Dual Graph Mesh

Author: Caio Valente (https://github.com/valentecaio/geometry)
Date: 2023-11-01

Introduction

Triangle meshes are a fundamental element in computer graphics, computational geometry, and scientific simulations. These meshes are versatile representations of complex surfaces, essential for applications like computer-aided design, computer graphics, and more. To optimize algorithms that work with these meshes, efficient data structures are crucial. This report explores an innovative approach to structure triangle meshes using principles from topology and graph theory. The script leverages dual graph representations to create a sophisticated and efficient mesh structure, with the goal of enhancing the performance of mesh-related algorithms. By combining these fields, the proposed technique has the potential to improve various applications in the domains of computer graphics and geometric analysis. Topology-based structures offer significant advantages, as they inherently contain information about adjacency relationships, which often leads to superior algorithm performance.

Dependencies

We will use the Python library MatPlotLib for plotting results and a Lua script (matplotlua) for parsing Lua data to matplotlib. As a consequence, we need python and matplotlib to run it. We also need the lua-cjson Lua package for parsing json data.

```
$ pip install matplotlib         # install python deps
$ sudo apt install lua-cjson      # install lua deps
```

Project Structure

```
.
├── mesh/
│   ├── mesh.txt        -- Input mesh file. A list of vertices and a list of triangles.
│   └── main.lua        -- Script to test the developed algorithm.
│
├── matplotlua/         -- Plot library. Used to plot input and output meshes.
│
└── utils/              -- Utils library. Useful methods reused by other modules.
```

How to execute tests

We just need to execute the enclosing_circle/main.lua file using the Lua interpreter:

```
$ cd mesh/ && lua main.lua
wrote output to mesh_adj.txt
```

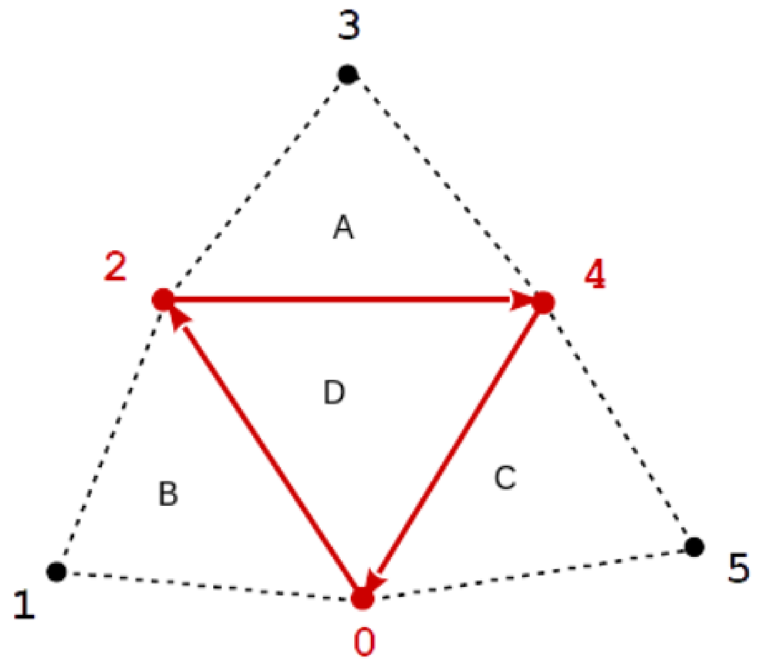
Algorithm

Our algorithm converts a simple triangle mesh, in the form of an incidence table, into a dual graph mesh structure. The input mesh is a list of vertices and a list of faces. As all faces are triangles, each face is a list of three vertex indices in the vertices list. Each vertex is a point in the plane.

The output mesh is also a list of vertices and a list of faces, but it contains more information. Each vertex has an additional field, adj_face, which is the id of a random adjacent face. Each face is a list of three elements, where each element is a pair of a vertex index and the index of the opposite face to this vertex.

To generate the dual graph mesh structure, we first create a list of vertices, which is initially the same as the input. Then, for each face, we store the list of vertices from the input and calculate the opposite face of each vertex. We determine the opposite face of each vertex in a face by finding another face in the list of faces that shares the two other vertices of the current face. If no opposite face exists, it means that this vertex is facing the border of the mesh. In this case, we store the id -1.

In the following image, the opposite face of The Vertex 2 in the central face D is the face C.



Be aware that the same vertex will have different opposite faces when present in different faces. The Vertex 2 is also present in the superior face A but, in this case, it does not have an opposite face.

We also calculate the center of each face, which is the barycenter of the triangle. The center of the faces are the actual vertices of the dual graph.

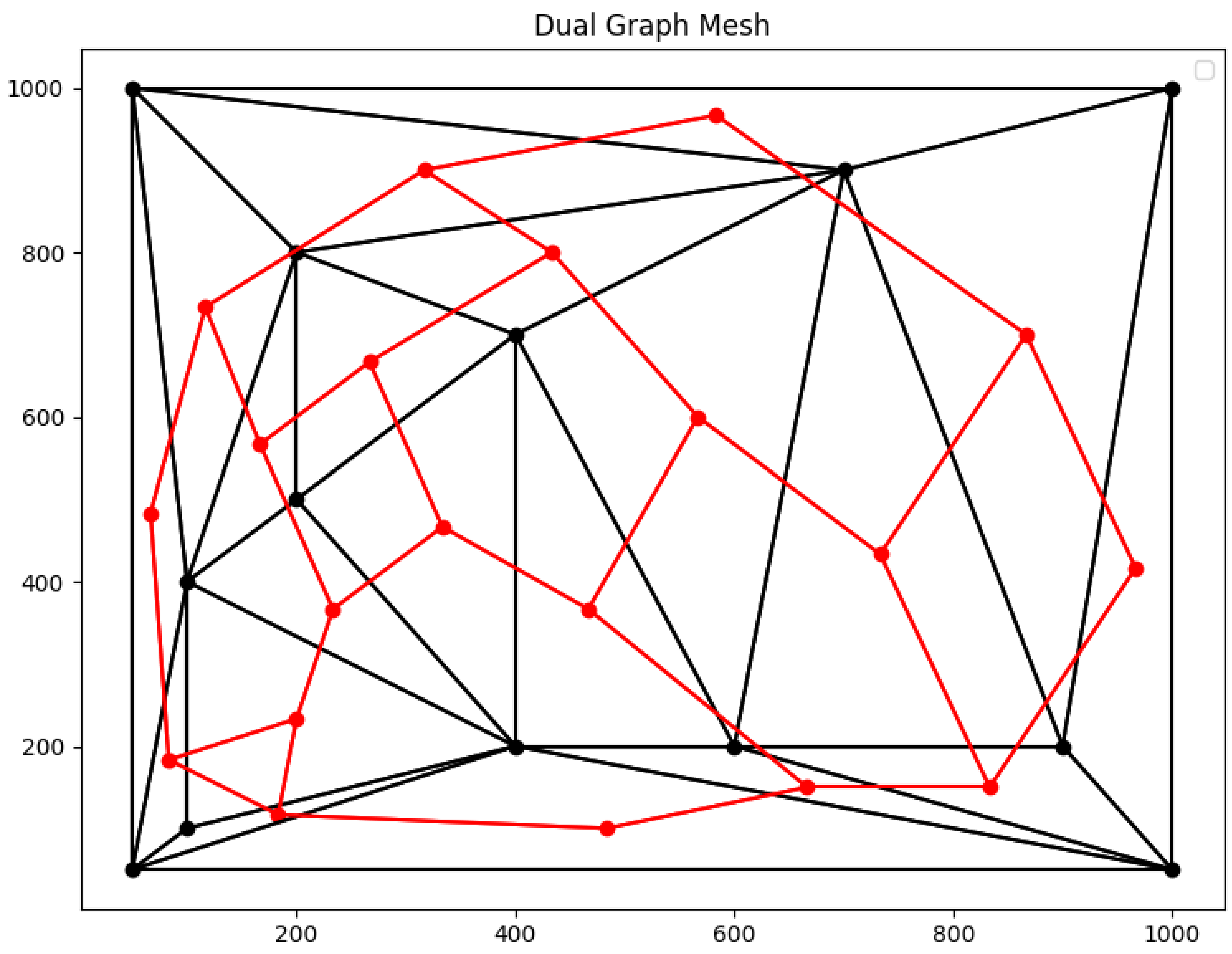
We also want to store a random adjacent face for each vertex. For this, we don't need to iterate over the vertices. Instead, when iterating over the faces, we can store the id of the current face in the adj_face field of each vertex of the face. This is a valid random adjacent face, because a face is adjacent to all of its vertices.

In the end, the algorithm only iterates over the faces of the mesh, but the process of finding the opposite face of a vertex may also iterate over all faces. This means that the algorithm is expected to be O(m^2) in the worst case, where m is the number of faces.

However, after we have the dual graph mesh structure, we can use it to consult most of the information about adjacency in O(1) time.

For example, if we want to know the opposite face of a vertex, we can just look it up in the dual graph mesh structure. We can also use the dual graph mesh structure to find the adjacent faces of a face, or the adjacent faces of a vertex, or the adjacent vertices of a vertex.

We can visualize the dual graph mesh, which is a graph where the vertices are the barycenter of the faces and the edges cross the edges that are shared by two faces in the original mesh. In the dual graph mesh, every edge represents a face-to-face adjacency in the original graph.



Black = input mesh. Red = dual graph mesh