Point Cloud to Tetrahedral Mesh Reconstruction

Eldad Peretz, Uria Dai, Nitzan Simchi

December 15, 2021

Abstract

The problem of transformation from point cloud to other representation techniques (i.e. mesh or signed distance functions) is a well known problem in computer graphics. In this work we show how to translate a point cloud into a corresponds tetrahedral mesh. We also deal with the movement crossing problem, define the TG data structure and define a convolutional network which utilizes the geometrical attributes of tetrahedral meshes. Lastly we show methods to increase movements range (explained in section 6) and efficient computation.

1 Introduction

A tetrahedron is a common 3D shape that consist of 4 vertices and 6 edges, and made out of 4 triangular faces. The tetrahedron is a convex shape and is the building block for our work. In the work we use tetrahedral meshes, that are a way to represent 3D shapes by a set of distinct tetrahedrons. A face of the tetrahedral mesh is on the **surface** if no other tetrahedron in the tetrahedral mesh has this face. A tetrahedron is in the **interior** of the tetrahedral mesh if non of its faces is on the surface. It's important to mention that in a valid tetrahedral mesh at most two tetrahedrons shares the same face and cannot share a part of it (i.e. there's no vertex in the interior of any face). Notice that for valid tetrahedral meshes, their surface is watertight. In this work we study the problem of point cloud to tetrahedral mesh translation. The outcome tetrahedral mesh must satisfy several conditions, such as:

- The faces are distinct. We will show that in most common solutions to the problem, the result is in crossing faces which are not valid tetrahedrons.
- The resulting tetrahedral mesh surface must be close in some sense to the original point cloud surface. In this work we try to minimize the chamfer distance between the tetrahedral mesh surface and the original point cloud surface.

In this work we continue the idea of [2] and use the following representation of tetrahedral meshes that, as we see next, with it we can guaranty the required properties described above. The representation is as follows: start with a tetrahedral mesh of a unit cube and for each tetrahedron in the set define an occupancy value, which is a binary value representing whether the tetrahedron is a part of the resulting tetrahedral mesh (i.e. inside the shape) or not. Then the tetrahedral mesh is simply the set of occupied (occupancy value of 1) tetrahedrons. Any interior face (that is not on the unit cube surface) is associated uniquely with two tetrahedrons that share it. An internal face is part of the surface if the occupancy of its two associated tetrahedrons is different (another way to think of it is that an internal face will be on the surface if and only if one of the tetrahedrons touching it is inside the shape [occupancy = 1] and the other is outside the shape [occupancy = 0]. Note that we can assert that by checking that the XOR of the occupancy values is 1). A face that is on the surface of the unit cube is part of

the surface if the associated (exactly one) tetrahedron has occupancy 1 (note that the previous condition is not well defined in this case as there are no 2 tetrahedrons touching the face). We call the tetrahedral mesh of the unit cube we described here UCTM - Unit Cube Tetrahedral Mesh. We will use this name often in this work as this is the whole world of each resulting tetrahedral mesh.

The most important thing about the UCTM is that the tetrahedral mesh it represents is valid, and therefore the surface is watertight. Another point worth mentioning is that the initial tetrahedral mesh of the unit cube enables us to control the resolution of the result, as we can choose to increase the number of tetrahedrons we initialize the UCTM with, and using more tetrahedrons will help the us get higher resolution and more accurate results. We discuss the computational problems with increasing the number of tetrahedrons and explain that in order to increase the resolution (i.e. decrease average edge size) by factor of k, we need to increase the amount of tetrahedrons in factor k^3 which dramatically increase the running time of the solution. In section 6 we discuss how to cope with the problem by subdivision method.

The input for the translation is a filled point cloud without normals. Given a point cloud without normals we can divide it to small pieces and fill each by finding and filling a convex hull (better methods in section 2). Therefore, one can use our translation in order to do tetrahedral mesh reconstruction of **a surface point cloud without normals**. To our knowledge, there are no papers on this setting. We show that using the tetrahedral mesh representation it is possible to reconstruct a tetrahedral surface mesh and therefore a triangular mesh of the surface which was not possible until now. In this work we extend point2mesh [3] paper to tetrahedral meshes. This extension solves inherent problem of point2mesh with same techniques and benefits because we used the tetrahedral mesh representation.

The method

1. Construct the initial UCTM
2. Apply a tetrahedral convolutional network and predict for each vertex v of the UCTM a movement vector m_v. (Optional: also for each tetrahedron a new occupancy value, see differential occupancy section for more information)
3. Move the vertices in the direction, calculate the losses described in section 5 and update network weights.
4. If possible (section 5.3 and 6) fix vertices at position. Else, reset to UCTM initial values.
5. repeat 1-4 until loss converges.

Our contributions

1. We continue the work on tetrahedral mesh field which is young in all related to deep learning methods.
2. To our knowledge, point2tet is the first to use filled point clouds in context of tetrahedral meshes reconstruction.
3. Solved problems from DefTet paper, specifically the face crossing problem by more sophisticated regularization.
4. Use iterative method to increase the movement range that (3) enables. (Section 6)
5. A novel convolutional network of tetrahedral meshes that utilizes the unique properties of tetrahedral meshes.
6. Discuss the computational problem in working with high resolution tetrahedral meshes and show a method to deal with that.

2 Filling a Point Cloud

The problem of filling a point cloud is essential in computer graphics. The input might vary from full point cloud with normals of the shape surface to a surface point cloud in which a little component of the normals has orientation mistakes and even point clouds of the surface without any normals. In this section we will show how a point cloud can be filled, which gives a strong motivation for our work. There are some shapes in which filling a point cloud is easy. A family of such shapes are the convex shapes, where we can compute a convex hull and use the section of filling a mesh [3.2] to fill the point cloud. Notice that for convex hull there is a way faster algorithm (logarithmic time in number of faces / vertices / edges) to test whether the point is inside the convex hull. This is not important for our work so we leave this fact as known.

2.1 Filling Point Cloud With Normals

Given the surface point cloud and normals the SDF can be approximated [SIREN PAPER]. We use such an approximation of the SDF, then sample points in the unit cube uniformly at random and for each point decide whether it is inside the shape or not. The decision on any point x is easy, as we apply the SDF and check the sign (if negative (i.e -1) or zero x is inside the shape and otherwise, outside). The examples below, and the way we created samples for our work, is based on this approach. [Figure 2]

2.2 Filling a General Point Cloud (Without Normals)

Now we assume the point cloud contains only the coordinates, without any additional data such as oriented normals or a mesh. A point cloud can be split into convex parts that their union covers the whole point cloud [Dani Work On Weakly...]. The idea is to leverage this fact and understand that given a weakly convex hull one can use Point2Mesh to create an enclosing mesh and then sample from the interior of this mesh. After sampling from all interiors we can attach them into a sample of the mesh

2.3 Using initial mesh

We use the following known claim without a proof.

Claim 2.1. *a point x is inside a mesh if and only if the ray in direction (0,0,1) from x intersects odd number of the mesh faces. The claim doesn't hold for intersecting a vertex or edge (which we show happen with negligible probability).*

Corollary 2.1.1. *The claim holds for arbitrary direction.*

Proof. For direction d we apply rotation in relation to x such that d transforms to (0, 0, 1). Therefore the ray from x in direction d transforms to a ray from x in direction (0, 0, 1). Note that the mesh after applying the rotation is still a triangular mesh, and therefore by applying the claim (2.1) on this ray and the rotated mesh we get the proof. □

Furthermore, the number of vertices is finite but the number of directions isn't. There are no two different directions that intersect the same vertex because the direction can be calculated uniquely given any point ($\neq x$) on the ray. Therefore we conclude that by choosing a random direction, the probability to intersect one of the vertices (similarly, can be proven for edges because edges are null sets) is negligible. In the following algorithm if we intersect an edge or a vertex we can repeat the process of choosing direction.

The algorithm we used is as following:

1. Sample a random point from the UCTM object

2. Check if it's inside or outside the shape by casting a ray in a random direction and counting the number of intersections it has with the mesh

3. If the intersection is a vertex, sample a new random direction and repeat step 2

3 Unit Cube Tetrahedral Mesh [1]

This is the main data structure we use along the work[2]. We implemented most of it from scratch according to our needs and requirements, and we expanded its functionality and efficiency as the project grew more and more complex and required more data structures in order to support more queries more efficiently. Here we will describe the main data structures and classes we use as part of the UCTM module.

3.1 Half Faces

Any face of the UCTM's tetrahedrons is split into two twin faces (HalfFaces). Each of the faces is associated with the same plane, but the plane normal is in an opposite direction to the tetrahedron it's involved in. Every tetrahedron is associated with four half faces that each of them satisfy that for all points in the tetrahedron, the signed distance of each point from each of the half faces is negative and for all points outside the tetrahedron there exists a half face such that the signed distance to that half face is positive. We will use the half face construction to solve the most challenging problem in this work, the movement crossing problem (to be discussed later).

3.2 Differential Point Cloud Sampling From UCTM

The procedure to sample points is as follows: The first step is to calculate the tetrahedrons' volumes (see equation (1)). Then we sample random points from each tetrahedron such that the amount of sampled points is proportional to its volume.

let a, b, c and d be the vertices of the tetrahedron:

$$volume = \frac{|(a-d)\cdot((b-d)\times(c-d))|}{6} \tag{1}$$

In order to uniformly sample from a tetrahedron, we toss 4 random numbers that sum up to 1 which we treat as aggregation weights of the tetrahedron's vertices. Each aggregated point, according to different randomly generated weights, is a single point sampled from the tetrahedron. We sample points proportional to the volume up to some limit, which is the sampling resolution.

Of course, random generator is not differential. But notice that we can treat (and we actually do) the generated weights as constants that are given in advance. That can be handled by any auto-grad library and the gradient propagates to the vertex generator multiplied by the random generated coefficient (in our case, the gradient propagates to a vertex movement network that we will define later).

[1] a friendly reminder: UCTM: Unit Cube Tetrahedral Mesh
[2] note that in the code we refer to the UCTM as QuarTet

3.3 Using UCTM for surface reconstruction

If the UCTM is associated with 0/1 occupancies values than the following procedure defines a unique watertight triangular mesh.

The procedure is based on the fact that each face is a part of exactly two tetrahedrons (unless it is on the boundary). We add a face to the mesh if the two tetrahedrons that shares this face has different occupancy values. If the face is on the boundary we add it only if the tetrahedron it belongs to has an occupancy value of 1.

Claim 3.1. *The resulting mesh is watertight*

Proof. By induction of the number of tetrahedrons with occupancy 1. The base case is trivial due to that a single tetrahedron is watertight. For the induction step, assume we have n tetrahedrons with occupancy 1; Denote this group by S. W.L.O.G. we assume that all the n tetrahedrons are connected together (i.e. for every tetrahedron $T1$ in S, there is another tetrahedron $S \ni T2 \neq T1$ such that $T1$ and $T2$ share a face). Let T be an arbitrary tetrahedron from S. From the induction hypothesis we know that $S' = S \setminus T$ represents a watertight mesh. Now we look on the mesh resulting from the addition of T to $S \setminus T$. Note that by adding T, the only changes to the surface are the removal of the faces that T "hides", those are the faces that connect T to the other tetrahedrons in S, and the addition of the faces of T that are not the latter. Now lets denote by F the set of faces that are "hidden" by T. Because we assume the regular form of the UCTM is as described in section 1, we know that the rest of the faces of T are connected to the vertices of S', with at most one vertex that is not part of S', which is a vertex of T. In both cases, we get a watertight mesh, as the faces added and reduced are touching the same vertices, with or without the vertex that is part of T but not in S'. □

3.4 Initial UCTM occupancies

To initialize the UCTM occupancies, we use the filled point cloud to estimate the fraction of filled volume by the shape for each of the tetrahedrons.

To estimate that volume we count the number of points in any of the tetrahedrons. Then we relate the maximal amount of points in a tetrahedron (or the average of the largest T elements where there is a tradeoff between large to small T we discuss later) as totaly filled. We divide each of the counts by this factor and occupancy of 1 is given only for tetrahedrons with approximated filled fraction of at least 0.2. There is a tradeoff of small and large T values between noice cancelation and accuracy. For our purpose because of the vertex movements, T=1 acts well.

3.5 Differential occupancies

The occupancies might be learnable. For that purpose we define the differential occupancy setting. In this setting the occupancies are real values between 0 to 1. Values greater than 0.5 are treated as "occupied" or more specifically, 1. Notice that binary values cannot be learned well and therefore this definition is essential for the problem.

4 Convolutional Neural Network on Tetrahedral Meshes

4.0.1 Tetrahedral Convolution

For each tetrahedron we define its neighborhood as the four tetrahedrons that share one of its faces (in case it does not have a neighbor for one of the faces, for example if its on the edge of the tetrahedral mesh, we use the tetrahedron itself as the neighbor, using its features the number of times necessary to achieve the correct number of feature vectors for the convolution, i.e. 4). Then we apply the convolution as following:

We take the features vectors of each neighbor, and stack all of them and the features vector of the current tetrahedron to create a single feature vector v. Let n be the number of features of each tetrahedron in the current convolution layer - then we will get a feature vector of size $5 \times n$ (here we use copies of the current tetrahedron's features if we don't have 4 neighbors). Next we apply a linear network on v to get the new features of the tetrahedron.

Note we only modify the current tetrahedron's features, as we don't want to modify the neighbors' features, because we want the same exact convolution on all tetrahedrons.

4.0.2 Tetrahedral Pooling

Set Collapse Definition: a set collapse on the k vertices $\{v_1, ..., v_k\}$ to vertex x is moving any v of the k vertices in the direction $x - v$ and retain the edges v is a part of. The following claims justifies the choice of edge collapse.

Claim 4.1. *After applying edge collapse we get a valid tetrahedral mesh.*

Claim 4.2. *Denote by n the number of tetrahedrons in the tetrahedral mesh. After collapsing an edge e, denote by k The number of tetrahedrons sharing the edge e, remaining tetrahedrons in the tetrahedral mesh is $n - k$.*

From claim 4.2 states that if we choose an initial tetrahedral mesh for the unit cube such that any vertex has relatively small amount of edges (as promised in the way we created it) then edge collapse in the tetrahedral mesh will perform locally. This is not an intuitive fact, we show a proof in the appendix. This is very important in order to design a pooling layer.

Corollary 4.2.1. *For arbitrary tetrahedral mesh, the outcome may be empty.*

An example is a tetrahedral mesh such that all tetrahedrons involved sharing the same edge. If we choose to collapse the shared edge then all the tetrahedral mesh collapses and we get an empty shape.

Notice that collapsing a face is equivalent to collapsing any two edges of the face. Also notice that every pooling must involve edge collapsing because in order to decrease resolution we must merge vertices and that is a collapse. Therefore we left to show the third option which is tetrahedron collapse and conclude that edge collapse decreases less resolution then the other two and therefore we prefer to use it, of course with some limitations that will follow from the following claim.

Claim 4.3. *For any tetrahedral mesh, if collapsing any of the tetrahedrons involved we get an empty shape.*

Therefore, the collapsing limitation is that we are not allowed to collapse 3 edges of the same tetrahedron (i.e. collapse the tetrahedron). We just to clarify that a tetrahedron can be removed as a consequence of collapse but the entire resulting tetrahedral mesh is not empty.

The unpooling is by a lookup table as in [4].

5 Tetrahedral Mesh Reconstruction Given Point Cloud

In this section we show our approach for that problem. We first describe the network used and the basic losses for the solution. Then we discuss the fundamental problem of tetrahedral mesh reconstruction we dealed with and the solution to that problem. In the next section we discuss on tetrahedrons subdivision and iterating methods that is used to increase performance and running time.

5.1 The network

The network is used to dynamically change the tetrahedral mesh. By generate movement direction for any vertex and move the vertex in this direction, or by changing the occupancy of tetrahedron in the tetrahedral mesh. Notice that if we sample points from tetrahedrons with high occupancy values, then both of that networks are essential. The problem is that the sampled points are calculated from the vertices locations only and are not affected by the occupancy. The number of points sampled from each of the tetrahedrons is depends on the occupancy but if we apply chamfer distance, the gradients does not propagate to the occupancy values. Therefore in the current implementation we do not change the initial occupancies and we only change the vertices locations. The initial occupancies captures the shape well as we see in section [3.4] by utilizing the fact we have a filled point cloud and we can calcualte approximatly the fraction of filled volume for each tetrahedron. The only problem is that we must choose between use the tetrahedron or not, and because we start from unit UCTM we get a spiky result. Notice that the technique of vertices movement is ideal for that setting, it just have to make the result look smoother as the shape must be there. But there's also a limitation, as discussed in section 6, there are point clouds that requires high resolution initial UCTMs. In this work we don't deal with this scenario and leave it to future work. We think that a good way to approach it is by defining a loss that does propagate the gradients to the occupancy values and we show in section 7.1. Of course we tried them and they gave poorly results, maybe a variant of them or another ideas can lead to suitable loss.

The network consists of a shared network which captures the semantic information of the shape. Then there are two heads (for occupancy and vertices movements generation) that utilize the information extracted by the shared network. The shared network consists an embedding layer that we found it is well suited for our overfitting task, more than random weights and a convolutional network for tetrahedral meshes. The heads consists of traditional linear layers.

5.2 Tetrahedral Mesh Reconstruction Loss

5.2.1 Point clouds distance loss

The first and main loss in this work is based on point clouds distance. Specifically, as described before in 3.2, we sample deferentially points in the interior of the tetrahedral mesh (from tetrahedrons with occupancy value of 1). Then we compare the sampled point cloud to the ground truth filled point cloud and minimize this distance.

As discussed at start, we can start with either the point cloud surface without normals or a filled point cloud (that can be created from the surface without normals as discussed). We noticed that if we add another loss which is to minimize the chamfer distance between the surface to the mesh derived from the UCTM we do not get any benefit. We think that for future work a promising direction is to find a loss that compares the resulting mesh and the shape surface, maybe the loss presented in neural subdivision paper [5] can be adapted to that.

5.2.2 The Movements Crossing Problem

The movement crossing problem, which is left as open in DefTet paper. The problem is that the vertices moves might cross other faces and if it does we don't have a tetrahedral mesh anymore. In order to get a meaningful results and to make the process automatically as possible with less as possible limitations that guaranties the vertices movements will not suffer from that loss. Further more, we show how with iterative method (section 6) we allow updates that weren't allowed before and proving facts that actually using the limitation we show in this section combined with the iterative method of section 6 and an assumption that the tetrahedrons are with angles between faces of at least a constant α small as needed but constant through all computation (that can be guaranteed by forcing it and simple loss that we know converged very fast for small α values). To describe the solution we need the following definition.

Vertex Tetrahedrons Group (in shortcut vertex TG) each vertex v is a part of M_v tetrahedrons $T_1...T_{M_v}$. This set is the tetrahedrons group of v. The faces that doesn't contain v are a convex hull. Lets look at the graph defined by the tetrahedral mesh vertices V and edges E, then for any two vertices that are not adjacent, their tetrahedrons groups are disjoint. Therefore if we move a set of vertices that every two are not adjacent, if we restrict the movement to stay in their tetrahedron group interior, then there are no crossings (because the tetrahedral groups interiors in this case are disjoint).

But as stated before, we allowed move all the vertices. Therefore we enable the movement to be in smaller set which is all the points in the tetrahedral group convex region x that $dist(v, x) > min_{f \in TGfaces} dist(x, f)$. We can restrict that using either **PGD** (projective gradient decent) or by loss. In this work we used the loss solution although the other solution is also possible. Lets look at the loss and argue that if the loss is 0 then there are no crosses. Notice that one can use instead PGD and solve this problem using the vertex tetrahedral group definition for all instances. We noticed that the loss does converges to 0 almost all times and therefore we decided to use that in this work.

$$\sum_v max(0, min_{f \in TGfaces} dist(v + move_v, f) - ||move_v||) \tag{2}$$

5.2.3 Improved Tetrahedral Mesh Reconstruction Loss

We add this two losses to encourage smoother reconstructed meshes.

straighten two adjacent faces We add a loss on the surface mesh faces which is for any two adjacent faces we encourage the angle between the planes of the faces to be close to π.

Low variance edges sizes / tetrahedrons volumes / faces sizes This loss minimizes the empirical variance of each of the following sets - edges sizes, tetrahedrons volumes and faces sizes [2].

The final loss is the aggregation of those two losses with, the chamfer distance based loss (5.2.1) and the movement crossing loss (5.2.2). We did use another many losses that we discuss in section 7.2.

6 Iterative approach and Tetrahedral subdivide

The iterative approach means that after T iterations, where T is an hyperparameter, the initial UCTM changes its vertices initial positions. This call **fix at position**. We can fix at position only if the moved vertices are a valid tetrahedral mesh. Another definition for this section is **divide tetrahedron** in which we divide a tetrahedron into 4 equal pieces. We add 4 edges from the tetrahedron center to each of its vertices. This is how the division is performed. The division is performed on the original UCTM (before movements) and therefore mostly done after applying **fix at position**.

6.1 Run time influence

Applying calculations on UCTM with many tetrahedrons is costly, in our work it adds additional run time proportional to the number of tetrahedrons in the UCTM. In order to get higher resolution results we need to minimize the tetrahedrons size. Decreasing their edge size by factor of k increases the number of tetrahedrons in the UCTM in factor k^3. Recall that all the computation cost is proportional (linearly) to the number of tetrahedrons in the UCTM, that leads to k^3 increase in run time for each iteration.

We show how we can use subdivide method combining with the solution to movement crossing problem in order to achieve results of same quality as applying point2tet on higher resolution UCTM with run time close to applying on much lower resolution UCTM.

6.2 Increase Movement Range

After enough iterations we get that vertices movement loss is 0. In that case, there are no crosses and if we fix the positions (i.e. we do not reset the UCTM to the initial locations but move each vertex in the resulting direction) and calculate again the half faces, then we have for any vertex another tetrahedral group that enables it to move further. This is important when we use high resolution UCTMs, because then the vertices movement allowed region is small.

6.3 Update Network

After subdivide we have to add to the embedding layer additional columns for the added tetrahedrons. We found that a good approach for that is giving them the column values of the tetrahedron they are generated from.

7 Future Work

7.1 Relating Occupancy and Point Cloud

We tried to make the occupancy learnable by defining a new type of weighted chamfer losses. The basic idea is to give each of the sampled points a weight which is the occupancy of the tetrahedron it came from. Then in calculating the chamfer distance between src to ground truth (gt) (the forward) we multiply the smallest distance of each point by the weight to encourage the network to decrease the occupancies where is far from the ground truth. And in the backward (gt to src) we multiply each distance by minus of the weight which encourages the right tetrahedrons (those that has a closest point in the ground truth) to be with higher occupancy. We still think it is an interesting direction that can do another step in the study of deep learning and tetrahedral meshes, as we get learnable occupancies this way.

7.2 Improved loss functions

References

[1] Crawford Doran, Athena Chang, and Robert Bridson. Isosurface stuffing improved: acute lattices and feature matching. In ACM SIGGRAPH 2013 Talks, pages 1–1, 2013.

[2] Jun Gao and Wenzheng Chen and Tommy Xiang and Clement Fuji Tsang and Alec Jacobson and Morgan McGuire and Sanja Fidler. Learning Deformable Tetrahedral Meshes for 3D Reconstruction. 2020

[3] Rana Hanocka, Gal Metzer, Raja Giryes, and Daniel Cohen-Or. Point2mesh: A self-prior for deformable meshes. 2020

[4] Hanocka, Rana and Hertz, Amir and Fish, Noa and Giryes, Raja and Fleishman, Shachar and Cohen-Or, Daniel. MeshCNN: A Network with an Edge. 2019

[5] Hsueh-Ti Derek Liu, Vladimir G. Kim, Siddhartha Chaudhuri, Noam Aigerman, Alec Jacobson. Neural Subdivision. 2020

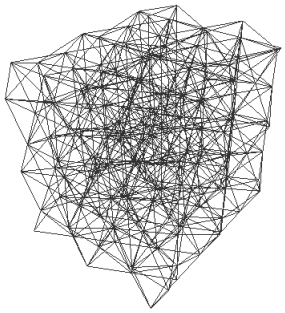
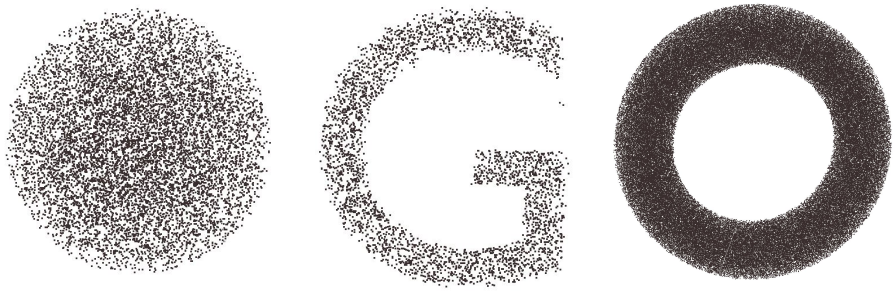
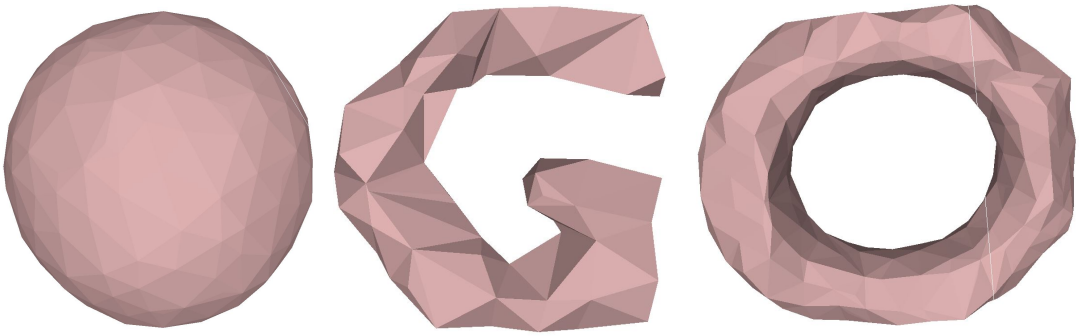


Figure 1: Example Tetrahedral Mesh - unit cube without the surface



(a) sphere (b) G (c) torus

Figure 2: filled point clouds examples with different density



(a) sphere (0.05) (b) G (0.07) (c) torus (0.05)

Figure 3: mesh reconstruction: shape(tetrahedrons volume)