Przetwarzanie i przechowywanie opisu siatki trójkątnej na płaszczyźnie

Opis projektu

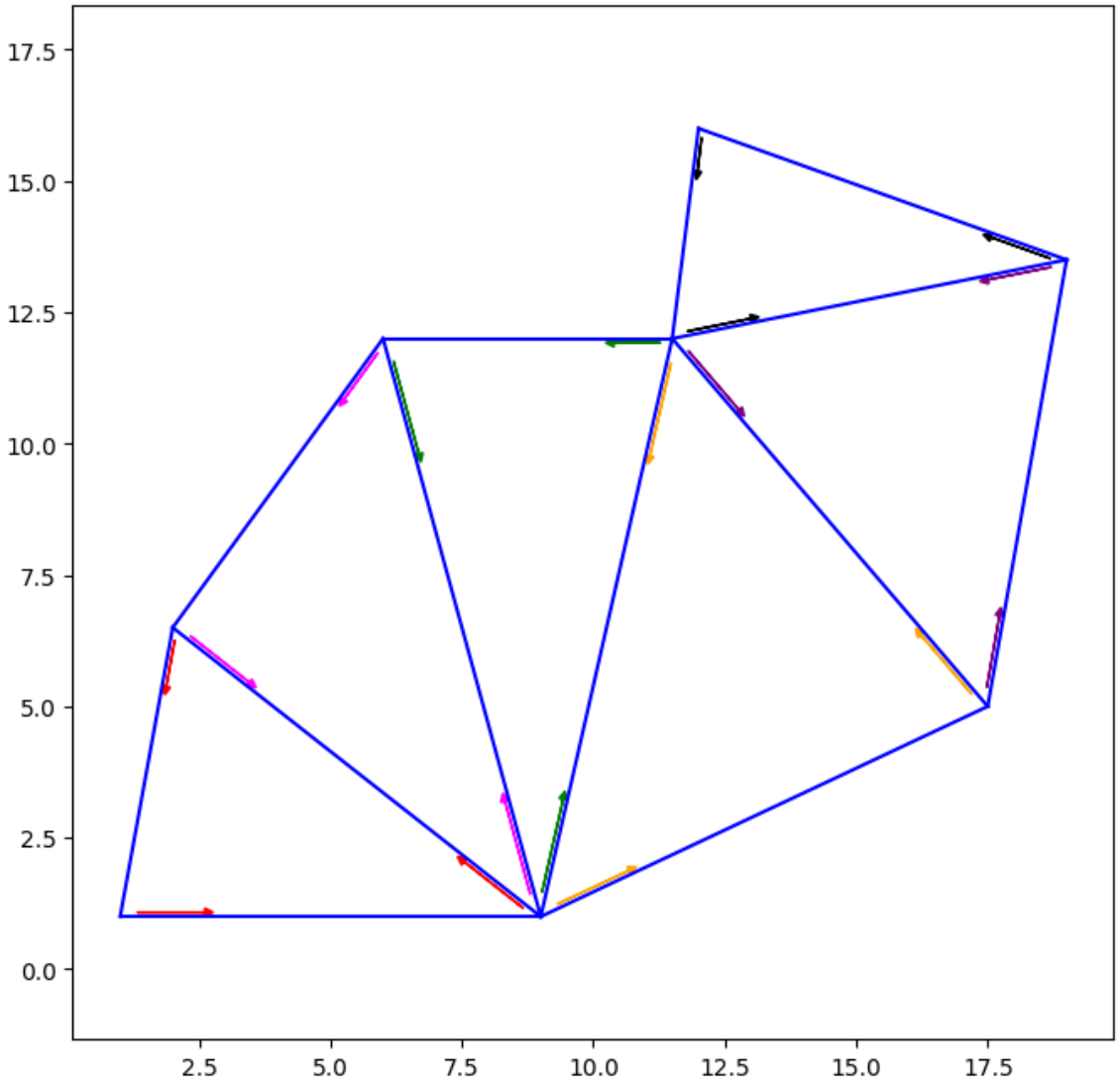
Porównanie Halfedge Data structure ze strukturą składającą się ze zbioru punktów i trójkątów na podstawie trzech operacji.

OP1: wyznaczanie otoczenia dla wybranego wierzchołka (kolejne warstwy incydentnych wierzchołków – należy rozpatrzyć otoczenia składające się z jednej warstwy oraz dwóch warstw),

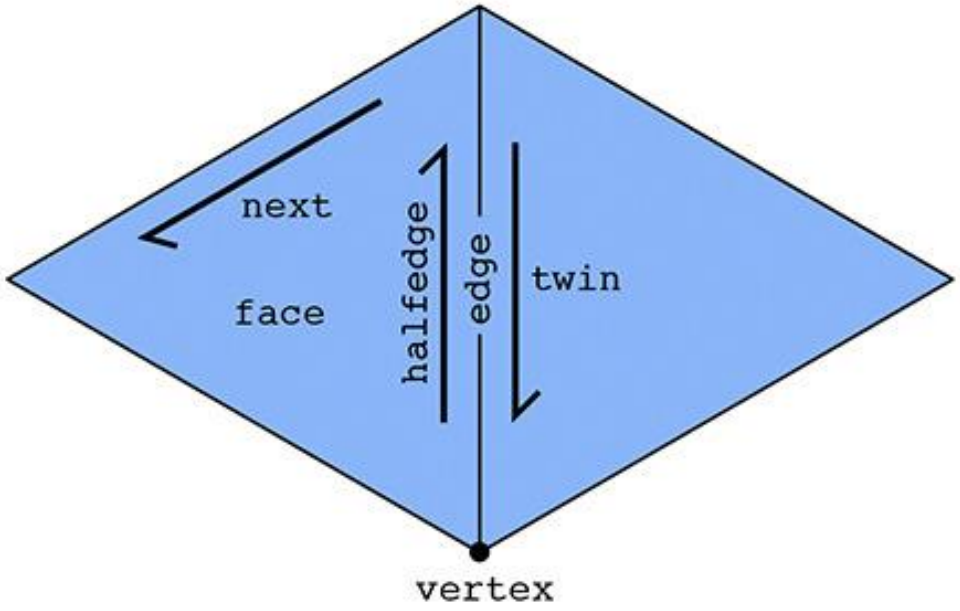
OP2: wyznaczanie otoczenia dla wybranego trójkąta (kolejne warstwy incydentnych trójkątów – należy rozpatrzyć otoczenia składające się z jednej warstwy oraz dwóch warstw),

OP3: przeglądanie incydentnych trójkątów od wybranego trójkąta w kierunku wybranego punktu (dla odszukania trójkąta zawierającego dany punkt).

Half Edge Data Structure



Halfedge Data Structure jest to struktura danych będąca listą wszystkich pół krawędzi, gdzie pół krawędź jest obiektem który przechowywuje informacje o wierzchołku z którego wychodzi pół krawędź, ścianie wewnątrz której jest pół krawędź, następnej w kolejności pół krawędzi oraz bliźniaczej pół krawędzi czyli takiej która ma początek w wierzchołku z którego wychodzi następnik pół krawędzi, a jej następnik ma początek w tym samym wierzchołku co badana pół krawędź



Potrzebne informacje o wierzchołkach oraz ścianach przechowywane są w odpowiednich obiektach na które to wskaźniki przechowują pół krawędzie.

Pół krawędzie leżące na tej samej ścianie są ułożone w cykl w kolejności odwrotnej do wskazówek zegara, a przejście na sąsiednie ściany odbywa się za pomocą bliźniaczej pół krawędzi.

```python
class HalfEdge:
    def __init__(self,v: 'Vertex', face: 'Face') -> None:
        self.vertex = #wskaźnik na wierzchołek
        self.twin = #wskaźnik na bliźniaczą półkrawędź
        self.face = #wskaźnik na ściane
        self.next = #wskaźnik na kolejną półkrawędź
        self.prev = #wskaźnik na poprzednią półkrawędź tożsame z next.next

class Vertex:
    def __init__(self,cords: Point, index: int) -> None:
        self.cords = #krotka zawierająca współrzędne
        self.index = #indeks wierzchołka w tabeli z punktami
        self.halfEdge = #wskaźnik na półkrawędź z która wychodzi z punktu

class Face:
    def __init__(self,points: list[Point],index: int) -> None:
        self.points = #krotka zawierająca indeksy wierzchołków trójkąta w liście z punktami
        self.index = #indeks ściany
        self.halfEdge = #jedna z trzech półkrawędzi leżących wewnątrz ściany
```

OP1 Przy użyciu Half Edge Data structure

```python
def apex_surroundings_halfedges(p,halfedges):
    def surr(p,halfedges):
        nonlocal p1
        surrounding = set()
        p_halfedges = []
        for halfedge in halfedges:
            if halfedge.vertex.index == p:
                p_halfedges.append(halfedge)
        while p_halfedges:
            he = p_halfedges.pop()
            if he.next.vertex.index != p1:
                surrounding.add(he.next.vertex.index)
            if he.prev.vertex.index != p1:
                surrounding.add(he.prev.vertex.index)
        return surrounding
    p1 = p
    s = list(surr(p,halfedges))
    s1 = copy.copy(s)
    for s_point in s1:
        s += surr(s_point,halfedges)
    return list(set(s))
```

- Szukamy sąsiadów wierzchołka startowego, następnie szukamy sąsiadów jego sąsiadów.
- Aby znaleźć sąsiadów wierzchołka najpierw znajdujemy wszystkie pół krawędzie wychodzące z tego wierzchołka a następnie dla każdej takiej pół krawędzi przechodzimy po jej ścianie i zapisujemy znalezione punkty

OP2 Przy użyciu Half Edge Data structure

```
def triangle_surroundings_halfedge(tri_id, halfedges):
    def surr(tri_id,halfedges):
        nonlocal tri_id1
        he = None
        for h in halfedges:
            if h.face.index == tri_id:
                he = h
                break
        surrounding = []
        if he.twin != None and he.twin.face.index != tri_id1:
            surrounding.append(he.twin.face.index)
        he = he.next
        if he.twin != None and he.twin.face.index != tri_id1:
            surrounding.append(he.twin.face.index)
        he = he.next
        if he.twin != None and he.twin.face.index != tri_id1:
            surrounding.append(he.twin.face.index)
        he = he.next
        return surrounding
    tri_id1 = tri_id
    surrounding = surr(tri_id,halfedges)
    s1 = copy.copy(surrounding)
    for face in s1:
        surrounding += surr(face,halfedges)
    return list(set(surrounding))
```

- Szukamy sąsiednich ścian ściany początkowej, następnie powtarzamy to dla znalezionych sąsiadów
- Aby znaleźć sąsiadów szukamy pół krawędzi należącej do badanej ściany, a następnie dodaje (jeżeli istnieją) ściany do których należą bliźniacze pół krawędzie powiązane z cyklem wewnątrz początkowej ściany

OP3 Przy użyciu Half Edge Data structure

```
def find_triangle_halfedge(halfedges,point,start_triangle):
    def surr(he):
        surrounding = []
        if he.twin != None:
            surrounding.append(he.twin)
        he = he.next
        if he.twin != None:
            surrounding.append(he.twin)
        he = he.next
        if he.twin != None:
            surrounding.append(he.twin)
        return surrounding
    he = None
    for h in halfedges:
        if h.face.index == start_triangle:
            he = h
            break
    triangle = [he.vertex.cords,he.next.vertex.cords,he.prev.vertex.cords]
    tri_ind = he.face.index
    stack = [(triangle,tri_ind,he)]
    visited = [tri_ind]
    while stack:
        t,t_ind,halfedge = stack.pop(-1)
        if point_in(t,point):
            return t_ind
        surrounding = surr(halfedge)
        for h in surrounding:
            new_tri = [h.vertex.cords,h.next.vertex.cords,h.prev.vertex.cords]
            new_tri_ind = h.face.index
            if new_tri_ind not in visited:
                stack.append((new_tri,new_tri_ind,h))
                visited.append(new_tri_ind)
```

- Tworzymy stos który przechowuje trójkąty, które należy sprawdzić
- Dopóki stos nie jest pusty lub nie znaleźliśmy już szukanego trójkąta, dodajemy na stos sąsiadów aktualnego trójkąta

Struktura złożona z listy punktów oraz listy połączeń

Struktura składa się z dwóch list, listy punktów reprezentowanych jako krotki zawierające współrzędne x, y oraz listę połąćzeń przedstawionych jako krotki zawierające indeksy punktów składających się na dane połączenie.

OP1 Przy użyciu listy punktów oraz listy połączeń

```python
def apex_surroundings_basic(point, points, connections):
    surrounding = []
    for i,j in connections:
        if i == point:
            if j not in surrounding:
                surrounding.append(j)
            for a,b in connections:
                if a == j and b != point and b not in surrounding:
                    surrounding.append(b)
                elif b == j and a != point and a not in surrounding:
                    surrounding.append(a)
        elif j == point:
            if i not in surrounding:
                surrounding.append(i)
            for a,b in connections:
                if a == i and b != point and b not in surrounding:
                    surrounding.append(b)
                elif b == i and a != point and a not in surrounding:
                    surrounding.append(a)
    return surrounding
```

- Iterujemy po wszystkich połączeniach i sprawdzamy czy jest to połączenie z interesującego nas punktu, jeżeli tak to ponownie iterujemy po wszystkich połączeniach aby znaleźć sąsiadów sąsiadów.

OP2 Przy użyciu listy punktów oraz listy połączeń

```python
def triangle_surroundings_basic(tri_id, triangles):
    triangle = triangles[tri_id]
    surrounding = []
    for i,(t1,t2,t3) in enumerate(triangles):
        if i != tri_id and (t1 in triangle or t2 in triangle or t3 in triangle):
            if i not in surrounding:
                surrounding.append(i)
    res = copy.copy(surrounding)
    for j in surrounding:
        temp_triangle = triangles[j]
        for k,(t1,t2,t3) in enumerate(triangles):
            if k != tri_id and (t1 in temp_triangle or t2 in temp_triangle or t3 in temp_triangle):
                if k not in res:
                    res.append(k)
    return surrounding, res
```

- Iterujemy po liście połączeń(przedstawionej jako trójkąty) i szukamy trójkątów sąsiadujących z trójkątem startowym
- Następnie dla każdego sąsiada szukamy jego sąsiadów w ten sam sposób

OP3 Przy użyciu listy punktów oraz listy połączeń

```
def find_triangle_basic(triangles,points,point,start_triangle):
    triangle = [points[triangles[start_triangle][0]],
    points[triangles[start_triangle][1]],
    points[triangles[start_triangle][2]]]
    stack = [(triangle,start_triangle)]
    visited = [start_triangle]
    while stack:
        t, t_id = stack.pop(-1)
        if point_in(t,point):
            return t_id
        surrounding = triangle_surroundings_basic(t_id,triangles)[0]
        visited += [t_id]
        for s in surrounding:
            if s not in visited:
                stack.append(([points[triangles[s][0]],points[triangles[s][1]],points[triangles[s][2]]],s))
```

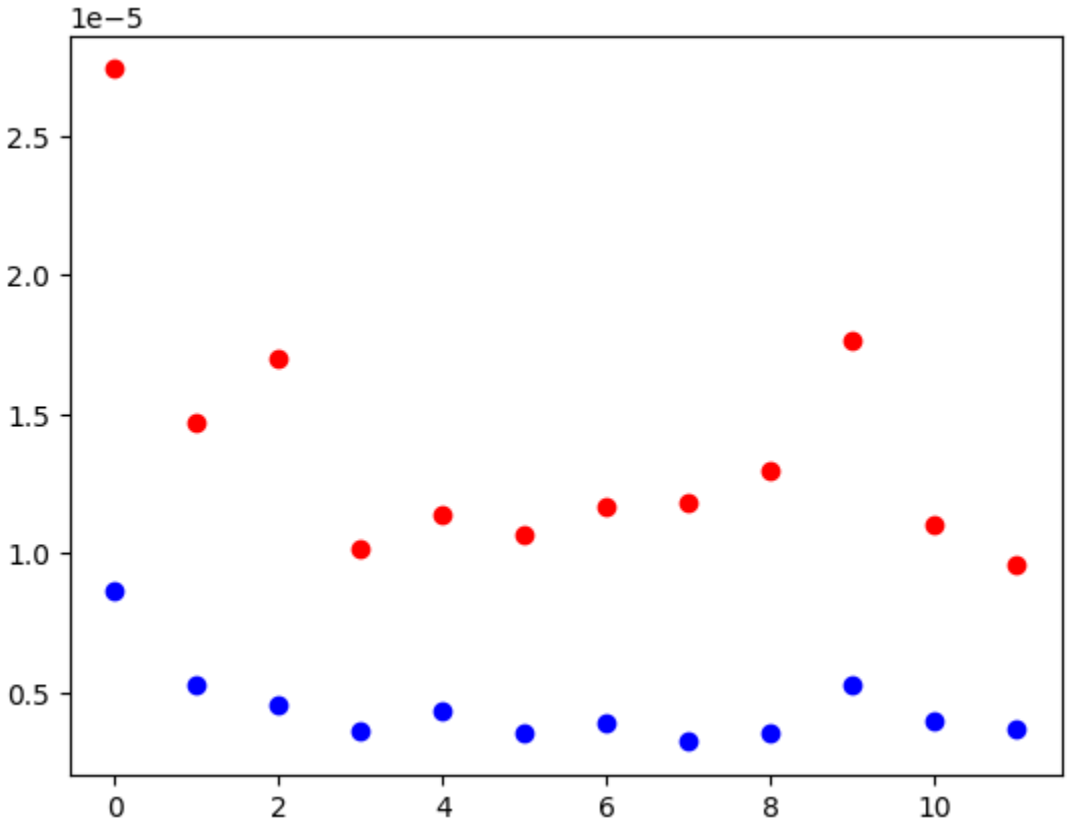
- Tworzymy stos przechowywujący trójkąty które należy sprawdzić
- Dopóki stos nie jest pusty lub nie znaleźliśmy już szukanego trójkąta, dodajemy sąsiednie trójkąty

Porównanie czasowe algorytmów

W OP1 algorytm oparty o Half Edge Data structure wypadł gorzej od swojego odpowiednika, w OP2 algorytm oparty o Half Edge Data structure wypadł lepiej, a w przypadku OP3 wyniki były zbliżone z delikatną przewagą dla algorytmu opartego o podstawową strukture

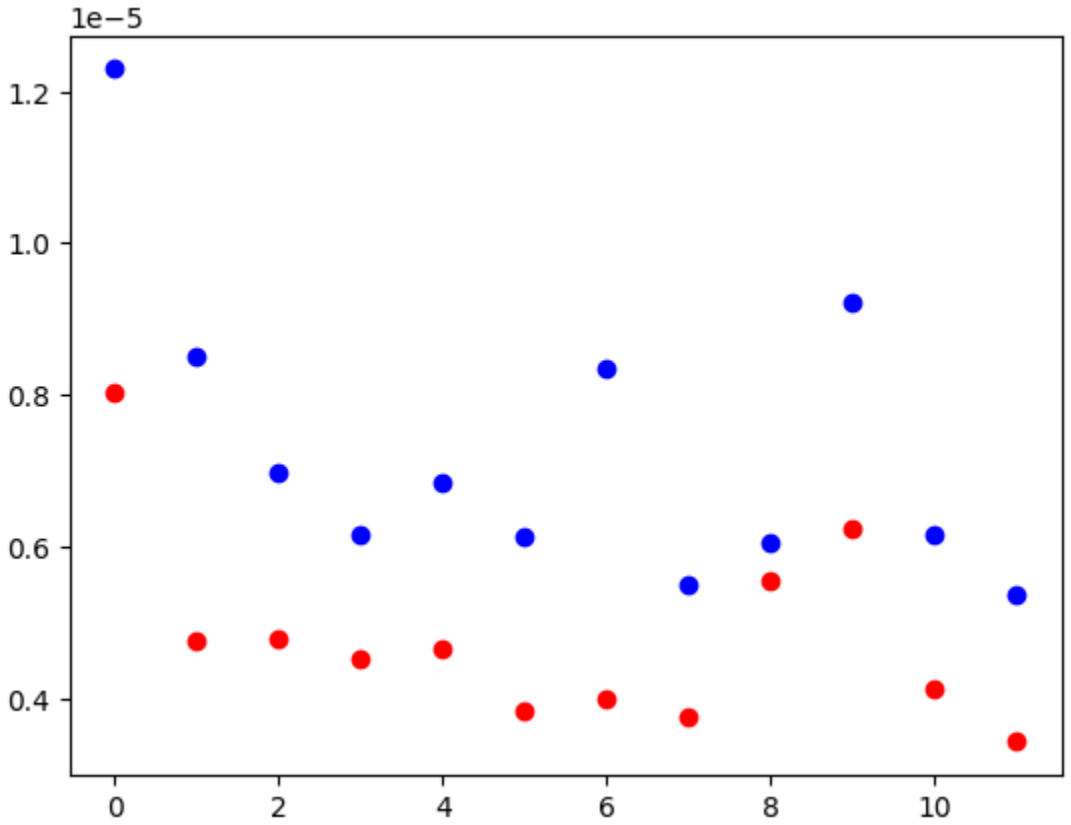
Wykres dla OP1

Czerwone – Half Edge Data Structure
Niebieskie – Lista punktów oraz lista połączeń



Wykres dla OP2

Czerwone – Half Edge Data Structure
Niebieskie – Lista punktów oraz lista połączeń



Wykres dla OP3

Czerwone – Half Edge Data Structure
Niebieskie – Lista punktów oraz lista połączeń