LITM: Layer-Interpolating Tet Mesh

Project 5 for CS6491 – Fall 2017

Anna Greene

Dibyendu Mondal

ABSTRACT

The goal of this project is to invent, justify, implement, debug, and learn advanced algorithms that construct, display, process, and encode triangle (tri) meshes and tetrahedron (tet) meshes. Here, we review the goals achieved, an outline of our solution, an overview of prior art, and followed by a step-by-step description of our approach.

1 Problem Statement

Given two clouds of balls located on two horizontal planes, we first aim to compute the Delaunay Tetrahedralization of the union of the balls. Each ball has a center (called 'sites') on one of the two planes (called the **floor** and the **ceiling**, respectively), and the edges between the balls will be represented as tubes with the same radius.

Then, we will compute a high-resolution water-tight triangle mesh that approximates the boundary of the union of all balls and tubes and render this mesh using smooth shading with both visible and hidden silhouettes drawn.

2 Goals Achieved

We have successfully calculated the Delaunay Tetrahedralization of the given clouds of balls. We have also rendered this mesh using smooth shading.

To find the Delaunay Tetrahedralization, we first find the Delaunay Triangulation of the clouds of points on the floor and on the ceiling, displayed in orange and green, respectively, in Figure 1. Then, we loop once over the points on the opposite plane in order to determine the edges that connect the two planes, displayed in cyan in Figure 2. Figure 2 shows the tubes at their appropriate radius, which is the desired result.

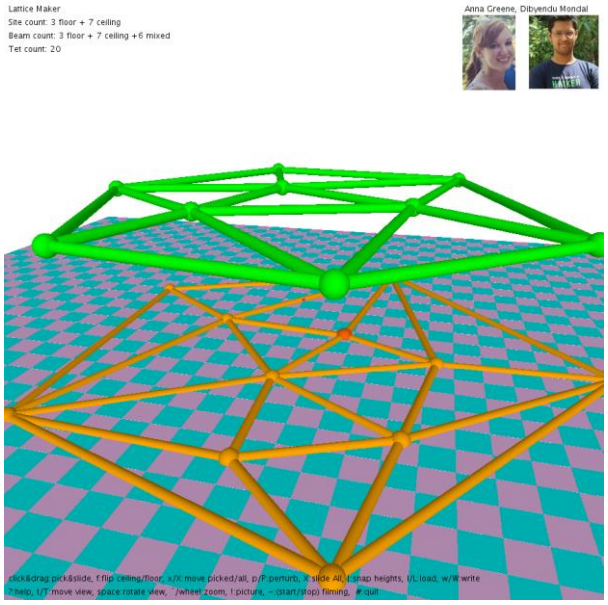

Fig 1. Delaunay Triangulation of the clouds of points on the floor (orange) and the ceiling (green).

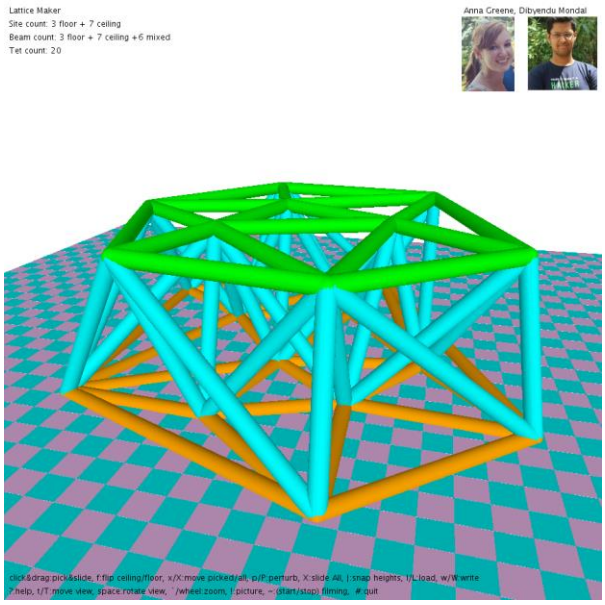

Fig 2. Delaunay Tetrahedralization of the union of the clouds of points on the floor and the ceiling.

Given the following loop structure to determine points A, B, and C that define a triangle on one of the planes and a fourth point M on the other plane, our solution has a complexity of n^4:

```
for (int i = 0; i < P.nv; i++) {
      for (int j = i+1; j < P.nv; j++) {
            for (int k = j+1; k < P.nv; k++) {
                  pt A = P.G[i];
                  pt B = P.G[j];
                  pt C = P.G[k];
                  ...
                  for (int m = 0; m < P.nv; m++) {
                        pt M = P.G[m];
                        ...
                  }
            }
      }
}
```

Since we store each triangle as we find it, our code to determine the tetrahedralization between the two planes loops over those existing triangles and all points on the other plane once, which has a complexity of n^2. Since $n^2 < n^4$, our overall complexity remains n^4. We also store each edge that is determined to be on the boundary of the P and Q clouds respectively (shown below in Figure 3), and use those values to determine any tetrahedra that are composed of two points on the ceiling and two points on the floor.

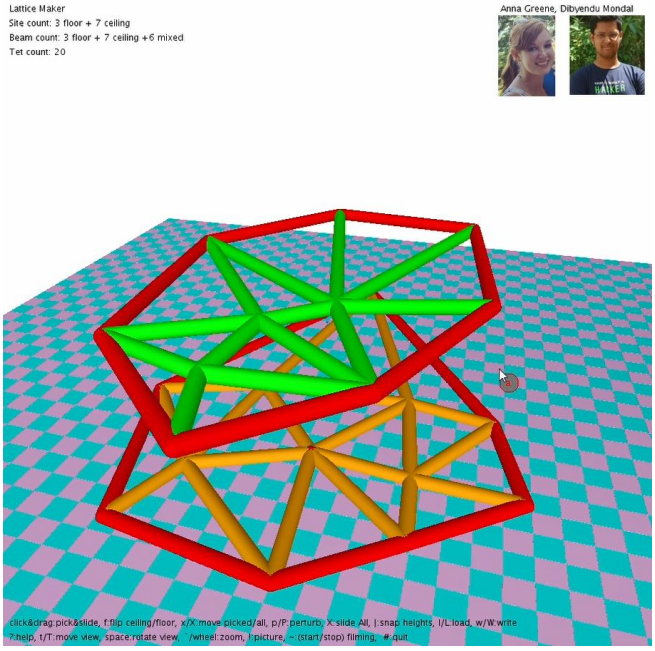

Fig 3. Boundary edges.

Our solution, however, assumes the following restrictions on the input. First, we need at least three balls on each plane, and the balls must be pairwise disjoint. We also assume a general configuration, where no four balls in a plane can lie on the same circle, and the clouds of balls are relatively similar (the cloud on the ceiling lies over the cloud on the ceiling), and there are no extreme outliers. Examples of bad configuration are shown below in Figures 4 and 5.

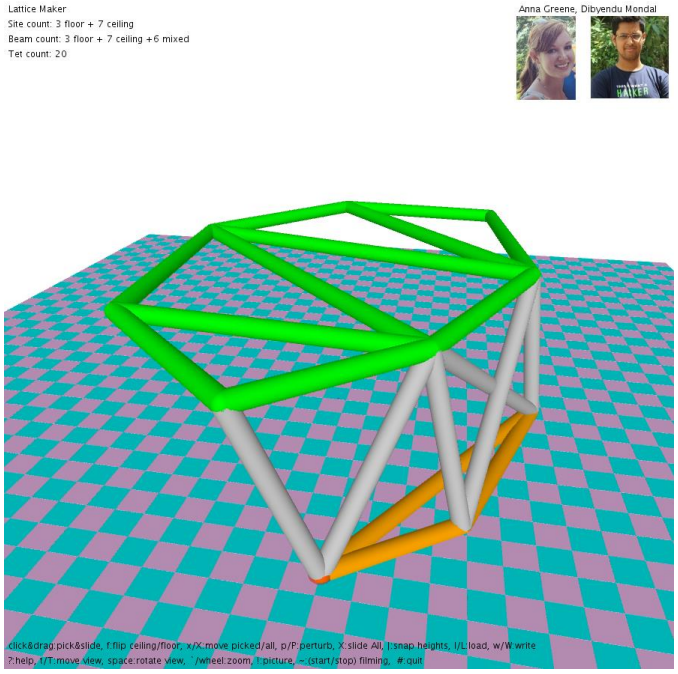

Fig 4. Bad configuration: the cloud of balls on the ceiling (green), is a lot more spread out than the cloud on the floor (orange).

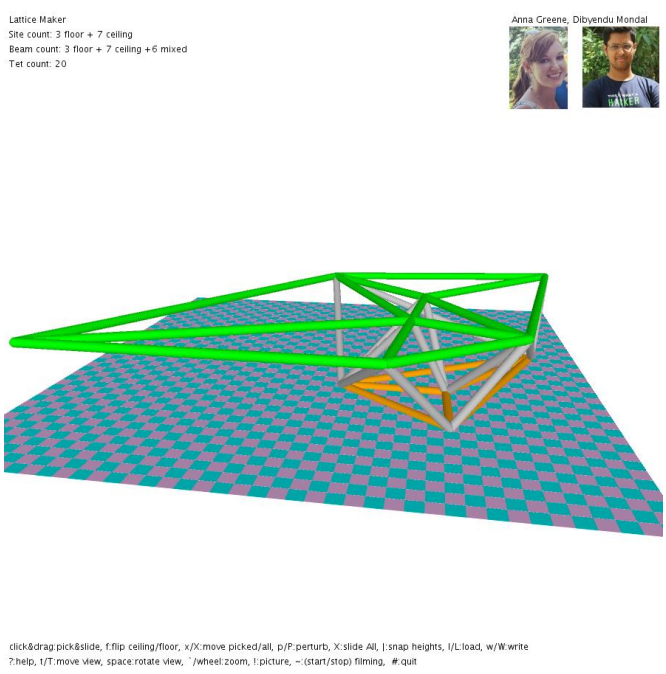

Fig 5. Bad configuration: the cloud on the ceiling (green) has a large outlier and is therefore not connected back to the cloud on the floor (orange).

Given these restrictions, our solution is reliable. Given the n^4 complexity, there is a limit to the number of balls that can be provided as input in the two clouds, as our program will slow drastically as more balls are added.

3 Outline of Solution

To find the Delaunay Tetrahedralization of union of all the sites on the two planes, we first find the triangle mesh of each of the two clouds of balls, and then continue on to find the tubes that connect the floor to the ceiling. To find the Delaunay Triangulation, we first calculate the circumcenter between any three balls (A, B, C) on the plane (first on the floor, then on the ceiling). Then we use a simple distance equation to determine if there is any fourth ball within that circle. If there is no fourth ball closer to the circumcenter that the three balls used to define it in the first place, then we store that triangle in either array Ptriangles or array Qtriangles, and store the tubes AB, BC, and AC in Ptubes or Qtubes to be drawn in a later routine.

When we store the triangle, we also update our array Pedges or Qedges, which store the tubes that are considered to be on the **boundary** of the Delaunay Triangulation. For each new triangle, we determine if any new edges AB, BC, or AC are already listed in Pedges or Qedges, depending on whether we are triangulating the floor or the ceiling. If a tube is already present, we remove it from that array, as that edge is now bordering two different triangles. If it is not yet listed, we add the new tube since we only know of one triangle that it is the edge of.

To calculate the tetrahedral mesh, we first loop over all the edges in Pedges and Qedges to find all tetrahedra that are defined as having two points of an edge on the ceiling and two points of an edge on the floor. We then find the edge on the ceiling which is at a minimum distance from the edge on the floor. We find the minimum distance by just checking the minimum distance between the two end-points of the edges. Next, we add tubes across the end-points and add a tube for the shortest diagonal of the quad.

Then, we calculate the circumcenter for a sphere between three points defined by one of our stored triangles (A, B, and C) and a fourth point (D) on the other plane. Once we have the circumcenter, we calculate the **bulge**, which we define as the vertical distance between the plane the triangle lives on and the "top" of the circle, which will be beyond the other plane. For example, while looping over Ptriangles, the bulge will be the distance between the floor, and the top of the circle, which lies above the ceiling. This bulge distance will always be larger than the distance between the floor and ceiling. When we find the smallest bulge for a given triangle, we store the edges DA, DB, and DC in array QPtubes to be drawn in a later routine.

Next, we add normal for all the vertices and pass them to the shader to do smooth shading.

4 Prior Art

In the paper by Li and Teng [9], they provide a refinement-based algorithm to generate well-shaped Delaunay meshes. The algorithm majorly has 4 components:

Enforce Empty Encroachment: For any encroached boundary segment, add its midpoint and update the Delaunay triangulation. For any encroached boundary triangle, add its circumcenter c and update the Delaunay triangulation. If c encroaches any boundary segment, they split the encroached boundary segment instead of adding c.

Clean Bad Elements: For any bad tetrahedron T, find a point p in its picking region whose insertion avoids creating small slivers. If such p does not exist, then add the circumcenter C_T of T. Here, tetrahedra with a large radius-edge ratio have priority over slivers to be split. If the circumcenter C_T encroaches boundary, they either Encroach Equatorial Sphere or Encroach Diametric Sphere.

In the paper by Lee and Schachter [6], they provide two algorithms for constructing a triangulation over a planar set of N points. The first algorithm is a divide and conquer algorithm which runs in *O(NlogN)* time and the second algorithm is an iterative algorithm which runs in $O(N^2)$ time.

The first algorithm stores an ordered adjacency list of points v_{i1}, v_{i2}, v_{i3}, … where (v_i,v_{ij}) j = 1,…,k is a Delaunay edge. First, they sort the set of N points in lexicographically ascending order. Next, they divide the set into two equal subsets such that the first sorted half is in the first set and the second sorted half is in the second set. Next, they recursively construct the Delaunay triangulations. To merge the two triangulations, they use the convex hull of the union of those two sets.

The second algorithm iteratively triangulates a set of points within a rectangular region. If the point set does not include all four vertices of the rectangle, the missing vertices are implicitly added. The algorithm uses the swapping approach developed by Lawson.

5 Step-by-Step Approach

Here, we will give an in-depth description of the methods that we created.

```
pt circumCenter(pt A, pt B, pt C)
{
  pt M = P(A, 0.5, V(A, B));
  vec N = cross(V(A, B), V(A, C));
  vec V0 = U(cross(N, V(A, B)));
  pt H = P(A, 0.5, V(A, C));
  vec AM = V(A, M);
  vec AH = V(A, H);
  float s = (dot(AH,AH) - dot(AM, AH))/(dot(V0, AH));
  pt center = P(M, s, V0);
  return center;
}
```

We represent the center of the circumcircle as P = M + s*V, where M is the midpoint of the edge AB, V is the vector perpendicular to edge AB and in the direction of the center, s is the distance from M to the center.
M = (A+B)/2 since it's the midpoint of the edge AB.
V = (ABxAC)xAB which is perpendicular to the normal of the triangle and the edge AB.
Let H = (A+C)/2 be the midpoint of AC.
AP = P-A
= M + s*V – A
= AM + s*V
Also, AP.AH = AH.AH (since projection of AP on AH is AH)
⇨ (AM+s*V).AH = AH.AH
⇨ s = (AH.AH-AM.AH)/V.AH
Now we have the values of M, s and V, so we can find P which is M+s*V

```
pt circumCenter(pt A, pt B, pt C, pt D)
{
  pt center;
  pt P = circumCenter(B,C,D);
  pt P0 = circumCenter(A,B,C);
  float s = dot(V(P,P0),V(C,A))/(dot(U(cross(V(C,D),V(C,B))),V(C,A)));
  center = P(P,s,U(cross(V(C,D),V(C,B))));
  return center;
}
```

First, we find the circumcenter of 2 triangles.

P = circumCenter(B, C, D) and

P0 = circumCenter(A, B, C)

The circumcenter of the sphere Q = P + s*V

where V = CDxCB/|CDxCB| i.e. normal to the triangle BCD

P + s*V = P0 + s'*V'

where V' = CAxCB/|CAxCB|

Now, do a dot product with CA on both the sides, we get,

P.CA + s*V.CA = P0.CA + 0 (since V' is perpendicular to CA)

=> s = PP0.CA/V.CA

```
boolean isInCircle(pt A, pt center, pt point)
{
  if(d(point,center) <= d(A,center))
    return true;
  else
    return false;
}
```

Here, we check if the distance between the point to the center is <= the distance from one of the points on the circle to the center, then we return true else we return false.

```
pt findBulge(pt A, pt B, pt C, int num)
{
  pt minPt = P(0,0);
  if(num == 1)
  {
    pt D = Q.G[0];
    pt quadCenter = circumCenter(D,A,B,C);
    pt triCenter = circumCenter(A,B,C);
    float bulge = d(triCenter,quadCenter) + d(D,quadCenter);
    float min = bulge;
    minPt = D;
    for(int n = 1; n < Q.nv; n++)
    {
      D = Q.G[n];
      quadCenter = circumCenter(D,A,B,C);
      triCenter = circumCenter(A,B,C);
      bulge = d(triCenter,quadCenter) + d(D,quadCenter);
      if(min > bulge)
      {
        min = bulge;
        minPt = D;
      }
    }
  }
  else
  {
    ……
  }
  return minPt;
}
```

To find the bulge, we first find the circumcenter of the quad ABCD and the circumcenter of the triangle ABC.

bulge = distance from point on ceiling to circumcenter of ABCD + distance from circumcenter of ABCD to circumcenter of ABC

We iterate through all the points on the ceiling to find the point with minimum bulge

```
void addTube(pt A, pt B, int num)
{
  boolean flag = false;
  if(num == 0)
  {
    for(int i = 0; i < Ptubes.nv; i+=2)
    {
      if((A.x == Ptubes.G[i].x) && (A.y == Ptubes.G[i].y) && (A.z == Ptubes.G[i].z) && (B.x == Ptubes.G[i+1].x)
&& (B.y == Ptubes.G[i+1].y) && (B.z == Ptubes.G[i+1].z))
      {
```

```
          flag = true;
          break;
        }
      }
      if(!flag)
      {
        Ptubes.addPt(A);
        Ptubes.addPt(B);
      }
    }
    else if(num == 1)
    {
      ……
    }
    else if(num == 2)
    {
      ……
    }
  }
```

To add a tube from A to B, we check if a tube already exists from A to B, if not then we add it.

```
  pts addEdge(pt A, pt B)
  {
    boolean flag = false;
    {
      for(int i = 0; i < nv - 1; i+=2)
      {
        if((A.x == G[i].x) && (A.y == G[i].y) && (A.z == G[i].z) && (B.x == G[i+1].x) && (B.y == G[i+1].y) && (B.z
== G[i+1].z))
        {
          flag = true;
          deletePt(i+1);
          deletePt(i);
          break;
        }
      }
      if(!flag)
      {
        G[nv]=A;
        G[nv+1]=B;
        nv+=2;
      }
    }
    return this;
  }
```

In this function, we add an edge between point A and point B. If an edge already exists, we delete that edge. So, at the end, we will have all the boundary edges.

Now, since we are done with the details of the helper functions, we will give an in-depth detail of our main code.

```
  for(int i = 0;i < P.nv; i++)
  {
    for(int j = i+1; j < P.nv; j++)
    {
      for(int k = j+1; k < P.nv; k++)
      {
        pt A = P.G[i];
        pt B = P.G[j];
        pt C = P.G[k];

        pt center = circumCenter(A,B,C);
        boolean flag = false;
        for(int m = 0; m < P.nv; m++)
        {
          pt M = P.G[m];
          if(M == A || M == B || M == C)
            continue;
          else
          {
            if(isInCircle(A,center,M))
            {
```

```
            flag = true;
            break;
          }
        }
      }
      if(!flag)
      {
        addTube(A,B,0);
        addTube(B,C,0);
        addTube(A,C,0);
         addEdge(A,B,0);
        addEdge(B,C,0);
        addEdge(A,C,0);
        Ptriangles.addPt(A);
        Ptriangles.addPt(B);
        Ptriangles.addPt(C);
      }
    }
  }
}

for(int i = 0; i < Ptriangles.nv-2; i+=3)
{
  pt A = Ptriangles.G[i];
  pt B = Ptriangles.G[i+1];
  pt C = Ptriangles.G[i+2];
  pt minPt = findBulge(A,B,C,1);
  addTube(minPt,A,2);
  addTube(minPt,B,2);
  addTube(minPt,C,2);
}
```

Here, first we iterate through the points thrice and find the circumcenter of those three points. Next, we check if any point on the same plane lies inside the circumcircle of those three points. If not, then we add a tube/beam between those points. We also add an edge between these two points. We also store those three points as a new triangle.

Next, we iterate through the triangles and find the minimum bulge point on the other plane and add a tube/beam from the minimum bulge point to the other three points.

```
 for (int i = 0; i < Pedges.nv-1; i+=2)
{
    pt A = Pedges.G[i];
    pt B = Pedges.G[i+1];
    pt C = Qedges.G[0];
    pt D = Qedges.G[1];


    float avgDist = (d(A,C)+d(B,D))/2;
    for (int j = 0; j < Qedges.nv-1; j+=2)
    {
        pt Ctemp = Qedges.G[j];
        pt Dtemp = Qedges.G[j+1];
        float tempAvg1 = (d(A,Ctemp)+d(B,Dtemp))/2;
         float tempAvg2 = (d(A,Dtemp)+d(B,Ctemp))/2;
         if (min(tempAvg1,tempAvg2) < avgDist)
         {
             if (tempAvg1 < tempAvg2)
             {
                   avgDist = tempAvg1;
                    C = Ctemp;
                    D = Dtemp;
             }
              else
             {
                   avgDist = tempAvg2;
                    C = Dtemp;
                    D = Ctemp;
              }
        }
    }

    EdgeTubes.addTube(A,C);
    EdgeTubes.addTube(B,D);
    float diagAC = d(A,D);
    float diagBD = d(B,C);
    if (diagAC < diagBD) EdgeTubes.addTube(A,D);
```

```
    else EdgeTubes.addTube(B,C);
  }
```

Next, we check for edges on Q which are closest to edges on P. Here we find the edge which has the minimum of the smallest distance between the endpoints of the edges. Then, we add tubes across the endpoints and a tube for the smallest diagonal.

```
void collar(pt P, vec V, vec I, vec J, float r, float rd)
{
  float da = TWO_PI/36;
  beginShape(QUAD_STRIP);
  for(float a=0; a<=TWO_PI+da; a+=da)
  {
    normal(U(V(r*cos(a),I,r*sin(a),J))); v(P(P,r*cos(a),I,r*sin(a),J,0,V));
    normal(U(V(rd*cos(a),I,rd*sin(a),J))); v(P(P,rd*cos(a),I,rd*sin(a),J,1,V));
  }
  endShape();
}
```

Here we add normal for all the vertices and pass them to the shader to do smooth shading.

6 Resources

Circumspheres:

[1] http://mathworld.wolfram.com/Circumsphere.html

[2] http://www2.washjeff.edu/users/mwoltermann/Dorrie/70.pdf

[3] https://www.jstor.org/stable/2973351?seq=1#page_scan_tab_contents

Tetrahedralization:

[4] http://wias-berlin.de/software/tetgen/examples.dragon.html

Triangulation of 3D surfaces from points:

[5] https://www.cs.purdue.edu/homes/aliaga/cs334-15fall/lectures/lec-voronoi-and-triangulation.pdf

Relevant Literature Survey:

[6] https://link.springer.com/article/10.1007/BF00977785

[7] https://dl.acm.org/citation.cfm?id=276894

[8] https://math.stackexchange.com/questions/1858388/tetrahedron-signed-distance-between-circumcenter-and-face

[9] https://dl.acm.org/citation.cfm?id=365416

[10] https://arc.aiaa.org/doi/pdfplus/10.2514/6.1987-1124