

Geometric Algorithms

Assignment 4

Laura Baakman (s1869140)

October 15, 2014

A

A.1 Creating the DCEL

To create a DCEL one needs a couple of base classes, namely `Vertex`, `HalfEdge` and `Face`.

Vertex

The definition of the class `Vertex` and its methods that are pertinent now are presented in Listing 1. In line with the provided definition a `Vertex` has a set of coordinates representing its location, `coordinates`, and an edge, `incident_edge`, that has the `Vertex` as its origin.

I have overridden the equality definition since using the default definition could lead to infinite recursion. As the default compares all attributes of the class, which would mean comparing two `Vertex`'s and two `HalfEdges`. Comparing two `HalfEdges` means comparing, among others, its origins which would lead to comparing two `Vertex`s which would lead to comparing two `HalfEdges` and so on and so forth.

Since each `Vertex` is uniquely defined by its `coordinates` we only compare those when checking the equality of two `Vertex`s.

Listing 1: The definition of the class `Vertex`.

```
def __init__(self, coordinates, incident_edge=None):
    """Construct a Vertex object."""
    super(Vertex, self).__init__()
    self.coordinates = coordinates
    self.incident_edge = incident_edge

def __eq__(self, other):
    """Check if two objects are equal by comparing only their coordinates."""
    if type(other) is type(self):
        return self.coordinates == other.coordinates
    return False

def __neq__(self, other):
    """Check if two objects are not equal."""
    return not self.__eq__(other)
```

HalfEdge

A `HalfEdge` is a directed edge which is represented by its origin, a `Vertex` and a twin, which is the `HalfEdge` with this `HalfEdge`'s origin as its destination and this `HalfEdge`'s destination as its origin. The implementation of the class `HalfEdge` and the methods relevant to this discussion are presented in Listing 2.

Furthermore each `HalfEdge` stores an incident face and a next and previous edge. The `incident_face` is the face that is to the left-handed side when walking along this edge. The attributes `nxt` represent the edge one should take on arriving on the `HalfEdge`'s destination

when traversing the boundaries of its `incident_face`, `prev` is the `HalfEdge` one came from on the same walk.

The method `get_destination` returns the destination of the `HalfEdge` this is the same as the origin of its twin.

The `__eq__` method of `HalfEdge` has been overridden to avoid infinite recursion when comparing `HalfEdge`'s and to make it possible to compare an `HalfEdge` without the `nxt`, `incident_face` or `prev` property. This will not lead to any problems since an `HalfEdge` is uniquely defined by its origin and destination.

Listing 2: The definition of the class `HalfEdge`.

```
def __init__(self, origin, twin=None, incident_face=None, nxt=None, prev=None):
    """Construct a HalfEdge object."""
    super(HalfEdge, self).__init__()
    self.origin = origin
    self.twin = twin
    self.incident_face = incident_face
    self.nxt = nxt
    self.prev = prev

def get_destination(self):
    """Return the destination of the halfedge as a vertex."""
    return self.twin.origin

def __eq__(self, other):
    """Check if two half edges are equal."""
    if type(other) is type(self):
        return (
            self.origin == other.origin and
            self.twin.origin == other.twin.origin
        )
    return False

def __neq__(self, other):
    """Check if two objects are not equal."""
    return not self.__eq__(other)
```

Face

A `Face` is defined by an `outer_component`: a `HalfEdge` that when traversed keeps the face on the left and a list `inner_components`: which stores `HalfEdges` of the outer boundaries of each hole in the `Face`. The definition of the class `Face` is provided in Listing 3.

Listing 3: The definition of the class `Face`.

```
def __init__(self, outer_component, inner_components=[]):
    """Construct a Face object."""
    super(Face, self).__init__()
    self.outer_component = outer_component
    self.inner_components = inner_components
```

From a Delauny Triangulation to a DCEL

To represent a DCEL I have introduced the method `from_delauny_triangulation` that given the triangles and vertices of a Delauny Triangulation as provided by `matplotlib.delaunay.delaunay()` and returns a DCEL which is defined as in Listing 4.

Listing 4: The constructor the class `DCEL`.

```
def __init__(self, vertices=[], edges=[], faces=[]):
    """Construct a DCEL object."""
    super(DCEL, self).__init__()
    self.vertices = vertices
    self.edges = edges
    self.faces = faces
```

To generate a DCEL from a Delauny Triangulation without geometric operations the following algorithm is used:

1. for each triangle t of the triangulation
 - (a) Add t 's vertices to the DCEL if they do not already exist, otherwise get the existing vertices from the DCEL.
 - (b) Add t 's edges and the twins of those edges to the DCEL if they do not already exist, otherwise get the existing edges from the DCEL.
 - (c) Add a face with one of the edges from the previous step to the as its outer component to the DCEL. The triangles in a Delaunay triangulation do not have inner faces.
 - (d) Set the next and previous edge of the edges from step 1b and add the face created in 1c as its face.
2. Add the face that has the triangulation as its inner component to the DCEL, and update the half edges that from the outer boundary of the triangulation.

The method `from_delaunay_triangulation` is presented in Listing 5 without its local methods, these are presented in Listing 8 and Listing 10. The code presented in Listing 5 executes step 1a and calls the necessary methods for the other steps.

Listing 5: The method `from_delaunay_triangulation()` without its local methods.

```
def from_delaunay_triangulation(cls, xl, yl, triangles):
    """ Construct a DCEL from the output of matplotlib.delaunay.delaunay."""
    dcel = cls()
    for t in triangles:
        triangle_vertices = [
            dcel.add_vertex(Vertex(x))
            for x in du.get_triangle_vertices(xl, yl, t)
        ]
        add_triangle_edges()
        add_containing_face_to_dcel()
    return dcel
```

The method `get_triangle_vertices()` is defined in the module `delaunyUtils`, see Listing 6 which contains methods that perform often used methods on the results of `matplotlib.delaunay.delaunay()`. This method returns the vertices of the triangle in counter-clockwise order, using the fact that the order of the vertices of a triangle as returned by `matplotlib.delaunay.delaunay()` is in clockwise order.

Listing 6: The method `get_triangle_vertices` in the module `delaunyUtils`.

```
def get_triangle_vertices(xl, yl, triangle):
    """Return the vertices of triangle in CCW, triangle is a list of indices in xl and yl."""
    return ([
        [xl[triangle[0]], yl[triangle[0]]],
        [xl[triangle[2]], yl[triangle[2]]],
        [xl[triangle[1]], yl[triangle[1]]]
    ])
```

The function `add_vertex`, presented in Listing 7, checks before adding a vertex if that Vertex is already in the DCEL. If that is the older Vertex is returned, otherwise the new Vertex is added and returned.

Listing 7: The method `add_vertex` in the class DCEL.

```
def add_vertex(self, vertex):
    """Add a vertex to the DCEL if the vertex doesn't already exists."""
    try:
        vertex_idx = self.vertices.index(vertex)
        return self.vertices[vertex_idx]
    except Exception:
        self.vertices.append(vertex)
        return vertex
```

The local method, `add_triangle_edges`, presented in Listing 8, adds the edges and the face of the current triangle to the DCEL. Which corresponds with step 1b through 1d.

Listing 8: The method `add_triangle_edges()`.

```
def add_triangle_edges():
    triangles_edges = []
    for vertex_idx, origin in enumerate(triangle_vertices):
        # Destination of the edge in this triangle that has vertex as origin
        destination = triangle_vertices[(vertex_idx + 1) % 3]
        edge_1 = HalfEdge(origin)
        edge_2 = HalfEdge(destination, twin=edge_1)
        edge_1.twin = edge_2
        edge_1 = dcel.add_edge(edge_1)
        edge_2.twin = edge_1
        edge_2 = dcel.add_edge(edge_2)
        edge_1.twin = edge_2
        triangles_edges.append(edge_1)

    triangle_face = Face(triangles_edges[0])
    dcel.faces.append(triangle_face)
    # Set previous and next of the edges
    for edge_idx, edge in enumerate(triangles_edges):
        edge.nxt = triangles_edges[(edge_idx + 1) % 3]
        edge.prev = triangles_edges[(edge_idx + 3 - 1) % 3]
        edge.incident_face = triangle_face
        triangle_vertices[edge_idx].incident_edge = edge
```

The function `add_edge`, see Listing 9, called in `add_triangle_edges` works the same as the method `add_vertex`. `edge_1` is the `HalfEdge` of the current triangle, since we know that the three vertices in `triangle_vertices` are in CCW-order we know that the destination of the current edge is the vertex before this edge in that list.

The constant setting of `edge_1` and `edge_2` is to ensure that the references to the twin of an edge contain the oldest instance of that edge, thus the `HalfEdge` returned by `add_edge`.

When the edges of the current triangle and their twins are added to the DCEL, `triangles_edges` contains all the `HalfEdges` that when traversed in their order in the list walk around the boundary of the current triangle while keeping that face on the left side.

We then create a new face, using one of the edges of the triangle as its incident edge, and add this to the DCEL. We then set the `nxt`, `prev` of the edges and the `incident_edge` of the vertices of the triangle using our knowledge of the order of the `HalfEdges` in `triangles_edges`.

Listing 9: The method `add_edge()` in the class DCEL.

```
def add_edge(self, edge):
    """Add a edge to the DCEL if the edge doesn't already exists."""
    try:
        edge_idx = self.edges.index(edge)
        return self.edges[edge_idx]
    except Exception:
        self.edges.append(edge)
        return edge
```

When we have walked through all triangles of the triangulation only the `HalfEdges` that make up the boundary of the triangulation do not have the attributes `nxt`, `prev` and `incident_face` set. These edges have been created since their twins are a `HalfEdge` of the triangles in the triangulation that have two or less neighbours.

The method `add_containing_face_to_dcel` adds the unbounded face whose hole is defined by these edges to the DCEL, see Listing 10.

Listing 10: The method `add_containing_face_to_dcel()`.

```
def add_containing_face_to_dcel():
    containing_face_edges = [edge for edge in dcel.edges if not edge.nxt]
    edge = containing_face_edges.pop()
    face = Face(outer_component=None, inner_components=[edge])
    dcel.faces.append(face)
    first_edge = edge
    previous_edge = [
        e for e in containing_face_edges if e.get_destination() == edge.origin
    ]
    edge.prev = previous_edge[0]
    while len(containing_face_edges) > 1:
        edge.incident_face = face
        next_edge = [
            e for e in containing_face_edges if e.origin == edge.get_destination()
        ]
```

```

    ]
    edge.nxt = next_edge[0]
    next_edge[0].prev = edge
    edge = next_edge[0]
    containing_face_edges.remove(next_edge[0])
    edge_2 = containing_face_edges.pop()
    edge.incident_face = face
    edge_2.incident_face = face
    edge_2.prev = edge
    edge_2.nxt = first_edge
    edge.nxt = edge_2

```

To make searching in this subset of edges more efficient the list with edges of the DCEL without incident face is copied and each HalfEdge that has received a `incident_face` is removed from the list.

A.2 Walking Along the Outer Boundary

To walk along the outer boundary I have defined the method `get_edges_inner_component()`, which given a Face and the index of an inner component of that Face returns a list of edges that walk along that Face. This list is generated by walking recursively along the Edges of the face until the current Edge is the edge where the walk started. See Listing 11 for the implementation of this method.

Listing 11: The method `get_edges_inner_component()` in the class Face.

```

def get_edges_inner_component(self, inner_component_idx=0):
    """Return all edges of this face in CCW order."""
    def get_edges_helper(current_edge, edges):
        if self.inner_components[inner_component_idx] == current_edge:
            return edges
        else:
            edges.append(current_edge)
            return get_edges_helper(current_edge.nxt, edges)
    return get_edges_helper(
        self.inner_components[inner_component_idx].nxt,
        [self.inner_components[inner_component_idx]]
    )

```

Running the script `assignment4A` with the flag `-dt` calls sets the method that displays the Delaunay Triangulation and highlights the outer boundary as the display method, see Listing 12 for the display method. The resulting image is presented in Figure 1. The method `as_points` returns the object it is called on as a list of defining points, see Listing 13 and Listing 14.

Listing 12: The part of the method `display_outer_boudary()` that draws the boundary.

```

def display_outer_boudary():
    """Display the Delaunay triangulation and its outer boundary."""
    global dcel
    face = [face for face in dcel.faces if face.inner_components]
    edges = face[0].get_edges_inner_component()
    glColor3f(0.0, 1.0, 0.0)
    glBegin(GL_LINES)
    for edge in edges:
        destination = edge.get_destination().as_points()
        glVertex2f(edge.origin.as_points()[0], edge.origin.as_points()[1])
        glVertex2f(destination[0], destination[1])
    glEnd()
    glutSwapBuffers()

```

Listing 13: The method `as_points()` in the class Vertex.

```

def as_points(self):
    """Return the vertex as a set of points."""
    return self.coordinates

```

Listing 14: The method `as_points()` in the class HalfEdge.

```

def as_points(self):
    """Return the edge as the coordinates of the origin and destination."""
    return [self.origin.coordinates, self.twin.origin.coordinates]

```

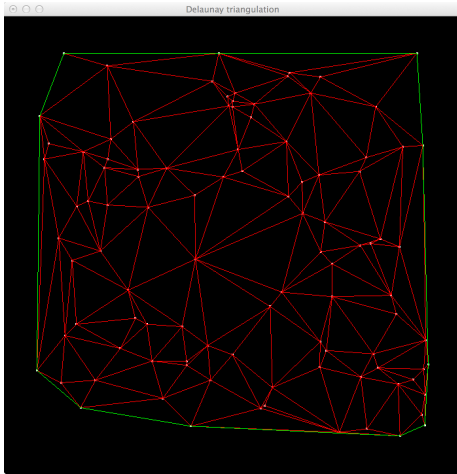


Figure 1: A Delaunay Triangulation with the outer boundary highlighted.

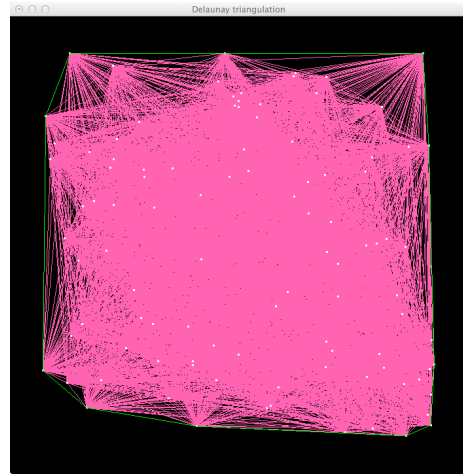


Figure 2: The vertices of the Delaunay Triangulation shown in Figure 1, the outer boundary of the triangulation and all possible line segments between the vertices of the triangulation.

To see if the plotted boundary is the convex hull of the points we have generated Figure 2, since all possible line segments between the vertices of the triangulation stay within the convex hull we can conclude that the outer boundary of the triangulation is indeed the convex hull. To generate this image one can run the script `assignment4A` with the flag `-ch`. The used display function is presented in Listing 15.

Listing 15: The method `display_inspect_convex_hull()`.

```
def display_inspect_convex_hull():
    """Display the convex hull of the points and show that it is the actual convex hull."""
    glClear(GL_COLOR_BUFFER_BIT)
    # Draw lines between all points
    glColor3f(1.0, 0.4, 0.7)
    glBegin(GL_LINES)
    for (x1, y1) in zip(x1, y1):
        for (x2, y2) in zip(x1, y1):
            glVertex2f(x1, y1)
            glVertex2f(x2, y2)
    glEnd()
    # Draw points
    glLineWidth(1.0)
    glColor3f(1.0, 1.0, 1.0)
    glPointSize(3)
    glBegin(GL_POINTS)
    for i in range(len(x1)):
        glVertex2f(x1[i], y1[i])
    glEnd()
    # Draw the outer boundary
    global dcel
    face = [face for face in dcel.faces if face.inner_components]
    edges = face[0].get_edges_inner_component()
    glColor3f(0.0, 1.0, 0.0)
    glBegin(GL_LINES)
    for edge in edges:
        destination = edge.get_destination().as_points()
        glVertex2f(edge.origin.as_points()[0], edge.origin.as_points()[1])
        glVertex2f(destination[0], destination[1])
    glEnd()
    glutSwapBuffers()
```