Doubly Connected Edge List

Crossed with a KD Tree to allow for orthogonal range search

Daniel Latimer

Advisor: Dr. Bradford Nickerson

Table of Contents

[Doubly Connected Edge List (DCEL) 2](#_Toc311821673)

[Construction 2](#_Toc311821674)

[Purpose of Project 3](#_Toc311821675)

[Range Search 3](#_Toc311821676)

[References 5](#_Toc311821677)

# Doubly Connected Edge List (DCEL)

The Doubly Connected Edge List is a winged edge data structure, meant for representing planar graphs. It provides queries to obtain the edges around a face, and edges incident on a vertex in O(m) time where m is the number of edges around the face or vertex. Because it can obtain this information so quickly it is an ideal data structure for making local edits [1].

The data structure is comprised of a series of edges. Each edge is made up of six references. Two of the references lead to sets of coordinates that define the start and end points of the edge. Another two of the references lead to the faces on either side of the edge. While the last two are references to other edges, the next clockwise edge on the Start vertex, and the next clockwise edge on the End vertex.

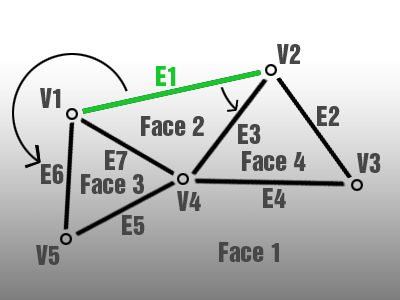
|  |  |
| --- | --- |
| Edge | |
| Start:VertexPtr | End:VertexPtr |
| Face1:Integer | Face2:Integer |
| StartNextCCWEdge:EdgePtr | EndNextCCWEdge:EdgePtr |

## Construction

The data must be set up in a specific way. The preconditions to creating a DCEL would be to have a valid set of lines that have a vertex at any point two lines cross; this would define a planar graph. We also the data set up in a way to find all edges on a vertex.

The next step would be to create a temporary data structure that is used in the construction of the DCEL. This edgeCycle datastructure holds information about the order in which edges are incident in a clockwise manner around each vertex. The structure is an array of two integers. The first of which containing a reference to the ending vertex of the particular edge around the vertex in question, the second integer is an index into the edgeCycle array of where to go to get the next clockwise edge around that vertex. It is used in conjunction with another array, edgeCycleVertexIndex, that is of size n, where n is the number of verticies. This array gives you the index into edgeCycle for the first edge of a given vertex E.g.

Once we have the edge cycles we can construct the edges of our DCEL. This is done by two functions, constructVertexCycles and constructFaceCycles. First we call constructVertexCycles. Its responsibility is to fill Edge.Start, Edge.End, Edge.StartNextCWEdge and Edge.EndNextCWEdge. It basically does this by iterating through the edgeCycle array, adding edges as it goes. It again uses the edgeCycle array to find which the next counter clockwise edge is on the start vertex and end vertex. Now that constructVertexCycles is complete we no longer need the edgeCycle and edgeCycleVertexIndex arrays.

In the example to the left we see edge 1 being processed. The edge data structure for edge 1 after running constructVertexCycles would look like this:

Edge { Start = 1, End = 2, Face1=NIL, Face2=NIL, StartNextCCWEdge=6, EndNextCCWEdge=3 }.

As you can see we still haven’t initialized Face1 or Face2. It is the responsibility of the next function, constructFaceCycles, to do so.

When we run the constructFaceCycles function we will iterate over the edges we have so far. Each time we find an edge that doesn’t have a face set we will assign a number for that face and follow the edges’ next counter clockwise edge variables to traverse all edges on that face. On each edge we encounter we will set the face number. The data structure for edge 1 of our example would look now like this:

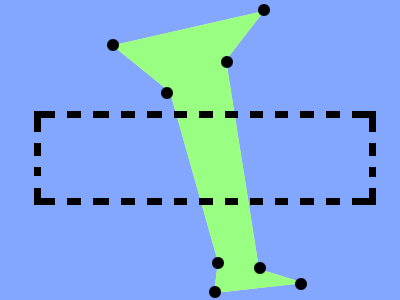
Edge { Start = 1, End = 2, Face1=1, Face2=2, StartNextCCWEdge=6, EndNextCCWEdge=3 }.

# Purpose of Project

The purpose of this project was to implement a Doubly Connected Edge List and include functionality to perform queries on it. A secondary purpose was to include another data structure to allow for orthogonal range search. A KD-Tree was determined to be the most suitable data structure to allow for this. The main idea was to implement the data structure and try querying it with different datasets to get an idea of the time it takes to perform certain queries.

# Range Search

The orthogonal range search process is performed by making use of the KD-Tree’s ability to search for points within a multi-dimensional orthogonal range. We can use this ability to find all verticies within the search range. The next step is to find all edges incident on those verticies. The DCEL provides a very efficient way to do this for a single vertex. In my trials more time was spent creating a unique list of these edges then acquiring the edges themself. We then take those edges and create a unique list of the faces on either side of each. Again, removing duplicates took longer than creating the list. With the list of faces we query the DCEL for all edges that make up each face. We then have all edges surrounding areas who have at least one vertex in the search area.

The weakness in this vertex based approach is we can miss areas that intersect our search rectangle, but do not have any verticies within it. Consider the following example to the left. We have a search rectangle that intersects both blue and green areas. However there are no verticies within the search area, so the query returns nothing. Obviously this query would be useful for some applications while not appropriate for others.

# References

[1] <http://www.holmes3d.net/graphics/dcel/>