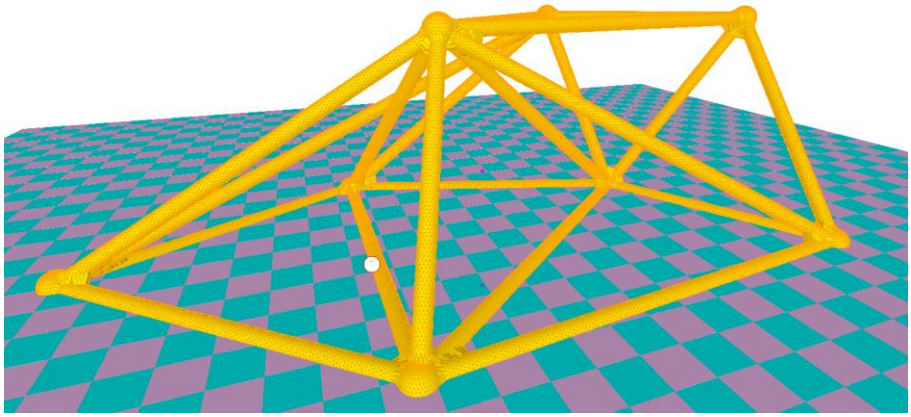
Space Ball Roller

Joonho Kim
CS 6491 Fall 2017
11/14/2017



Problem Statement

We are given two horizontal planes called the floor and ceiling each with a set of balls (sites). Our assignment is to find the Delaunay Tetrahedralization created from edges between the floor and ceiling.

Once given the Delaunay Tetrahedralization, we then form a triangulation of the union of the tetrahedron surfaces to form one continuous triangulation.

Contributions and Level of Completion

I have completed all the requirements to this project.

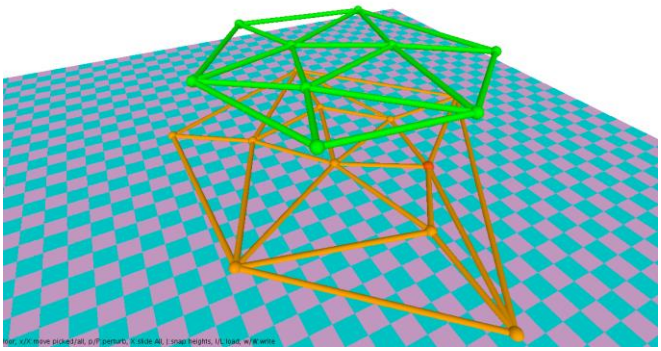
- I have fully completed the Delaunay Triangulation of the floor and ceiling in $O(n^3)$. This is acceptable because it's simple and there aren't that many balls per plane.
- I have fully completed the Delaunay Tetrahedrilization of the union of floor and ceiling points. This was done in $O(n^3)$ by finding tetrahedrons of floor triangles and ceiling points, ceiling triangles and floor points, and floor/ceiling edge pair tetrahedrons.
- I have found a point cloud suitable for my ball-roll. I sample a triangle pattern on my beams as well as on my beam-end spheres.
- I have fully completed the water-tight triangulation.
    - I have optimized this by voxelizing the space into voxels of dimension 200x200x200. When trying to roll the ball, we take the voxel the ball is currently in as well as all any neighboring voxels if they are within the ball-diameter distance. This significantly increased the speed of my program

The restriction to my ball-rolling triangulation is the points must not form perfect squares or rectangles. This is because perfect squares and rectangles have two triangles that share the same circumcenter, so if we roll the ball, the ball doesn't actually roll. This causes a lot of bugs from floating point arithmetic determining whether the opposite point is rolled onto or not. To solve this issue, I created a more jagged triangle pattern without any perfect squares or rectangles.

I think the algorithm I've implemented gives reliable solution with good running time.

Solution Outline

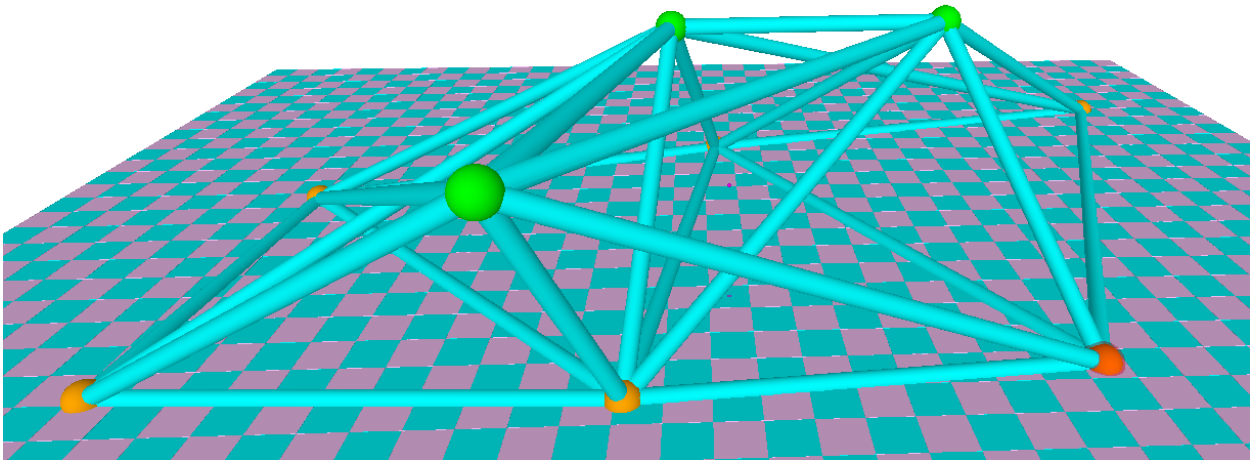
We first find the Delaunay Triangulation of both the floor and ceiling. We compute these with the Bowyer-Watson triangulation algorithm.



To find the Delaunay Tetrahedrilization of the floor and ceiling, we find Delaunay tetrahedrons using 3 sets of data:

- Floor triangles and ceiling points
- Ceiling triangles and floor points
- Edge pairs of floor and ceiling edges

Having the floor and ceiling be planar to their own set of points avoids us from having to run Bowyer-Watson in a 3 dimensional space.

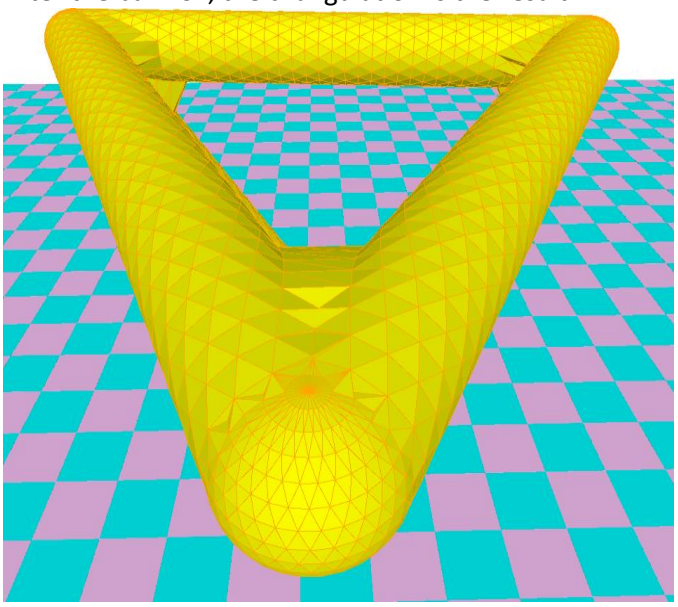


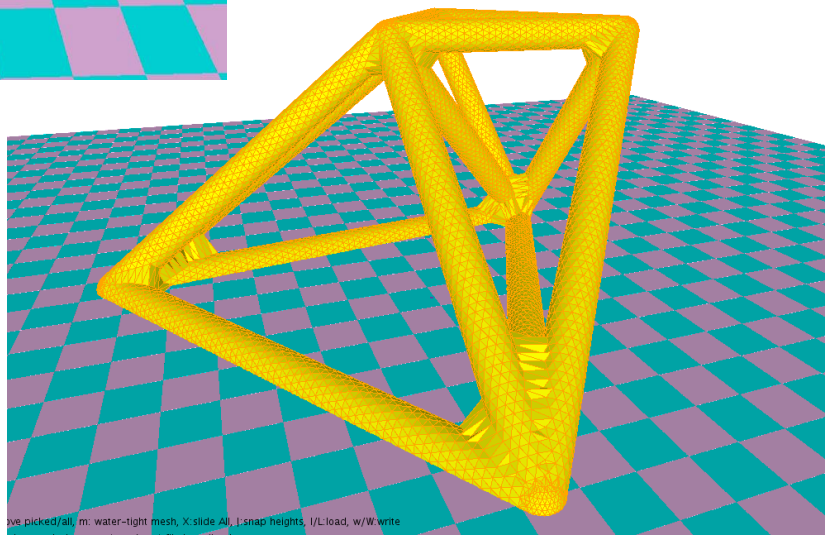
With the tetrahedalization, we want to sample a point cloud to run our ball-rolling triangulation algorithm on. We take all the edges and turn them into beams (cylinders) and uniformly sample points on their surfaces. We also uniformly sample points from the floor and ceiling points using a spherical pattern.

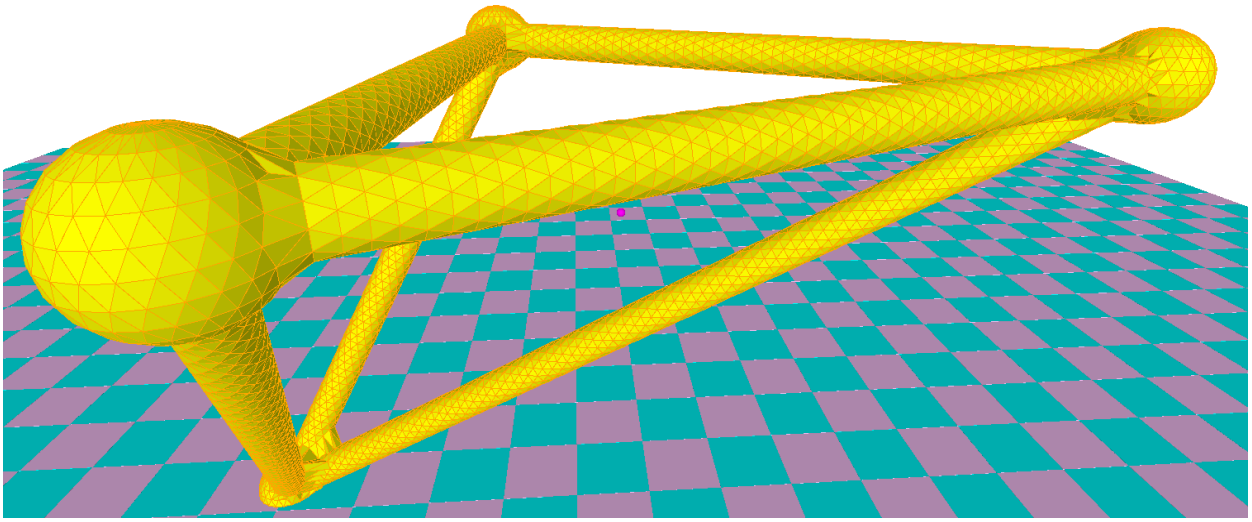
With this point cloud, we voxelize the space into 200x200x200 dimension voxels and place points in their respective voxels. This voxelization helps speed up the algorithm by not having to compare a point to every other point all the time.

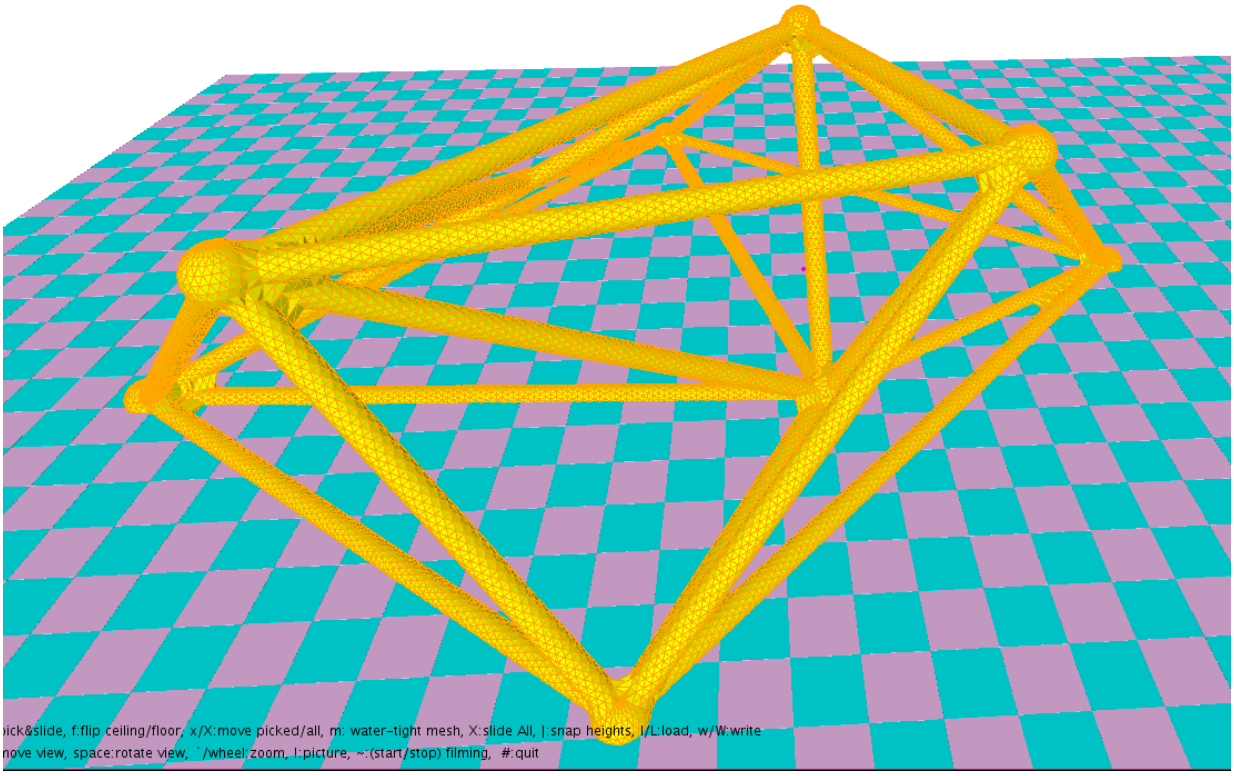
With this voxelized points, we run our ball-rolling algorithm starting on the side of a cylinder. From the current triangle the ball is on, the ball will attempt to roll across two of its edges (excluding the edge it came from) and for each edge find the first points that the ball will "roll onto" with minimal angular change. The point is found by creating a candidate list of points that the ball can land on. We then reduce the candidates by which ball rolls will contain other points, and then finally we find the point that will create the minimum amount of angular roll. We complete a breadth first search from our starting triangle.

After the ball-roll, the triangulation is the result.








ick&slide, f:flip ceiling/floor, x/X:move picked/all, m: water-tight mesh, X:slide All, |:snap heights, l/L:load, w/W:write
ove view, space:rotate view, `/wheel:zoom, !:picture, ~:(start/stop) filming, #:quit

Prior Art

I read the Delaunay Triangulation page on Wikipedia to understand what the topic is. I use the Incremental Bowyer-Watson algorithm to create the triangulation. I found it interesting that the algorithm online starts with a "super" triangle as part of the triangulation and then removes triangles connected with the super triangle. It seems inefficient, but I used it and it creates a nice triangulation. I'll probably rewrite the code to remove it, but for now it's just a draft. The triangulation idea does seem very simple and intuitive.

I also used slides form Jarek Rossignac's class, but most of the project I figured out myself. I followed the advice of Dr. Rossignac to do ball-rolling to find a triangulation, and I used the voxel space idea from my experience in *Simulation of Biology* by Greg Turk to speed of the algorithm by not having to compare a point to every other point but only points within its voxel and neighboring voxels.

Step-by-Step Guideline

We assume that you are given the base code from the assignment.

1. Delaunay Triangulation

We first find the Delaunay triangulation for the floor and the ceiling. The Delaunay triangulation of a set of points is the triangulation in which each circumcircle created by a triplet of points (triangle) does not contain any other point inside of it.

We use the Bowyer-Watson triangulation algorithm on both planes. The Bowyer-Watson algorithm starts off with a "super" triangle enclosing the entire set of points as the base triangulation and then inserts each point one-by-one while flipping triangles whose circumcircle encloses the new point. To find the circumcircle, we find the circumcenter C of the 3 points (A, B, C) with a radius d(C,A).

2. Delaunay Tetrahedrilization

With the floor and ceiling triangulations, we then find tetrahedrons from three different datasets.

- Floor triangles with ceiling points
- Ceiling triangles with floor points
- Floor edges with ceiling edges

We can assume this because the floor points and ceiling points are planer to their respective planes.

Each of these datasets gives 4 points. We create a tetrahedron from these four points by creating two triangles from these points. We then find the circumcenters of each triangle and shoot a normal vector from the circumcenter. Calculate when the circumcenters hit and that is the circumcenter of the tetrahedron.

3. Point Cloud Sampling

From the tetrahedralization, we then create a list of all edges in our tetrahedralization. For each edge, we sample points on the surface to create a point cloud representing a beam (cylinder). To sample the

points, start from one end of the edge and move up the edge in increments of 2*PI*rt/$circumPts$ where rt is the radius of the beam and $circumPts$ is how many points to sample around the circumference. At each step along the length of the beam, sample points around the circumference with radius rt. Also at every other step along the beam, we add an extra 2PI/circumPts/2 degrees to the circumference sampling. This prevents the samples from creating perfect square and rectangles. We don't want these because these quadrilaterals contain two triangles that share the same circumcenter. In our ball-rolling algorithm (depending on floating point error) this will register as no rolling or contained inside another possible ball. We want to avoid this by creating a more jagged beam sample points.

We also want to sample the ends of the beams as spheres. This is simply sampling circumPts points at circumPts/2 heights of the sphere.

4. Voxel Space

To allow our program to compare points faster, we want to voxelize the space. We create a class called VoxelSpace that has a 3D array of lists that hold points. Because the voxel space is an imaginary box, we give VoxelSpace the most positive and negative coordinates as corners ((0,0,0) and (2000, 2000, 600)). For each point, we then add it to the appropriate array entry by integer dividing the points x, y, and z coordinates with the voxelLength (200). These will give us proper indices. Any coordinates that fall outside the box will be indexed at the closest index.

5. Ball-Rolling

To do our ball rolling, we start off by selecting a triangle in the middle of a beam. From our starting triangle we will try to roll over two edges and find points across the edges that the ball will "roll onto". To find these points, we can recognize that the ball will roll over this edge as an axis of rotation. We can also recognize that midpoint of this axis and the center of the ball will be aligned with the circumcenters of any triangle formed by a point the ball rolls onto but these circumcenters cannot extend beyond the length of V(midpoint of the axis, center of ball). We maintain a list of points that fit these requirements.

To check which points to consider, we take all the points in the current voxel the ball is in as well as any of the voxel points the ball can roll into. We check if the edge midpoint can extend into a voxel by adding a vector of length voxelLength in that voxel's direction.

Once we have the candidate of roll-able points, we then create tetrahedrons from the edge and the points and see if they contain any other points. These points are discarded and the rest are kept in a list of betterCandidates.

For the rest of these points, we calculate the angle between the axis of rotation and current ball position and the new ball possible ball position. We take the ball position with the smallest angle. If the ball ends up completely upside down, then we return null.

We continue these steps in a BFS fashion by adding these new triangles into a queue.