University of Louisville Instructor **Dr. Aly A. Farag**

Electrical and Computer Engineering **Summer 2009**

ECE 600: Introduction to Shape Analysis

Lab #3 – Polygon Triangulation

(Assigned Thursday 6/4/09 – Due Thursday 6/11/09)

In this lab, we will discuss the details of implementing polygon triangulation by ear removal, refer to lecture notes for more theoretical details. We will start with a few representation issues.

Table of Contents

Table of Code Snippets

1. Representation Issues

1.1 Representation of a Point

We will use our Point2D class to represent a point in a polygon, each point will have its xy-coordinates.

1.2 Representation of a Polygon

In previous labs, we implemented a polygon as a doubly-linked list whose data stored in Point2D instances, however for our purposes now, we need to define a new class called Vertex2D which will be used to define the basic building block of a polygon. It will contain (1) *vertexID* which is a number assigned to the vertex inside a polygon where vertices are ordered/labeled in a counterclockwise orientation. (2) *point* (of type Point2D) which maintains the coordinates of the point maintained by this vertex (3) *isEar* which is the ear status of the current vertex, it is true if and only if the current vertex is an ear within its polygon and (4) *next, prev* which are the next and previous vertices respectively. The following code snippet shows the Vertex2D class.

Code 1 - Vertex2D class

```
classdef Vertex2D < handle
    % in this class we will define the basic building block of a polygon

    properties
        vertexID    % index or id number assigned to the current vertex
        % which reveals its order within the polygon
        point       % point coordinates for this vertex
        isEar       % ear status, true if this vertex is an ear
        next        % next vertex
        prev        % previous vertex
    end

    methods
        %% constructor
        function obj = Vertex2D(vertexID,point,prev,next,isEar)
            switch nargin,
                case 0,
                    obj.vertexID = -1;
                    obj.point = [];
                    obj.isEar = 0;
                    obj.next = [];
                    obj.prev = [];
                case 1,
                    obj.vertexID = vertexID;
                    obj.point = Point2D(0,0);
                    obj.isEar = 0;
                    obj.next = [];
                    obj.prev = [];
                case 2,
                    obj.vertexID = vertexID;
                    obj.point = point; % it is of type Point2D
                    obj.isEar = 0;
                    obj.next = [];
                    obj.prev = [];
                case 3,
                    obj.vertexID = vertexID;
                    obj.point = point; % it is of type Point2D
                    obj.isEar = 0;
                    obj.next = [];
                    obj.prev = prev;
                case 4,
                    obj.vertexID = vertexID;
                    obj.point = point; % it is of type Point2D
                    obj.isEar = 0;
                    obj.next = next;
                    obj.prev = prev;
                case 5,
                    obj.vertexID = vertexID;
                    obj.point = point; % it is of type Point2D
                    obj.isEar = isEar;
                    obj.next = next;
                    obj.prev = prev;
            end
        end

        %% get/set access methods
        function vertexID = get.vertexID (obj)
            vertexID = obj.vertexID;
        end

        function obj = set.vertexID(obj,vertexID)
            obj.vertexID = vertexID;
        end

        function point = get.point (obj)
            point = obj.point;
        end

        function obj = set.point(obj,point)
            obj.point = point;
        end

        function isEar = get.isEar (obj)
            isEar = obj.isEar;
        end

        function obj = set.isEar(obj,isEar)
            obj.isEar = isEar;
        end
        function next = get.next (obj)
            next = obj.next;
        end

        function obj = set.next(obj,next)
```

```
            obj.next = next;
        end

        function prev = get.prev (obj)
            prev = obj.prev;
        end

        function obj = set.prev(obj,prev)
            obj.prev = prev;
        end

        %% functions related to doublely linked nodes
        function insertAfter(newNode, nodeBefore)
            % insertAfter  Inserts newNode after nodeBefore.
            disconnect(newNode);
            newNode.next = nodeBefore.next;
            newNode.prev = nodeBefore;
            if ~isempty(nodeBefore.next)
                nodeBefore.next.prev = newNode;
            end
            nodeBefore.next = newNode;
        end

        function insertBefore(newNode, nodeAfter)
            % insertBefore  Inserts newNode before nodeAfter.
            disconnect(newNode);
            newNode.next = nodeAfter;
            newNode.prev = nodeAfter.prev;
            if ~isempty(nodeAfter.prev)
                nodeAfter.prev.next = newNode;
            end
            nodeAfter.prev = newNode;
        end
        function disconnect(node)
            % DISCONNECT  Removes a node from a linked list.
            % The node can be reconnected or moved to a different list.
            prev = node.prev;
            next = node.next;
            if ~isempty(prev)
                prev.next = next;
            end
            if ~isempty(next)
                next.prev = prev;
            end
            node.next = [];
            node.prev = [];
        end

        function head = getHead(node)
            head =[];
            while 1
                if node.isHead()
                    head = node;
                    break;
                end
                node = node.prev;
            end
        end

        function tail = getTail(node)
            node = node.gotoHead();
            tail = [];
            while 1
                if node.isTail()
                    tail = node;
                    break;
                end

                node = node.next;
            end
        end

        function node = gotoHead(node)
            while 1
                if node.isHead()
                    break;
                end

                node = node.prev;
            end
        end

        function ishead = isHead(node)
            if node.vertexID == 0
                ishead = 1;
            else
                ishead = 0;
            end
        end

        function istail = isTail(node)
            n = getNoOfVertices(node);
            if node.vertexID == (n-1)
                istail = 1;
            else
                istail = 0;
            end
        end

        function n = getNoOfVertices(node)
            node = node.gotoHead();
            n = 0;
            while 1
                n = n+1;
                node = node.next;
                if node.isHead()
                    break;
                end
            end
        end

        function TF = eq(v1,v2)
            if (v1.vertexID == v2.vertexID)
                TF = 1;
            else
                TF = 0;
            end
        end

        function disp(vertex)
            % DISP  Displays a link vertex.
            disp('Doubly-linked list vertex with data:')
            disp('vertexID      X      Y       isEar      Next      Prev');
            disp([ num2str(vertex.vertexID)  '        ' ...
                   num2str(vertex.point.x) ...
                   '        '  num2str(vertex.point.y) '         '...
                   num2str(vertex.isEar) ...
                   '         ' num2str(vertex.next.vertexID) ...
                   '         ' num2str(vertex.prev.vertexID)]);
        end
    end
end
```

Now, we can use this new class as the node-structure to build up our polygon. The following code snippet shows how to construct this list from user input.

Code 2 – How to construct the linked list which maintains the vertices of the polygon

```
x = [0 10 12 20 13 10 12 14 8 6 10 7 0 1 3 5 -2 5];
y = [0 7 3 8 17 12 14 9 10 14 15 18 16 13 15 8 9 5];
starting_point = Point2D(x(1),y(1));

figure
starting_point.draw('DrawHandle',gca,...
    'ColorMarkerStyle','r*:','LineWidth',2);
hold on

vertices = Vertex2D(0,starting_point);
for i = 2 : length(x)
    curPoint = Point2D(x(i),y(i));
    curPoint.draw('DrawHandle',gca,...
        'ColorMarkerStyle','r*:','LineWidth',2);
    hold on

    curVertex = Vertex2D(i-1,curPoint);
    curVertex.insertAfter(vertices);
    vertices = curVertex;
end

vertices.next = getHead(vertices);
vertices = vertices.gotoHead();
```

```
vertices.prev = getTail(vertices);

% number of points
n = vertices.getNoOfVertices();
```

Task 0: Extend your GUI to allow the user interactively select polygon vertices instead of having fixed xy arrays as given in Code(2).

At all times, a variable *vertices* is maintained which points to some vertex of type Vertex2D. This will serve as the "head" of the list during iterative processing. Loops over all vertices will take the form shown in the following code snipped. Care must be exercised if the processing in the loop deletes the cell to which vertex points.

```
% starting from the beginning
vertices = vertices.gotoHead();
while 1
    % processing
    vertices.point.draw('DrawHandle',haxes,...
        'ColorMarkerStyle','b*:','LineWidth',2);
    pause (0.5);

    % go to the next vertex
    vertices = vertices.next;

    % exit if you returned to the begining vertex (head of the list)
    if vertices.isHead()
        break
    end
end
```

1.3 Polygon Definition

We can now construct another class called Polygon2D inorder to encapsulate main functionalities which will be detailed in the following subsections. This class will have the *vertices* variable which will point to the first vertex of the polygon.

We will use the convention of listing the vertices of a polygon in a counterclockwise order such that if you walked along the boundary of the polygon visiting the vertices in that order, the interior of the polygon would be always to your left.

Code 3 - Polygon2D class

```
classdef Polygon2D
    % in this class we will define basic tools for polygons in 2D
    properties
        vertices = Vertex2D();
    end
    methods
        %% the constructor - called when you create an instance of this
        % class
        function obj = Polygon2D(vertices)
            if nargin >0
                obj.vertices = vertices;

            else
                obj.vertices = Vertex2D();
            end
        end

        %% get access methods
        function vertices = get.vertices (obj)
            vertices = obj.vertices;
        end

        %% set access methods
        function obj = set.vertices(obj,vertices)
            obj.vertices = vertices;
        end

        %% drawing a polygon
        function draw(obj, varargin)
           % getting the line points in a suitable structure for display
            current_node = obj.vertices.getHead();
            while (1)
                next_node = current_node.next;
                line = Line2D(current_node.point,next_node.point);
                line.draw(varargin{:});
                current_node = next_node;

                if current_node.isHead()
                    break;
                end
            end
        end

        %% transform the polygon
        function obj = transform(obj,mat,aboutpoint)
            transformed_shape = obj;

            line_node1 = obj.vertices.point - aboutpoint;
            line_node1 = line_node1.transform(mat);
            transformed_shape.vertices.point = line_node1;
            current_node = transformed_shape.vertices;
            while (1)
                next_node = current_node.next;
                line_node2 = next_node.point - aboutpoint;
                line_node2 = line_node2.transform(mat);
                next_node.point = line_node2;
                current_node = next_node;
                if current_node.isHead()
                    break;
                end
            end
            obj = transformed_shape;

        end

        %% getting the mid (centroid) polygon shape
        function centroidpoint = getMidPoint(obj)
            centroidpoint = Point2D();
            current_node = obj.vertices;
            centroidpoint.x = current_node.point.x;
            centroidpoint.y = current_node.point.y;
            node_count = 1;
            while (1)
                node_count = node_count + 1;
                next_node = current_node.next;
                centroidpoint.x = centroidpoint.x + next_node.point.x;
                centroidpoint.y = centroidpoint.x + next_node.oint.y;
                current_node = next_node;

                if current_node.isHead()
                    break;
                end
            end

            centroidpoint.x = centroidpoint.x /node_count;
            centroidpoint.y = centroidpoint.y /node_count;
        end
    end
end
```

1.4 Code for Area

Computing the area of a polygon is now a straightforward implementation of the following theorem (Refer to lecture notes for more theoretical details).

Theorem 7 - Area of Polygon: *Let* $\mathcal{P}$ *be a simple polygon (convex or nonconvex), having vertices* $v_0, v_1, \dots, v_{n-1}$ *labeled counterclockwise, and let* p *be any point in the plane, then*

$$\mathcal{A}(\mathcal{P}) = \mathcal{A}(p, v_0, v_1) + \mathcal{A}(p, v_1, v_2) + \mathcal{A}(p, v_2, v_3) + \cdots + \mathcal{A}(p, v_{n-2}, v_{n-1}) + \mathcal{A}(p, v_{n-1}, v_0)$$

Selecting the first vertex in the polgon to be p, we will have two functions, the first one computes the area of a triangle given its vertices, i.e computing $\mathcal{A}(p, v_i, v_{i+1})$ and the second one is summing these areas up to get the area of the whole polygon, i.e. $\mathcal{A}(\mathcal{P})$.

1.4.1 Area of Triangle

Using the determinant form, we have the following lemma,

Lemma 5: *Twice the area of a triangle* $t = (a, b, c)$ *is given by,*

$$2\mathcal{A}(a,b,c) = \begin{vmatrix} a_0 & a_1 & 1 \\ b_0 & b_1 & 1 \\ c_0 & c_1 & 1 \end{vmatrix} = (b_0 - a_0)(c_1 - a_1) - (c_0 - a_0)(b_1 - a_1)$$

Re-write the preceeding formula to compute $\mathcal{A}(p, v_i, v_{i+1})$ we will obtain,

$$2\mathcal{A}(p, v_i, v_{i+1}) = \begin{vmatrix} p.x & p.y & 1 \\ v_i.x & v_i.y & 1 \\ v_{i+1}.x & v_{i+1}.y & 1 \end{vmatrix}$$
$$= (v_i.x - p.x)(v_{i+1}.y - p.y) - (v_{i+1}.x - p.x)(v_i.y - p.y)$$

Now, we can add the following function to our Polygon2D class, refer to the following code snippet.

Code 4 - Area of triangle

```
classdef Polygon2D
    % in this class we will define basic tools for polygons in 2D
    properties
        vertices = Vertex2D();
    end

    methods
        %%
        % other functions here
        %%
    end

    % static functions
    methods (Static)
        % area of a triangle given its vertices
        function A = getTriangleArea(p,vcur,vnext)
            % p,vcur,vnext are polygon vertices
            p     = p.point;
            vcur  = vcur.point;
            vnext = vnext.point;
            A = (1/2) *((vcur.x - p.x)*(vnext.y - p.y) - …
                        (vnext.x - p.x)*(vcur.y - p.y));
        end
    end
 end
```

1.4.2 Area of Polygon

Now, we are ready to write the function which will go over all the vertices in a given polygon and compute its area using its first vertex as the point *p*.

Code 5 - Area of polygon

```
classdef Polygon2D
    % in this class we will define basic tools for polygons in 2D
    properties
        vertices = Vertex2D();
    end

    methods
        %%
        % other functions here
        %%

        % area of the whole polygon
        function A = getArea(obj)
            % initialization
            A = 0;

            % we assume that our p will be the first vertex in the polygon
            p = obj.vertices;

            % now the current vertex will be the next one
            vcur = p.next;
            while(1)
                vnext = vcur.next;
                A = A + obj.getTriangleArea(p,vcur,vnext);
                vcur = vcur.next;

                if vcur.next == obj.vertices % we reached to where we started
                    break;
                end
            end
        end
    end
end
```

Task 1 : report your results when you select a polygon with counterclockwise and clockwise orientations, you might start with a simple case just one triangle and see what happens.

2. Segment Intersection

We still have one further step to be able to develop an algorithm to triangulate a given polygon, in this subsection we will discuss how can we detect an intersection between two given segments.

2.1 Left Predicate

Checking whether two segments intersect can be established by determining whether or not a point is to the left of a directed line.

Two points given a particular order (a, b) determine a directed line moving from the first point a to the second point b. If another point c is to the left of this directed line, then the area of the counterclockwise triangle, $\mathcal{A}(a, b, c)$, is positive. Therefore we may implement the Left predicate by a single call to getTriangleArea( ).

When c is collinear with ab? Then the determined triangle has zero area. Since it will be useful to distinguish collinearity, we write a separate *Collinear* predicate for this, as well as *LeftOn* predicate, giving us the equivalent of =, <, and <. Again we are adding to our Polygon2D class, refer to the following code snippet.

Code 6 – Left, LeftOn and Collinear Predicates

```
classdef Polygon2D
    % in this class we will define basic tools for polygons in 2D
    properties
        vertices = Vertex2D();
    end

    methods
        %%
        % other functions here
        %%
    end

    % static functions
    methods (Static)
        %%
        % other functions here
        %%

        % determining whether a vertex c lies on the left of a directed
        % line connecting vertex a and vertex b
        function isLeft = Left(a,b,c)
            A = Polygon2D.getTriangleArea(a,b,c);
            if A > 0
                isLeft = 1 ;
            else
                isLeft = 0;
            end
        end

        % determining whether a vertex c lies on the left of or on a directed
        % line connecting vertex a and vertex b
        function isLeftOn =  LeftOn (a,b,c)
            A = Polygon2D.getTriangleArea(a,b,c);
            if A >= 0
```

```
                isLeftOn = 1 ;
            else
                isLeftOn = 0;
            end
        end

        % determining whether a vertex c is colliner with ab
        function isCollinear =  Collinear (a,b,c)
            A = Polygon2D.getTriangleArea(a,b,c);
            if A == 0
                isCollinear = 1 ;
            else
                isCollinear = 0;
            end
        end
    end
 end
```

2.2 Proper Intersection

If two segments ab and cd intersect in their interiors, then c and d are split by the line L_1 containing the segment ab. And likewise, a and b are split by the line L_2 containing the segment cd. Note that neither of these conditions alone is sufficient to guarantee intersection, however we should make sure first that we do not have the case where three of the four endpoints are collinear. This is referred to as *proper intersection* where we force non-collinearity when two segments intersect at a point interior to both.

Let's add this function to our Polygon2D class,

Code 7 – properIntersection predicate

```
classdef Polygon2D
    % in this class we will define basic tools for polygons in 2D
    properties
        vertices = Vertex2D();
    end

    methods
        %%
        % other functions here
        %%
    end

    % static functions
    methods (Static)
        %%
        % other functions here
        %%

        % checking whether two segments ab and cd are properly intersected,
        % that is they intersect in their interior without having three of
        % the end point being collinear
        function isIntersect = properIntersection(a,b,c,d)
            if Polygon2D.Collinear(a,b,c)||Polygon2D.Collinear(a,b,d)|| ...
               Polygon2D.Collinear(c,d,a)|| Polygon2D.Collinear(c,d,b)

                isIntersect = 0;
            else
                % we will xor - true if its arguments are different ...
                isIntersect= xor(Polygon2D.Left(a,b,c),Polygon2D.Left(a,b,d))
                             && ...
                           xor(Polygon2D.Left(c,d,a),Polygon2D.Left(c,d,b));
            end
        end
    end
 end
```

There is unfortunate redundancy in this code, in that the four relevant triangle areas are being computed twice each. This redundancy could be removed by computing the areas and storing them in local variables. I would argue against storing the areas, as then the code would not be transparent. I prefer to sacrifice efficiency for clarity and leave properInterection as is. In this instance, properInterection is precisely the function needed to compute clear visibility.

2.3 Improper Intersection

Now we should deal with the special case of *improper intersection* between two segments, this occurs when an endpoint of one segment lies somewhere on the other segment, this can only happen if there points are collinear, however collinearity is not a sufficient condition. Hence what we need is to decide if an endpoint of a segment lies between the endpoints of the other segment.

If the point *c* is known to be collinear with *a* and *b,* the betweenness check can proceed as follows; If *ab* is not vertical, then *c* lies on *ab* if and only if the *x* coordinate of *c* falls in the interval determined by the *x* coordinates of *a* and *b*. If *ab* is vertical, then a similar check on *y* coordinates determines betweenness.

Let's add this function to our Polygon2D class,

Code 8 – Between predicate

```
classdef Polygon2D
    % in this class we will define basic tools for polygons in 2D
    properties
        vertices = Vertex2D();
    end

    methods
        %%
        % other functions here
        %%
    end

    % static functions
    methods (Static)
        %%
        % other functions here
        %%

        % checking whehter a point c lies between two point a and b
        function isBetween = Between (a,b,c)
            if ~Polygon2D.Collinear(a,b,c)
                % if they are not collinear, no way c will be between a and b
                isBetween = 0;
            else
                % now if ab is not vertical, check betweenness on x, else
                % on y
                a = a.point;
                b = b.point;
                c = c.point;
                if a.x ~= b.x
                    isBetween = ((a.x <= c.x)&&(c.x <= b.x))|| ...
                                ((a.x >= c.x)&&(c.x >= b.x));
                else
                    isBetween = ((a.y <= c.y)&&(c.y <= b.y))|| ...
                                ((a.y >= c.y)&&(c.y >= b.y));
                end
            end
        end
    end
end
```

2.4 Segment Intersect

We finally can present code for computing segment intersection. Two segments intersect iff they intersect properly or one endpoint of one segment lies between the two endpoints of the other segment. The check for improper intersection is therefore implemented by four calls to Between, refer to the following code snippet.

Code 9 – Intersect predicate

```
classdef Polygon2D
    % in this class we will define basic tools for polygons in 2D
    properties
        vertices = Vertex2D();
    end

    methods
        %%
        % other functions here
        %%
    end

    % static functions
    methods (Static)
        %%
        % other functions here
        %%

        % now checking whether two segments (ab and cd) do intersect or not
        function isIntersect = Intersect (a,b,c,d)
            if Polygon2D.properIntersection(a,b,c,d)
                isIntersect = 1;
            else
                if Polygon2D.Between(a,b,c)|| Polygon2D.Between(a,b,d)|| ...
                   Polygon2D.Between(c,d,a)||Polygon2D.Between(c,d,b)

                    isIntersect = 1;
                else
                    isIntersect = 0;
                end
            end
        end
    end
end
```

3. Triangulation

Having developed segment intersection code, we are nearly prepared to write code for triangulating a polygon.

3.1 Diagonals

In order to perform polygon triangulation, we need first to know how to find a diagonal of the given polygon. Recall that diagonals are characterized by two main conditions: (1) non-intersection with polygon edges and (2) being interior to the polygon.

If we ignore the second condition, finding a diagonal will be straightforward: Consider a potential diagonal s connecting between a pair of polygon vertices v_i and v_j, for every edge e of the polygon $\mathcal{P}$ not incident to either v_i or v_j, check if e intersect s, as soon as an intersection is detected, s will be declared not to be a diagonal, if no such edge intersects s, then s might be a diagonal, since we have already ignored the second condition, we should check whether it is interior or exterior to the polygon.

The following code snippet checks whether a segment joining two vertices may or may not be a diagonal, checking whether it is internal or external to the polygon will be deferred to subsequent section.

One more function in our Polygon2D class, Code 10 – maybeDiagonal predicate

```
classdef Polygon2D
    % in this class we will define basic tools for polygons in 2D
    properties
        vertices = Vertex2D();
    end

    methods
        %%
        % other functions here
        %%

        % given two vertices, lets check whether it is a candidate diagonal
        % or not, here we only check that it doesn't intersect the boundary
        % of the polygon
        function isDiagonal = maybeDiagonal(obj,vi,vj)
            % lets start for the beginning of the polygon vertices and
            % start checking out whether the given segment vivj intersects
            % the boundary of the polygon
            vcur = obj.vertices;

            while(1)
                vnext = vcur.next;
                % skip edges incident to vi or vj
                if (vcur.point ~= vi.point)&&(vnext.point ~= vi.point)&& ...
                        (vcur.point~=vj.point)&&(vnext.point~=vj.point)&&...
                        obj.Intersect(vi,vj,vcur,vnext)

                    isDiagonal = 0;
                    return;
                end

                vcur = vcur.next;
                if vcur == obj.vertices % we reached to where we started
                    break;
                end
            end

            % if we got here, there is not intersection with any edge
            isDiagonal = 1;
            return;
        end
    end
end
```

3.2 InCone Predicate

Now it is time to check whether a candidate diagonal is really a diagonal, that is it is interior to the given polygon. Refer to lecture notes for more theoretical details.

InCone determines if one vector *B* lies strictly in the open cone counterclockwise between two other vectors *A* and C. The latter two vectors will lie along two consecutive edges of the polygon, and *B* lies along the diagonal. Such a procedure will suffice to determine diagonals.

Code 11 – InCone predicate

```
classdef Polygon2D
    % in this class we will define basic tools for polygons in 2D
    properties
        vertices = Vertex2D();
    end

    methods
        %%
        % other functions here
        %%
    end

    % static functions
    methods (Static)
        %%
        % other functions here
        %%

        %InCone determines if one vector B lies strictly in the open cone
        %counterclockwise between two other vectors A and C. The latter two
        %vectors will lie along two consecutive edges of the polygon, and B
        %lies along the diagonal. Such a procedure will suffice to
        %determine diagonals.
        function isInCone = InCone(a,b)
            % a and b are two polygon vertices
            % getting the neighboring vertices to a
            a_plus  = a.next;
            a_minus = a.prev;

            % if a is a convex vertex ...
            if Polygon2D.LeftOn(a,a_plus,a_minus)
                isInCone = Polygon2D.Left(a,b,a_minus) && ...
                           Polygon2D.Left(b,a,a_plus);
            else % a is reflex
                isInCone = ~(Polygon2D.LeftOn(a,b,a_plus) && ...
                             Polygon2D.LeftOn(b,a,a_minus));
            end
        end
    end
 end
```

3.3 Diagonal Predicate

We now have developed code to determine if *ab* is a diagonal:

iff maybeDiagonal(a,b) , InCone (a,b) , and InCone (b,a) are true.

The InCone calls serve both to ensure that *ab* is internal and to cover the edges incident to the endpoints not examined in maybeDiagonal.

Code 12 – Diagonal predicate

```
classdef Polygon2D
    % in this class we will define basic tools for polygons in 2D
    properties
        vertices = Vertex2D();
    end

    methods
        %%
        % other functions here
        %%
        % given two vertices, determine whether it is a diagonal
        function isDiagonal = Diagonal(obj,a,b)
            isDiagonal = Polygon2D.InCone(a,b) && Polygon2D.InCone(b,a) ...
                         && obj.maybeDiagonal(a,b);
        end
    end
end
```

Task 2: *properIntersection* Detail exactly what properIntersection (Code 7) computes if the *if*-statement is deleted. Argue that after this deletion, Intersect (Code 9) still works properly.

Task 3: *Inefficiencies in Intersect.* Trace out (by hand) Intersect (Code 9) and determine the largest number of calls to getTriangleArea (Code 4) it might induce. Design a new version that avoids duplicate calls.

Task 4: *Saving intersection information.* Work out a scheme to avoid testing the same two segments for intersection twice. Analyze the time and space complexity of the new algorithm.

Task 5: *InCone improvement.* Prove that *ab* is in the cone at *a* iff at most one of these three Lefts are false: Left(a, a^+, *b)*, Left(a, *b*, a^-*)*, Left(a, a^-, a^+*)*.

Task 6: *Diagonal improvement.* Prove that either one of the two calls to InCone in Diagonal (Code 12) can be removed without changing the result.

3.4 Triangulation by Ear Removal

We are now prepared to develop code for finding a triangulation of a polygon. The algorithm can be summarized as follows;

Algorithm: Triangulate via ear removal

```
(1) Initialize the ear tip status of each vertex
(2) While $n > 3$ do
        a. Locate an ear tip $v_2$ where $(v_1, v_3)$ is a diagonal.
        b. Update the ear tip status of $v_1$ and $v_3$ where $v_1$ is an ear if $v_0 v_3$ is a diagonal and $v_3$ is an ear if $v_1 v_4$ is a diagonal.
        c. Cut of the ear $v_2$.
```

The first task is to initialize the ear status isEar that is a part of the vertex structure (Code 1). This is accomplished by one call to Diagonal per vertex. See the following code snippet. In order to store the triangulation, we will construct two new members in the Polygon2D class, the first one is vertexList which is a 2D matrix with number of rows equal to number of vertices and each column maintains a coordinate, i.e. first column is x-coordinate and second column is y-coordinate, the second member is triangleList which is also a 2D matrix with number of rows equal to number of triangles resulted from the triangulation, which is n-2 where n is the number of vertices of the polygon, triangleList will have three columns which will maintain the indices of the vertices in the vertexList which make up each triangle, i.e. the first column will have the index of the first vertex in a triangle and same for the second and third column, order in counterclockwise manner. See the following figure for illustration. Two other members should be added to Vertex2D class, isAdded which is a flag to indicate whether a vertex is added to the vertexList, and index_in_vertexList which is the index of a previously added vertex in the vertexList.

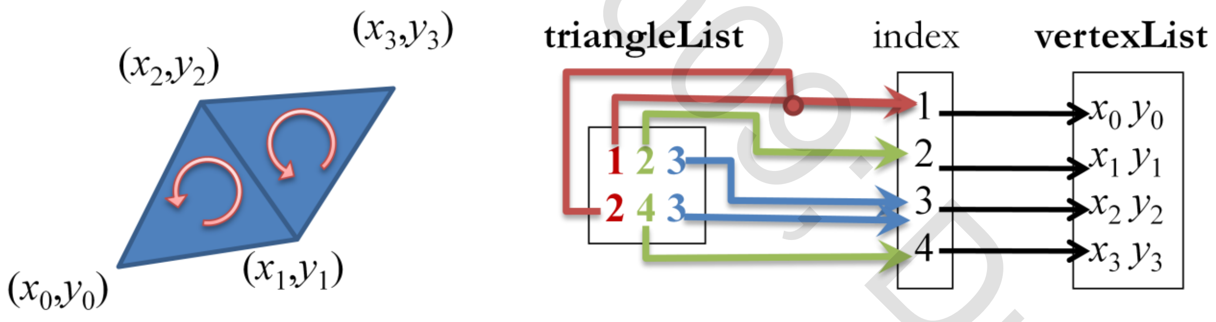

Figure 1 – Illustration of triangulation data structure

Code 13 – EarInit

```
classdef Polygon2D
    % in this class we will define basic tools for polygons in 2D
    properties
        vertices = Vertex2D();
        vertexList   = [];
        triangleList = [];
    end

    methods
        %%
        % other functions here
        %%

        % initialize the ear status of each vertex in the polygon
        function obj = EarInit(obj)
            % start from the begining
            v1 = obj.vertices;
            while(1)
                % get the neighboring vertices of v1
                v2 = v1.next;
                v0 = v1.prev;

                v1.isEar = obj.Diagonal(v0,v2);
                v1 = v1.next;

                if v1 == obj.vertices % we reached to where we started
                    break;
                end
            end

            % now we end the loop when we go to the first vertex
            % lets update the given polygon
            obj.vertices = v1;
        end
    end
end
```

The main Triangulate code consists of a double loop. The outer loop removes one ear per iteration, halting when $n = 3$. The inner loop searches for an ear by checking the precomputed *v2.isEar* flag, where *v2* is the potential ear tip. Once an ear tip is found, the ear status of *v1* and *v3* are updated by calls to *Diagonal*, the diagonal representing the base of the ear is displayed, and the ear is removed from the polygon. This removal is accomplished by rewiring the next and prev pointers for *v1* and *v3*.

Code 14 – Triangulate

```
classdef Polygon2D
    % in this class we will define basic tools for polygons in 2D
    properties
        vertices = Vertex2D();
        vertexList   = [];
        triangleList = [];
    end

    methods (Static)
        %%
        % other functions here
        %%
        function drawDiagonal(vi,vj,varargin)
            % getting the line points in a suitable structure for display
            line = Line2D(vi.point,vj.point);
            line.draw(varargin{:});
        end
    end

    methods
        %%
        % other functions here
        %%

        % make a clone of a given polygon in the memory
        function clone_obj = Clone(obj)
            clone_obj = Polygon2D();
            curVertex = obj.vertices;
            clone_obj.vertices = curVertex;
            while(1)
                curVertex = curVertex.next;
                if curVertex == obj.vertices % we reached to where we started
                    break;
                end

                curVertex.insertAfter(clone_obj.vertices);
                clone_obj.vertices = curVertex;
            end
            clone_obj.vertices.next = getHead(clone_obj.vertices);
            clone_obj.vertices = clone_obj.vertices.gotoHead();
            clone_obj.vertices.prev = getTail(clone_obj.vertices);
        end

        function [obj,verticesID] = add2vertexList(obj,curVertices)
            verticesID = [];
            for i = 1 : length(curVertices)
                if ~curVertices{i}.isAdded
                    x = curVertices{i}.point.x;
                    y = curVertices{i}.point.y;
                    obj.vertexList(end+1,:) = [x y];
                    curVertices{i}.index_in_vertexList = …
                            size(obj.vertexList,1); % number of rows

                    curVertices{i}.isAdded = 1;
                end
                verticesID(i) = curVertices{i}.index_in_vertexList ;
            end
        end

        function obj = add2triangleList(obj,verticesID)
            obj.triangleList(end+1,:) = …
                   [verticesID(1) verticesID(2) verticesID(3)] ;
        end

        % now lets triangulate the given polygon
        function obj = Triangulate(obj,varargin)
            n = obj.vertices.getNoOfVertices();

            % intialize ear status
            obj.EarInit();
```

```
% since triangulation will change the linked list structure but
% cutting off ears, lets generate a clone of the given polygon
% in the memory so that we will operate on it and leave the
% original one.
clone_obj = obj.Clone();

clone_obj.vertices = clone_obj.vertices.getHead();

% starting with an empty vertexList and triangleList
obj.vertexList   = zeros(0,0);
obj.triangleList = zeros(0,0);

% each step of outer loop removes an ear
while (n > 3)
    % inner loop search for an ear
    v2 = clone_obj.vertices;

    while(1)
        if v2.isEar
            % ear found, get neighboring vertices
            v3 = v2.next;
            v4 = v3.next;
            v1 = v2.prev;
            v0 = v1.prev;

            % display the diagonal v1v3
            Polygon2D.drawDiagonal(v1,v3,varargin{:});
            pause(0.5);

            % update earity of diagonal endpoints
            v1.isEar = clone_obj.Diagonal(v0,v3);
            v3.isEar = clone_obj.Diagonal(v1,v4);

            % now v1, v2, v3 construct a triangle
            % lets update the triangulation datastructre
            % first the vertex list, add these vertices if they
            % are not added before
            curVertices{1} = v1;
            curVertices{2} = v2;
            curVertices{3} = v3;
            [obj,verticesID] = obj.add2vertexList(curVertices);

            % now adding to the triangleList
            obj = obj.add2triangleList(verticesID);

            % cut off the ear v2
            v1.next = v3;
            v3.prev = v1;
            clone_obj.vertices = v3;
            n = n -1;
            break % out of inner loop and resule the outerloop
        end
```

```
                    v2 = v2.next;
                end
            end

            % adding the last triangle (since we exit when n = 3, i.e.
            % there are three more vertices not yet visited)
            curVertices{1} = v2.next;
            curVertices{2} = v2.next.next;
            curVertices{3} = v3.next.next.next;
            [obj,verticesID] = obj.add2vertexList(curVertices);

            % now adding to the triangleList
            obj = obj.add2triangleList(verticesID);
        end
    end
end
```

Now you can test your triangulation as follows,

Code 15 - Test Triangulation Code

```
% drawing a polygon
vertices = vertices.gotoHead();
polygon = Polygon2D(vertices);
polygon.draw('DrawHandle',gca,...
    'ColorMarkerStyle','g*:','LineWidth',2);

polygon = polygon.Triangulate('DrawHandle',gca,...
    'ColorMarkerStyle','r*:','LineWidth',2);
pause(0.5);
triplot(polygon.triangleList, ...
        polygon.vertexList(:,1),polygon.vertexList(:,2),'b');
```

Task 7: *Repeated intersection tests.* Triangulate (Code 14) often checks for the same segment/segment intersections. Modify the code so that you can determine how many unnecessary segment/segment intersection tests are made. Test it the polygon whose xy coordinates are given in Code(2).

Task 8: *Convex polygons.* Analyze the performance of Triangulate when run on a convex polygon.

Task 9: *Spiral.* Continue the analysis using the polygon whose vertices are given in the following table, this is an example that forces the inner ear loop to search extensively for the next ear. Does Triangulate continue to traverse the boundary in search of an ear? More specifically, if the polygon has *n* vertices, how many complete circulations of the boundary will the pointer *v2* execute before completion?

X	17	1	4	4	9	12	15	15	15	3	11	15	15	15	9
Y	1	3	6	7	11	14	17	1	3	7	14	7	8	9	14

Task 10: *Ear list* . The inner loop search of Triangulate can be avoided by linking the ear tips into their own (circular) list, linking together those vertices *v* for which v.isEar == 1 (true) with pointers *next_ear* and *prev_ear* in the vertex structure. Then the ear for the next iteration can be found by moving to the next ear on this list beyond the one just clipped. Implement this improvement, and see if its speedup is discernible on the example given in Task 9.

Task 11: *3-coloring*. Use your triangulation to 3-color a given polygon and indicate the minimum number of cameras required to guard such polygon.

Task 12: Write a report to summarize the theoretical background needed for this lab and your experimental results. Your report should begin with a cover page introducing the project title and group members. It is important to note that all figure axes should be labeled properly. You are required to submit your MATLAB codes (fully commented) with a readme file describing your files and how to use them in terms of input and output.

Good Luck