Exercise 3

for the lecture

Computational Geometry

Dominik Bendle, Stefan Fritsch, Marcel Rogge and Matthias Tschöpe

December 10, 2018

Exercise 1 (Doubly-Connected Edge List) (2 + 1 + 1 points)

a) Which of the following equations are always true?.

$$\text{Twin}(\text{Twin}(\vec{e})) = \vec{e} \tag{1}$$

$$\text{Next}(\text{Prev}(\vec{e})) = \vec{e} \tag{2}$$

$$\text{Twin}(\text{Prev}(\text{Twin}(\vec{e}))) = \text{Next}(\vec{e}) \tag{3}$$

$$\text{IncidentFace}(\vec{e}) = \text{IncidentFace}(\text{Next}(\vec{e})) \tag{4}$$

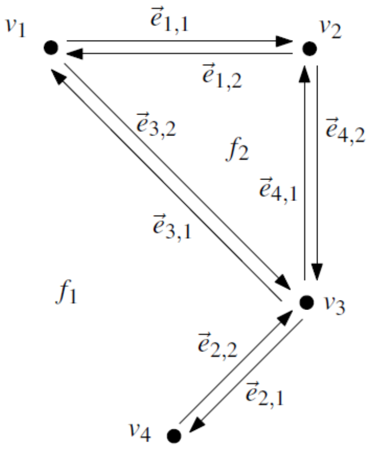
(1) True. See definition of Twin.

(2) True. See definitions of Prev and Next.

(3) False. Consider figure 1 as counterexample, with $\vec{e} := \vec{e}_{2,2}$.

$$\begin{aligned}
&\text{Twin}(\text{Prev}(\text{Twin}(\vec{e}))) \\
=&\text{Twin}(\text{Prev}(\text{Twin}(\vec{e}_{2,2}))) \\
=&\text{Twin}(\text{Prev}(\vec{e}_{2,1})) \\
=&\text{Twin}(\vec{e}_{4,2}) \\
=&\vec{e}_{4,1} \\
\neq&\vec{e}_{3,1} \\
=&\text{Next}(\vec{e}_{2,2})
\end{aligned}$$

(4) True. See defintion of IncidentFace.



Vertex	Coordinates	IncidentEdge
v_1	$(0,4)$	$\vec{e}_{1,1}$
v_2	$(2,4)$	$\vec{e}_{4,2}$
v_3	$(2,2)$	$\vec{e}_{2,1}$
v_4	$(1,1)$	$\vec{e}_{2,2}$

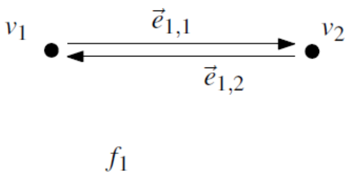
Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

Half-edge	Origin	Twin	IncidentFace	Next	Prev
$\vec{e}_{1,1}$	v_1	$\vec{e}_{1,2}$	f_1	$\vec{e}_{4,2}$	$\vec{e}_{3,1}$
$\vec{e}_{1,2}$	v_2	$\vec{e}_{1,1}$	f_2	$\vec{e}_{3,2}$	$\vec{e}_{4,1}$
$\vec{e}_{2,1}$	v_3	$\vec{e}_{2,2}$	f_1	$\vec{e}_{2,2}$	$\vec{e}_{4,2}$
$\vec{e}_{2,2}$	v_4	$\vec{e}_{2,1}$	f_1	$\vec{e}_{3,1}$	$\vec{e}_{2,1}$
$\vec{e}_{3,1}$	v_3	$\vec{e}_{3,2}$	f_1	$\vec{e}_{1,1}$	$\vec{e}_{2,2}$
$\vec{e}_{3,2}$	v_1	$\vec{e}_{3,1}$	f_2	$\vec{e}_{4,1}$	$\vec{e}_{1,2}$
$\vec{e}_{4,1}$	v_3	$\vec{e}_{4,2}$	f_2	$\vec{e}_{1,2}$	$\vec{e}_{3,2}$
$\vec{e}_{4,2}$	v_2	$\vec{e}_{4,1}$	f_1	$\vec{e}_{2,1}$	$\vec{e}_{1,1}$

Figure 1: An example of a doubly-connected edge list for a simple subdivision. Figure is taken from [Ber+08].

b) Give an example of a doubly-connected edge list where for an edge $\vec{e}$ the faces IncidentFace($\vec{e}$) and IncidentFace(Twin($\vec{e}$)) are the same.

Consider this example:



Vertex	Coordinates	IncidentEdge
v_1	$(0,4)$	$\vec{e}_{1,1}$
v_2	$(2,4)$	$\vec{e}_{1,2}$

Face	OuterComponent	InnerComponents
f_1	**nil**	**nil**

Half-edge	Origin	Twin	IncidentFace	Next	Prev
$\vec{e}_{1,1}$	v_1	$\vec{e}_{1,2}$	f_1	$\vec{e}_{1,2}$	$\vec{e}_{1,2}$
$\vec{e}_{1,2}$	v_2	$\vec{e}_{1,1}$	f_2	$\vec{e}_{1,1}$	$\vec{e}_{1,1}$

Figure 2: A very simple example of a doubly-connected edge list.

This is a doubly-connected edge list where $f_1 = \text{IncidentFace}(\vec{e}_{1,1}) = \text{IncidentFace}(\text{Twin}(\vec{e}_{1,1})) = \text{IncidentFace}(\text{Twin}(\vec{e}_{1,2})) = \text{IncidentFace}(\vec{e}_{1,2}) = f_1$ holds.

c) Given a doubly-connected edge list representation for a subdivision where

$$\text{Twin}(\vec{e}) = \text{Next}(\vec{e})$$

holds for every half-edge $\vec{e}$. How many faces can the subdivision have at most?

In this case we have exactly 1 unbounded face. There are only doubly-connected edges like in figure 2 because this is the only case where $\text{Twin}(\vec{e}) = \text{Next}(\vec{e})$ holds. (Of course we might have many of these edges and some of them might be sharing some nodes). By definition each half-edge bounds only one face. We can easily apply the prove from **b)** here and use it to show that this is always the same face. This shows that there can only be exactly 1 unbounded face.

Exercise 2 (Planar Subdivision and Point Set) (4 points)

Let S.vertices be a list of vertices, which has the always the edge in counter-clockwise direction as incident face. The edges in counter-clockwise direction are the inner edges.

```
V = sort(S.vertices)
F = [ ]

for p ∈ P
    face = null
    vertices = V
    while size(vertices) > 0
        current_face = null
        v = vertices.pop()
        i = v.incident_edge
        n = i.next_edge
        if is_left(point, i)
            current_face = i.incident_face
        while i != n
            vertices.remove(n.origin)
            if current_face != null
                if is_right(point, n)
                    current_face = null
            n = n.next_edge
        if current_face != null
            face = current_face
    F.append(face)
```

F lists the face in which each point lies. If a face is null, that means the point lies outside of all polygons.

The complexity of this algorithm is n*m, as for each of the m points, each of the n edges has to be checked.

Exercise 3 (Planar Subdivision and Point Set) (1 points)

Exercise 4 (Planar Subdivision and Point Set) (1 points)

Exercise 5 (3-Coloring of Simple Polygons) (5 points)

For saving space, we only hand in our function for computing the 3-coloring, and skip all helper functions like reading the ply files or drawing the results. We encode all three colors by the numbers 0, 1 and 2. After that, we encode those numbers by real colors. But this is not part of this function.

```
# tri is the abbreviation of triangle.
def coloring(tri, colors, next_triangle, used_triangle):
    if tri != []:
        p1 = tri[next_triangle][0]
        p2 = tri[next_triangle][1]
        p3 = tri[next_triangle][2]

        used_triangle[next_triangle] = 1

        # This can only be true in the first call of coloring(), because initial are all colors -1
        if max(colors) == -1:
            colors[p1] = 0
            colors[p2] = 1

        if colors[p1] == -1 and colors[p2] != -1 and colors[p3] != -1:
            all_possible_colors = [0,1,2]
            all_possible_colors.remove(colors[p2])
            all_possible_colors.remove(colors[p3])
            colors[p1] = all_possible_colors[0]

        if colors[p2] == -1 and colors[p1] != -1 and colors[p3] != -1:
            all_possible_colors = [0,1,2]
            all_possible_colors.remove(colors[p1])
            all_possible_colors.remove(colors[p3])
            colors[p2] = all_possible_colors[0]

        if colors[p3] == -1 and colors[p1] != -1 and colors[p2] != -1:
            all_possible_colors = [0,1,2]
            all_possible_colors.remove(colors[p1])
            all_possible_colors.remove(colors[p2])
            colors[p3] = all_possible_colors[0]
```

```python
        p1_p2_triangles = [i for i in range(len(tri)) if p1 in tri[i] and p2 in tri[i] and not p3 in tri[i]]
        p1_p3_triangles = [i for i in range(len(tri)) if p1 in tri[i] and p3 in tri[i] and not p2 in tri[i]]
        p2_p3_triangles = [i for i in range(len(tri)) if p2 in tri[i] and p3 in tri[i] and not p1 in tri[i]]

        children = p1_p2_triangles + p1_p3_triangles + p2_p3_triangles

        for face in children:
            if used_triangle[face] != -1:
                children.remove(face)

        for child in children:
            coloring(tri, colors, child, used_triangle)

    else:
        if min(colors) == -1:
            coloring(tri, colors, 0, used_triangle)

    return colors
```

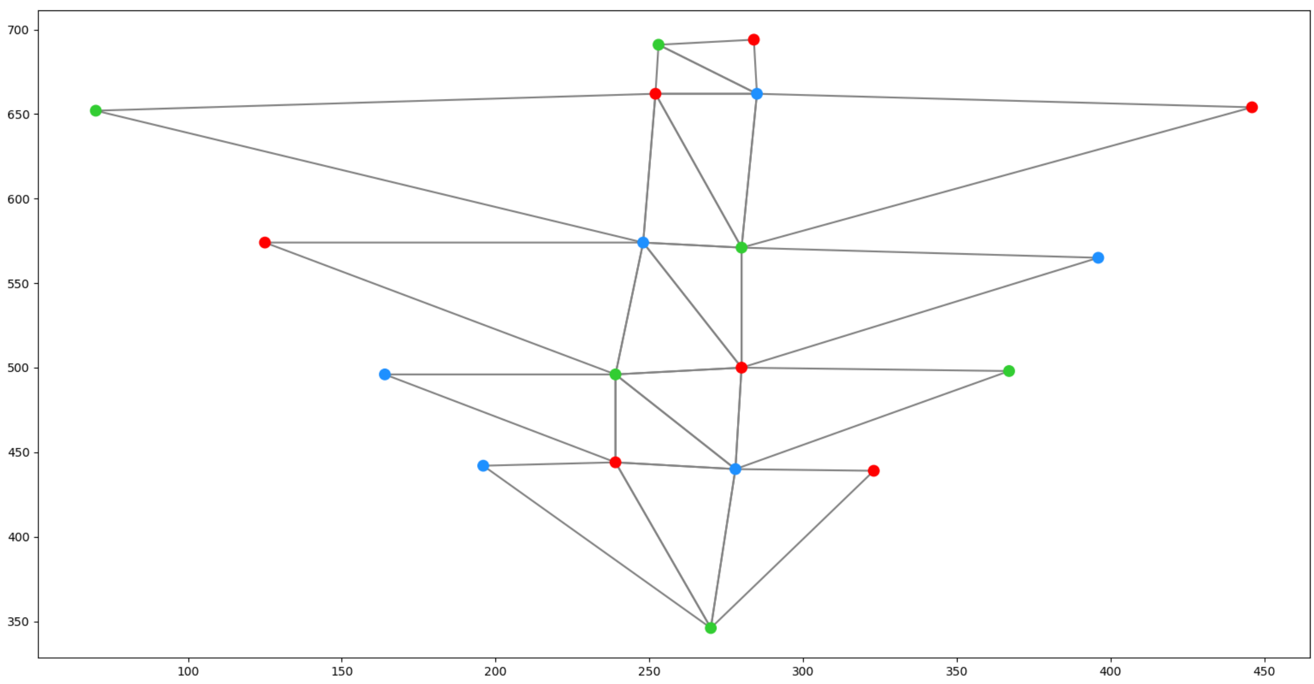


Figure 3: 3-coloring of the dataset: simplePolygonTriangulated_1.ply.

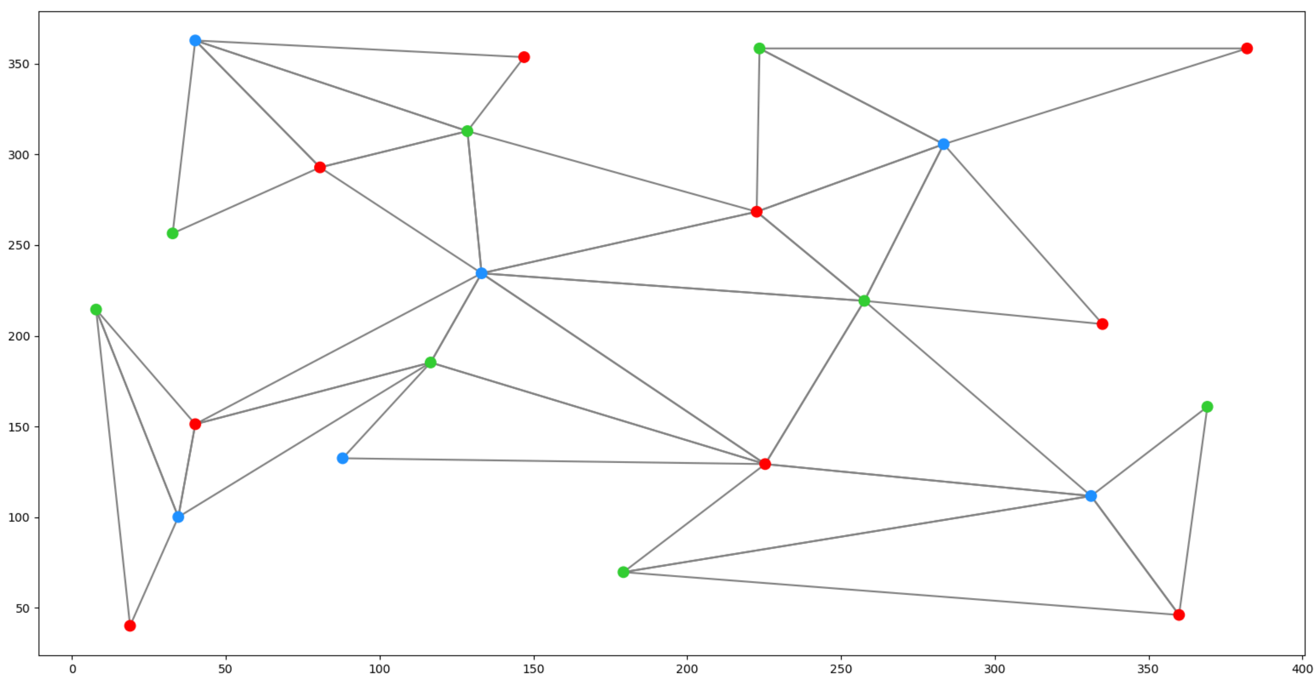


Figure 4: 3-coloring of the dataset: simplePolygonTriangulated_2.ply.

References

[Ber+08] Mark de Berg et al. Computational geometry: algorithms and applications. Springer-Verlag TELOS, 2008.