A Tool for Quick freehand drawing of 3D shapes on the Computer

Rohit Sud (rohit@gatech.edu) and Abhishek Venkatesh (venky@gatech.edu)

College of Computing, Georgia Institute of Technology, Atlanta GA

Abstract

In this paper we describe a way to create simple 3D shapes by interpreting freehand sketches drawn by the user. The user creates a polyloop by tracing a curve using a mouse. The curve is cleaned up by removing the non-manifold points and hairs. The cleaned loops properly oriented to define the interior of the shape. We then distribute points in the interior of the shape, compute a triangulation of those points and finally bulge the shape using the ball transform to get a 3D shape. Our method allows arbitrary number of genus to be present in the 3D shape and is easily able to generate a variety of shapes.

Keywords

Curves, Smoothing, sidewalks, graph, Delaunay triangulation, bulge, trimming, meshing, corner table.

Problem statement

The aim of the project is to create a tool for drawing freehand curves representing shapes (polyloop curves) and then be able to inflate the loops into 3D surface. The application should automatically remove the non-manifold points in the curve and present a set of polyloops to the user which are not self intersecting and at the same time similar to the shape which was intended by the user. The application then *inflates* or bulges the shape using the *ball* transform. It also produces a triangulation for the inflated shape using a variant of constrained Delaunay triangulation to represent the 3D shape intended by the freehand curve.

Introduction

There are many advanced software in the market which allows us to create complex and accurate 3D shapes. But most of involve selecting a primitive and the drawing that primitive. Our motivation towards developing this tool is to create a tool which may not produce very accurate 3D objects but is as intuitive as drawing shapes on a paper. There are many applications like animation, where accuracy may not be that important and our tool will give reasonable results. The advantage is that drawing simple shapes is almost as simple as drawing on a paper and the user doesn't have to bother choosing primitive shapes or vertices.

In our tool, we ask the user to draw freehand curve representing the boundary of the shape he intends to create. But user may draw a curve which may be self intersecting and might contain unwanted parts. Our tool cleans up the user curve and presents him a set of polyloops which do not contain any non-manifold point but tries to resemble as close as possible to the shape intended by the user. Once the user is happy with the loops we present him he can bulge the shape and create a 3D shape. We also calculate the triangle mesh of the 3D object which the user can save to a file and load into the mesh viewer to do other operations.

Section I talks about the main algorithm used to reduce the user drawn curve (with non-manifold points) to a set of manifold polyloops with possibly holes. Section II describes the bulging algorithm. Section III describes the algorithm for generating a 3D triangle mesh for the 3D shape, Section IV consists of the helper algorithms used in section I,II and III to solve this problem. Section V shows some results and shapes drawn through our application. Section VI concludes the paper some possible improvements.

Section I

Trimming: Cleaning up the freehand curve drawn by the user.

We represent the user drawn curve using a set of sampled vertices (P[ ]) in order. Each consecutive vertex in this array of points P[] represents an edge. The algorithm proceeds as follows: The sampled user curve is first cleaned up. It might happen that many vertices are very close to each other. The x,y co-ordinates of a group of vertices close to each other is set to a common (x,y) coordinate. Next we calculate the intersecting points of all edges and insert them in the array P[] using Algorithm 1 and 2 in section IV. The intersection points are important because they help us identify the loops in the user drawn curve. We explicitly need to do this because while sampling the user drawn curve it is not guaranteed that the all intersecting point of the edges would be part of array P. (This is done by insertNonManifoldPts() in the accompanying code).We now create a new array of points GP[] from P such that no 2 vertices in GP are very close to each other. This is done by adding vertices to GP from P only if the new point to be added from P is not very close to an already existing point in GP. But the point to note is that now consecutive points in the array GP do not represent an edge. So we also maintain a list of valid edges GE[] as we add vertices from P to GP. Each element of GE stores the index of 2 vertices from GP to denote an edge. So we basically have a graph representation of the user drawn curve including all the intersecting point (non-manifold points). While creating the graph we take care not to insert duplicate edges or zero length edges in the array GE.

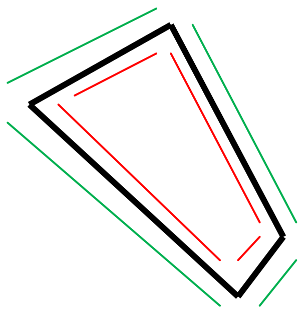


Figure 1 : Sidewalks (red and green) around roads (edges of polyloop)

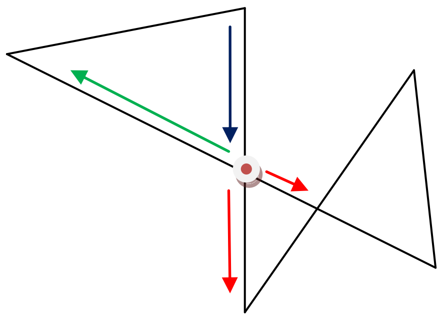


Figure 2 : Illustrating a right turn at a non-manifold point to trace an inner loop, we take the Green path

Next we use the analogy of 'sidewalks' to identify loops in our graph. Let the edges of the polyloop represent *roads*. Now these roads might cross each other at *crossings*. Imagine that the road is composed of 2 *sidewalks* on either side of itself. Therefore we model the edge as to having 2 sidewalks on the left side (ls) and right side (rs). Now identifying the loop is straightforward. Pick a sidewalk which is not covered and keep taking the next sidewalk without crossing the road (or in other words keep taking right turns at each crossing). When you come back to the point from where you stared you have a loop. While we traverse a sidewalk we also paint that sidewalk to denote that it has been covered. We use Algorithm 3 to identify the next sidewalk to be taken (or the next right turn). We keep generating loops till no sidewalk is left to be painted. If the starting sidewalk is clockwise we get an interior loop and if the starting sidewalk is clockwise then we get an exterior loop.

The code snippet for finding loops is given below:

```
void FindLoops(){
 for(int i=0;i<nGE;i++)// for each edge ini the graphs
 {
   if(GE[i].ls==0)// if left side walk is not yet painted
   {
     int ei=i,S,E; S=GE[ei].S; E=GE[ei].E;
     LOOPS[nloops]=new Loop(); //create a new loop do{
     LOOPS[nloops].Vi[LOOPS[nloops].nVi]=S;/*add the starting
vertex of the sidewalk to the Loop */
     LOOPS[nloops].nVi++; int
     rvi=getRightVertex(S,E);// get the next sidewalk
     ei=getEdgeFromVertices(E,rvi);// get the edge index
     if(ei==-1)// invalid edge
      {nloops--;break;}
     if(E==GE[ei].S)
       {S=GE[ei].S;E=GE[ei].E;GE[ei].ls=nloops+1;}
     else
       {S=GE[ei].E;E=GE[ei].S;GE[ei].rs=nloops+1;}
     if(S==GE[i].S&&E==GE[i].E)// detect the starting point
        break;
}while (true)
 }
 same thing is done for right sidewalks. ..... ....
}
```

So we have now identified all the loops L[] in our Graph GE or non-manifold polyloop. Each entry in the array L[] consists of ordered set of indices of vertex in GP[](defined earlier). Each consecutive index in L[i] represents an edge of the Loop L[i]. The last edge is formed by combining the last and first entry of L[i]. Now the problem boils down to removing loops from L[] in such a way that the final set of loops have a close resemblance with the original freehand shape drawn by the user and also does not contain any non-manifold point. We define a rule that the loop with the least absolute area is picked and if it has a non-manifold point it is deleted and the remaining loops and edge table GE are updated. We use algorithm 4 for computing the area of a polyloop and use its absolute value. But deletion of a loop may result in splitting up of already existing loops and addition of new edges. See Algorithm 5 in section IV for the details of how to delete a loop. We keep on deleting the loops till we reach a state when there are no non-manifold points left in the graph/loops. The motivation for choosing the smallest area is that in general the user would draw the intended part of the shape bigger than the unwanted part of the shape.

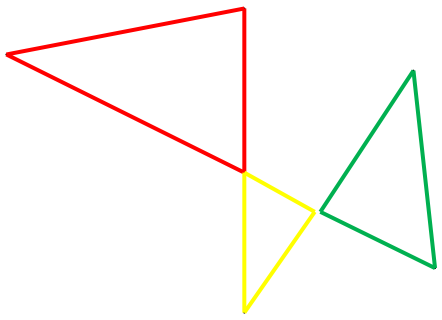


Figure 3: Result of loop detection algorithm. For clarity, the loop encompassing the complete polyloop is not shown.

Code Snippet for reducing a set of non-manifold loops to a set of manifold loops is given below:

```
void ReduceToManifoldPoly()
{
 for(int i=0;i<nloops&&checkForNonManifoldPts()>0;i++)
 {
   int delLoopi=getMinLoopArea();
   if(1!=CheckIfLoopIsNonManifold(delLoopi))
  {
    LOOPS[delLoopi].done=1;
    continue;
  }
LOOPS[delLoopi].done=1;
 RemoveLoopsEdges(delLoopi);
}
 DrawFinalReducedPoly();
}
```

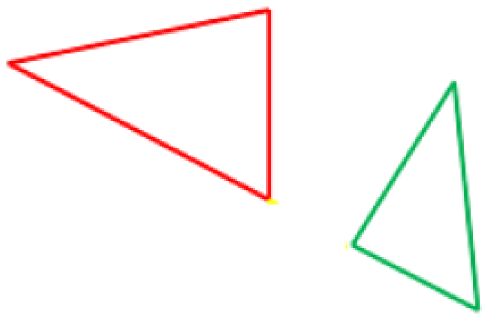


Figure 4: Result of Loop Deletion to remove non-manifold points

For our loop removal algorithm, we are not only interested in the loop with the smallest area but it should also have a manifold point. If we do not do that we will be removing all the loops from the polyloop till we have just one loop left!

For removing a loop we remove its entry from a Global Edge table which contains all the edges that form the polyloop. We remove the loop using Algorithm 5(section IV). Then we try to repair/ stitch-back the loops that were broken as we removed the edges of the smallest polyloop.

Section II

Bulge

After trimming we are left with a set of clean manifold polyloops which represents the user intended shape. We would like to bulge this shape to produce a 3D shape. In order to do so we need to interpret the loops and identify the interior and exterior parts of the shape. Then we create a 3D point cloud (each point having x,y,z co-ordinates)by sampling the interior of the shape which we call as Bulge.
Overview of the Bulge algorithm is as follows:

1) Identify the interior: Orient the loops (clockwise or anti-clockwise) so that the interior of the shape always lies towards the left of the oriented loop edge.
2) Calculate Height Map: for the all the points (in 2D) which lie in the interior of the shape we calculate the height at that point.
3) Distribute Points inside the shape.
4) Compute the Z-co-ordinate of the points added in step 3) using the height map computed in step 2). The output of this step is the 3D point cloud

(bulge) which will be later triangulated in "Meshing"(described later).

The details of the above steps follows:

1) Identifying the interior: We orient the loops (output of the trimming step) such that the interior of the shape always lie to the left of an oriented loop edge. This can be done for a loop: LOOP[i] by computing the parity of the number of loops containing LOOP[i]. If the parity is odd then we orient the loop as clockwise else counter-clockwise. Example, the parity for Loop 'A' in figure 5 is even so it will be oriented counter-clockwise. Loop 'B' (parity=odd) will be oriented clockwise and Loop 'C' counter-clockwise.(The green shaded region is the interior)

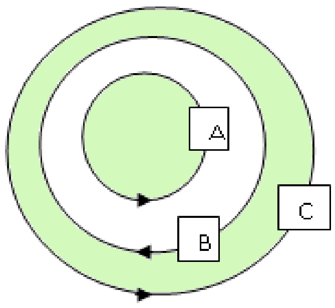


Figure 5: Orienting the polyloops using parity.

2) Calculating the Height Map: We have used the Ball transform (Figure 6) for calculating the height of each point in the interior of the shape which gives a smooth and bulging shape to the interior height map.

Ball Transform: The height at a point 'P' inside the given shape 'S' is the maximum height at 'P' of all the semi-spheres with base disk completely inside 'S'. The algorithm for Bulge is given in section IV.

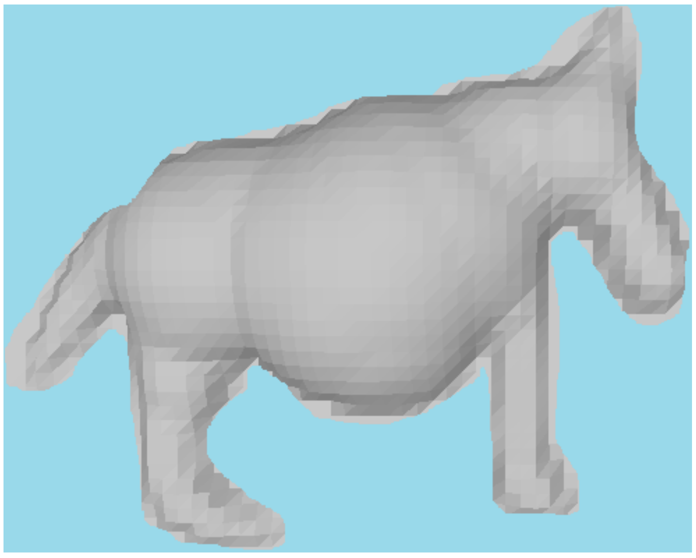


Figure 6: Ball Transform

3) Distributing Points inside the shape:

```
for each edge: edge[i]
{
 pt M=mid point of edge[i]
 vec N=edge[i].left().unit();
 int foundedge=0;
 for(j=0; foundedge==0&& j<BIGNUMBER;j++)
 {
  pt R= (M.x+N.x*j,M.y+N.y*j);// move R along the normal //from
 the mid of the edge.
  for each edge: edge[k]
  if(M.disTo(R)>= edge[k].disToPt(R))
  {
   foundedge=1;
   //Now Add vertices along the line segment 'M' to 'R' using the
   //formula: pt(M.x+N.x*d,M.y+N.y*d)
   for(float a=astartval;a<3.14f/2;a+=aincrementval)
   {
    float d=minD*(1-cos(a));
    appendVertex(new pt(M.x+N.x*d,M.y+N.y*d));
   }
   appendVertex(R);
   break;
  }
 }
}
```

Here we can control the number of points added by changing the variable *aincrementval.* If it is smaller you will get more points. The algorithm can be visualized by means of figure 7

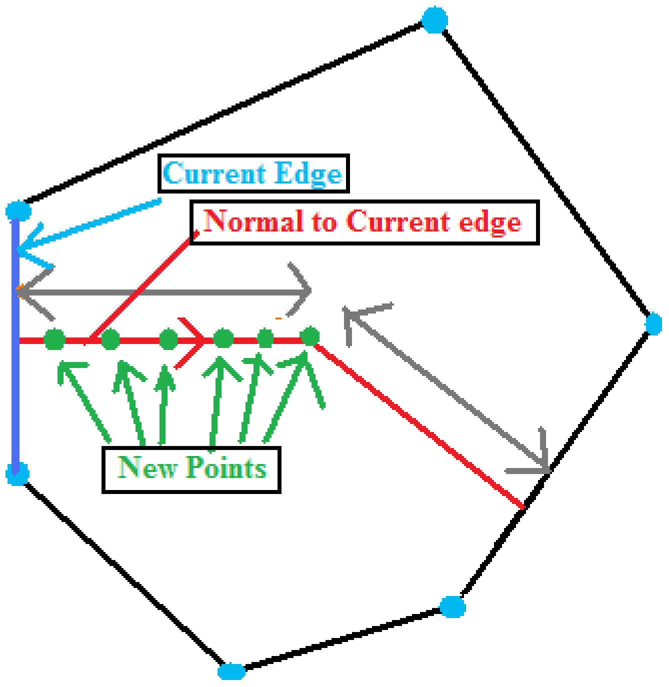


Figure 7: Computing new points in the polyloop interior.

Though the above approach gives us a uniform points distribution in case of convex polyloops, it leaves some areas without any points around concave vertices of the polyloop. To address that we can add vertices along the fan of normals emerging from the concave vertex as shown in figure 8:

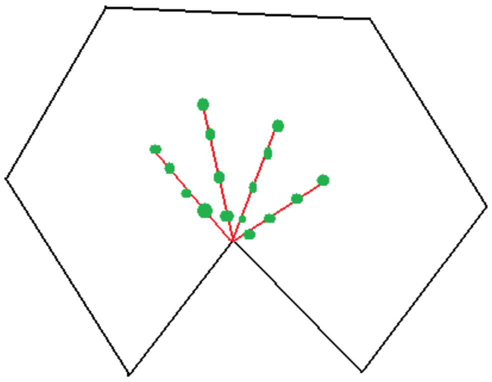


Figure 8: Handling concave vertices.

4) Compute the Z-co-ordinate of the point cloud: Once we have the height map and the set of points, the z-coordinate of a point 'P' is simply the height at point 'P' in the heightmap.

Now we have a point cloud and a list of edges which we will triangulate as described in the next section.

Section III

Meshing

Meshing here refers to the triangulation of the bulge we have previously calculated. Our input is the set of 3D point cloud (we do not consider the height information of the points as the point cloud was sampled taking the bulge height into consideration) and the edges of the oriented manifold loops.

INPUT:

i) GT[]: The 3D point cloud where each entry in GT stores the corresponding x,y,z co-ordinate of a vertex in the 3D point cloud.

ii) edge[]:We extract the edge information from the oriented loops and fill it in a common array edge[]. Here each entry has a starting(S) and an ending vertex (E) index denoting an edge. The corresponding co-ordinate of the starting and ending point of this edge is found by indexing S and E into GT[]. We also keep a marker ("used" field) for each edge to remember if we have already used that edge for the triangulation process.

OUTPUT:

i) V[]: The V-table represents the triangulation of the point cloud. Each set of consecutive 3 rows stores the vertex index of each of the 3 corners of a triangle.

ii)nV: (number of triangles)*3

The algorithm for computing the triangle mesh is as follows:

```
for each edge in edge[]
 if(edge[i].used==false)
 {
  edge[i].used=true;
  pt M=mid point of the edge;
  vec N= unit normal to edge;
 float R=Choose a Radius starting from –ve infinity (just a
 big number)
 while(!(found a 3rd point for triangulation))
 {
  pt O= (M.x+R*N.x,M.y+R*N.y);// center of the disk
  //moving along 'N' towards edge[i]. Note that here starting
 //value of 'R' is negative.e
  R+=.1;// move the center towards the edge
  for each vertex index 'k'
  {
  if(k==edge[i].S||k==edge[i].E) continue;
  else
  {
   if(dot(M.makeVecTo(GT[k]),N)>ZERO
  &&GT[k].disTo(O)<=GT[edge[i].S].disTo(O))
  {
    set the flag that we found the 3rd point for
    triangluation.

    // Now add the other 2 edges of the triangle.
   appendEdge(edge[i].S,k);
   appendEdge(k,edge[i].E);

   Remove any hairs. We do this by checking for duplicate
  edges and removing both the instances.

   // Now add the 3 rows of vertex indices representing
   //the triangle in the 'V' table in the proper order.
   V[nV++]=edge[i].S;
   V[nV++]=edge[i].E;
   V[nV++]=k;
   break;
   }// end if
   }//end else
  }//end for
 }//end while
}
}// end if
```

The algorithm can be visualised by means of figure 9. Here we move a disk from infinity along the outward normal of the edge (starting point of the normal is the mid-point of the edge) so that it touches one more point in addition to the 2 vertices of the edge. This basically amounts to fitting a circumcircle for 3 points so that no other point of the point cloud is inside this circle.

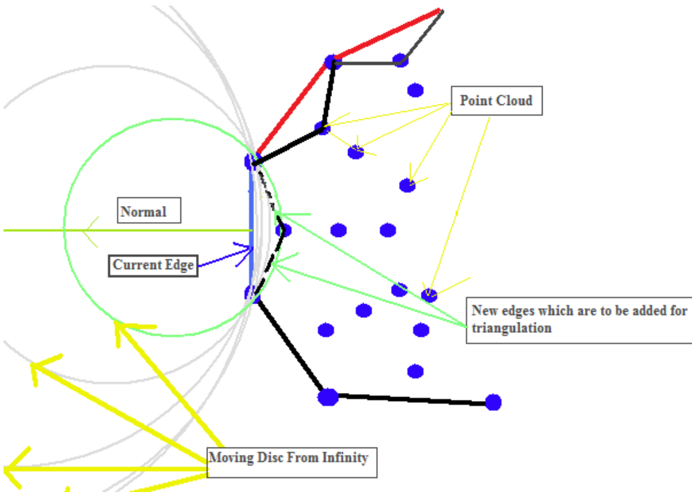

Point Cloud
Normal
Current Edge
New edges which are to be added for triangulation
Moving Disc From Infinity

Figure 9: Fitting the circumcircle to 2 points

The mesh that is generated is not closed (open from the bottom). We want to close the mesh so we simply take the mirror of the top and create the bottom half of the mesh as shown in figure 10

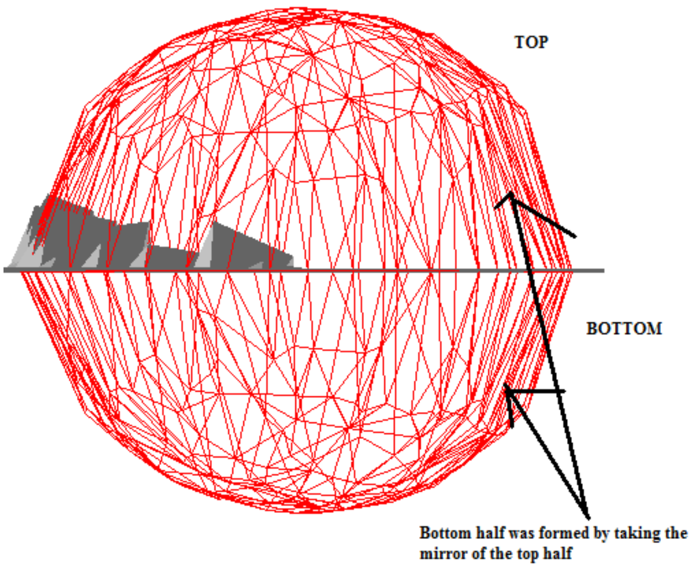


Figure 10: Creating the mirror image of the polyloop mesh.

Section IV

Algorithm 1

Algorithm to determine if two line segments intersect each other

Consider 2 line segments AB and PQ. They intersect each other if and only if, for both the line segments, the end points of one line segment are not on the same side of the other line segment.

Mathematically,

$$sgn\left(\overrightarrow{AQ}.left\left(\overrightarrow{AB}\right)\right) \neq sgn\left(\overrightarrow{AP}.left\left(\overrightarrow{AB}\right)\right)$$

and vice-versa.

Algorithm 2

Algorithm to determine point of intersection of 2 lines [1]

Let the equation of the lines be

$$\vec{P} = \overrightarrow{P_0} + \mu\left(\overrightarrow{P_0} - \overrightarrow{P_1}\right)$$, and

$$\vec{P} = \overrightarrow{Q_0} + \gamma\left(\overrightarrow{Q_0} - \overrightarrow{Q_1}\right)$$

Let $\vec{P}$ be the point of intersection of 2 lines and $\vec{N}$ be the normal such that $\vec{N}.\left(\overrightarrow{Q_0} - \vec{P}\right) = 0$.

Solving the line equations simultaneously and solving for μ, we get

$$\mu = \frac{\vec{N}.(Q_0 - P_0)}{\vec{N}.(P_1 - P_0)}$$

Therefore the point of intersection $\vec{P}$ is given by

$$\vec{P} = \overrightarrow{P_0} + \frac{\vec{N}.(Q_0 - P_0)}{\vec{N}.(P_1 - P_0)}\left(\overrightarrow{P_0} - \overrightarrow{P_1}\right)$$

Algorithm 3

Algorithm to determine the rightmost turn from a non-manifold point

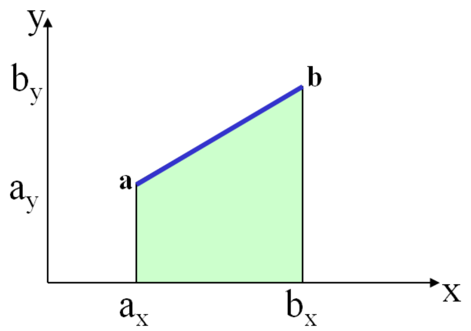
Consider a non-manifold point P. Let AP be an incident edge at P and PQ_i be emerging edges from P. To determine the rightmost turn from AP, we first compute the slope of AP and then the slope of each PQ_i. Using the slope, compute the angle between 0 to 2π for the slope.

Then we compute the difference between the angles formed by AP and the horizontal axis and PQ_i and the horizontal axis. The value of i for which PQ_i gives us the minimum difference with respect to the edge AP gives us the right turn vertex from AP.

Algorithm 4

Algorithm to determine the area of a polyloop [2]

Consider a polyloop made of a set of edges E_i. To compute the area of the polyloop we consider the signed area under formed by the trapezium made by an edge and the reference axis under it.



$$\text{area}(\mathbf{a},\mathbf{b}) := (a_y+b_y)(b_x-a_x)/2$$

Figure 11: Calculating area of a trapezium

We add up all the areas along with the sign to get a signed area. Note that the area will come out to be positive if our polyloop was oriented in a clockwise manner. For our purpose we take the absolute value of the area as a measure of the size of the polyloop.

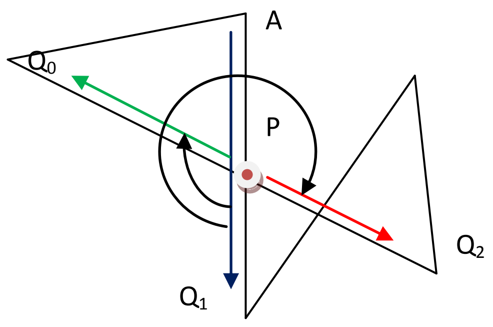


Figure 12: Illustrating a right turn at a non-manifold point. Green line represents the path taken (PQ_0).

Algorithm 5

Algorithm to remove non-manifold loops from a polyloop

Consider a polyloop (non-manifold) which has a loop to be removed. The loop has at least one non-manifold point as shown.

We maintain a global edge table (GE) which keeps track of all the valid edges remaining in the polyloop.

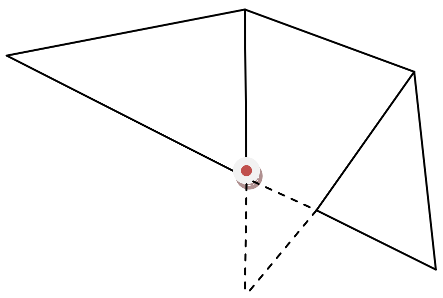


Figure 13: Removal of a loop, manifold point + smallest area

When we delete a loop L_i, we simply remove the corresponding edges of L_i from GE. The length of Loop L_i is marked as 0 so that in future it is not considered anymore in the algorithm.

Now we patch-up or stitch the remaining loops. For doing that, we observe that when we remove a set of edges from the polyloop, all the other loops:

a) Would not be affected as they didn't share any vertex (or they share a single vertex but no edge) with the deleted polyloop. So these loops need not be processed.
b) Or will have some vertices common with the removed loop and hence will be broken into

c) newer loops and open ends joined. We use the following approach to break these loops into new loops:

The edges of the loop L_j which has to be stitched are marked as true (T) or false (F) based on the updated GE. Now we have connected series of ‘F’s and ‘T’s. We simply throw away this loop and create new loops out of a series of connected ‘T’s.

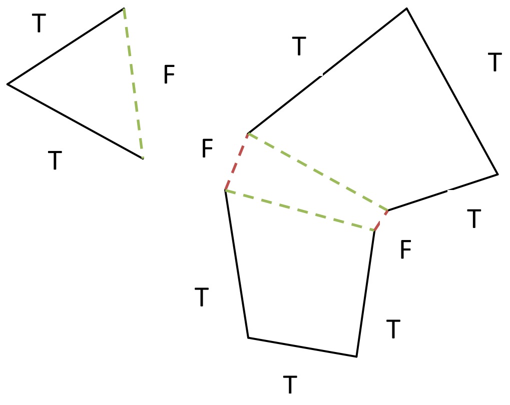


Figure 14: Figures showing the stitching rule for broken loops. The loops are represented by the series TTF and TTTFTTTF. When we re-stitch the loops, we remove the F edges and obtain a loop with TT_ and TTT_ and TTT_ where _ represents a new edge inserted to complete the loop.

Algorithm 6

Algorithm to compute the height map using the ball transform [2]

```
void ballTransform() {
for each point P(x,y) in the plane
 if (P(x,y is inside the shape ‘S’){
/*update the height map for the sphere with center P(x,y) and radius =
P(x,y).disToclosestEdgeIn(S)*/
scanSphere(P(x,y),P(x,y).disToclosestEdgeIn(S));
}}
void scanSphere(pt O, float r)
{
 for each point ‘SP’ in the disc with center =O and
 radius ‘r’
 {
  if(height at ‘SP’ of sphere with center O and radius r
   is > the height in the height map)
  {
    Make the height of heightmap at SP= height at ‘SP’
    of sphere with center O and radius r;
  }
 }
}
```

Section V

In this section we show a sample shape which was drawn using the above approach. Each of the following figures show the intermediate state followed by the final 3D triangle Mesh.

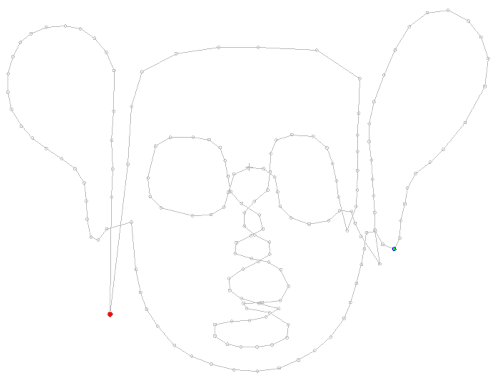


Figure 15: User Draws Curve

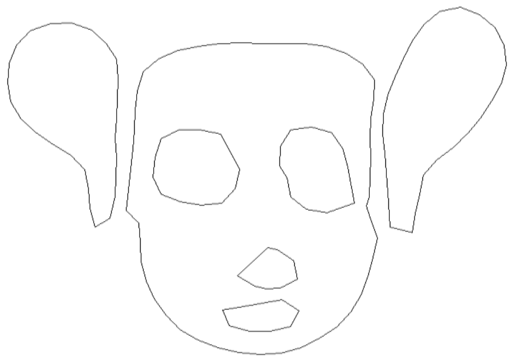


Figure 16: After Trimming we have manifold polyloops.

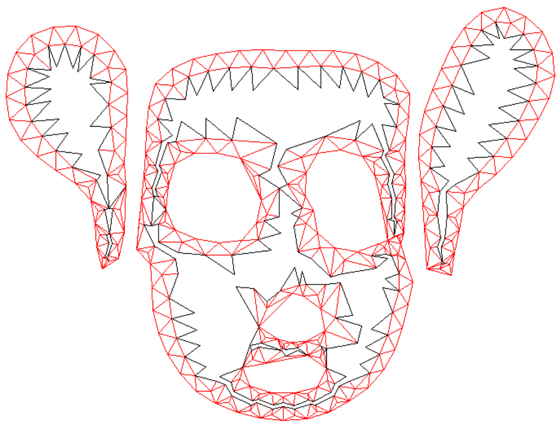


Figure 17: Intermediate step of Meshing of the point distribution. The black edges are not yet used and the red ones have been used for triangulation

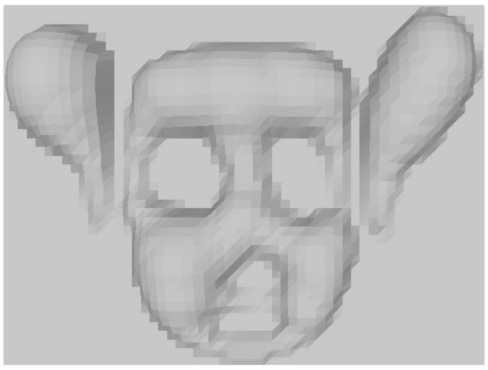


Figure 18: Ball Transform

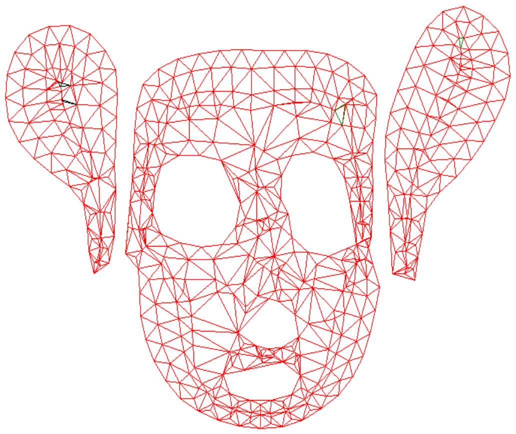


Figure 19: Final 3D Mesh formed by combining Ball transform and point distribution

Section VI

Conclusion:

By this implementation, we have provided the user with an easy and an intuitive way to create 3D shapes from simple 2D curves. The user also has the flexibility to create non-connected 2D curves which can be used to create multiple 3D shapes (including Genus).

Our polyloop stitching algorithm preserves more holes than the algorithm that was originally suggested in class for implementation. It does have a drawback in the sense that it doesn't preserve the parity of the loops, but for the current application, that doesn't matter too much.

The triangulation algorithm functions well for most cases and produces reasonably good results for concave vertices too. But it has a drawback of taking a long time to complete. A suitable modification for improving the speed and hence the efficiency of the procedure can be taken up as future work.

The bulge method produces good approximation of the 3D surface with a very simple algorithm. It can be further improved by considering triangles instead of squares or grids to compute the height of a particular point.

As an extension to this project we can generate the corner table out of the triangle mesh and allow the user do smoothing, subdividing and various other operations on the mesh.

Improvements:

- The algorithms that are being used by us are not optimized as of now. Consequently they take a lot of time to process and display the triangulation.
- There is no UI for the user to change the parameters for re-sampling and the size of the triangulation. But this is just a UI limitation and is possible with the current code by changing the variable in the accompanying code
- We could also try to triangulate the mesh in 3D.

References

[1] Dollins, S. C. (2002). Retrieved from Handy Mathematics Facts for Graphics: http://www.cs.brown.edu/~scd/facts.html

[2] Rossignac, J. Retrieved from CS6491 Foundations of Computer Graphics, Modelling, and Animation: http://www.gvu.gatech.edu/~jarek/courses/6491/

[3] Delaunay Triangulation http://en.wikipedia.org/wiki/Delaunay_triangulation

[4]Tai, Zhang and Phong. Prototype Modelling from Sketched Silhouettes based on Convolution Surfaces

[5] Jarek R. Rossignac. Solid and Physical Modelling.

[6] Paul Chew. Voronoi Diagram applet http://www.cs.cornell.edu/Info/People/chew/Delaunay.html

[7]Dey, Li and Ray.Polygonal Surface meshing with Delaunay refinement.

[3] Teddy: A Sketching Interface for 3D Freeform Design,Takeo Igarashi†, Satoshi Matsuoka‡, Hidehiko Tanaka†University of Tokyo, ‡ Tokyo Institute of Technology.