GAS project documentation 65165 Serra Sergio

Project structure

As regards the structure of the project, it was decided to organize it in data_structures, algorithm, drawable and static.
Going into more detail, we see below the composition of each of the folders listed above:

- ➔ Data_structures
  - ◆ Dag
  - ◆ Triangle
  - ◆ TriangleNode
- ➔ Algorithm:
  - ◆ DelaunayT
  - ◆ LegalizeEdge
- ➔ Drawable:
  - ◆ DrawableDelaunayTriangulation
  - ◆ DrawableVoronoi
- ➔ Static:
  - ◆ Adjacent

Data_structures: Triangle

The triangle class was thought of as a class made up of what was essential to be called triangle: its points. The triangle's constructor then takes care of assigning a value to its points. The class will also have the methods to return the points of which the triangle is composed and a *getOppositePoint* method, used during the legalization of the edge to find the point not in common among the triangles being considered.

Data_structures: TriangleNode
The TriangleNode class is the class that allows you to manage triangles. Previously, it was said that the Triangle class contains the exact definition of a triangle. Is the TriangleNode class that deals with the management of triangles. In fact, it is through this class that the relationship between the triangles is managed. The node will contain within it a triangle and may have kinship relations with other nodes, each containing a triangle. The constructor therefore, in addition to passing the associated triangle as a parameter, will also pass the indices of the child nodes and of the parent node that node can have. The children index will initially be 0, since the node has just been created.

The class will therefore contain all the methods that allow you to return the triangle associated with the node, the indices of the children and the father in the vector and the methods that let you know if the node is a leaf or has children. The children's index can be set using a special *setFigli* method.

As said initially, this class will handle the relationships between nodes. For each node a vector of adjacents has been defined, containing the indices of the nodes (and therefore of the triangles associated with them) in the vector. Consequently, the methods relating to the addition, substitution, and return of the adjacent have been defined in this class.

Regarding the triangulation displaying, some triangles must be skipped. Since the dag is composed of a vector of nodes, it has been chosen to insert in this class the methods that allow to understand if the node, and therefore the triangle associated with it, must be skipped during displaying.

Data_structures: Dag
The idea was to define the DAG as a vector of nodes. Each node will contain all the information in terms of kinship and relations with the other nodes. The first element of the DAG will be the Bounding Triangle node, and once a point will fall on a triangle associated with the first node, three new nodes will be formed with their triangles and relations, accessible by taking the dimension of the vector. More specifically, once a point falls into the first triangle, 3 more nodes will be formed, bringing the vector to dimension 4. To access the first new node, we will use *dagNode.size () - 3*, for the second *dagNode.size () -2* and *dagNode.size () - 1* for the third. Instead in the case of flip, since two new nodes are created and therefore two new triangles, it will be accessed through *dagNode.size () -2 for the first* and *dagNode.size () - 1 for the second.* Obviously, this applies to all nodes: given a node it is possible to access through the indexes to all the nodes that have a relationship with it.

All the triangulation is thought going to access the nodes through their index.

This class will consequently contain all the methods that allow to add, return, delete and manage some of the information of nodes contained in the vector.

Algorithm: DelaunayT
The essentiality of the Delaunay algorithm is to add the points to the triangulation and go to check some important aspects such as the triangle where the point falls so as to be able to create the new triangles in the correct triangle (*searchTriangle* method), to understand if the inserted point is has already been inserted previously (*pointAlreadyExist* method) and that to consider all the aspects for a good triangulation between which the legalization and the adjacencies. As for adjacencies, the Delaunay algorithm will rely on a class called Adjacent with static methods so as to manage the adjacent triangles created during triangulation. Initially the Delaunay algorithm will manage the insertion of the Bounding Triangle, the triangle in which all the points will then fall.

Algorithm: LegalizeEdge
This algorithm is called to verify the legality of the edge of the new triangles created by going flip when needed. Like the Delaunay algorithm, it will be based on the class called Adjacent with static methods to correctly reassign the adjacencies in case of overturning.

Drawable: DrawableDelaunayTriangulation
This is the class that allows you to draw the object of triangulation. It will have as attribute the vector of the nodes so as to recover all the triangles to be drawn and some attributes that allow to define the color of the points and the triangles. Among the attributes there is also a boolean:

drawActive, set to true if in addition to the triangles the Bounding Triangle must also be printed and the sides to it incident, false otherwise. Through the draw method it will be possible to draw the triangles obtained from the vector through the method *takesNodes*. Only the appropriate triangles will be printed: as mentioned previously, the Delaunay algorithm sets the nodes that must not be printed to true. In the *draw* method we used the *drawPoint2D* and *drawTriangle2D* method to print the triangle composed of its points.

Drawable: DrawableVoronoi
This is the class that allows you to draw the Voronoi diagram. As in the previous one, there will be some attributes that allow you to set the color of the points and in this case lines. Furthermore, this class will also have the *takesNodes* method to take the updated vector from which to derive the printable triangles. To draw the Voronoi diagram we used the Delaunay triangulation which is dual with respect to Voronoi. In order to obtain the Voronoi diagram from the Delaunay triangulation the circumcenter of the triangles that make up the Delaunay triangulation has been calculated, and each circumcenter has been connected to the circumcenter of the adjacent ones of the triangle under examination. The method used was draw aided by the *drawPoint2D* and *drawLine2D* method for printing points and lines.

Static: Adjacent
The class that deals with assigning the correct adjacencies is precisely Adjacent, composed of static methods. Each method has a precise function: setBrotherAdjacent sets the adjacent siblings of each node, *addAdjacentinCommonwithParent* searches for the adjacent of the parents in order to assign them as adjacent to the appropriate children. This method invokes another method: *checkAdjacency* that performs the comparison of the edges in common between the children and the adjacent ones of the father, assigning and appropriately reassigning these adjacent ones. To manage adjacencies, instead, after a reversal occurs, the *FlipAdjacence* method is used which in turn uses s*etBrotherAdjacent* and *checkAdjacency*.

GitHub: https://github.com/UNICAAS2/as2-project-sserra74.git