Layer-Interpolating Tet Mesh

CS6491 Fall 2017, Project 5, 11/28/2017

Yaohong Wu Jin Lin

1. Problem Statement

1.1 Construction of Tetrahedron Mesh

Given points on two planes, one called the floor and the other called the ceiling, construct a tetrahedron mesh that "connects" these two planes. For each tetrahedron, it has vertices on the floor as well as on the ceiling.

1.2 Approximation of the Boundary of the "Framework"

Given the tetrahedron mesh mentioned above, compute a triangle mesh to approximate the boundary of its "framework", which is made of balls (vertices) and beams (edges).

2. Contributions and Level of Completion

2.1 Main Contributions

- Implemented Delaunay Tetrahedralization based on empty sphere property.
- Applied Ball Pivoting Algorithm to reconstruct surface given oriented points.
- Utilized Octree to accelerate spatial query.
- Designed a simple method to compute an appropriate radius of the pivoting ball using sampling density and our knowledge of the original surface.

2.2 Analysis of Completion

The naïve implementation of Delaunay Triangulation for vertices on a plane costs $O(V^4)$, where V is the number of vertices. The Delaunay Tetrahedralization based on the two triangle meshes on the floor and the ceiling costs $O(TV)$ (for 1-3 or 3-1 case) and $O(E^2V)$ (for 2-2 case), where T is

the number of triangles on the floor or the ceiling, E is the number of edges for triangle mesh on the floor or the ceiling. Our naïve implementation is very slow theoretically. However, since V is not large in our case, it doesn't look that slow.

For surface reconstruction based on Ball Pivoting Algorithm, the time complexity is $O(T) + O(V^2)$, where $O(V^2)$ is spent on finding a seed triangle and $O(T)$ is for expanding the triangle mesh. We assume that after we build the Octree and can quickly access a small neighborhood, either checking empty ball configuration or finding a hit during pivoting will only cost constant time. Our implementation can reconstruct a triangle mesh with about 60k triangles in less than 20 seconds, given that we can quickly find a safe triangle to start with and have fast access to local neighborhoods.

We have to admit that due to numerical errors, some specific configurations (such as 4 vertices on the same plane), there exists some wrong border edges or even holes in our reconstructed meshes. We add a post processing stage to fix triangular holes, as suggested in Reference 1. More details can be found in Section 5.4.4.

3. Outline of Approach

3.1 Construction of Tetrahedron Mesh

3.1.1 Construction of Planar Triangle Mesh

We first construct two triangle meshes, one for points on the floor and another for points on the ceiling. Inspired by the empty circle property of Voronoi Diagram, we implement a naïve method to construct a triangle mesh for points on a plane. More specifically, we check three points at a time and see if there is any point inside the circumcircle defined by these three points. If the circumcircle is empty, meaning no points inside it, we make a triangle using by connecting these three points.

3.1.2 Connection of Two Planar Triangle Meshes

After we get the two triangle meshes on the floor and the ceiling, we can construct a tetrahedron mesh by "connecting" them. More specifically, there are three cases, namely, connecting a triangle on the floor with a point on the ceiling (3-1 case), connecting a triangle on the ceiling with a point on the floor (1-3 case), and connecting an edge on the floor with an edge on the ceiling (2-2 case).

For 3-1 or 1-3 case, we iterate the collection of triangles on one plane and find an "appropriate" point on the other plane for each triangle. The "appropriate" point for a triangle is the one that gives minimum "bulge" w.r.t the triangle. The algorithm for 3-1 case is shown below.

```
Algorithm for 3-1 case: (assume that 3 points on the floor and 1 point on the ceiling)
FOR each triangle ABC on the floor
  FOR each point P on the ceiling
    compute current bulge for P w.r.t. ABC
    update minimum bulge w.r.t ABC and optimal point P*
  connect triangle ABC and P*
```

For 2-2 case, we use a naïve method based on the empty sphere property. For each edge AB on the floor and each edge CD on the ceiling, we construct a sphere defined by A, B, C, D, and check if it is empty. If it is empty, we connect A to C and D, and connect B to C and D. The algorithm for 2-2 case is shown below.

```
Algorithm for 2-2 case:
FOR each edge AB on the floor
  FOR each edge CD on the ceiling
    compute circumcenter F of A, B, C, D and corresponding radius r
    FOR each point P, other than A, B, C, D, on the floor or the ceiling
      IF distance(P, F) < r THEN
        sphere defined by A, B, C, D is not empty
    IF sphere defined by A, B, C, D is empty THEN
      construct edges AC, AD, BC, BD
```

3.2 Sampling and Reconstruction

3.2.1 Sampling from the Boundary of the "Framework"

Since the "framework" is made of balls and cubes, we can first sample the surface of each ball and the surface of each cube. We parameterize the surface of a ball and the surface of a cube, and generate points on these surfaces. It should be noticed that we will generate points which are not on the surface we are interested in. For example, it is possible that some part of a ball is inside a cube. However, the Ball Pivoting Algorithm will "ignore" these useless points if we find a "good" place to start with. Thus, it would not be a big problem.

3.2.2 Reconstruction Using Ball Pivoting Algorithm

Once we have points on the target surface, we can use some mesh reconstruction method to construct a triangle mesh that approximates the original surface. We employ the Ball Pivoting Algorithm for this task. A brief introduction to Ball-Pivoting Algorithm is in Section 3.2.

Basically, we start from a “safe” triangle and then expand our triangle mesh step by step. We pivot a ball with some appropriate radius around the current boundary of our mesh, and add new vertices, new edges and new triangles to our mesh. We call the boundary of our mesh, that can be potentially expanded, the “front”.

4. Discussion of Prior Art

4.1 Voronoi Diagram and Delaunay Triangulation

It is known that the Delaunay Triangulation of a point set is the dual graph of the corresponding Voronoi Diagram. Thus, the Delaunay Triangulation has an elegant property, that is, a circle circumscribing any Delaunay triangle does not contain any other points in its interior. This is the foundation of our naïve implementation of triangulation and tetrahedralization.

4.2 Ball-Pivoting for Surface Reconstruction

Simply speaking, the Ball-Pivoting Algorithm uses a “growing” scheme to construct a triangle mesh. We start by finding a valid triangle and placing a ball that touches the three vertices. We then rotate the ball w.r.t. some boundary edge, meaning that it keeps touching the two endpoints of the edge of current triangle mesh until it hits a point. We update our triangle mesh and its corresponding boundary after each ball pivoting process. Finally, when we cannot advance from the boundary of the triangle mesh, we output the triangle mesh we find.

As in the original paper of this algorithm, some useful preprocessing techniques may be applied. For example, to find a target point during ball pivoting process, we only need to check a small spatial neighborhood of the edge around which the ball rotates. Thus, we can partition the whole space into small cubic cells and only check a specific set of cells when doing ball pivoting.

The mesh updating and front advancing part could be tricky. The original paper introduces two useful operations, namely join and glue, to tackle this task.

5. Details of Implementation

5.1 Construction of Tetrahedron Mesh

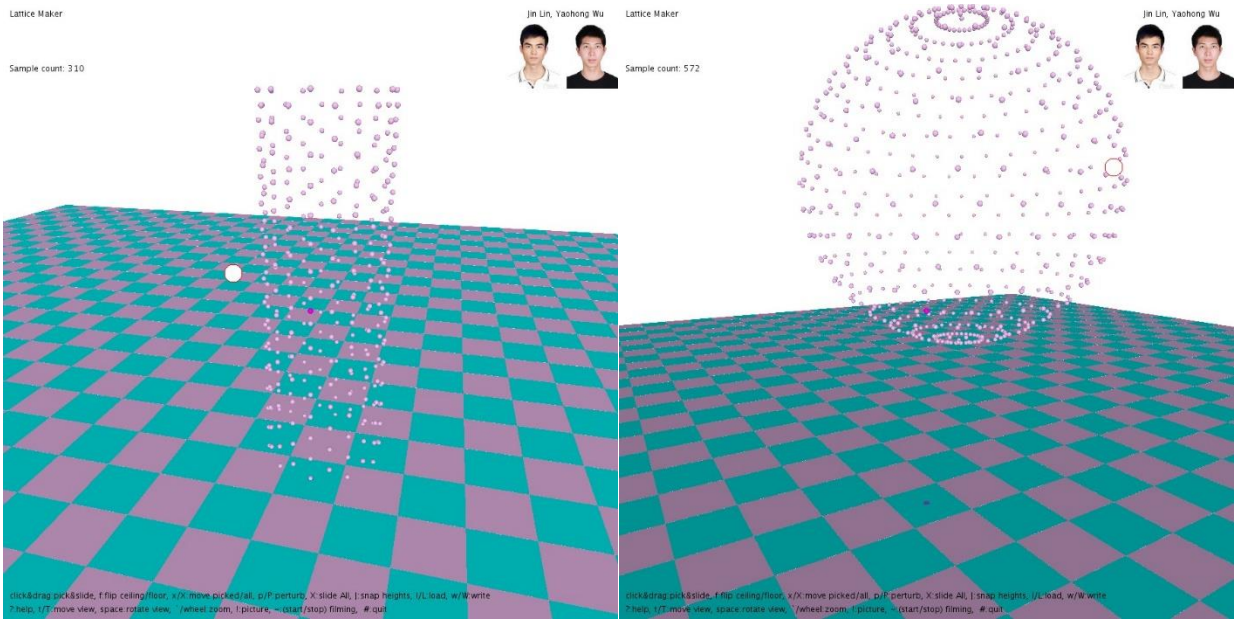
Our implementation is naïve. The framework of our algorithm is in Section 2.1. The main recipe for this implementation is to compute the circumcenter of a triangle and a tetrahedron.

5.2 Surface Sampling

To sample a beam or a cylinder, we can parameterize it first. For a point on the side surface of a cylinder, we use two parameters w, h to represent it. More specifically, w is the "angle" of this point w.r.t. the circle through this point and parallel with the bottom of the cylinder. h is the "height" of this point w.r.t. the bottom. To generate points in a more general configuration, we can apply a "zigzag" strategy. For example, at height h_i, we sample points at angles $0, \frac{\pi}{2}, \pi, \frac{3\pi}{2}$. Then at next height h_{i+1}, we sample points at angles $\frac{\pi}{4}, \frac{3\pi}{4}, \frac{5\pi}{4}, \frac{7\pi}{4}$.

To sample a ball or a sphere, we can also parameterize it use two parameters. We use two angles, α and β, i.e. "longitude" and "latitude" respectively. Obviously, $0 \leq \alpha < 2\pi$ and $-\frac{\pi}{2} \leq \beta \leq \frac{\pi}{2}$. A pair of α and β represents a point on the surface of a ball. Similar "zigzag" strategy can also be applied here.

Following pictures show results of sampling a cylinder and a sphere.



5.3 Surface Reconstruction by Ball Pivoting

5.3.1 Find a Seed Triangle

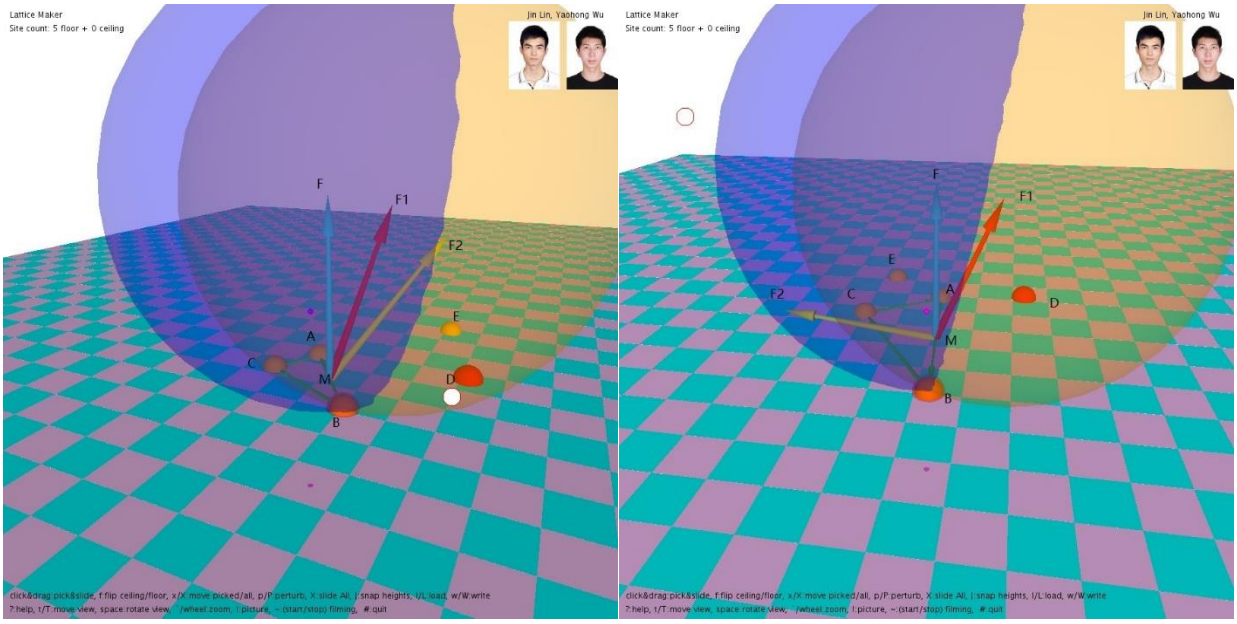
To find a safe place to start with, we use brute force method here. We check if there are three vertices in empty ball configuration. In fact, we can fix one vertex and find the other two. This trick is important because we don't want to find a seed triangle under the surface, which could happen because we sample some annoying vertices under the surface. We start at some middle

position of a beam, and use brute force method to find the other two. After we find three vertices, we construct a triangle and put it to our mesh. The three edges now become our front.

5.3.2 Mesh Expansion

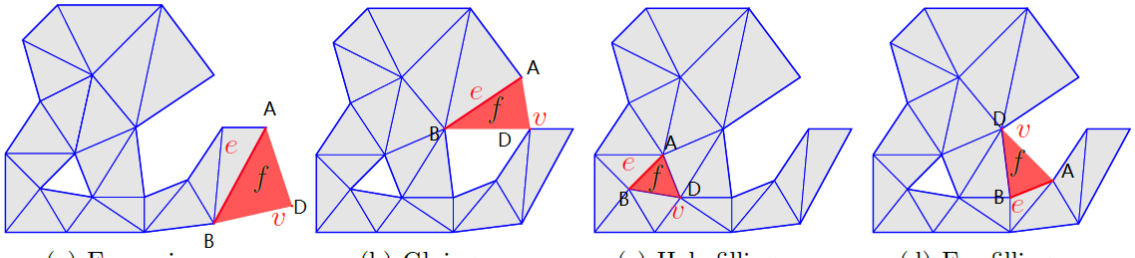
After we have our initial mesh, i.e. a triangle, we try to expand its boundary by ball pivoting. For each edge in the front, i.e. the boundary, we pivot a ball around it and find a valid hit point. By "valid", we mean the hit point and the two endpoints of the edge should in empty ball configuration, and the pivoting angle should be as small as possible.

To decide the pivoting angle, we should consider the direction of the edge. Thus, we should associate each front edge a direction and carefully manage these directions. More specifically, assume that the new center of the pivoting ball is *F'* and original center is *F*, the edge is from *A* to *B* with middle point *M*. We want the $\angle \mathrm{FMF}'$ to be some value in [0, 2π]. To accomplish this, we compute $\mathrm{MF} \times \mathrm{MF}'$, if it is not aligned with AB, i.e. $AB \cdot (MF \times MF') < 0$, then $\angle FMF' = 2\pi - \angle FMF'$. In the left picture below, the hit is D because it corresponds to a smaller pivoting angle, i.e. $\angle FMF_1 < \angle FMF_2$. In the right picture, $\angle FMF_2 > \pi$.



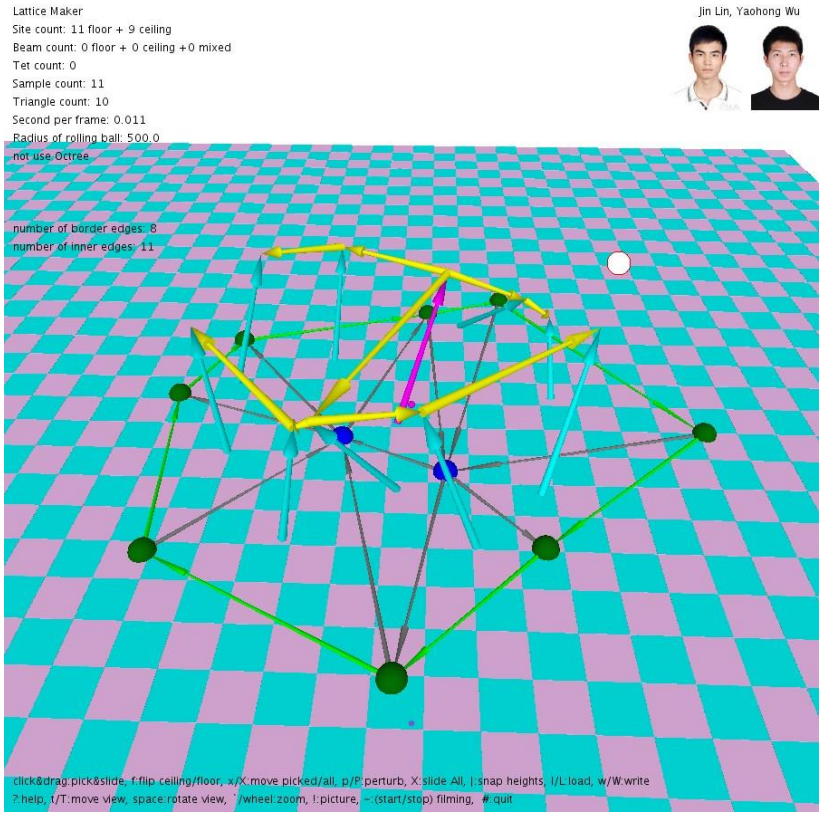
Some obvious rejection tests should also be performed to generate a manifold mesh. This will not only speed up the whole process but also help produce a good triangle mesh. For example, if three vertices don't have compatible normals, then they should not form a good triangle. Besides, if a vertex is inner, we should never hit it again.

After we find a hit, we should update our mesh and the front. The tricky part is updating front. If we pivot around edge AB, and we find a hit D. There are totally 4 cases to update the front depending on the occurrence of DA and BD in the front. For example, if DA is already in the front, then DA and AD, which we should create immediately, will “cancel out”. Thus we should remove DA from the front and do not need to create AD. Same story for DB and BD. Note that generally, we will not generate the same edge with the same direction twice. The following picture shows possible cases during mesh expansion. This picture is from Reference 1.



(a) Expansion case (b) Gluing case (c) Hole filling case (d) Ear filling case

The following picture shows a toy example of Ball Pivoting Algorithm. The green vertices are border vertices and the blue vertices are inner vertices. The green edges are border edges and the grey edges are inner edges. The yellow path is the ball path constructed by the different centers of the pivoting ball. The magenta arrow starts at the seed triangle we find. The cyan arrows correspond to other triangles we find during the expansion.



5.4 Performance Improvement

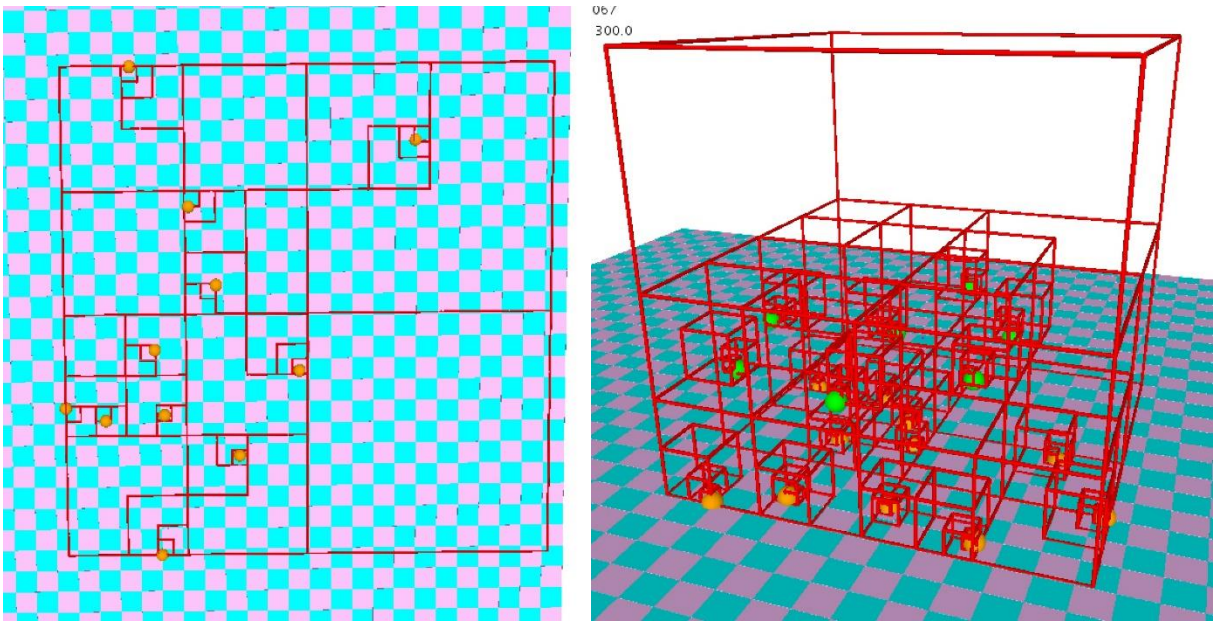
5.4.1 Octree Acceleration Structure

The Ball Pivoting Algorithm requires a large amount of spatial queries, in checking whether a ball is empty, or finding the closet hit when pivoting a ball. We do not need to look at all the input vertices each time we do these things. Instead, what we are interested in is just a small neighborhood of some point each time we pivot a ball or do empty test.

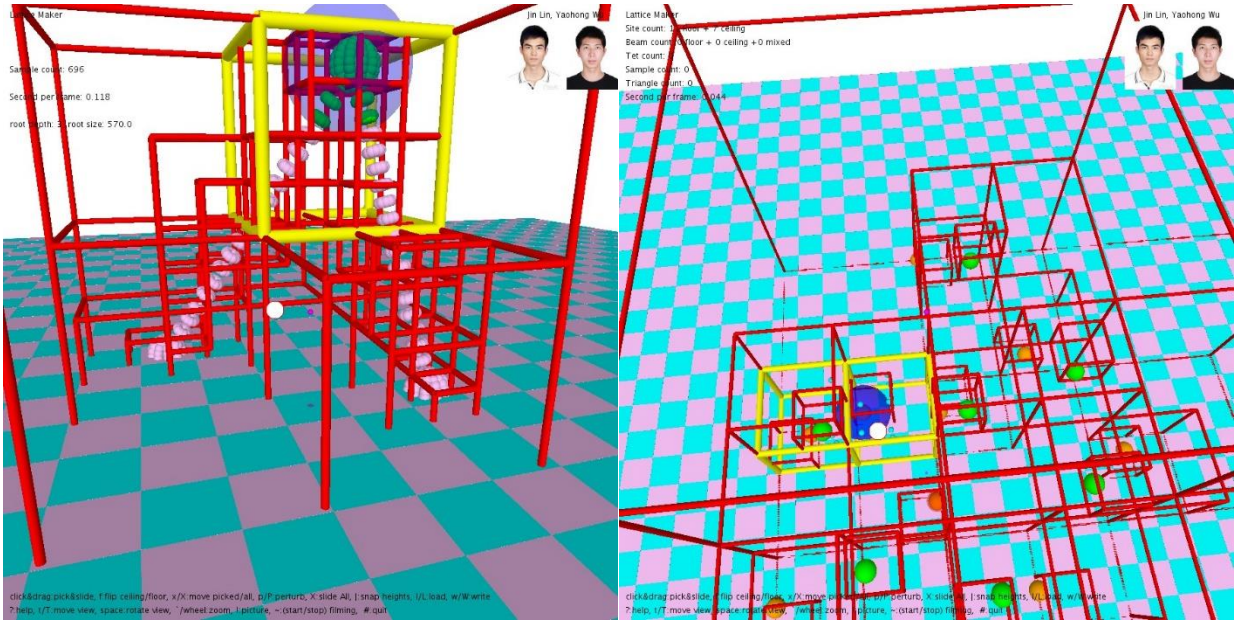
To fast access the neighborhood we are interested in, we add a preprocess stage, i.e. building an octree for the input point cloud. We start from a bounding box that contains our scene and partition this box into small cubic cells. Generally, a cell will be partitioned into 8 smaller cells. But not all 8 cells will be created because some may not contain vertices. What we get at last is an unbalanced octree.

To get the neighborhood of a point, we first locate the cell, at some level corresponding to the radius of the neighborhood, of this point and find its neighboring cells that intersect with the neighborhood.

The octree visualization shows in the following pictures.



The following pictures show the neighborhood we are interested in. The yellow cells are the cells that intersect with the blue sphere which is the neighborhood we want. In the left picture, the vertices in the blue sphere are shown in green.



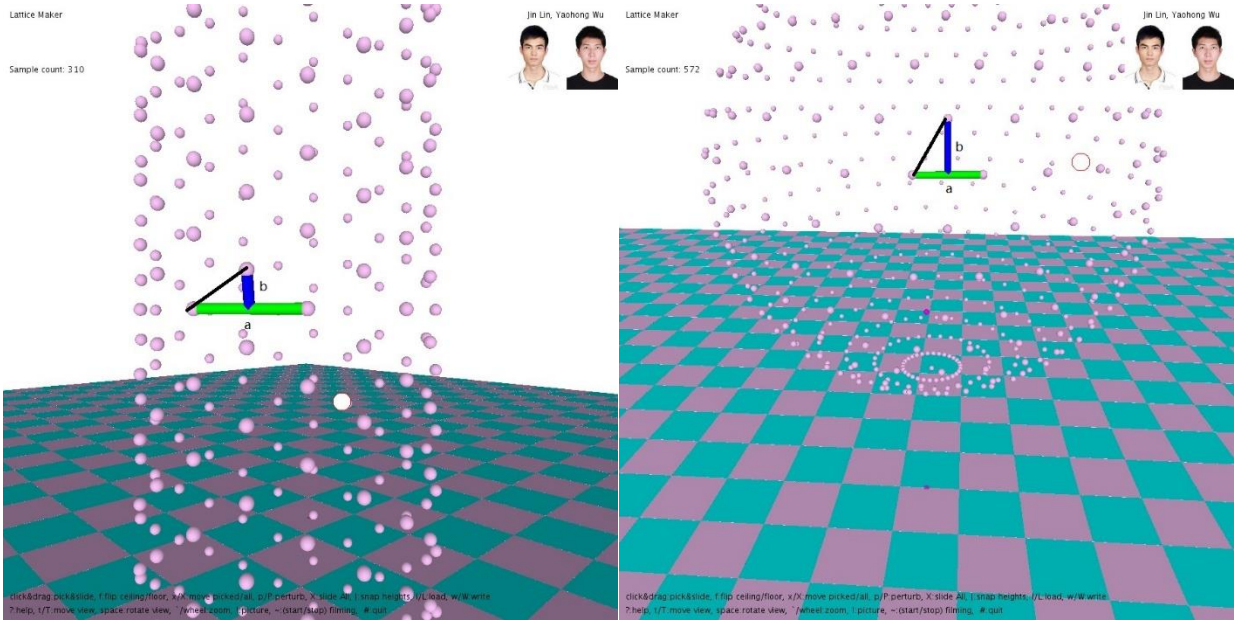
5.4.2 Front Updating using Hashing

In front updating, we need to find if an edge, that connects the hit point we find, has already existed in the front. We use a hash map to store the adjacent edges of a vertex and hence we can quickly tell whether an edge connecting a vertex exists in the front.

5.4.3 Choice of Radius of Pivoting Ball

The performance of ball pivoting algorithm is sensitive to the radius of the pivoting ball. In our context, we can compute an appropriate radius based on our sampling density and our knowledge of the original surfaces.

Let's consider sampling a cylinder first. If we fix at some height h and sample uniformly in [0,2π] as angle w. The arc distance between two adjacent points is $a = dw \times R$, where R is the radius of the cylinder and dw is the angle between these two points w.r.t. the center of the circle at that height. Similarly, if we fix at some angle and sample uniformly in [0, H], where H is the height of the cylinder, then the distance between two adjacent points is $b = H/n$, where n is the number of samples. For each point sampled in this way, we grow an artificial ball with radius r centered at this point. It is easy to know that if $r \geq max(a, b)$, then this ball will not be "empty", meaning there are other vertices in it. It is safe to let $\mathrm{r} = \max(\mathrm{a}, \mathrm{b})$ and use it as the radius of the pivoting ball. Since we use "zigzag" strategy, we should modify b as $\sqrt{(a/2)^2 + b^2}$. Because arc distance between two points on the surface is greater than their Euclidean distance, the r we compute will be a little bit larger, which is not a problem for ball pivoting. The following picture illustrate this computation.



Similarly, for sampling a sphere with radius R, if we fix at a latitude and sample uniformly in $[0, 2\pi]$ as longitude, then the arc distance between two adjacent points is $a = d\alpha \times r \leq d\alpha \times R$, where r is the radius corresponding to the circle at that latitude and $d\alpha$ is the angle between these two points w.r.t. the center of that circle. If we fix at a longitude and sample uniformly in $[-\pi/2, \pi/2]$ as latitude, then the arc distance between two adjacent points is $b = d\beta \times R$, where $d\beta$ is the angle between these two points w.r.t. the center of the sphere. Similarly, we can find an appropriate r in this scenario.

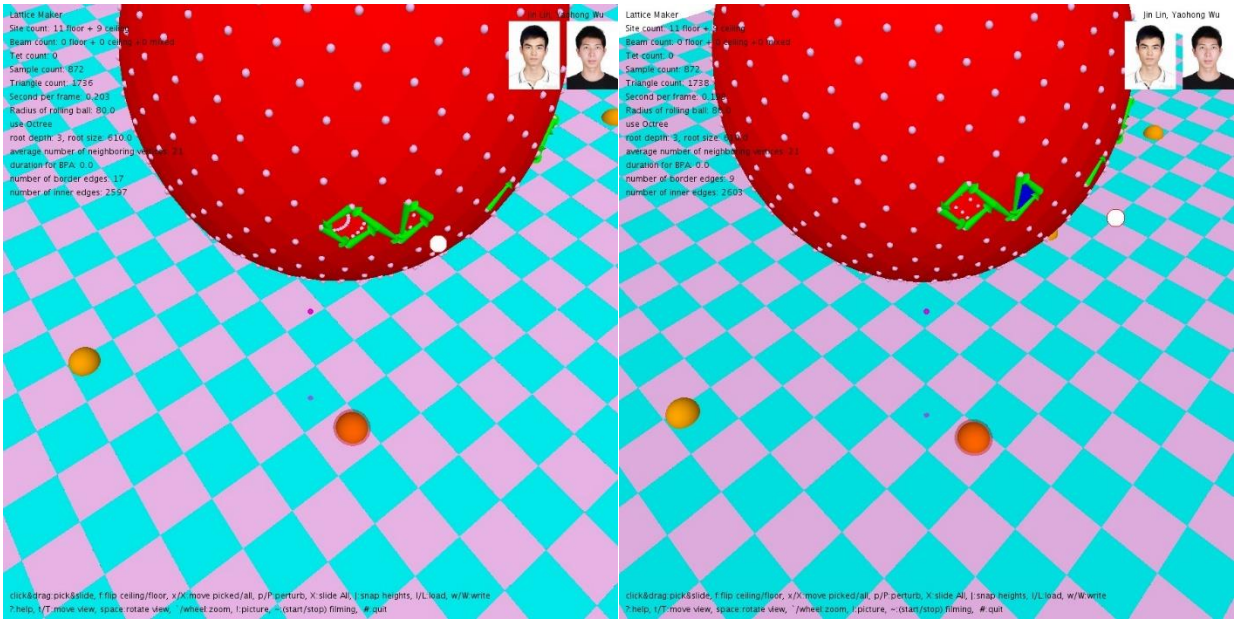
If there are multiple cylinders and multiple spheres, we can combine the possible radii we compute for cylinders and spheres. For example, if there is one cylinder and one sphere, then the radius can be $(r_1 + r_2)/2$, where r_1 is the appropriate radius we compute for vertices on the cylinder and r_2 is the appropriate radius we compute for vertices on the sphere.

5.4.4 Fix Triangular Holes

Sometimes some wrong border edges, or even holes, may be generated due to numerical issues or some specific configurations. We do not know how to completely remove the holes.

A simple method is implemented to remove triangular holes. We iterate over the boundary edges we found. If there are there edges forming an oriented triangle, then we add this triangle to our mesh and remove the three edges from boundary.

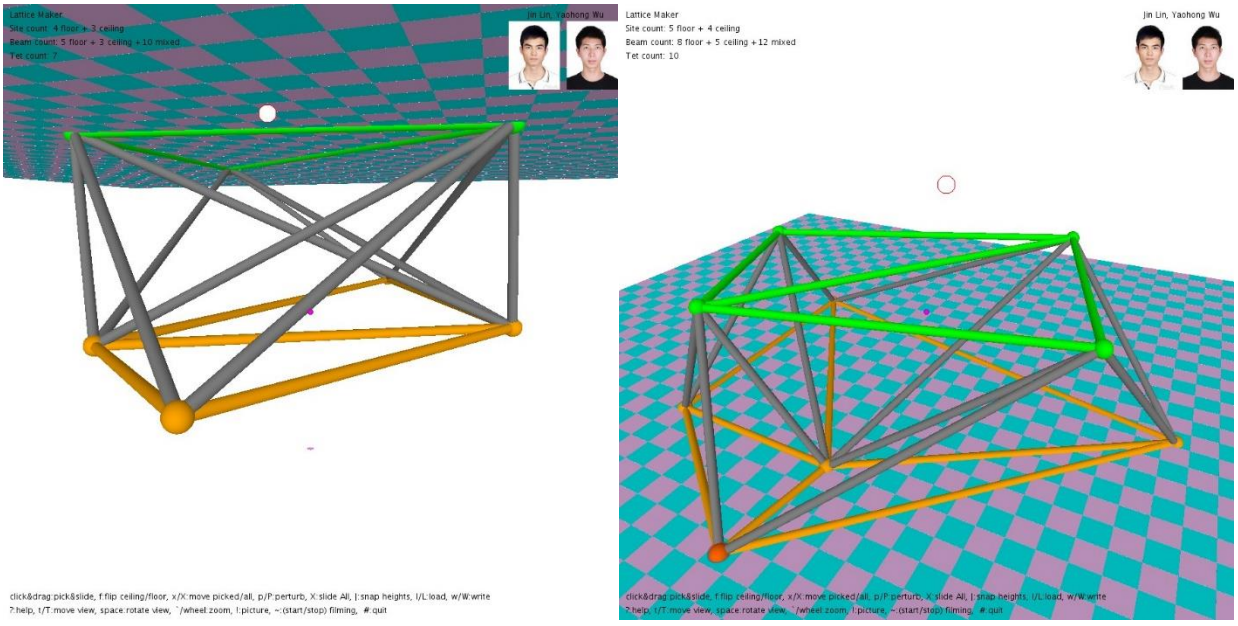
In the left picture shown below, green edges are boundary edges. We can see that there is a triangular hole. In the right picture shown below, we fix this triangular hole by adding a triangle shown in blue.



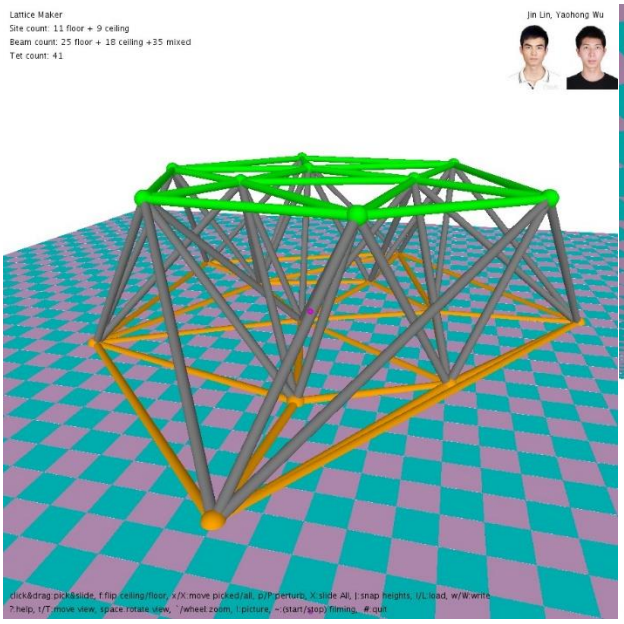
6. Results

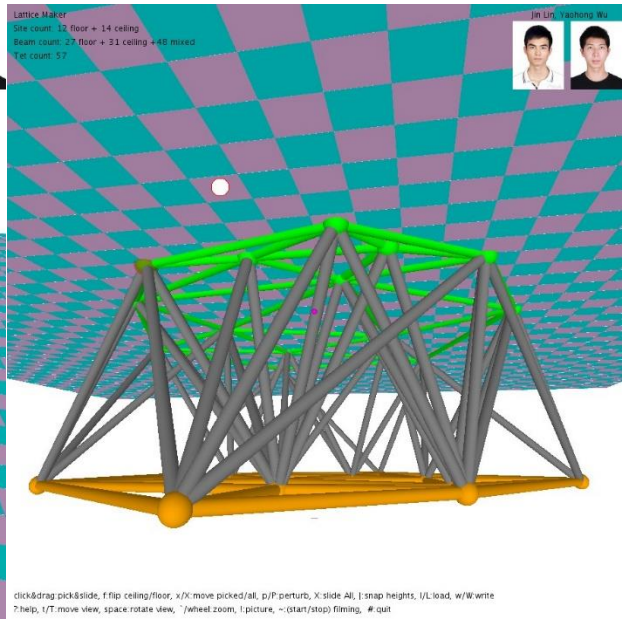
6.1 Results of Tetrahedralization

In the following two pictures, the input sizes are small. Note that the numbers of beams are correct though some beams are shared between several tetrahedra.



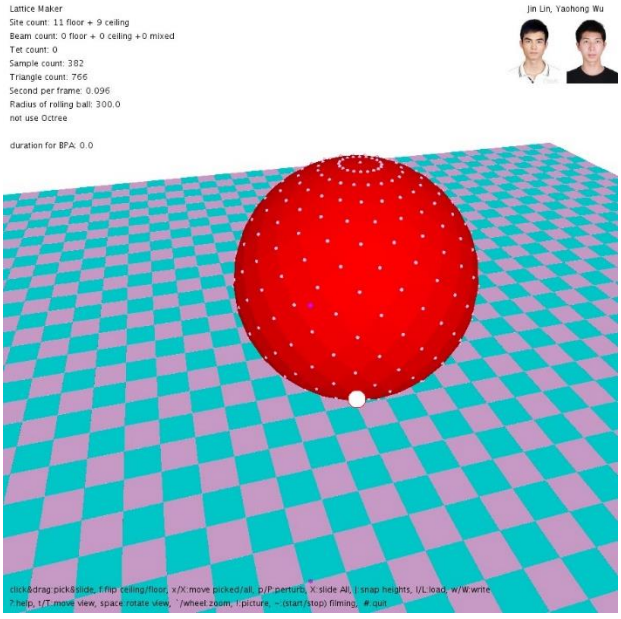
The following two pictures show results on inputs with bigger sizes.

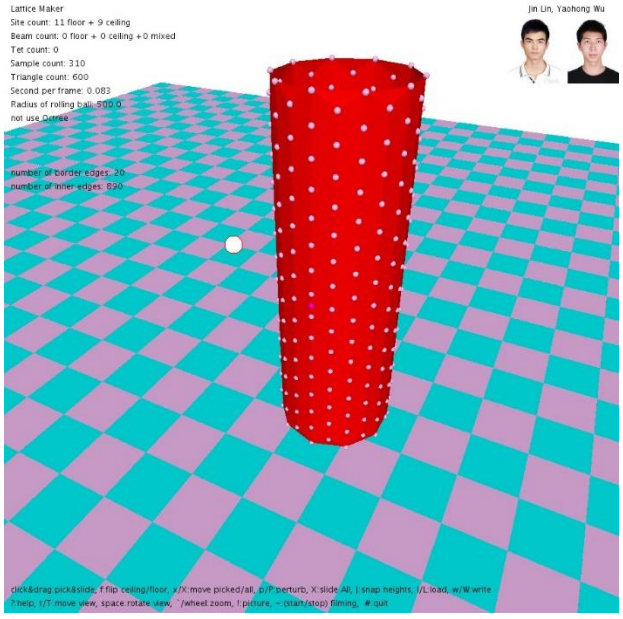




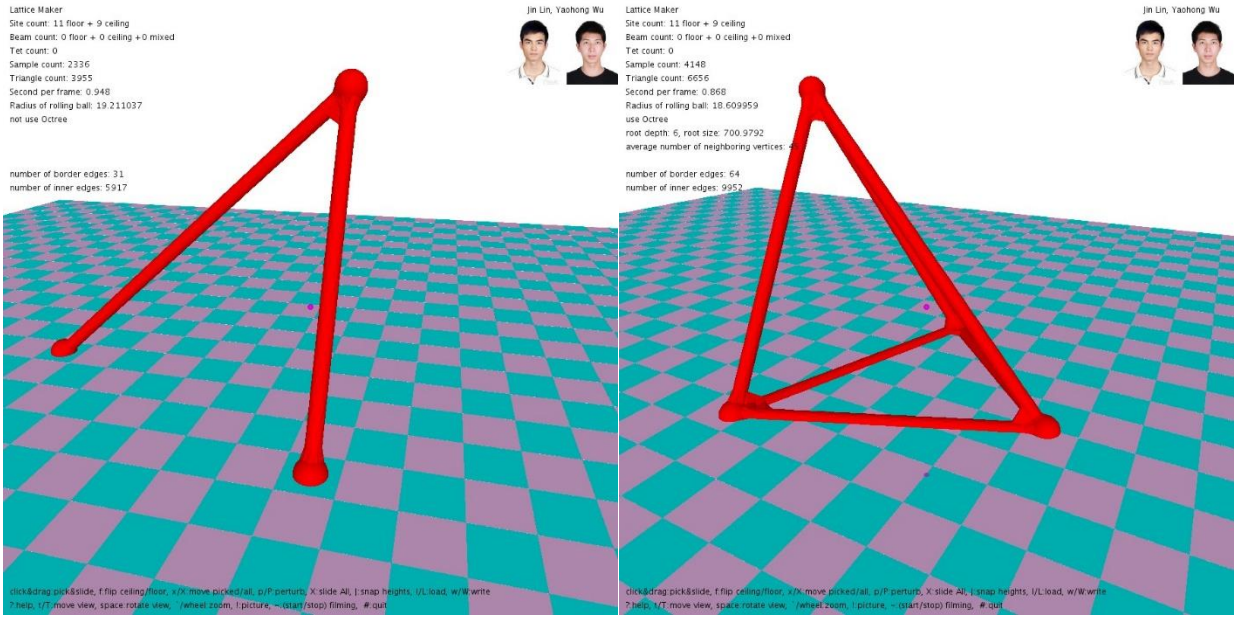
6.2 Results of Surface Reconstruction

Vertices sampled on a ball and a beam.

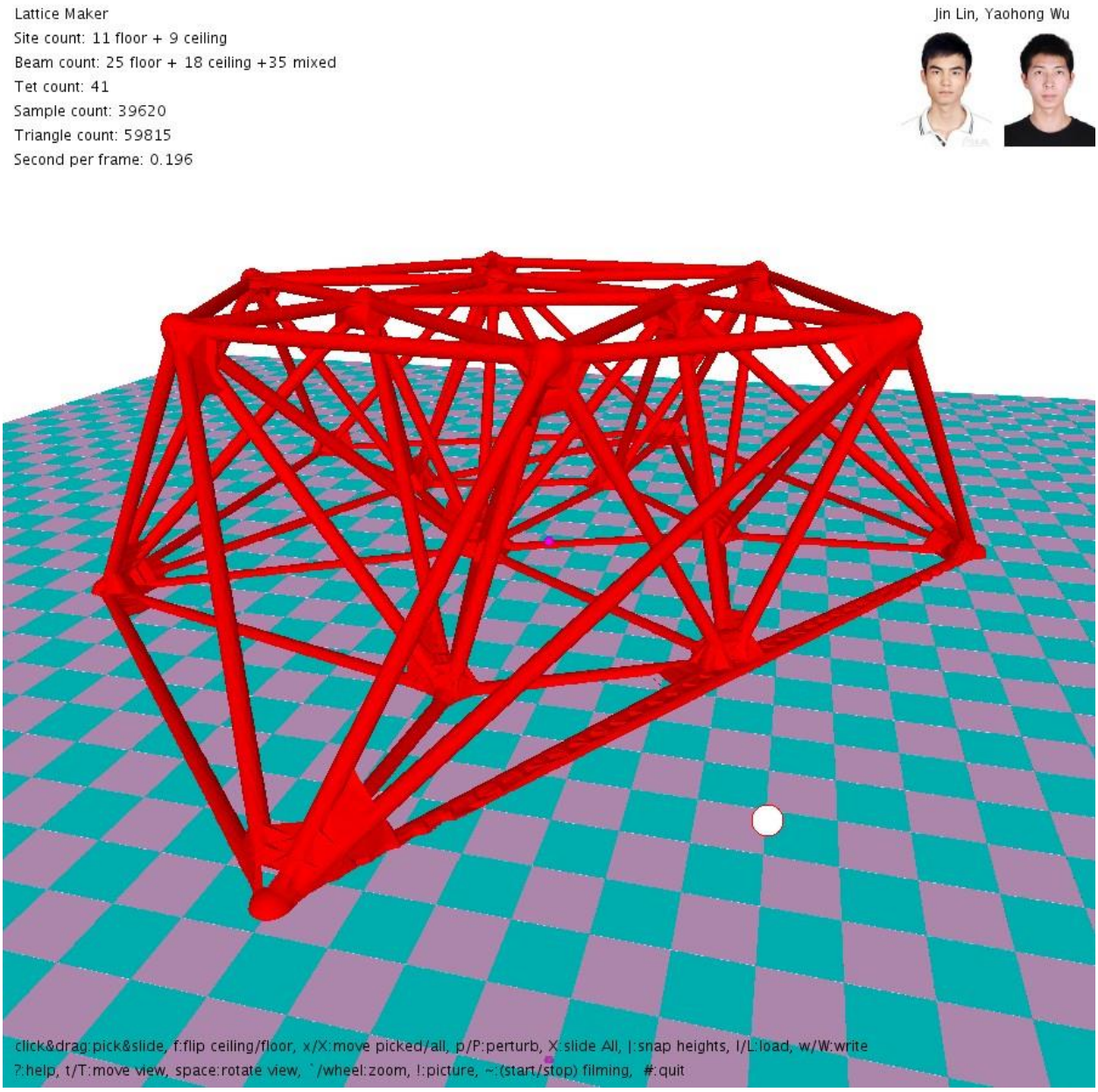




Vertices sampled on simple shapes constructed by beams and balls.



Vertices sampled on beams and balls of a tetrahedra mesh we build in the first part. It only takes less then 20 seconds to reconstruct this mesh though there are around 60k triangles. However, the mesh is not perfect as it contains some holes.



7. References

1. http://www.ipol.im/pub/art/2014/81/article_lr.pdf

2. https://vgc.poly.edu/~csilva/papers/tvcg99.pdf