포함하지 않는 이유: 길이보존
    - 14번: 1 ≤ x < 2, 3 ≤ x < 4
    - 8번: 1 ≤ x < 2
    - 4번: 1 ≤ x < 2, 3 ≤ x < 4



36

특수한 경우에 대한 처리 이것도 뭔가 제대로 설명아예 안하심

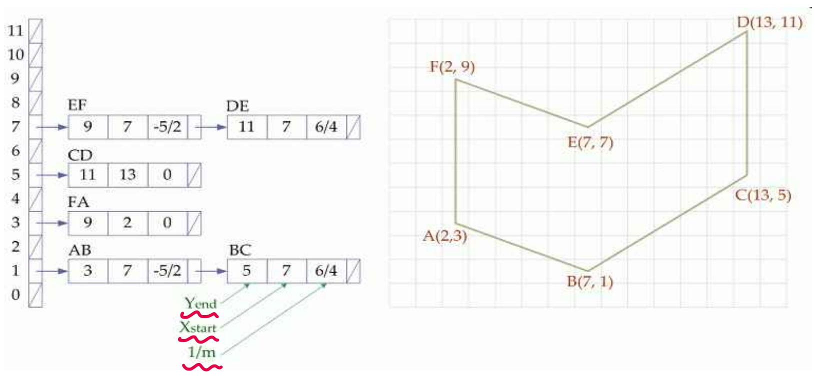




- 길이보존
  - 극대점: 교차하지 않은 것으로 간주(H, F, C)
  - 극소점:각각 교차한 것으로 간주: 2번(G, A)
- 극대극소: 1번 교차(B): 2개의 선분으로 분할
- 주사선과 평행
  - 선분이 없는 것으로 간주(DE)
  - CD, FE에 의해서 처리됨

37

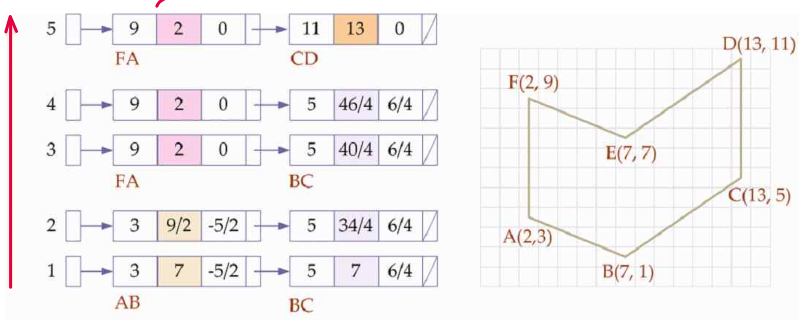
선분 리스트(선분 테이블)



38

활성화 선분 리스트(Active Edge List)

항상 짝수개임



39

알고리즘 수업때 대충 넘어가심

```
struct Point2D { int x, y; };
struct EdgeRec {          // 에지 레코드 관리를 위한 클래스
    int     yUpper;
    double  xIntersect, dxPerScan;
    EdgeRec* next;

    EdgeRec() { next = NULL;  }
    EdgeRec(Point2D* lower, Point2D* upper, int yComp) {
        dxPerScan = (double)(upper->x-lower->x) / (upper->y-lower->y);
        xIntersect = lower->x;
        yUpper  = (upper->y < yComp) ? upper->y - 1 : upper->y;
        next = NULL;
    }
    void AddNext(EdgeRec* e) { ... }            // e를 다음에 추가
    void AddSorted(EdgeRec *edge) { ... }       // 정렬된 위치에 삽입
    void DeleteNext() { ... }                   // edge를 리스트에서 제거
    void DeleteAll() { ... }                    // 이후의 모든 에지 제거
};
```

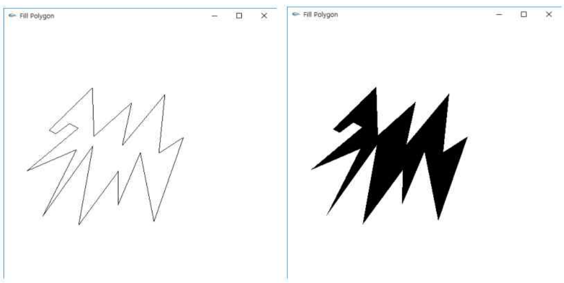
40

```
// global variables -------------------------------------------------
static EdgeRec     edges[4096];
static EdgeRec     active;

// 모든 스캔라인에 대해 에지 리스트를 만듦.
static void buildEdgeList(int cnt, Point2D* pts) { ... }
// active 리스트에 해당 스캔라인의 에지 추가
static void buildActiveList(int scan) { ... }
// active 리스트 갱신. 끝난 에지 삭제. 그렇지 않으면 xIntersect 갱신
static void updateActiveList(int scan) { ... }
// active 리스트 안의 에지들을 xIntersect 순으로 재정렬
static void resortActiveList() { ... }
// active 리스트를 이용해 현재 스캔라인(scan)을 그림
```



```
static void fillScan(int scan)
{
    EdgeRec *p1 = active.next;
    while (p1 != NULL) {
        EdgeRec *p2 = p1->next;
        glBegin(GL_LINES);
            glVertex2i(round(p1->xIntersect), scan);
            glVertex2i(round(p2->xIntersect), scan);
        glEnd();
        p1 = p2->next;
    }
}
// Main routine: 정점수가 count인 정점 배열과 화면 크기를 받아 그림
void scanFill(int count, Point2D* points, int ww, int wh)
{
    buildEdgeList(count, points);

    for (int i = 0; i < wh; i++) {
        buildActiveList(i);
        if (active.next != NULL) {
            fillScan(i);
            updateActiveList(i);
            resortActiveList();
        }
    }
}
```

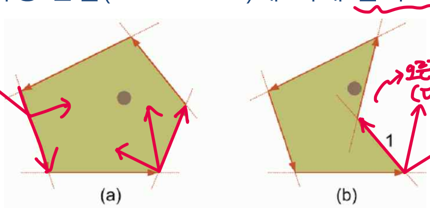
- [Lab 9-5] Scan-line polygon fill 알고리즘을 구현하여 프로그램 9.5를 완성하라. 자세한 알고리즘은 인터넷을 참고하라. 생각보다 알고리즘이 복잡할 것이다. 다음과 같이 먼저 다각형을 지정해야 한다. 이를 위한 사용자 인터페이스도 구상해야 한다. 2장을 복습하라. 다음과 같은 실행결과를 참고하라.



9.6 점과 다각형의 내부외부 판정

- 평면에서 주어진 점이 다각형의 내부에 있는지 외부에 있는지를 판단
  - 예들 들어 평면에 다양한 다각형 객체를 그리고 하나를 선택할 때 마우스 클릭으로 할 수 있는데, 이때 클릭한 점의 위치가 어느 다각형의 내부에 있는지를 판단

- 진행방향의 왼쪽이 내부 외적
  - 볼록 다각형에서만 성립
  - 오목 다각형의 경우 다각형 분할(Tessellation)에 의해 볼록 다각형의 집합으로 변형

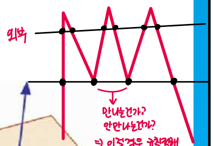


오른쪽
(다른선과 부딪히다함)

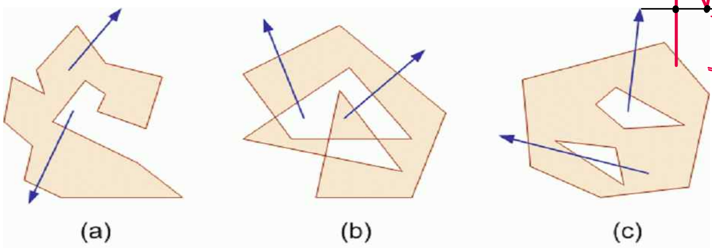
홀수 규칙(Odd Parity Rule, Even-Odd Rule)

홀수-내부, 짝수-외부

- 볼록, 오목에 무관하게 내외부 판정
- 내부점으로부터 외부를 향한 직선은 다각형과 반드시 홀수 번 교차



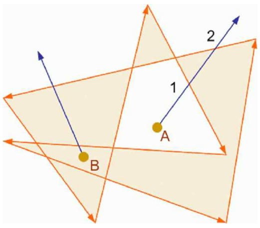
만나는건가?
안만나는건가?
=> 이럴경우 규칙정해 예외처리함
ex) 조금올려서 두번만나게 함(나머지도 똑같이 적용)
or 조금 내려서 아예 안만나게 함(나머지도 똑같이 적용)

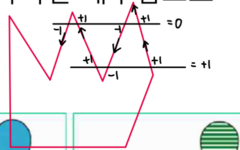


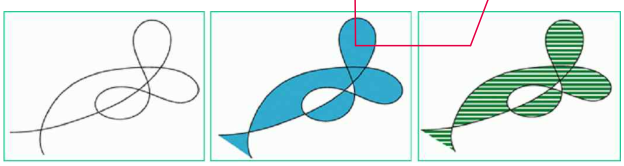
넌 제로 와인딩 (Non-Zero Winding Rule)

- 선분의 방향을 고려
- 감싸기 수(Winding Number)
  – 선분이 반 시계방향으로 그 점을 몇 번이나 감싸는가. 0으로 초기화. 선분의 오른쪽에서 왼쪽으로 건너가면 +1, 왼쪽에서 오른쪽으로 건너가면 -1. 최종 감싸기 수가 0이 아니면 내부점으로 간주

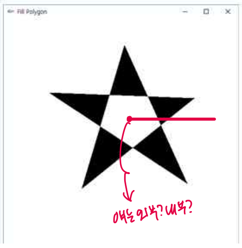
0 = out
0 != in







- [Lab 9-6] 프로그램 9.6을 완성하라. 다음은 화면의 세 점에 대한 내외부 판정 결과이다. 별모양의 중심부에서 두 알고리즘의 실행 결과가 in과 out으로 다른 것에 유의하라. 넌제로 와인딩



C:\WINDOWS\system32\cmd.exe

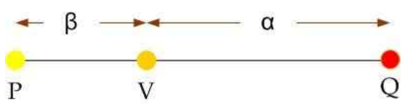
```
num cross =1 in
num winding=1 in
num cross =2 out
num winding=2 in
num cross =0 out
num winding=0 out
```

47

9.7 보간법

- 보간(interpolation)이 필요한 경우
  - 색, 깊이, 법선 벡터, 텍스쳐 등
  - 보다 매끈하고 실감나게 보이도록
  - 선과 면 등에 대해 처리

- 선분의 무게중심 좌표
  - (α, β): 가중치
  - 색이 연속적이 되도록.



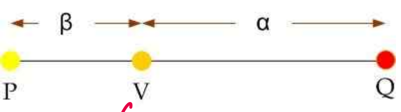
$$V(t) = P + t(Q - P) = (1-t)P + tQ = \alpha P + \beta Q$$

$$0 \leqq \alpha, \beta \leqq 1, \ \alpha + \beta = 1$$

48

무게중심 좌표 (선분)

- 선분의 무게중심 좌표 (α, β)
  - (α, β): 가중치 , blending 비율
  - 색이 연속적이 되도록.



↳ P,Q의 선형결합으로 표현 (일정 비율로 혼합)

$$V(t) = P + t(Q - P) = (1-t)P + tQ = \alpha P + \beta Q$$

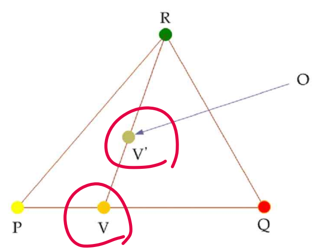
$$0 \leqq \alpha, \beta \leqq 1, \ \alpha + \beta = 1$$

무게중심 좌표 (삼각형)

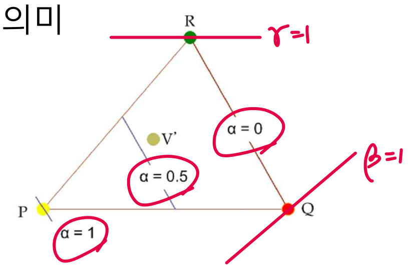
- 삼각형의 무게중심 좌표 (α, β, γ)

$$\begin{aligned} V' &= V + s(R - V) \\ &= P + t(Q-P) + s(R - (P + t(Q-P))) \\ &= (1 - t - s + st)P + (t - st)Q + sR \\ &= \alpha P + \beta Q + \gamma R \end{aligned}$$

$$0 \leqq \alpha, \beta, \gamma \leqq 1, \ \alpha + \beta + \gamma = 1$$



- α 의 의미



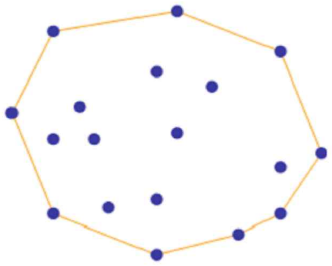
무게중심 좌표 (컨벡스 헐)

- 컨벡스 헐(Convex Hull)
  - 주어진 점을 모두 포함하는 가장 작은 볼록 다각형
- Convex Hull Property
  - 다음 식으로 표현된 정점 V는 항상 컨벡스 헐 내부에 존재

컨벡스헐 특징

$$V = t_1 P_1 + t_2 P_2 + \dots + t_n P_n$$

$$0 \leqq t_1, t_2, \dots, t_n \leqq 1,\ t_1 + t_2 + \dots + t_n = 1$$



51

무게중심 좌표 계산

여기까지만 알면될듯?

$$\alpha = area\,(V'QR)/area\,(PQR)$$

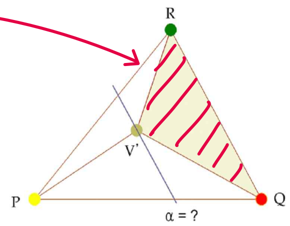
$$\beta = area\,(V'RP)/area\,(PQR)$$

$$\gamma = area\,(V'PQ)/area\,(PQR)$$

$$\alpha = abs\,((V' - Q) \times (R - Q))/abs\,((P - Q) \times (R - Q))$$

$$\beta = abs\,((V' - R) \times (P - R))/abs\,((P - Q) \times (R - Q))$$

$$\gamma = abs\,((V' - P) \times (Q - P))/abs\,((P - Q) \times (R - Q))$$



2차원 투상

면적

$$area\,(PQR)$$

$$= \frac{1}{2} \begin{vmatrix} P_x & Q_x & R_x \\ P_y & Q_y & R_y \\ 1 & 1 & 1 \end{vmatrix}$$

$$= \frac{1}{2} \left(\begin{vmatrix} Q_x & R_x \\ Q_y & R_y \end{vmatrix} + \begin{vmatrix} R_x & P_x \\ R_y & P_y \end{vmatrix} + \begin{vmatrix} P_x & Q_x \\ P_y & Q_y \end{vmatrix} \right)$$

$$= \frac{1}{2} (Q_x R_y - R_x Q_y + R_x P_y - P_x R_y + P_x Q_y - Q_x P_y)$$

$$= \frac{1}{2} ((Q_x - P_x)(R_y - P_y) - (R_x - P_x)(Q_y - P_y))$$

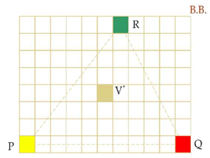
면적

52

무게중심 좌표에 의한 보간

화면상의 모든 화소 테스트? → 비효율적

- 경계상자(BB: Bounding Box)
  - 다각형을 둘러싼 최소크기 4각형



- 보간 과정 (BB 내 모든 화소에 대해)
  1. 해당 화소가 삼각형 내부인지 판단
  2. 화소가 내부이면 무게중심 좌표를 계산
  3. 색과 깊이를 보간

색

$$r = \alpha P_r + \beta Q_r + \gamma R_r$$

$$g = \alpha P_g + \beta Q_g + \gamma R_g$$

$$b = \alpha P_b + \beta Q_b + \gamma R_b$$

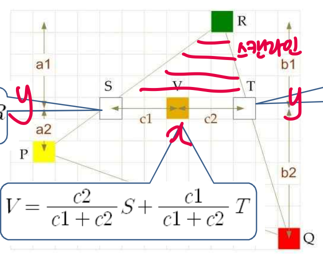
깊이

$$z = \alpha P_z + \beta Q_z + \gamma R_z$$

무게중심 좌표보다 더 많이 쓰임

양방향 선형보간(Bilinear Interpolation)

- Bilinear
  - Y 방향 보간에 의해 S, T를 구함 → 선형보간
  - X 방향 보간에 의해 V를 구함 → 선형보간 → 양선형 보간
  - 무게중심 좌표와 일치(모든 화소를 대상으로 삼각형 내부인지 판단할 필요X)
  - 연산 속도는 더 빠름



$$S = \underbrace{\frac{a1}{a1+a2}}_{+} P + \underbrace{\frac{a2}{a1+a2}}_{=1} R \quad y$$

항상 1

$$T = \frac{b1}{b1+b2} Q + \frac{b2}{b1+b2} R \quad y$$

$$V = \frac{c2}{c1+c2} S + \frac{c1}{c1+c2} T \quad x$$

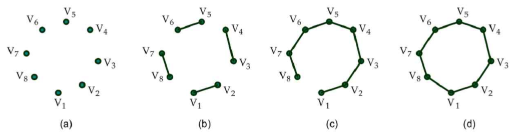
① 엣지 먼저 보간

② 엣지를 사용하여 내부점을 보간 (스캔라인 Fill)

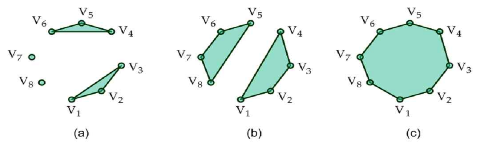
지난번 시험에 나옴

9.8 OpenGL의 그래픽 기본요소

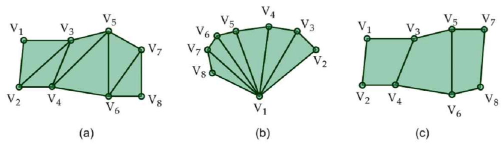
- GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP



- GL_TRIANGLES, GL_QUADS, GL_POLYGON



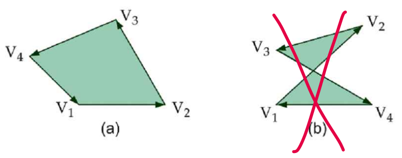
- GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUAD_STRIP



제약조건 (다각형에 대한 GL의 제약조건)

3가지 조건 만족 해야함

- 단순 다각형(Simple Polygon) - 엣지가 겹치면 안됨



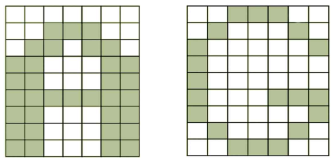
단순O
볼록X

- 볼록 다각형(Convex Polygon)
- 평면 다각형(Flat Polygon)

- 만족하지 않으면
  - 실행결과 예측불가
  - 다각형을 삼각형으로 분할하면 됨 (프로그래머의 책임)

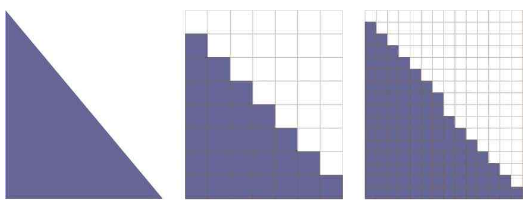
9.9 비트맵과 포스트스크립트

- 편집기 예
  - 비트맵 : Adobe Photoshop
  - 포스트스크립트 : Adobe Illustrator (벡터)
- 비트맵(Bitmap)
  - 래스터 모니터 영상, 스캐너로 읽은 영상, 팩스에 인쇄된 영상
  - 페인트 브러시로 만든 영상
  - 영상을 구성하는 개별 화소의 색을 표현하고 저장
  - 예: 7X9 = 63 개의 화소배열



비트맵(Bitmap)

- 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

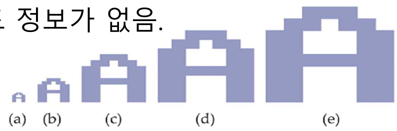


(a) (b) (c)

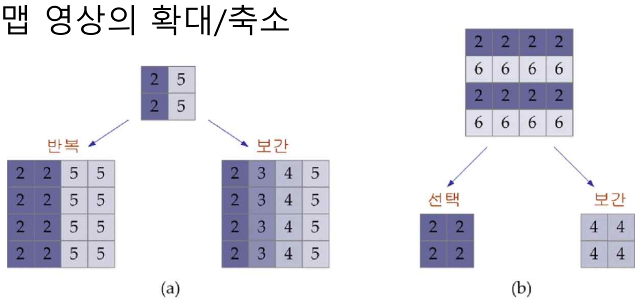
비트맵의 변환

시간걸림 ex) 90도

- 확대, 회전 등 — 문제 생김
  - 필연적으로 에일리어싱을 수반
  - 추가의 화소를 채우기 위한 별도 정보가 없음.



- 비트맵 영상의 확대/축소



59

포스트 스크립트(Postscript)

- 벡터 그래픽 장비로부터 유래
  - 화소라는 개념이 없음. 무한 해상도
- 실제로는 영상을 그려내는 방식
  - 물체(객체, Object)단위로 물체를 표현
  - 화소 대신 정점좌표를 사용

- 비트맵 그리기 = PAINTING
- 포스트스크립트 그리기 = DRAWING
- 영상선택
  - 비트맵: 비트 단위, 포스트스크립트: 물체(객체) 단위

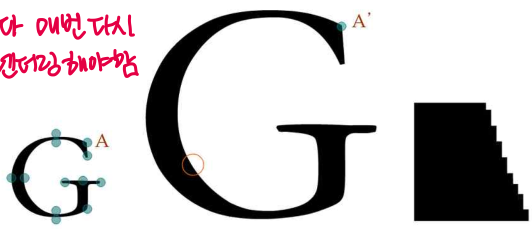
60

비트맵 영상과 벡터 영상 비교

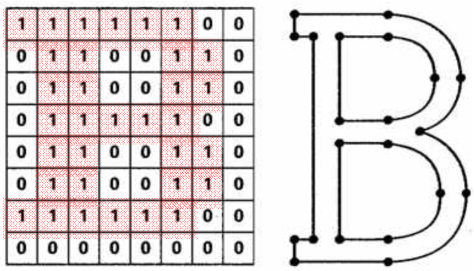
포스트 스크립트 폰트

- 영상의 윤곽선을 수식으로 표현
  - 특징적인 정점의 좌표, 이를 연결하는 보간 곡선의 수식을 명시
  - 특징적인 정점 = 제어점(Control Point) (픽셀값X, 제어점O)
  - 확대된 정점 위치에 보간 곡선을 다시 적용
    - 비트맵 보다 매끄러운 곡선
    - 에일리어싱 완화

바뀔때마다 매번 다시 전체를 렌더링해야함



(a) (b) (c)



63

그래픽 파일 형식

X 그냥 있다는 것만 보고 넘어가자~
skip

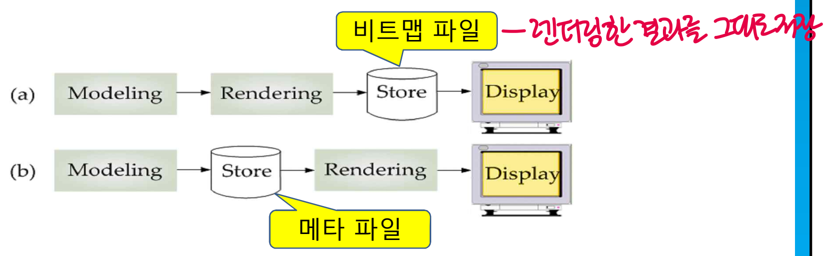
- 영상압축
  - 무손실 압축(Lossless Compression),손실압축(Lossy Compression)
- BMP(BitMapped Picture)
  - 마이크로 소프트 윈도우즈 운영체제의 기본 비트맵 파일. 일반적으로 압축을 가하지 않은 파일.
- GIF(Graphic Interchange Format)
  - 무손실 압축을 사용한 비트맵 파일. 8비트 컬러 256 컬러 중 하나를 투명성을 구현하는데 사용
- GIF 89a(Graphic Interchange Format 89a)
  - 애니메이션을 위한 파일 형식으로서 하나의 파일에 일련의 영상을 저장. Moving GIF. 프레임 재생률 제어가능. 256 컬러. 사운드 추가할 수 없음. 단순한 웹 애니메이션

64

- PNG (Portable Network Graphics)
  - W3C에서 추천 파일형식. 향상된 투명성 제어기능. 무손실
- JPEG(Joint Photographic Expert Group)
  - JPEG은 엄밀한 의미에서 일종의 압축 기법. 파일 형식이 아님. 24비트 컬러를 지원. 손실압축.
- TIFF(Tagged Image File Format)
  - 8비트, 24비트 컬러 지원. JPEG 및 기타 압축방법을 수용

비트맵 파일과 메타 파일

차이점만 알아두기 !!



- 메타파일
  - ~~렌더링 결과 저장: 비트맵 파일~~
  - 모델링 결과와 렌더링 명령어 저장
  - 예: 포스트스크립트 파일(PDF(Postscript Description File))
    - 0 1 0 setrgbcolor　　현재 색을 녹색으로 설정
    - 0 0 128 128 rectfill　　외부 사각형을 채움
    - 1 0 1 setrgbcolor　　현재 색을 자홍으로 설정
    - 32 32 64 64 rectfill　　내부 사각형을 채움

메타 파일 종류 X

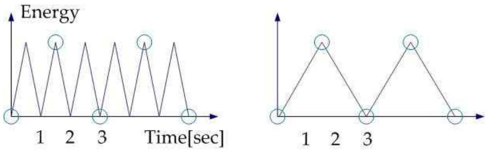
- EPS(Extended PostScript)
  - 포스트스크립트, 비트맵, 텍스트를 동시에 저장.
- SWF(Shockwave Flash)
  - 플래시 애니메이션을 위한 파일형식.
  - 웹 애니메이션에서 사실 표준
- WMF(Windows Meta File)
  - 마이크로소프트 윈도우즈에서 사용하는 파일
- SVG(Scaleable Vector Graphic)
  - W3C 추천하는 그림파일 형식
  - XML(Extensible Markup Lang)에서 자주 사용
- PICT(PICTure)
  - 매킨토시에서 사용하는 표준 메타파일 형식.

9.10 앤티에일리어싱

시험 100%

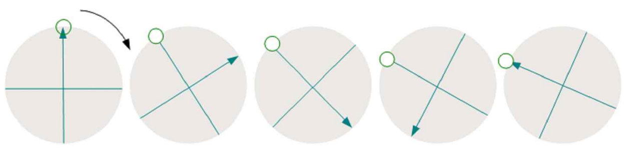
- 에일리어싱(Aliasing)
  - 신호의 Under sampling으로 인함 원래 신호가 가지고 있는 것보다 샘플링 주파수가 낮아서 발생
  - 신호의 복원?
  - 나이퀴스트 주파수

나이퀴스트 주파수 => 원본 신호가 가지고 있는 주파수의 2배 이상은 샘플링 해야 신호복원가능



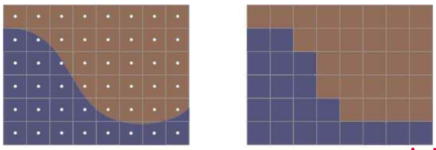
(a) (b)

- Stroboscopic Effect
  - 시간적 에일리어싱



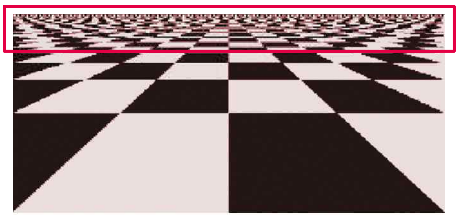
점 샘플링과 모아르 패턴

- 점 샘플링으로 인한 에일리어싱



(a) want (b) reality

- 모아르 패턴
  - 뒷 부분의 높은 주파수를 화소 크기가 수용하지 못함

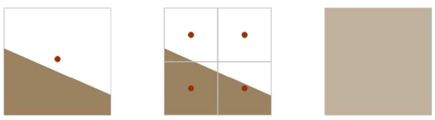


69

앤티 에일리어싱(Anti-Aliasing) (에일리어싱 완화 방법)

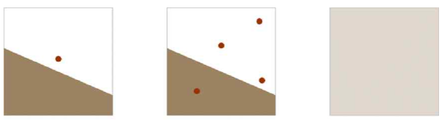
중간중간 어중간한 색을 사용하여 멀리서 보면 매끈해보이게!!

- 수퍼 샘플링(Super-Sampling) (영역 샘플링보다 단순함)
  - 부분화소에서 샘플링. 사후 필터링
  - 부분화소의 평균값을 반영



(a) (b) (c)

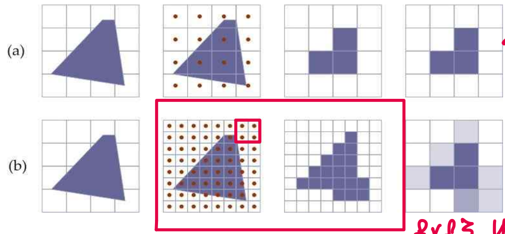
  - 지터에 의한 수퍼 샘플링
    - 물체 자체가 불규칙이라면 불규칙 샘플링이 유리



(a) (b) (c)

70

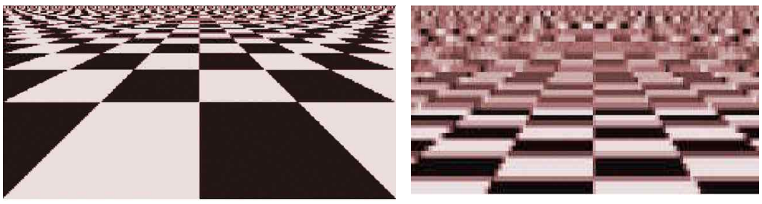
수퍼 샘플링 VS 영역 샘플링 ⇒ 차이점 설명

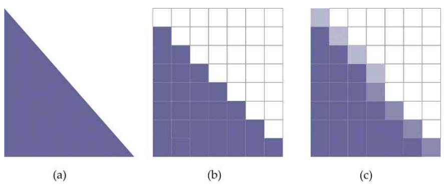


→ 원래 그려야할 해상도 4x4

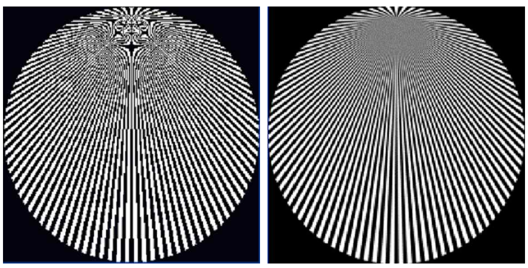
원하는 위치에

8x8로 나눈후 점을 찍고 다시 픽셀 4개를 하나로 묶어 이 픽셀안에 찍힌 점의 수가 많을수록 진하게 칠함





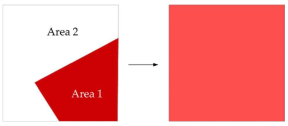
- 포인트 샘플링, 지터링에 의한 수퍼 샘플링

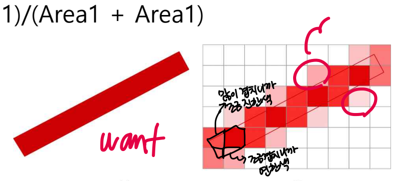


영역 샘플링(Area-Sampling)

↳계산량↑ 시간↑

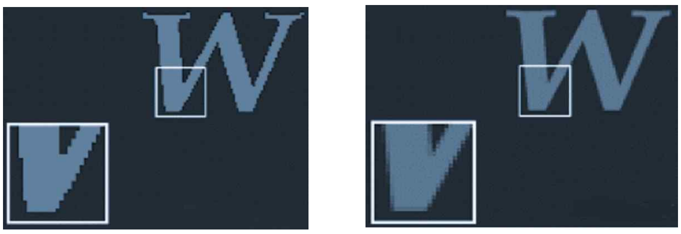
- 면적에 비례. 사전 필터링
  - (백색 × Area2 + 적색 × Area1)/(Area1 + Area1)





(a) (b)

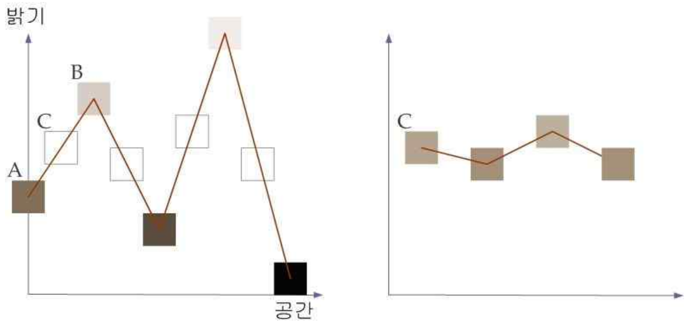
- 포인트 샘플링, 영역 샘플링



73

영상 필터링

- 화소그룹 처리(Pixel Group Processing)
  - 어떤 화소의 색에 인접화소의 색이 영향을 주는 것.
  - Ex. 저역통과 필터(LPF: Low-Pass Filter) 또는 블러링(Blurring)



(a) (b)

74

컨볼루션 마스크(Convolution Mask)

- Blurring

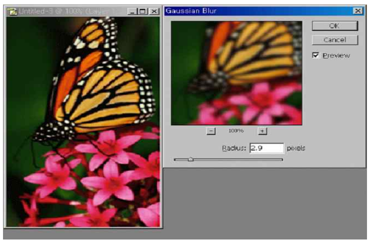
1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

(a)

117	117	27	27
117	117	27	27
117	117	27	27

87	57

(b)



- Sharpening

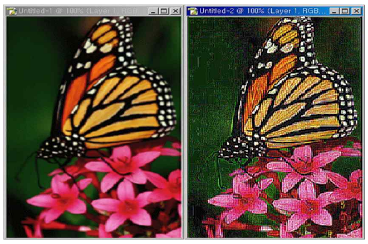
-1	-1	-1
-1	9	-1
-1	-1	-1

(a)

117	117	51	27
117	117	51	27
117	117	51	27

317	-45

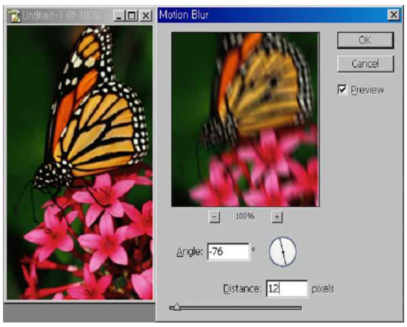
(b)



75

모션 블러(Motion Blur)

- 컨볼루션 마스크가 중앙 화소를 중심으로 방향성을 지님
  - 물체가 움직이는 방향에 있는 화소들에 대해서만 가중치를 적용



76

블러링에 의한 앤티-에일리어싱

- 수퍼 샘플링에 비해 고속처리
  - 수퍼 샘플링은 원래 화면 해상도 보다 훨씬 많은 샘플링을 요구
  - 블러링은 해상도를 그대로 둔 채 인접 화소 정보 만을 이용
- 블러링은 수퍼샘플링에 비해 실질적 해상도 저하

77

9장 연습문제

78

감사합니다!

79