q5

Thursday, May 02, 2024 9:15 PM

5) Write a program to fill a polygon using Scan line fill algorithm.

```
#include <iostream>
#include <vector>
#include <utility> // for pair
#include <algorithm>


#include <math.h>
#include "dda.cpp"


#include  <conio.h>
#include  <math.h>


#include <cmath>




#include <graphics.h>


using namespace std;


// Structure to represent a node in the adjacency list
struct Node {
   int vertex; // Index of the vertex
   pair<int, int> point; // Coordinates of the vertex
   Node(int v, pair<int, int> p) : vertex(v), point(p) {}
};




// Function to add an edge between vertices u and v in the adjacency
list
void addEdge(vector<vector<Node> >& adjList, int u, int v, pair<int, int>
point_u, pair<int, int> point_v) {


   adjList[u].push_back(Node(v, point_v));
   adjList[v].push_back(Node(u, point_u));

   // For undirected graph
}


class EdgeNode
{
   public:
   int vertex1; // Index of the vertex
   int vertex2;
   double x;
   double y;
   double m_inv;
   EdgeNode *ptr;


   EdgeNode(){


      this->x = 0.0;
      this->y = 0.0;
      this->m_inv = 0.0;
      ptr = NULL;
```

```
    }

 static bool compareX(const EdgeNode* a, const EdgeNode* b) {
        return a->x < b->x;
    }




};

class vertex_ptr{

    public:
        EdgeNode *ptr;

        vertex_ptr(){
            ptr = NULL;
        }
};









class array_linked_list
{
    public:
    vertex_ptr *arr;
    int s;

    array_linked_list(int n){
        arr = new vertex_ptr[n];  //
        s = n;
    }



};



void print_global_edge_table(array_linked_list *ged,int n){



    cout<<"\n\nPRINTING GLOBAL EDGE TABLE :: "<<endl;

    for(int i=1;i<=n;i++){
        cout<<"for y is  "<<i<<" "<<endl;
        cout<<"------------------------------------------------------------------------
--"<<endl;

        EdgeNode *temp = ged->arr[i].ptr ;
```

```
        while(temp  != NULL){  // reached to last node
              cout<<"for edge : "<<temp->vertex1<<" , "<<temp-
>vertex2<<" , x min : "<<temp->x<<" , y max : "<<temp->y<<" , 1/m is :
"<<temp->m_inv;
           cout<<endl;
           temp = temp->ptr;
        }
        cout<<"----------------------------------------------------------------------
-\n\n"<<endl;
    }


}









void print_edge(vector<pair<int, int> >& edges){



//cout<<"----------------edge list-----------------------------"<<endl;
    for(int i=0;i<edges.size();i++){
       // cout<<edges[i].first<<" , "<<edges[i].second<<endl;
    }
//cout<<"------------------------------------------------------------"<<endl;
}







void scan_line(vector<vector<Node> >& adjList,int n,int y,
array_linked_list *ged,  vector<pair<int, int> >& edges){




    // find all vertex which has y =4 , and than find adjacent edges



    // and store some flags that will be used to know that those edges,
has already been used







    vector <int> vertices;


        for (size_t i = 0; i < adjList.size(); ++i) {



        const Node& node = adjList[i][0];
        if(node.point.second == y){



            vertices.push_back(int(i));


        }
    }



    // now creating GLOBAL EDGE TABLE for each vertex
    // to keep the edges which are already covered;



    int flag = 0;



    // ged->arr[1] is  a pointer
```

```
// y_max will be used later

int i = y;

 // filling according to increasing y , and increasing x




       for(int j=0;j<vertices.size();j++){    // this access all vertex with y = 4,
or i=4  , or some other

              // when j = 0 than we are at 1st vertex whose y =4
              // when j = 1 than we are at 2nd vertex whose y =4

              // for each vertex  iterate its adjacent vertex
             vector<int> temp;

              for(int k= 0; k < adjList[vertices[j]].size(); k++){  // this in 1 loop
gives all adjacent to 1 vertex

                 const Node& node = adjList[vertices[j]][k];
                 // here  we have all ajacent edges releated to some vertex
j
                 // iteration is giving all vertex , wrt j


                 temp.push_back(node.vertex);  // contains the adjacent
vertexs

              }

              // now for each edge we will make a node

              // temp has all the vertex adjacent

              // vertices[j] gives vertex for which we will find edge

              int x1 = adjList[vertices[j]][0].point.first;
              int y1 = adjList[vertices[j]][0].point.second;

              // v1 is  vertices[j]

              //   v2 inside ,,  if  ( v1,v2) is found in any pair in edge  vectore
than we will skip it




              for(int p=0;p<temp.size();p++){

                 // this is v2
```

```
                int x2 = adjList[temp[p]][0].point.first;
                int y2 = adjList[temp[p]][0].point.second;


                // check for complete list of edges for already done


                for(int c=0; c<edges.size();c++){


                   if(edges[c].first == temp[p] && edges[c].second ==
vertices[j] ){
                      flag = -1;
                         break;
                   }


                    if(edges[c].first ==  vertices[j]  && edges[c].second ==
temp[p] ){
                      flag = -1;
                         break;
                   }
                }









                if(flag == -1){
                   flag =0;
                    // calculate for next edge
                   continue;
                }




             //    cout<<"working for :: (x1,x2) : ( "<<x1<<" , "<<y1<<" )"<<"
and "<<" , ( "<<x2<<" , "<<y2<<" ) "<<endl;


                int minX = (y1 < y2) ? x1 : y2;  // for x which ever has
minmum y
                int maxY = (y1 > y2) ? y1 : y2;


                if( ged->arr[i].ptr == NULL){


                   ged->arr[i].ptr = new EdgeNode();


                   ged->arr[i].ptr->x = minX;
                   ged->arr[i].ptr->y = maxY;







                   ged->arr[i].ptr->m_inv =  (x2*1.0 - x1*1.0)*1.0 / (y2*1.0 -
y1*1.0 )*1.0 ;


                ged->arr[i].ptr->vertex1 =  vertices[j];
                ged->arr[i].ptr->vertex2 =  temp[p];


                 edges.push_back(make_pair(vertices[j], temp[p]));



                }else{
```

```
                    EdgeNode *tempp = ged->arr[i].ptr ;


                    while(tempp->ptr  != NULL){  // reached to last node
                          tempp = tempp->ptr;
                    }


                    tempp->ptr = new EdgeNode();
                    tempp->ptr->x = minX;
                    tempp->ptr->y = maxY;
                    tempp->ptr->vertex1 =  vertices[j];
                    tempp->ptr->vertex2 =  temp[p];


                    tempp->ptr->m_inv =  (x2*1.0 - x1*1.0)*1.0 / (y2*1.0 -
y1*1.0 )*1.0 ;


                    edges.push_back(make_pair( vertices[j], temp[p]));


                }

            }


      }
}




void move_edges(int y_min, vector<vector<Node> >& adjList,
vector<EdgeNode *>& active_edges , array_linked_list *ged ){




//cout<<"=============================================
==============\n";
      EdgeNode *temp = ged->arr[y_min].ptr ;


      while(temp  != NULL){  // reached to last node
        //     cout<<temp->vertex1<<" , "<<temp->vertex2<<endl;
            active_edges.push_back(temp);
         temp = temp->ptr;
      }
//cout<<"=============================================
================\n";




}


void remove_from_active_edge(vector<EdgeNode *>& active_edges,int
y){


   for(int i=0;i< active_edges.size();i++){
     // cout<<"v1 : "<<active_edges[i]->vertex1<<" , "<<active_edges[i]-
>vertex2<<endl;


      if(active_edges[i]->y == y){
         // remove it
       //  cout<<"remove it : "<<endl;
            int indexToRemove = i;
```

```
                if (indexToRemove >= 0 && indexToRemove <
active_edges.size()) {
                    active_edges.erase(active_edges.begin() +
indexToRemove);
                } else {
                  // std::cout << "Index out of range" << std::endl;
                }
        }
    }

}




void print_active_edge_table(vector<EdgeNode *>& active_edges ){



cout<<"-----------------------------------------ACTIVE EDGE TABLE-------
----------------------------------------\n"<<endl;
    for(int i=0;i<active_edges.size();i++){
        cout<<"v1 : "<<active_edges[i]->vertex1<<" , v2 is :  "
<<active_edges[i]->vertex2<<endl;
    }
cout<<"-----------------------------------------------------------------------------
-----------------------------\n"<<endl;



}


void make_pair_and_print(vector<EdgeNode *>& active_edges,int y ){


    int x1 = 0;
    int y1 = y;


    int x2 = 0;
    int y2 = y;




    for(int i =0 ;i<active_edges.size(); i +=2){


            x1 = active_edges[i]->x;
            x2 = active_edges[i+1]->x;


          cout<<"pair : "<<x1<<" , "<<y1<<"   and  "<<x2<<"  ,  "
<<y2<<endl;


            dda(x1,y1,x2,y2,GREEN);


    }
}


void increment_x(vector<EdgeNode *>& active_edges){


        for(int i =0 ;i<active_edges.size(); i++){


            if(!isinf(active_edges[i]->m_inv)){
                active_edges[i]->x = active_edges[i]->x + active_edges[i]-
>m_inv ;
            }


        }
```

```
}




void scan_algo(array_linked_list *ged,int y_min, int y_max,
vector<vector<Node> >& adjList){


    int y = y_min;
    vector<EdgeNode *> active_edges ;  // active edge table is empty


    int get = 5;
    int aet = 0;




    while(y <= y_max){   // repeat until aet and get is empty


        //move from et to aet those y_min = y
        move_edges(y,adjList, active_edges,ged);
        // remove those has y = y_max


        remove_from_active_edge(active_edges,y);


        // sort aet on x basis
      std::sort(active_edges.begin(), active_edges.end(),
EdgeNode::compareX);


      //  cout<<"sorted list :============================\n";



       // cout<<"=================check if there is any changes in
GLOBAL EDGE TABLE==============\n";
      // print_global_edge_table(ged,y_max);


        // make pairs from aet using y


      //  print_active_edge_table(active_edges);
        make_pair_and_print(active_edges,y);



        y +=1;

        // increment with slope


        if(y< y_max){
            increment_x(active_edges);
        }else{
            break;
        }




    }
```

```
}

int main() {

    int gd = DETECT, gm;
    char pathtodriver[] = "";
    initgraph(&gd, &gm, pathtodriver);

    int numVertices = 6;

    // Initialize adjacency list
    vector<vector<Node> > adjList(numVertices);

    // Store points associated with each vertex
    vector<pair<int, int> > points;

    int y_min = 100;
    int y_max = 400;
    points.push_back(make_pair(200,100));
    points.push_back(make_pair(50,300));
    points.push_back(make_pair(80,400));

    points.push_back(make_pair(200,350));
    points.push_back(make_pair(300,400));
    points.push_back(make_pair(350,310));

    // Add some edges
    addEdge(adjList, 1, 0, points[0], points[1]);
    addEdge(adjList, 2, 1, points[1], points[2]);
    addEdge(adjList, 3, 2, points[2], points[3]);
    addEdge(adjList, 4, 3, points[3], points[4]);
    addEdge(adjList, 5, 4, points[4], points[5]);
    addEdge(adjList, 0, 5, points[5], points[0]);

  vector<pair<int, int> > edges;

    array_linked_list *ged =  new array_linked_list(y_max+1);
```

```
  for(int i= y_min ;i <= y_max ;i++){
       scan_line(adjList,numVertices,i,ged,edges);
  }

//print_global_edge_table(ged,y_max);



scan_algo(ged,y_min,y_max,adjList);


cout<<"done"<<endl;



getch();
closegraph();









    return 0;
}
```