Polygon Filling

Graphics 1 CMP-5010B

Dr. David Greenwood

david.greenwood@uea.ac.uk

SCI 2.16a University of East Anglia

Spring 2022

Content

- Polygon Filling
- Scan Line Algorithm
- Boundary Fill Algorithm

Polygon Filling

Identify pixels that belong to the *interior* of a polygon. Once identified, we can:

- pass the pixel to the rasteriser
- assign colour to the pixel
- assign a depth value to the pixel
- sample a texture for the pixel

Polygon Filling

- A polygon is a set of vertices that are connected by *edges*.
- We need *efficient* algorithms to fill polygons.
- We can extend ideas from line drawing to polygon filling.
- Not all polygons are handled equally!

Convex Polygons

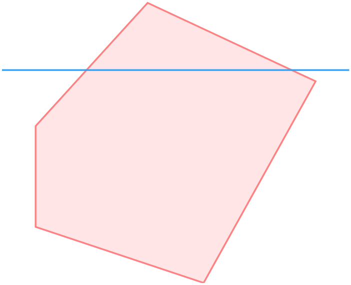

Figure 1: convex polygon

- interior angles $\leq 180^\circ$
- scan lines enter the interior once and exit once
- triangles are always convex

Concave Polygons

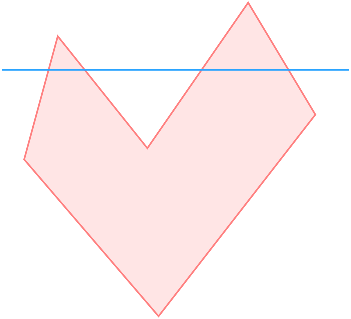

Figure 2: concave polygon

- arbitrarily complex polygons
- scan lines enter and exit many times
- more difficult to fill

Scan-Line Algorithm

The scan-line algorithm must work for **both** convex and concave polygons.

Scan-Line Algorithm

```
for line in y=0 to y=height:
    counter = 0
    for pixel in x=0 to x=width:
        if edge:
            counter +=1
        if counter is odd:
            draw(line, pixel)
```

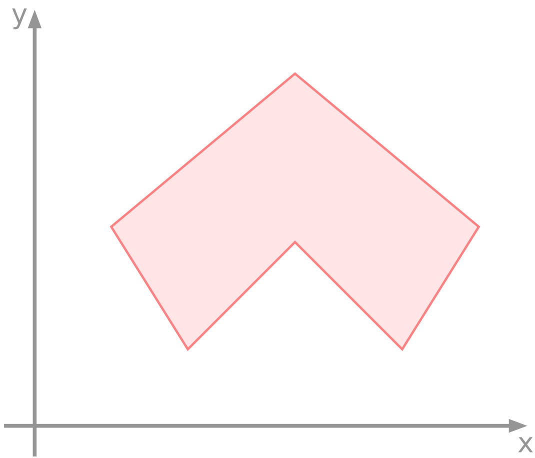

Figure 3: concave polygon

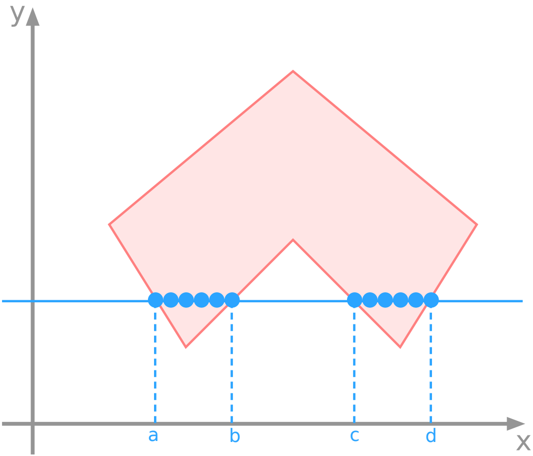

Figure 4: concave scan

Scan-Line Algorithm

The algorithm seems to work well.

- Have we considered all cases?

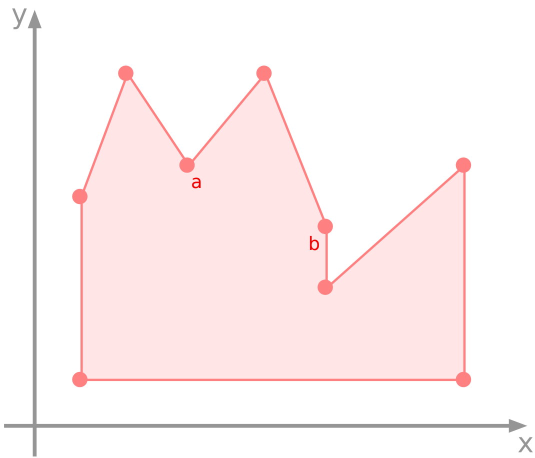

Figure 5: complex polygon

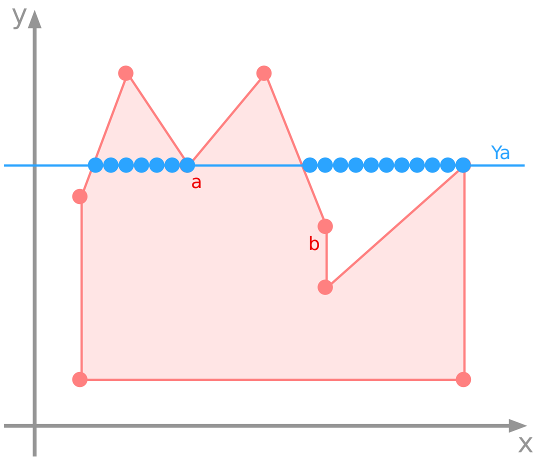

Figure 6: scanning problem

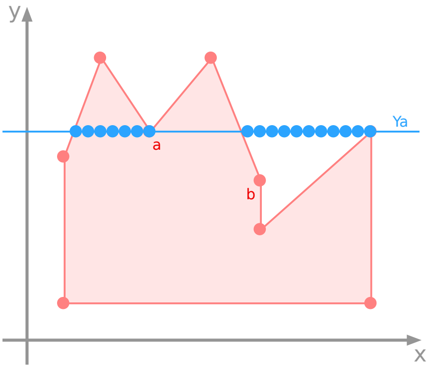

Figure 7: naive algorithm wrongly fills the cavity

- Enter the left edge, increment the counter and draw.
- Pass through vertex *a*, increment the counter and stop drawing.
- Leave the right edge, increment the counter and draw.

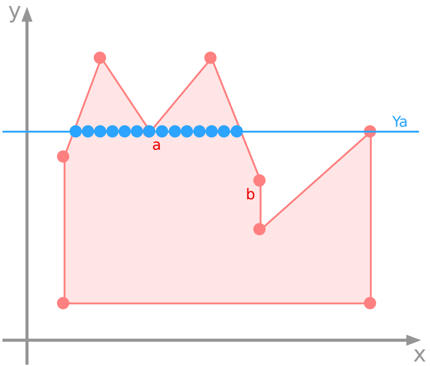

Figure 8: Counting vertex *a* twice provides a solution.

Solution:

– count the vertex *twice*

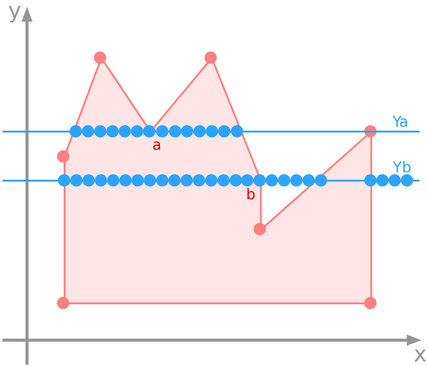

Figure 9: Counting vertices twice does not always work.

Problem:

- Counting the vertex twice does not always work!

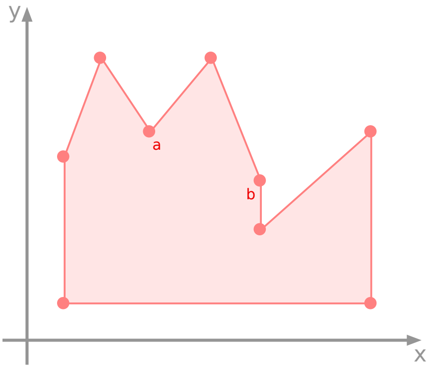

Figure 10: Difference between vertex *a* and *b*.

- consider the *edges* at each vertex
- edges through vertex *b* are **monotonic** in *y*

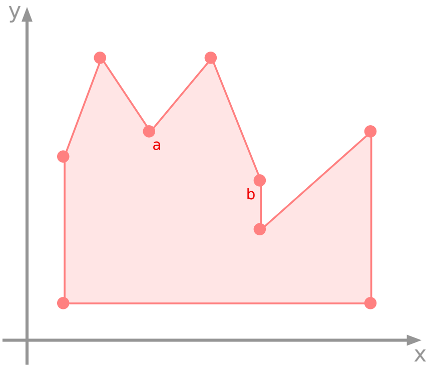

Figure 11: Difference between vertex *a* and *b*.

If we move around the polygon in a clockwise direction:

- edges that enter and leave vertex **a** go in **opposite** y directions.
- edges that enter and leave vertex **b** go in the **same** y direction.
- edges through vertex **b** are **monotonic** in y

We can *split* the vertex for *monotonic* edges:

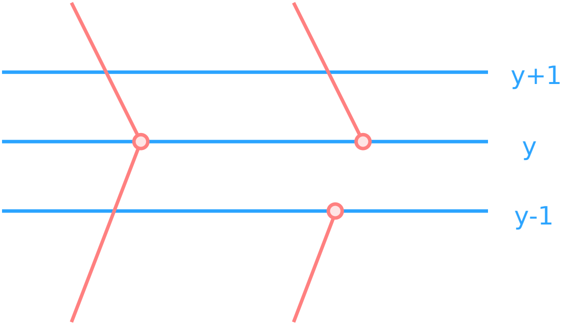

Figure 12: split vertex

The **lower** edge is shortened to create two *new* edge points.

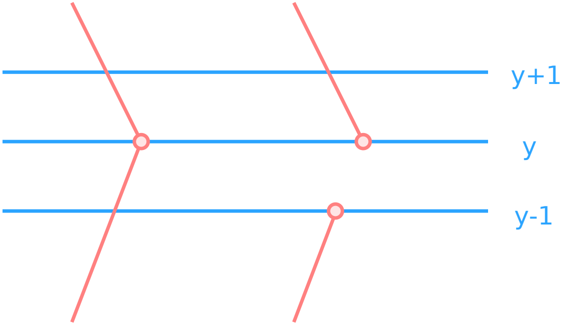

Figure 13: split vertex

Scan-Line Algorithm

```
process vertices of monotonic edges

for line in y=0 to y=height:
    counter = 0
    for pixel in x=0 to x=width:
        if edge or edge-point:
            counter +=1
        if vertex:
            counter +=2
        if counter is odd:
            draw(line, pixel)
```

Scan-Line Implementation

Expanding the pseudocode.

Scan-Line Implementation

The first step is to build an array of *linked lists*, called a Bucket Sorted Edge Table (**BSET**).

Scan-Line Implementation

Each **node** in the *linked list* has 3 members related to a vertex, and a pointer to the next node:

- y value of the *other* vertex on the edge
- x value of *this* vertex
- inverse slope of the edge
- pointer to the next node

Scan-Line Implementation

To determine edge intersections it uses the familiar slope of a line:

$$m = \frac{y_{k+1} - y_k}{x_{k+1} - x_k} = \frac{1}{x_{k+1} - x_k} \Rightarrow x_{k+1} = x_k + \frac{1}{m}$$

Scan-Line Implementation

Before we start to build the Bucket Sorted Edge Table (**BSET**), we *split* any vertices on **monotonic** edges.

- The BSET is built from the vertex with the lowest y value to the vertex with the highest y value.
- If the vertex is part of two edges, the first node is for the left edge, and the second node is for the right edge.
- Split vertices have only one edge, so only one node is entered.

BSET Example

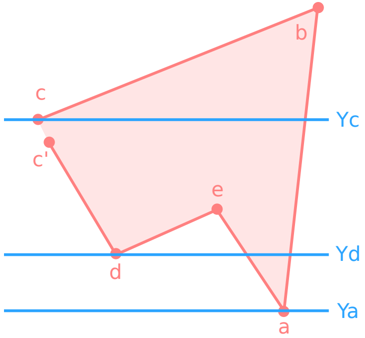

Figure 14: polygon scan

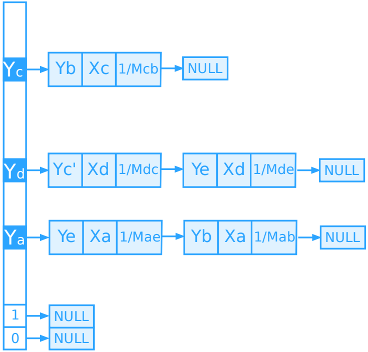

Figure 15: BSET

Scan-Line Run Time

The BSET is an initialisation step.

- It is created once.

At runtime, we use another data structure:

- Active Linked List (**ALL**).

Active Linked List

- Initially, the ALL points to NULL.
- Search the BSET for the first non NULL entry.
- Set the ALL to the first non NULL entry.

Active Linked List

For our example, the ALL is first set to y_a.

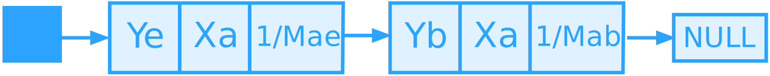

Figure 16: Active List

The `draw` function will now draw from x_a to x_a, that is, just a single point.

Active Linked List

Next, the scan line moves up to $Y_a + 1$.

- There is **no** entry in the BSET for this y value.
- Therefore the current ALL has the x values updated:

$$x_a^{'} = x_a + \frac{1}{m_{ae}}, \quad x_a^{''} = x_a + \frac{1}{m_{ab}}$$

Active Linked List

Now, we have a new ALL:

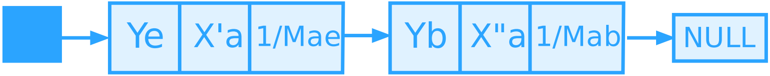

Figure 17: Updated Active List

- The `draw` function will now draw from x_a' to x_a''.
- The x values are updated for each line
- until a new BSET entry is found.
- In our example, when we reach y_d.

Active Linked List

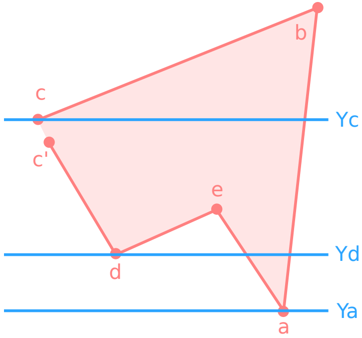

Figure 18: y_d scan

- Scan line is now at y_d.
- Fetch BSET entry for y_d.
- merge with the ALL in increasing order of x values.

Active Linked List

Now, we have the ALL:

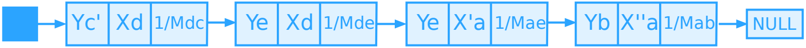

Figure 19: y_d Active List

- We draw from x_d to x_d **and** $x_a^{'}$ to $x_a^{''}$.
- All x values are then updated for each line with the inverse slope.

Active Linked List

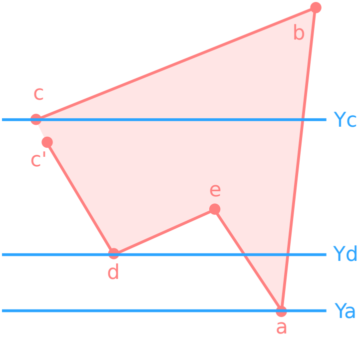

Figure 20: y_e scan

What happens at y_e?

Active Linked List

We monitor the maximum y value of the nodes in the ALL.

- when we exceed any maximum y value, we remove those nodes from the ALL.

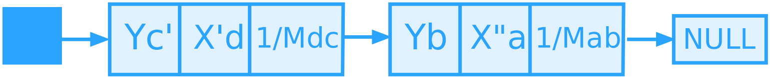

Figure 21: remove y_e entries

Active Linked List

In our example, we have one more fetch from the BSET.

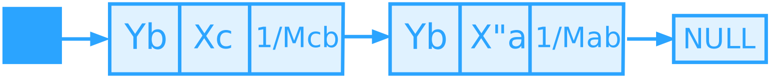

Figure 22: y_c

We have merged the y_c entry and removed the $y_{c'}$ nodes.

Scan-Line Implementation

We observe that splitting the c vertex automatically avoids double drawing of monotonic vertices.

Boundary Fill

Another popular method for filling polygons.

Boundary Fill

idea:

- find the edges of the polygon.
- initialise a seed pixel
- from the seed, recursively colour the neighbours.
- stop when polygon is filled.

Connectivity

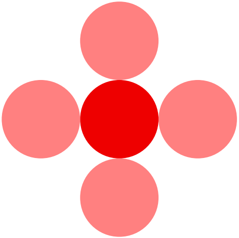

Figure 23: 4 connectivity

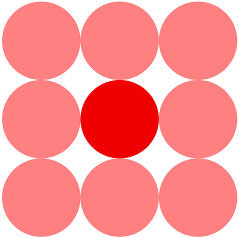

Figure 24: 8 connectivity

Four Connectivity

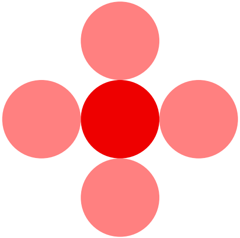

Figure 25: 4 connectivity

Four connectivity requires fewer recursive calls, but more steps to complete.

Four Connectivity

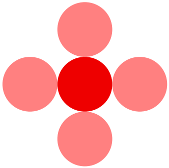

Figure 26: 4 connectivity

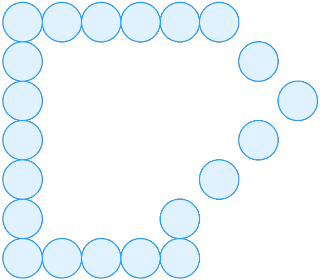

Figure 27: 4 fill

Eight Connectivity

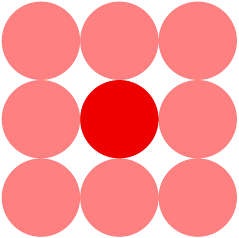

Figure 28: 8 connectivity

Eight connectivity usually completes a fill with fewer steps.

Eight Connectivity

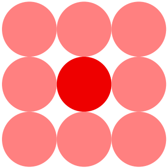

Figure 29: 8 connectivity

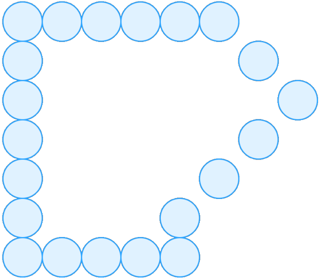

Figure 30: 8 fill

Eight Connectivity

Eight connectivity fills thin bridges more reliably.

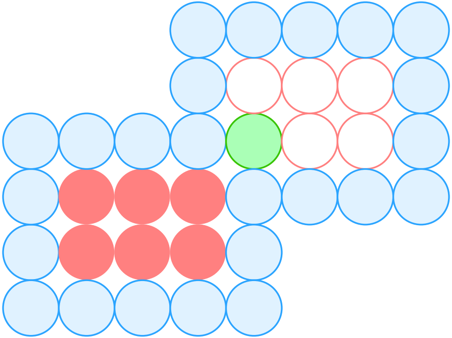

Figure 31: filling thin bridges

Boundary Fill

```
func fill4(x, y, fill_colour, edge_colour):
    if pixel(x, y) == edge_colour:
        return
    if pixel(x, y) == fill_colour:
        return

    draw(x, y, fill_colour)

    fill4(x+1, y, fill_colour, edge_colour)
    fill4(x-1, y, fill_colour, edge_colour)
    fill4(x, y+1, fill_colour, edge_colour)
    fill4(x, y-1, fill_colour, edge_colour)
```

Boundary Fill

Some caveats:

- recursive algorithm - so not memory efficient.
- leaks due to unclosed boundary
- premature stop if interior pixel is already fill colour.

Summary

- Polygon Filling
- Scan Line Algorithm
- Boundary Fill Algorithm

Reading:

- Hearn & Baker, *Computer Graphics with OpenGL*, 4th Edition, Chapter 4.10