Computer Graphics

Prof. Jibum Kim

Department of Computer Science & Engineering

Incheon National University

- **Polygon rasterziation**

- One of the major advantages that the first raster systems brought to users was the ability to display **filled polygons**

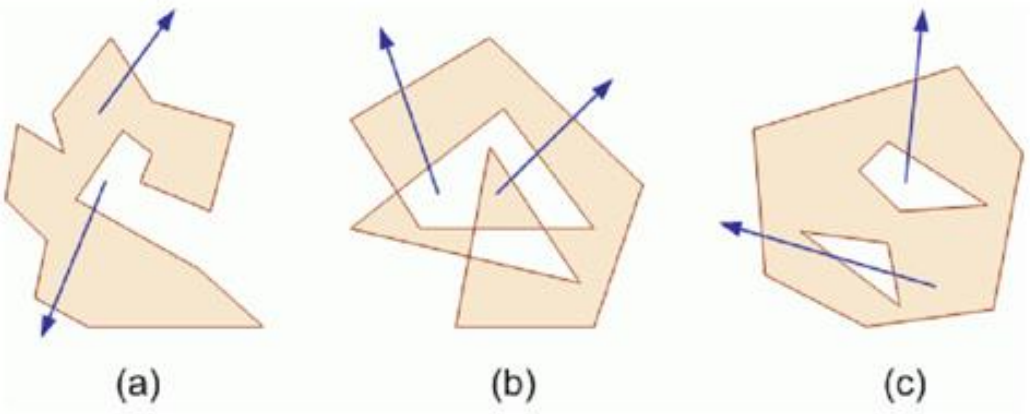
- Unlike rasterization of lines, where a single algoirithm dominates, there are many viable methods for rasterizating polygons

- The choice depends heavily on the implementation architecture

- **Inside-outside testing**
- **(odd-even test)**

- **The process of filling the inside of a polygon with a color is equivalent to deciding which points in the plane of polygon are interior (inside) points**
- **The odd-even test is the most widely used test for making inside-outside decisions**

- Suppose that p is a pont inside a polygon. Any ray emanating from p and going off to infinity must cross an odd number of edges
- Any ray emanating from a point outside the polygon and entering the polygon crosses an even number of edges before reaching infinity
- Hence, a point can be defined as being inside if after drawing a line through it and following this line, starting on the outside, we cross an odd number of edges before reaching it

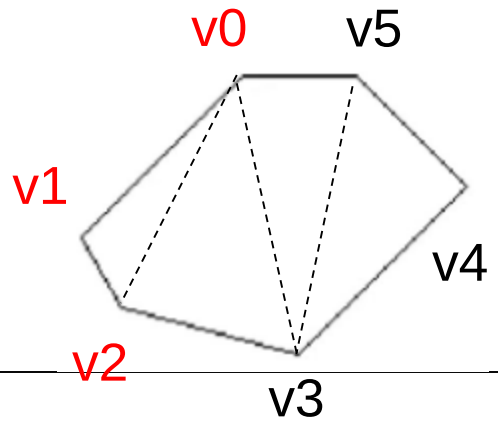


- For the star-shaped polygon in this figure, we obtain the inside coloring shown (특이, singularity case?)
- Odd even testing is easy to implement and integrates well with the standard rending algorithms
- Usually, we replace rays through points with scan lines and we count the crossing of polygon edges to determine inside and outside
- Scanline: one row of pixels

- **Tessellation (triangulation) an arbitrary simple polygon**

- Certain architecture renders only triangles (e.g., webGL) that are flat and convex, we have the problem with more general polygons
- One approach is to work with the application to ensure that they only generate triangles
- Another is to provide software that can tessellate a given polygon into triangles
- A good tessellation should not produce triangles that are long and thin

- **Simple triangulation algorithm for simple polygon with n vertices**
- **Assume our polygon is specified by an ordered list of vertices v_0, v_1,…,v_{n-1}**
- **Thus, there is an edge from v_0 to v_1, from v_1 to v_2, and finally from v_{n-1} to v_0**
- **The first step is to find the leftmost vertex, v_i, a calculation that requires a simple scan of the x components of the vertices**
- **Let v_{i-1} and v_{i+1} be the two neighbors of v_i. These three vertices form the triangle v_{i-1}, v_i, v_{i+1}**
- **We can proceed recursively by removing v_i from the original list, and have a triangle and polygon with n-1 vertices**

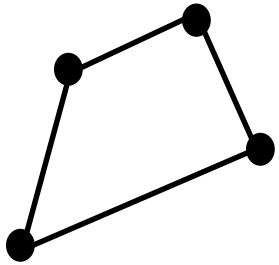


- **Flood-fill algorithm**

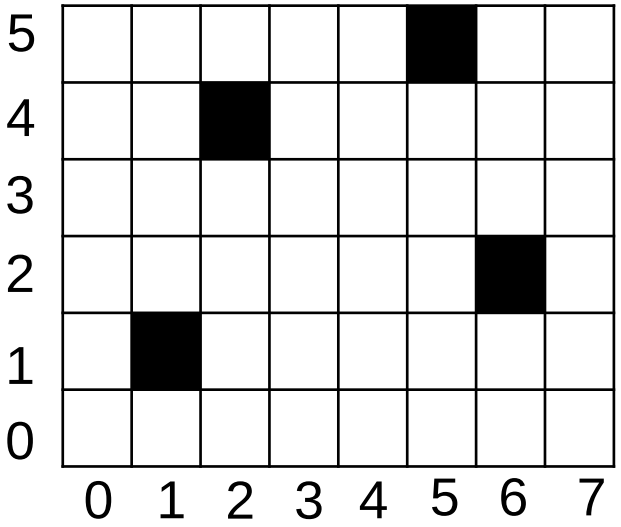
- Another approach to rasterizing the polygon P is to fill first its boundary by repeatedly applying Bresenham's line rasterizer to each edge and then fill its interior

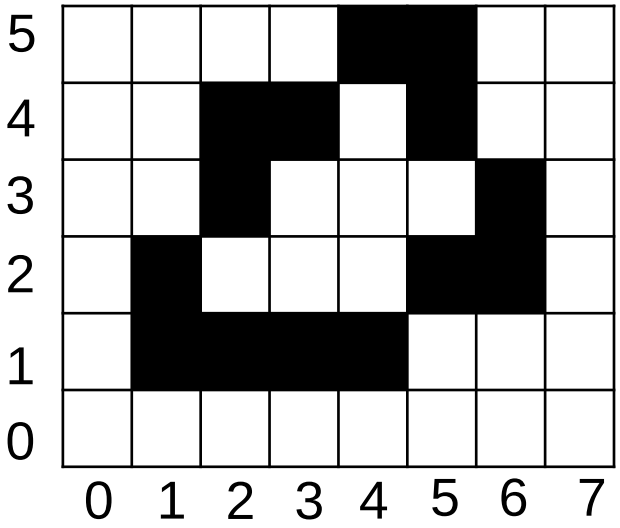
- Once the boundary of P has been fill, a flood-fill algorithm fills its interior

- **1. Rasterize edges first, Line rasterization (예: 브레스넘 알고리즘)**

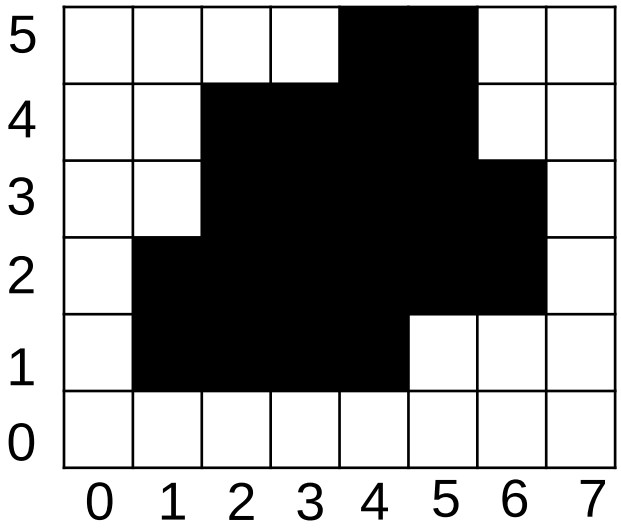


Polygon





- **2. Polygon filling**



- Start with a pixel p known to be in P's interior, found by performing the parity test. Fill P
- Examine four of p's neighboring pixels, the ones to its N, S, E, W.
- Of these pixels (4-adjacent to p), fill those that don't belong to the boundary and have not yet been filled
- Next, examine the pixels 4-adjacent to the ones just filled and, again, fill those don't belong to the boundary and have not yet been filled
- Continue until no more pixels can be filled

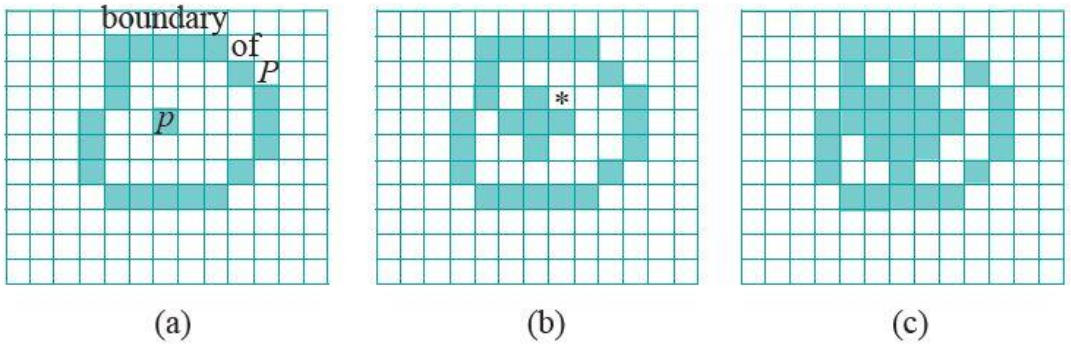
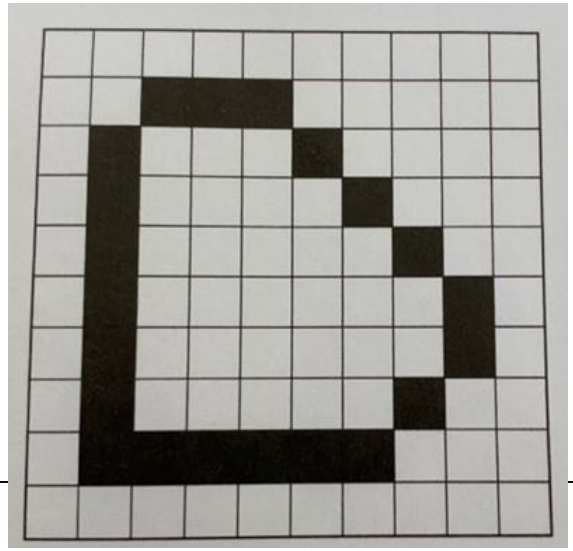


Figure 14.26: Flood-fill: (a) Initially (b) Fill pixels 4-adjacent to p (c) Fill pixels 4-adjacent to the ones filled in the previous step. The starred pixel of (b) is examined by both its south and west neighbors at this step.

- Suppose we have only two colors: background (white) and a foreground (drawing, black)
- We can use the foreground color to rasterize the edges
- If we can find an initial point (x, y) inside the polygon – a seed point – then we can look at its neighbors recursively, coloring them with the foreground color if they are not edge points
- The folldfill algorithm can be expressed in pseudo code, assuming that there is a function "read_pixel" that returns the color of a pixel

```
function floodFill(x, y)  {
   if (readPixel (x, y) == WHITE)
  {
     writePixel(x, y, BLACK);
     floodFill(x-1, y);
     floodFill(x+1, y);
     floodFill(x, y-1);
     floodFill(x, y+1);
   }   }
```



- OpenGL에는 frame buffer에서 특정 pixel에 대한 정보 (예: color)를 읽을 수 있는 **'glReadPixels'** 함수가 있음
- glReadPixels - OpenGL 4 Reference Pages (khronos.org)
- 아래 사용 예
- x, y: 읽을 pixel의 위치 (window coordinates로)
- 읽을 pixel block 크기 (가로, 세로): 하나만 읽으면 모두 1로 줌
- color buffer (GL_RGB), 자료형 (GL_FLOAT)

```
struct Color {
    GLfloat r;
    GLfloat g;
    GLfloat b;
};
```

```
Color getPixelColor(GLint x, GLint y) {
    Color color;
    glReadPixels(x, y, 1, 1, GL_RGB, GL_FLOAT, &color);
    return color;
}
```

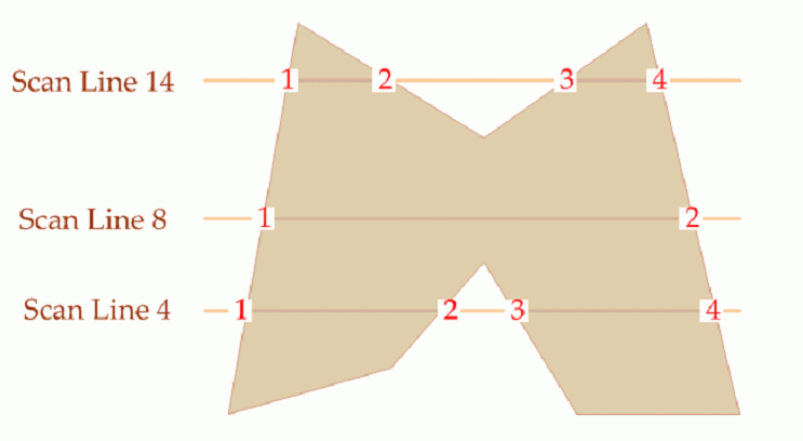
- 특정 픽셀의 색을 정할때는 OpenGL에서 ‘GL_POINTS’를 사용하고 ‘glColor3f’로 그 픽셀의 색을 지정할 수 있음
- 아래 예

```
void setPixelColor(GLint x, GLint y, Color color) {
    glColor3f(color.r, color.g, color.b);
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
    glFlush();
}
```

- Flood fill 알고리즘 코드 예
- 실행후 마우크 왼쪽 클릭
- https://www.dropbox.com/s/uxdxvlgo96sbroj/floodfill.txt?dl=0

- **Scan line fill algorithm**

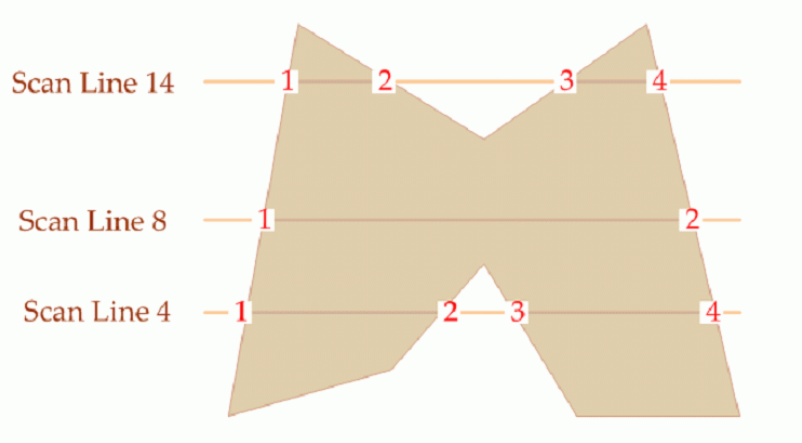
- polygon 내부의 색을 칠하는 또다른 방법은 주사선 채움 알고리즘 (scan line fill algorithm)을 이용하는 것이다
- Scanline: one row of pixels
- Scan line filling 알고리즘은 polygon을 rasterization 하는 방법으로 앞에서 배운 odd-even test를 사용 한다
- 화면 아래부터 row-by-row로 주사선 (scan line)과 다각형의 edge들과의 교차점을 계산한다. 각 주사선에서 홀수 번째의 교차점부터 짝수 째의 교차점 직전 구간에 있는 화소들을 모두 칠한다. 이를 통해 다각형 내부만 골라서 칠함



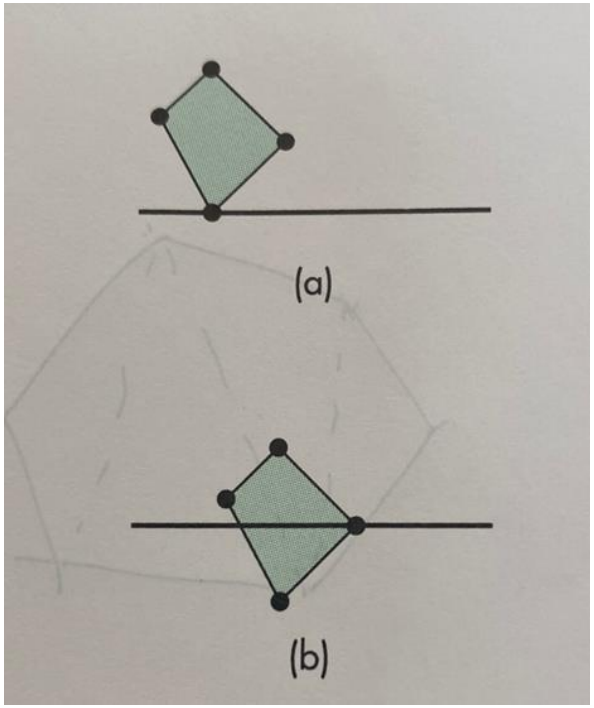
- **Scanline fill algorithm의 Pseudo-code**

```
for (each scanline L)
{
   Find intersections of L with all edges of P
   Sort the intersections by increasing x-value
   Fill pixels runs between all pairs of intersections
}
```

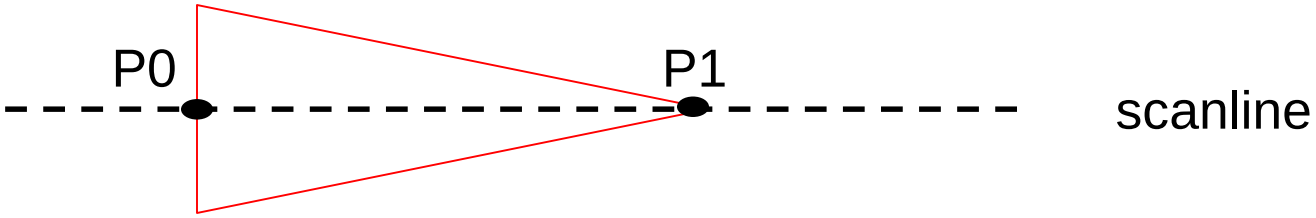
- **예: scanline 4는 polygon의 edge와 4번 intersection 생김**
- **교차점을 왼쪽부터 1, 2, 3, 4라고 하면 1-2 칠함, 3-4 칠함**



- **Singularity problems (특이 case)**
- **Special case of a vertex lying on an edge**
- **If we are using an odd-even test, we have to treat these two cases differently**
- **(a) 아래 교차점에서 scanline과 두번 교차했다고 하면 어떤 문제?**
- **(b) 오른쪽 교차점에서 scanline과 세번 교차했다고 하면 어떤 문제?**



- **Scanline fill 알고리즘의 예외 처리 (singularity 문제)**
- **어떤 scanline이 동시에 polygon의 여러 edge와 교차할 수 있다. P1은 scanline과 2번 교차한다**
- **만일 2번 교차한 걸로 하면 어떤 문제가 생기나?**



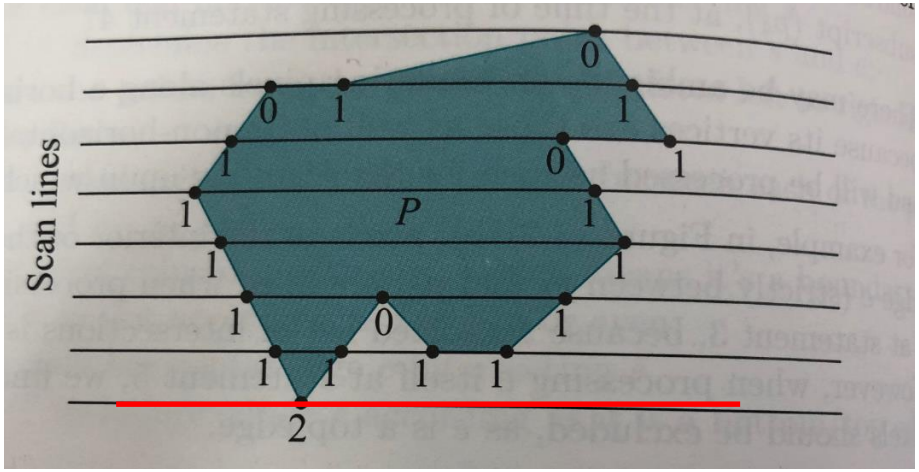
- **1번교차 – 2번 교차까지 칠해짐 , 3번 교차부터 칠해짐 -**

- **Scanline filling 알고리즘의 예외 처리**
- **앞의 scanline filling 알고리즘의 수정 (추가)**

Find the intersections of the scanline with all edges of P
Discard intersections with horizontal edges and with the upper endpoint of any edge

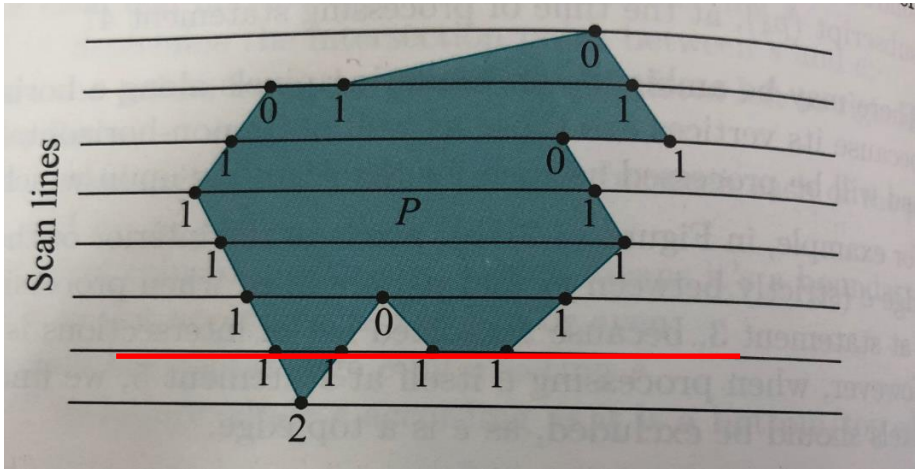
```
for (each scanline L)
{
   Find intersections of L with all edges of P
   Discard intersections with horizontal edges and with the  upper endpoint of any edge
   Sort the intersections by increasing x-value
   Fill pixels runs between all pairs of intersections
}
```

- 예: scanline과 polygon의 edge와의 교차 횟수 (교차점 개수) 표시



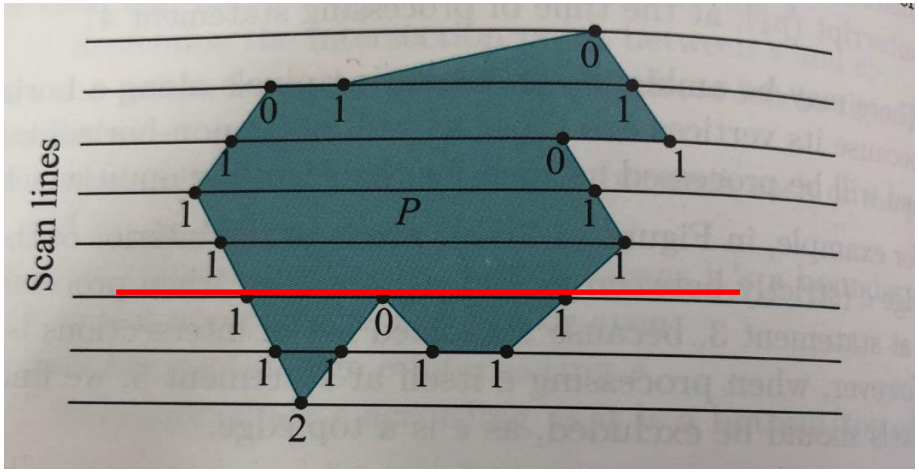
- 1. 제일 아래 scanline: polygon edge와 2번 교차, 교차점 p1
- 즉 p1 하나 칠해짐, output: {P1}

- 예: scanline과 polygon의 edge와의 교차 횟수 (교차점 개수) 표시



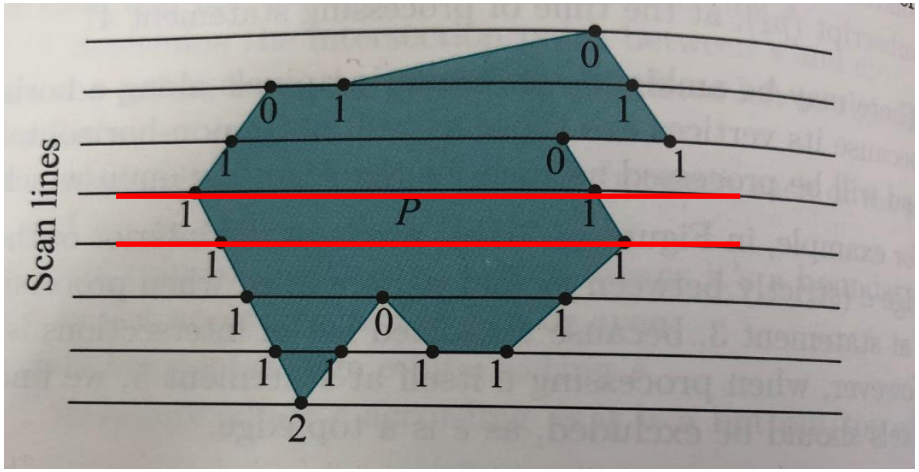
- 2. 2번째 scanline과의 교차점을 왼쪽 부터 p1,p2,p3,p4
- output: {p1, p2, p3, p4}
- p1-p2까지 칠해짐, p3-p4까지 칠해짐

- 예: scanline과 polygon의 edge와의 교차 횟수 (교차점 개수) 표시



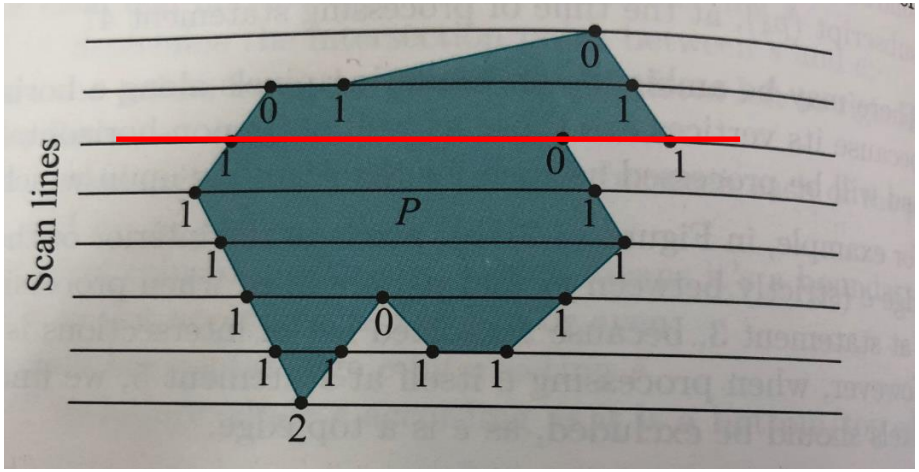
- 3. 3번째 scanline과의 교차점을 왼쪽 부터 p1,p2,p3
- p2는 0 intersections (why?), output{p1, p3}
- p1부터 p3까지 칠해짐

- 예: scanline과 polygon의 edge와의 교차 횟수 (교차점 개수) 표시



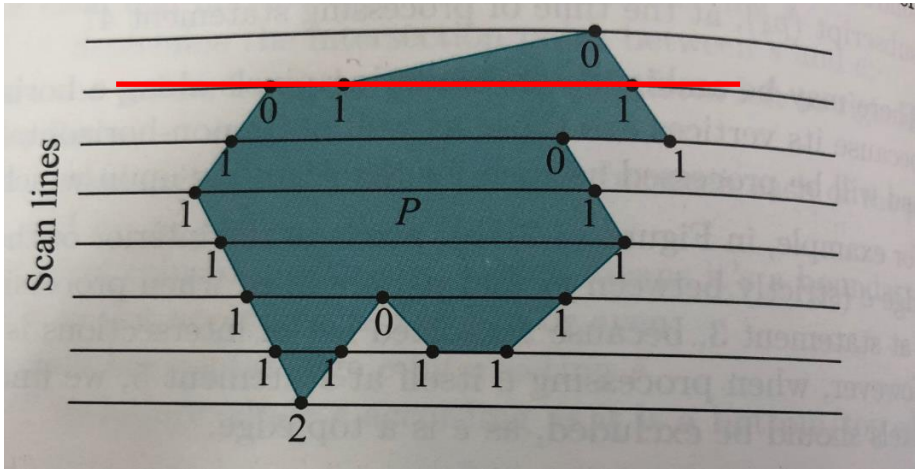
- 4. 4번째 5번째 scanline과의 교차점을 왼쪽 부터 p1, p2
- output: {p1, p2}
- P1부터 p2까지 칠해짐

- 예: scanline과 polygon의 edge와의 교차 횟수 (교차점 개수) 표시



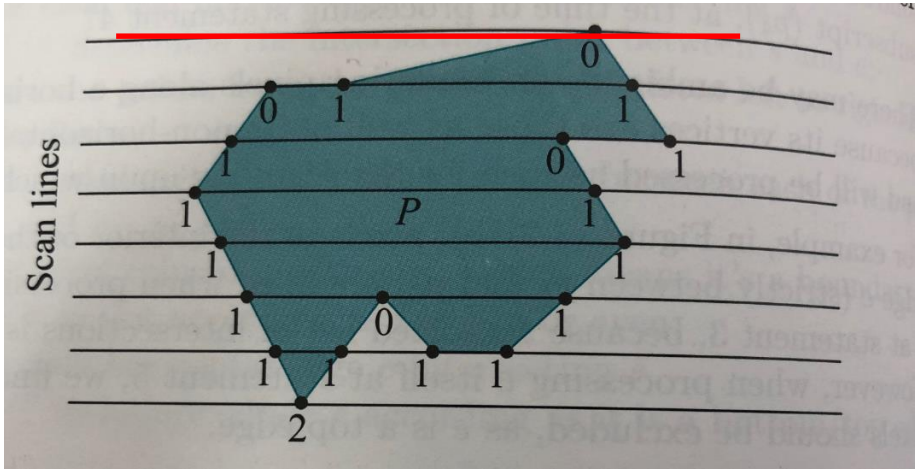
- 5. 6번째 scanline과의 교차점을 왼쪽 부터 p1,p2,p3
- P2는 0 intersections, output: {P1, P3}
- P1부터 P3까지 칠해짐

- 예: scanline과 polygon의 edge와의 교차 횟수 (교차점 개수) 표시



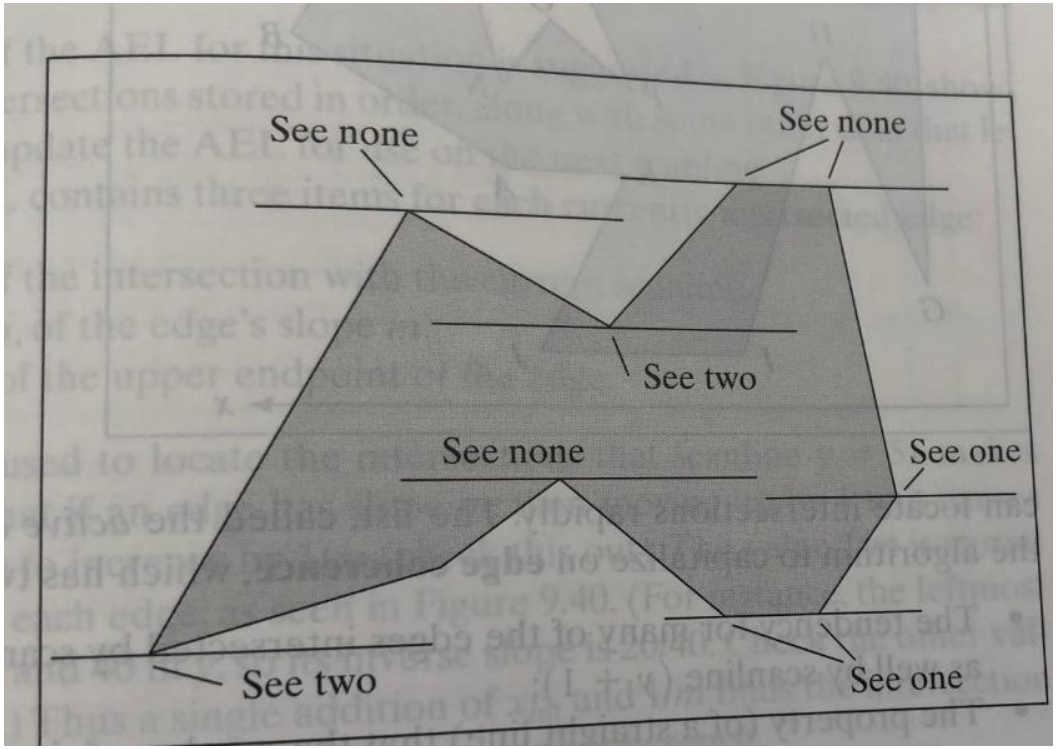
- 6. 7번째 scanline과의 교차점을 왼쪽 부터 p1,p2,p3
- P1은 0 intersections, output: {p2, p3}
- P2부터 P3까지 칠해짐

- 예: scanline과 polygon의 edge와의 교차 횟수 (교차점 개수) 표시



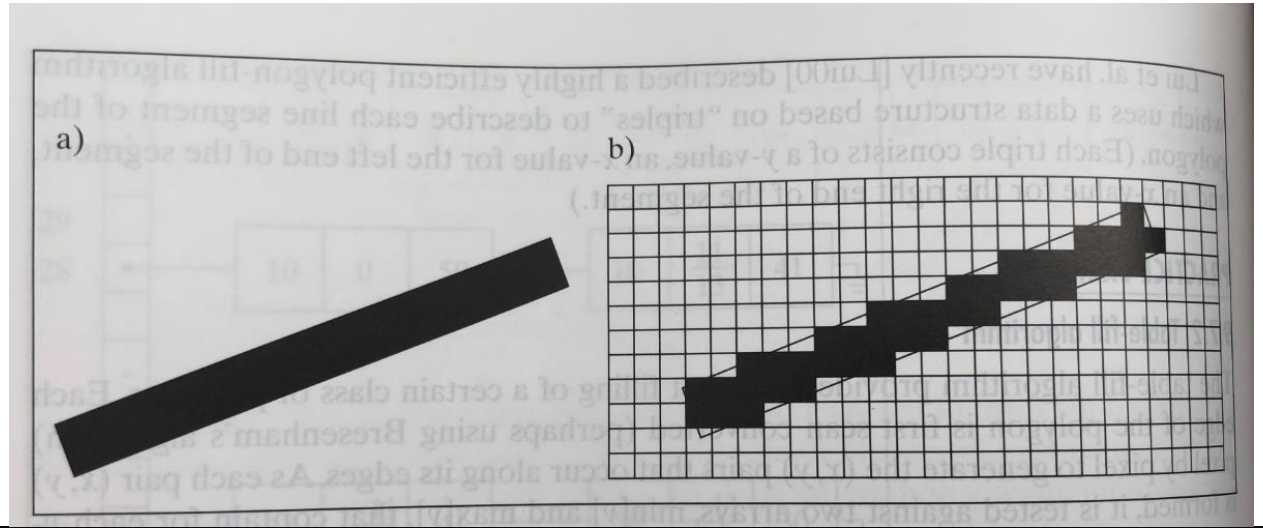
- 7. 8번째 scanline과의 교차점을 왼쪽 부터 p1
- P1은 0 intersections, output: none
- 칠해지지 않음

- 다음은 polygon의 끝점들에서 scanline과 polygon의 edge들과의 교차 횟수를 표시한 것이다. 맞는지 확인해 보자

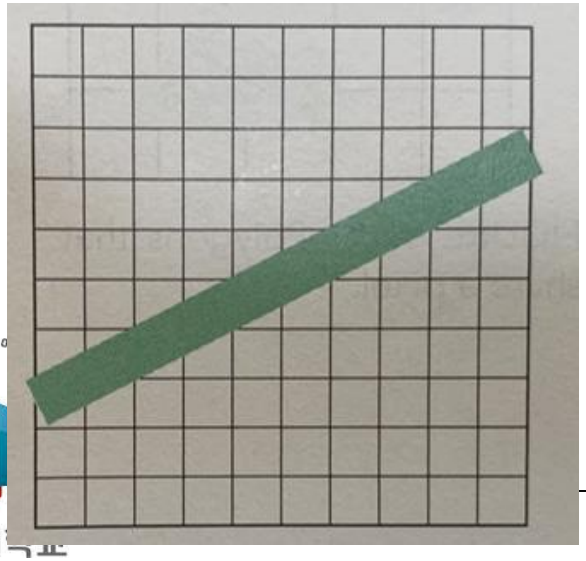


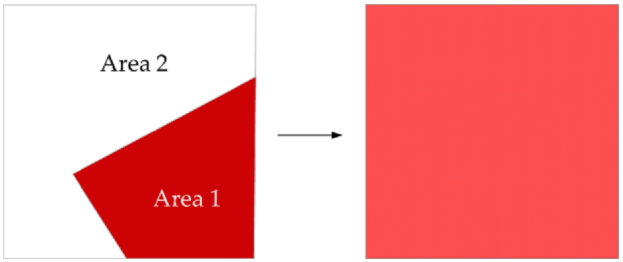
- **Anti-aliasing**

- Rasterized line segments and edges of polygons look jagged
- This type of error arises whenever we attempt to go from the continuous representation of an object, which has infinite resolution, to a sampled approximation, which has limited resolution
- The name aliasing has been given to this effect

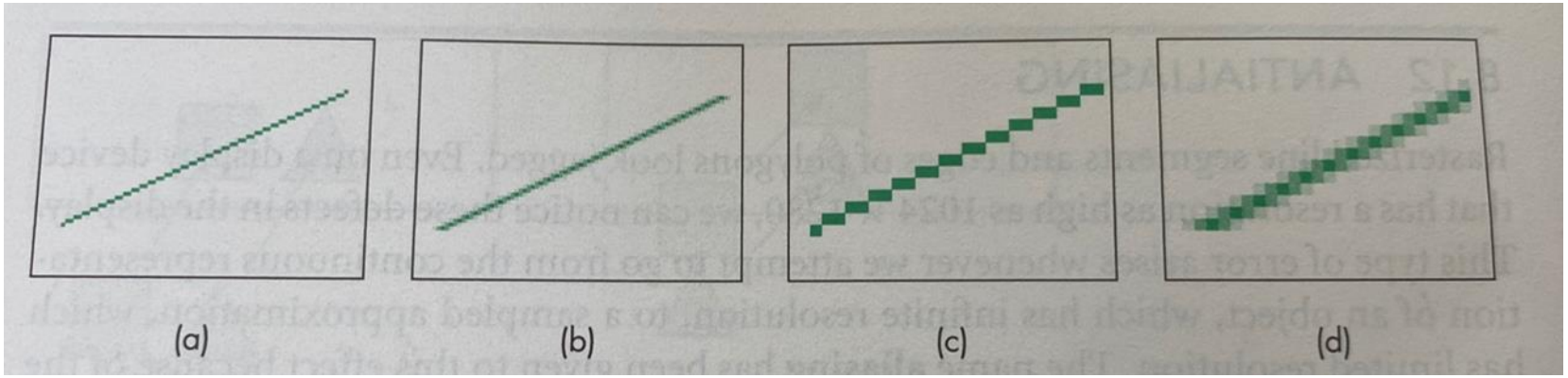


- **1. Anti-aliasing by area averaging**
- **See the below ideal raster line with one pixel wide**
- **We shade each pixel by the percentage of the ideal line that crosses it, we get the smoother-appearing image**
- **This technique is known as antialiasing by area average**
- **예: Pixel 색 = (백색 × Area2 + 적색 × Area1)/(Area2 + Area1)**

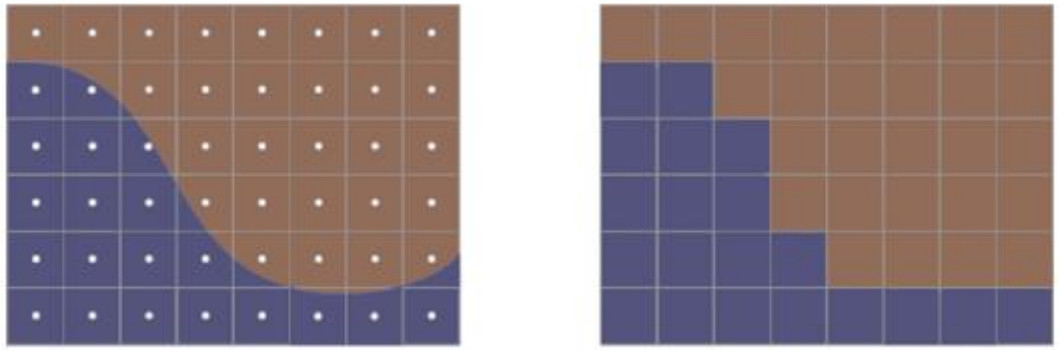




- Aliased versus antialiased line segments
- (a) aliased line segment (b) antialaised line segment
- (c) (a)번 확대 (d) (b)번 확대

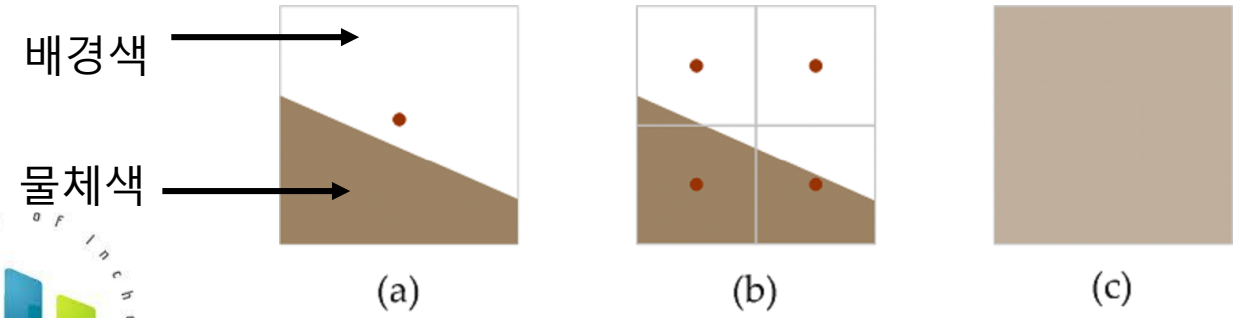


- **Point sampling and aliasing**
- **컴퓨터 그래픽스에서는 공간적 aliasing이 문제가 된다 이는 무한히 섬세한 모양을 가진 사물을 일정 pixel 단위로 화면에 나타내야 하기 때문에 발생한다**
- **예를 들어 아래와 같이 pixel의 정중앙 또는 임의의 위치에서, point sampling을 수행한 후에, 그 point의 색을 해당 pixel의 색으로 취하였다고 하자**
- **(a) Point sampling 전 (b) Point sampling 후. Aliasing 생김**

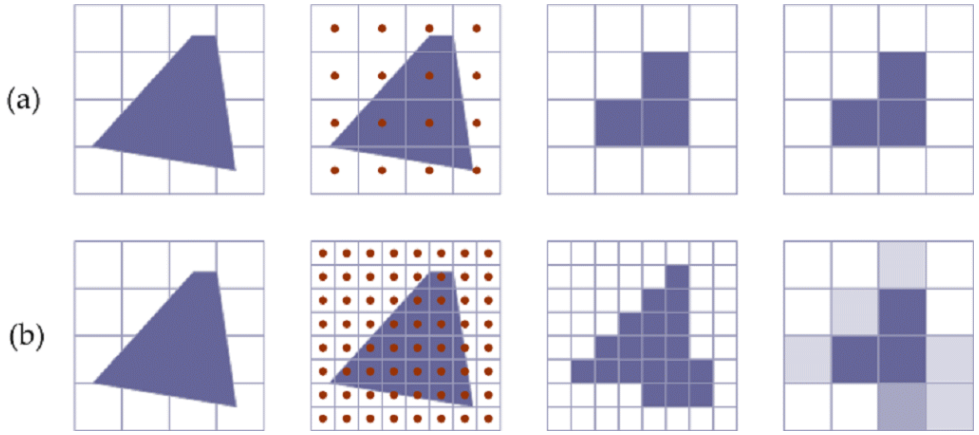


(a) (b)

- **2. Anti-aliasing by super sampling**
- **Pixel을 더 작은 단위로 분할하여 부분 화소 (subpixel)로 분할하여 화소 밝기를 계산, 최종적으로 이를 평균하여 하나의 화소 단위로 뿌림**
- **Super 샘플링시 : 화소색 = (물체색x2/4+ 배경색x2/4)**



- 예: point 샘플링과 2x2 super 샘플링 비교
- 경계 부분에 물체색과 배경색의 중간 밝기를 지닌 화소가 생긴다



Point sampling

Super sampling

- 계단 모양의 외곽선이 Super 샘플링 후에 많이 완화되어 보인다

- **OpenGL에서의 anti-aliasing**

- Points, lines, polygons에 대해서 각각 anti-aliasing 적용 가능

```
glEnable(GL_POINT_SMOOTH);
glEnable(GL_LINE_SMOOTH);
glEnable(GL_POLYGON_SMOOTH);
```

- glHint(); 함수
- OpenGL에 힌트를 전달 G_LICEST라면 가장 질이 좋은 anti-aliasing 사용하라는 의미

- Anti-aliasing 시에 색을 혼합하기 위하여 blending 사용
- glEnable(GL_BLEND)

- 예:
- https://www.dropbox.com/s/38fkvq9fk5twtts/antialiasing.txt?dl=0
- Space key: anti-aliasing on-off
- Arrow key: 회전
- Page up/down key: line 두께 조절