1 Introduction

In membrane biophysics, the problems concerning endo- and exo-cytosis (cellular processes that bring material into or expel material from the cell) and virus budding (a mechanism by which a virus leaves the infected cell in search of a new host) involve understanding the systems of cell membranes with protein particles embedded (see figure 1). A simple model for such a system would be a membrane with interacting particles that carry charges and exert attractive or repulsive forces on each other. The interacting particles represent proteins on the cell membrane. One of the models (see paper [1] is called Finite Element Method, which triangulates the surface to do a good approximation (see figure 2). In figure 2, the triangulated surface represents the actual surface, and the dot points represent the protein particles. An important step in Finite Element Method involves finding the location of a point in 3D space efficiently, and it is the therefore the goal of this project.

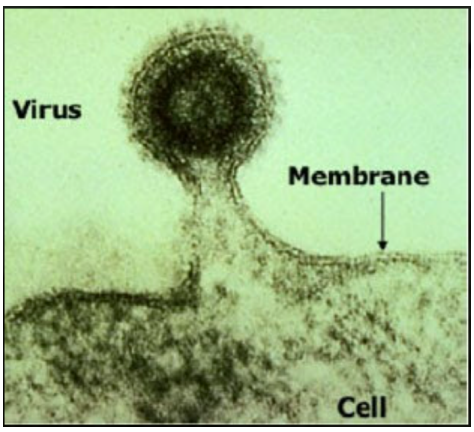

Figure 1: Virus budding process of cells.[H.R. Gelderblom, Robert-Koch-Institute, Berlin]

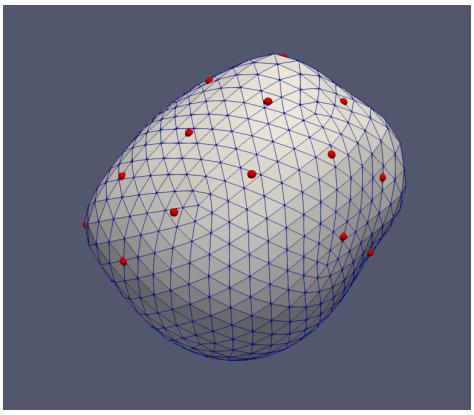

Figure 2: Schematic of particles on triangulated surfaces.

2 Literature Review

In this seciton, we focus on algorithms with limit in 2D space in a hope that it can be extended to point-location problems in 3D. There are many existing algorithms and data structures for graphs and triangulations, and we will only discuss three different methods here. For a more detailed discussion, we refer readers to papers about other methods such as Triangulation Refinement [3] and Monotone Subdivisions [2].

2.1 Slab Decomposition & Trapezoidal Decomposition

In **slab decomposition**, we store each edge segment using the face directly above it in a particular slab. A slab is defined as the space between two vertical lines. To search for a query point q, first we do a binary search along the x-axis to find the slab in which q is located, then do another binary search within the slab to find the edge segment directly below q, then the face associated with that edge segment is the face we're looking for.

Time complexity Given n edges, there are at most $2n$ vertices in any planar subdivision; and a slab can cross at most n edges. Hence time complexity is $O(\log 2n + \log n) = O(\log n)$.

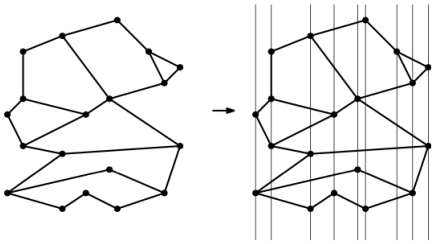

Figure 3: Slab decomposition.

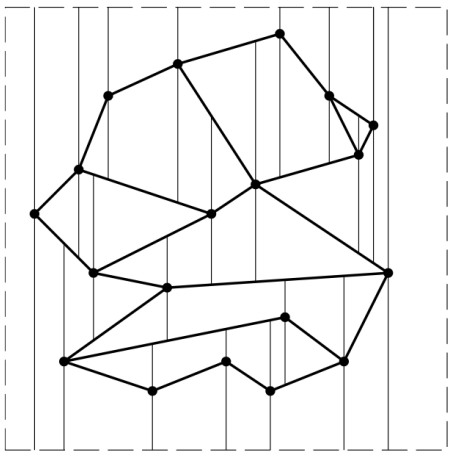

Figure 4: Trapezoidal decomposition.

Even though slab decomposition is straight-forward conceptually, the corresponding data structure requires quadratic complexity. A different but related method has been developed, and it is called **trapezoidal decomposition**. In trapezoidal decomposition, we simply make the vertical lines stops when it reaches a line segment. To represent a trapezoidal/triangular face, we use 4 elements given that the trapezoid itself is denoted as Δ.
A randomized incremental algorithm (elements are added in a uniformly randomized order) both computes the trapezoidal decomposition (*trapezoidal map*) $T(S)$ of a set S of n line segments in general position (meaning there are no crossing line segments and no two distinct endpoints lie on a common vertical line) and a search structure D for $T(S)$ in $O(n \log n)$ expected time. The expected size of the search structure is $O(n)$ and for any query point q the expected query time is $O(logn)$. Several facts about trapezoidal decomposition are listed below.

2.2 Jump and March Algorithm

Jump-and-March algorithm only applies to Delaunay triangulations, and it consists of three steps (see paper [4]). In the first step, we randomly sample m points out of n nodes; then we pick the sample point that's closest to the query point; in the final "march" step, we connect those two points and traverse through all triangles intersected by the line segment until we reach the one where the query point lies. The steps are summarized as follows: Given a Delaunay triangulation of n points $\{X_1, X_2, \ldots, X_n\}$.

1. Select m points $Y_1, ..., Y_m$ at random and without replacement from $X_1, X_2, \ldots, X_n$.
2. Determine the index $j \in 1, \ldots, m$ minimizing the distance $d(Y_j, q)$. Set $Y = Y_j$.
3. Locate the triangle containing query point q by traversing all triangles intersected by the line segment (Y, q).

3 Spherical Triangulation

Since our ultimate purpose is to locate a point within a triangulation embedded in 3D space, we will first look at how to approach that on a triangulated spherical surface (figure 5). Generally, we need two

coordinates to describe the surface which are denoted by (θ, ϕ) in convention. And in spherical case, we can use spherical coordinates directly where θ and ϕ are defined as the angles depicted in figure 5.

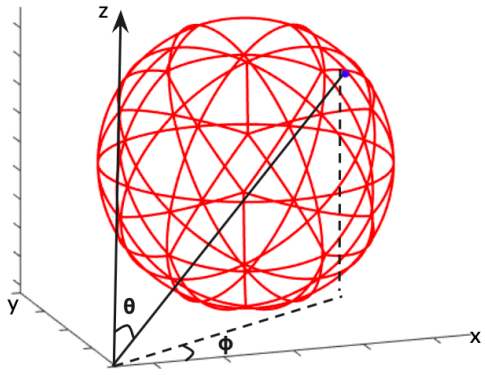

Figure 5: Triangulated spherical surface with (θ, ϕ) coordinates.

The aforementioned algorithms in 2D space turn out to be inapplicable to 3D space, because there will be distortion issues with the curves on surface and their projection on plane. Take the stereographic projection of a triangulated spherical surface as an example. figure 6 is the projection of the vertices connected by straight lines, and figure 7 adds the projection of the curves on the surface. As can be noted, the projection of the query point lies in the intersection of a straight-line triangle and a curved-line triangle, which means the output of searching algorithms in 2D deviates from the actual face the point locates in.

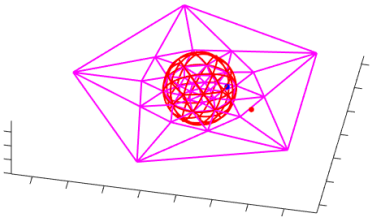

Figure 6: Stereographic projection of vertices

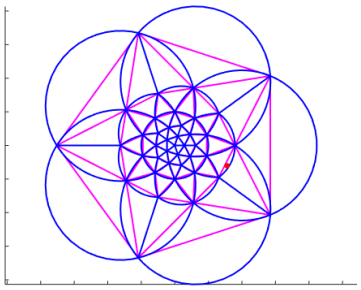

Figure 7: Stereographic projection of surface curves

Fortunately, in the case of spherical triangulation, there is an algorithm called Ray-triangle Intersection that can be applied directly in 3D space without the need for projection. By constructing a set of linear equations.

$$
\begin{aligned}
P &= w\mathbf{A} + u\mathbf{B} + v\mathbf{C}, \quad w = 1 - u - v \\
P &= \mathbf{O} + t\mathbf{D} \\
\mathbf{O} + t\mathbf{D} &= w\mathbf{A} + u\mathbf{B} + v\mathbf{C}
\end{aligned}
$$

given that

$$\begin{aligned}\mathbf{A} &= V_1 - V_0 \\ \mathbf{B} &= V_2 - V_1 \\ \mathbf{C} &= V_0 - V_2 \\ \mathbf{D} &= \langle \sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta \rangle\end{aligned}$$

we can solve for values t, u, v by solving the following matrix equation

$$\begin{bmatrix} -\mathbf{D} & (\mathbf{B}-\mathbf{A}) & (\mathbf{C}-\mathbf{A}) \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \mathbf{O} - \mathbf{A}$$

This algorithm would determine the point of intersection of a ray (vector) and a triangular faces in 3D space, and we can do a brute-force search on all faces using this algorithm.

4 A More Efficient Approach: Binning Triangles

Although we can apply the ray-triangle intersection method to all faces on the surface, it's not as efficient because it searches through an unnecessary amount of triangles. Given a global representation of the query point, its approximate position in the space can be roughly determined; hence it is unnecessary to search faces that locate far from the point. Based on that idea, we developed an intermediate step to narrow down the faces to search, and it is called **Binning Triangles**.
As the name suggests, the method divides the space into integer cubes using integer arithmetic, then it bins triangles to each cube in a way that each cube is associated with nearby triangles. Here is a picture in 3D space:

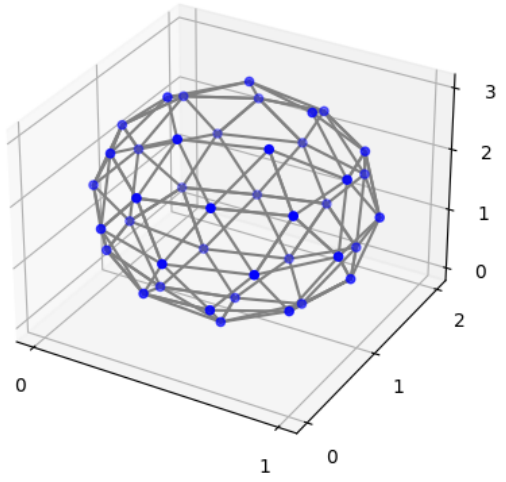

Figure 8: Binning triangles in 3D space

To given a 1D analogy, take a look at the figure 9.

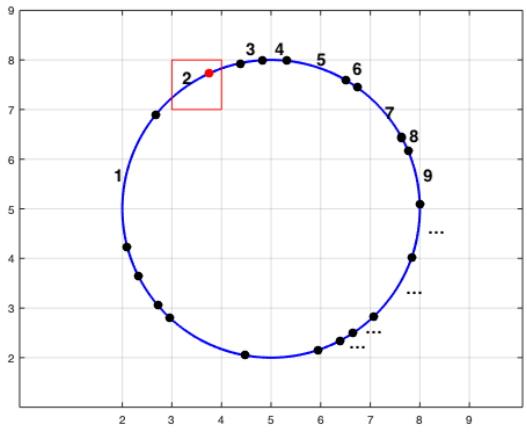

Figure 9: 1D analogy of Binning Triangles

Here the blue circle acts like the surface in 3D space, and the black dots is analogous to vertices in 3D space, and the line segments $1 \dots n$ correspond to faces in 3D space. We define each integer cube by the point at its lower left corner. For example, the cube with red border in the picture will be denoted as $(3, 7)$. The algorithm involves several steps shown in the pseudo-code below.
It takes the vertices and faces of the triangulated surface as inputs and divide the 3D space into cubes. Then it loops through each face, computes *listOfCubes* which are the neighboring cubes and creates a temporary hash table *tempHS* to link each face with nearby cubes. After that, it loops through each cube in the space, filters out the faces to which the cube is linked, and creates a new hash table that connects each cube with intersecting faces (opposite to *tempHS*).

Algorithm 1 Binning Triangles Algorithm

Input: $n \leftarrow$ vertices, $con \leftarrow$ connectivity
Output: A hash table with entries like {cube: face1, face2, ...} for each cube in the space.

```
face ← con
while do face ≠ None
    v1, v2, v3 ← coordinates of three vertices
    xmin ← floor[min(v1_x, v2_x, v3_x)], xmax ← ciel[max(v1_x, v2_x, v3_x)]
    ymin ← floor[min(v1_y, v2_y, v3_y)], ymax ← ciel[max(v1_y, v2_y, v3_y)]
    zmin ← floor[min(v1_z, v2_z, v3_z)], zmax ← ciel[max(v1_z, v2_z, v3_z)]
    listOfCubes = all cubes with index (i, j, k) s.t. i is between xmin and xmax, j between ymin
and ymax, and k between zmin and zmax.
    Append {face : listOfCubes} to a temporary hash table tempHS.
end while
cube ← cubes in space
while do cube ≠ None
    f ← tempHS.keys()
    if cube ∈ f.values() then
        Construct a hash table with entries {cube : f} or append f to cube.values().
    end if
end while
```

Here a sample hash table for the example considered in figure 9: (each cube represented by its lower left corner)

Cube	Element ID
(2,7)	2
(3,7)	2
(5,2)	1
(...,...)	...
(7,6)	5,6,7
(...,...)	...
(4,7)	2,3,4
(...,...)	...

Since constructing this hash table is a matter of integer arithmetic, it is very efficient time-wise. Going back to our original problem, to locate a point in 3D space given its (θ, ϕ) parametrization, we first found its Cartesian coordinates (x, y, z). Then by taking the integer part of (x, y, z), we locate the cube in which the point lies. Now we just need to use ray-triangle intersection to search through the faces associated with that particular cube, instead of searching through all faces.

5 Future Work and Conclusion

Now we have the algorithm working in 3D space, a premise to apply it on any general surface is globally parameterizing the surface in a way similar to the spherical coordinates representation for points on spheres. And we will use the method in paper [5] called Periodic Global Parameterization. Using θ and ϕ as parameterizing variables, we can obtain the trigonometric functions of θ and ϕ by minimizing energy function 1.

$$F = \int_S (||\nabla\theta - \omega\vec{K}||^2 + ||\nabla\phi - \omega\vec{K}^{\perp}||^2)dS \tag{1}$$

Details of the steps are included in the **Appendix**. The next step will be implementing the algorithm in paper [5] to calculate the accurate values for θ and ϕ on each vertex.
So far we learned about the point location algorithms in 2D such as slab decomposition and jump-and-march, and we found the ray-triangle intersection method that works directly in 3D space. We also came up with Binning Triangles Algorithm as an intermediate step to accelerate the process. Although the algorithm for a general surface has not been fully solved due to the global parameterization issue, we do have potential directions for the future, such as the Periodic Global Parameterization and the Global Conformal Surface Parameterization, which can be incorporated into the project.

A Periodic Global Parameterization

A.1 Discretized Energy

To minimize the energy function of the surface, we start by minimizing the energy along every edges of the triangles on the mesh.
The algorithm minimizes the energy:

$$F^* = \sum_T \sum_{i=0,1,2} \lambda_{i,T} F_{T,i}, \tag{2}$$

where $F_{T,i}$ are edge based energies computed for edges $i = 0, 1, 2$ of the triangle T. This term is given by

$$F_{T,i} = ||\mathbf{M}^{r_{i\oplus 2}} \mathbf{X}_{i\oplus 2} - \mathbf{B}_i \mathbf{M}^{r_{i\oplus 1}} \mathbf{X}_{i\oplus 1}||^2 \tag{3}$$

and $\lambda_{i,T}$ are obtained by solving the matrix equation (see Appendix B of []. Here

$$\mathbf{M} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{4}$$

$$r_i = \text{argmax} \left(\vec{K_1} \cdot \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \vec{K_i} \right); r \in \{0, 1, 2, 3\} \tag{5}$$

$$\mathbf{X}_i = \begin{pmatrix} \mathbf{U}_i \\ \overline{\mathbf{V}_i} \end{pmatrix} = \begin{pmatrix} \cos\theta_i \\ \sin\theta_i \\ \cos\phi_i \\ \sin\phi_i \end{pmatrix} \tag{6}$$

$$\mathbf{B}_i = \begin{pmatrix} \cos\delta_i & -\sin\delta_i & 0 & 0 \\ \sin\delta_i & \cos\delta_i & 0 & 0 \\ 0 & 0 & \cos\delta_i^{\perp} & -\sin\delta_i{}^{\perp} \\ 0 & 0 & \sin\delta_i^{\perp} & \cos\delta_i^{\perp} \end{pmatrix} \tag{7}$$

$$\delta_i = \omega/2(\vec{K'}_{i\oplus 1} + \vec{K'}_{i\oplus 2}) \cdot \vec{e_i}; \quad \delta_i^{\perp} = \omega/2(\vec{K'}_{i\oplus 1}^{\perp} + \vec{K'}_{i\oplus 2}^{\perp}) \cdot \vec{e_i} \tag{8}$$

$$\vec{K_i}' = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}^{r_i} \vec{K_i} \tag{9}$$

A.2 Derivation of the Gradient

Given a mesh with N vertices, to derive the gradient of the energy, we start with

$$F^{**} = F^* + \epsilon \sum_i \left((||\mathbf{U}_i||^2 - 1)^2 + (||\mathbf{V}_i||^2 - 1)^2 \right) \tag{10}$$

and define a projection vector $\mathbf{P}_i$ of size $4 \times 4N$ as below:

$$\mathbf{P}_i = \{\mathbf{0}_{4\times 4} \cdots \mathbf{I}_{4\times 4} \cdots \mathbf{0}_{4\times 4}\}$$

and rewriting $F_{T,i}$ using $\mathbf{P}_i$:

$$F_{T,i} = ||\mathbf{M}^{r_{i\oplus 2}} \mathbf{P}_{i\oplus 2} \mathbf{X} - \mathbf{B}_i \mathbf{M}^{r_{i\oplus 1}} \mathbf{P}_{i\oplus 2} \mathbf{X}||^2$$

Now here are the steps:

$$F = \sum_T \sum_i F_{T,i} \tag{11}$$

$$\nabla F = \sum_T \sum_i \nabla F_{T,i} \tag{12}$$

$$\begin{aligned}
\nabla F_{T,i} &= \nabla ||\mathbf{M}^{r_{i\oplus 2}}\mathbf{X}_{i\oplus 2} - \mathbf{B}_i\mathbf{M}^{r_{i\oplus 1}}\mathbf{X}_{i\oplus 1}||^2 \\
&= \nabla[(\mathbf{AX})^\top(\mathbf{AX})]; \quad \mathbf{A} = \mathbf{M}^{r_{i\oplus 2}}\mathbf{P}_{i\oplus 2} - \mathbf{B}_i\mathbf{M}^{r_{i\oplus 1}}\mathbf{P}_{i\oplus 2} \\
&= \nabla[\mathbf{X}^\top(\mathbf{A}^\top\mathbf{A})\mathbf{X}] \\
&= \nabla[\mathbf{X}^\top\mathbf{CX}]; \quad \mathbf{C} = \mathbf{A}^\top\mathbf{A} \\
&= (\mathbf{C} + \mathbf{C}^\top)\mathbf{X}; \quad \text{since } \nabla[\mathbf{X}^\top\mathbf{CX}] = (\mathbf{C} + \mathbf{C}^\top)\mathbf{X} \\
&= (2\mathbf{A}^\top\mathbf{A})\mathbf{X}
\end{aligned}$$

which is a $4N \times 1$ matrix.
To calculate the second part of ∇F^{**}, define two other projection vectors: a $2 \times 4N$ vector $\mathbf{Q}_i$, and a $2 \times 4N$ vector $\mathbf{R}_i$ as below:

$$\mathbf{Q}_i\mathbf{X} = \mathbf{U}_i; \quad \mathbf{Q}_i = \left\{\mathbf{0}_{2\times 4} \cdots \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdots \mathbf{0}_{2\times 4}\right\}$$

$$\mathbf{R}_i\mathbf{X} = \mathbf{V}_i; \quad \mathbf{R}_i = \left\{\mathbf{0}_{2\times 4} \cdots \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdots \mathbf{0}_{2\times 4}\right\}$$

Now we calculate the second part (denoted as $\sum_i g_i$) below:

$$\begin{aligned}
g_i &= \epsilon\left((||\mathbf{Q}_i\mathbf{X}||^2 - 1)^2 + (||\mathbf{R}_i\mathbf{X}||^2 - 1)^2\right) \\
&= \epsilon\left((\mathbf{X}^\top\mathbf{Q}_i^\top\mathbf{Q}_i\mathbf{X} - 1)^2 + (\mathbf{X}^\top\mathbf{R}_i^\top\mathbf{R}_i\mathbf{X} - 1)^2\right)
\end{aligned}$$

$$\begin{aligned}
\nabla g_i &= 2\epsilon\left\{(\mathbf{X}^\top\mathbf{Q}_i^\top\mathbf{Q}_i\mathbf{X} - 1)(2\mathbf{Q}_i^\top\mathbf{Q}_i\mathbf{X}) + (\mathbf{X}^\top\mathbf{R}_i^\top\mathbf{R}_i\mathbf{X} - 1)(2\mathbf{R}_i^\top\mathbf{R}_i\mathbf{X})\right\} \\
&= 4\epsilon\left\{(\mathbf{X}^\top\mathbf{Q}_i^\top\mathbf{Q}_i\mathbf{X} - 1)(\mathbf{Q}_i^\top\mathbf{Q}_i) + (\mathbf{X}^\top\mathbf{R}_i^\top\mathbf{R}_i\mathbf{X} - 1)(\mathbf{R}_i^\top\mathbf{R}_i)\right\}\mathbf{X}
\end{aligned}$$

Then, we have

$$\nabla F^{**} = \sum_T \sum_{i=0,1,2} \nabla F_{T,i} + \sum_i \nabla g_i. \tag{13}$$

References

[1] Sanjay Dharmavaram and Luigi E. Perotti. "A Lagrangian formulation for interacting particles on a deformable medium". In: *Computer Methods in Applied Mechanics and Engineering* 364 (2020), p. 112949. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2020.112949. URL: https://www.sciencedirect.com/science/article/pii/S0045782520301328.

[2] Herbert Edelsbrunner, Leonidas J. Guibas, and Jorge Stolfi. "Optimal Point Location in a Monotone Subdivision". In: *SIAM Journal on Computing* 15.2 (1986), pp. 317–340. DOI: 10.1137/0215023. eprint: https://doi.org/10.1137/0215023. URL: https://doi.org/10.1137/0215023.

[3] David Kirkpatrick. "Optimal Search in Planar Subdivisions". In: *SIAM Journal on Computing* 12.1 (1983), pp. 28–35. DOI: 10.1137/0212002. eprint: https://doi.org/10.1137/0212002. URL: https://doi.org/10.1137/0212002.

[4] Ernst P. Mücke, Isaac Saias, and Binhai Zhu. “Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations”. In: *Computational Geometry* 12.1 (1999), pp. 63–83. ISSN: 0925-7721. DOI: https://doi.org/10.1016/S0925-7721(98)00035-2. URL: https://www.sciencedirect.com/science/article/pii/S0925772198000352.

[5] Nicolas Ray et al. “Periodic Global Parameterization”. In: *ACM Trans. Graph.* 25.4 (Oct. 2006), pp. 1460–1485. ISSN: 0730-0301. DOI: 10.1145/1183287.1183297. URL: https://doi.org/10.1145/1183287.1183297.

Abstract

Student: Xinran(Joyy) Wan
Mentor: Sanjay Dharmavaram, Department of Mathematics

Research Topic, Inquiry Methods, Results

The research question of my project is about algorithms to locate a point on a triangulated surface in 3D space, and the method of inquiry includes literature-reading, writing scripts in Python/Matlab/C++, and discussions between the mentor and other professionals. I read and learned about the existing algorithms in 2D space and implemented the algorithm called ray-triangle intersection that works directly in 3D space. We learned that our original idea of using projections to apply information from 2D to 3D doesn't work because of the distortion issues with lines on the triangulated surface, so we turned to looking for algorithms that works directly in 3D space, and that's where ray-triangle intersection comes into the picture. We tested the method on triangulated spherical surface, then I implemented an algorithm that acts as an intermediate step called Binning Triangles which makes the process of locating points faster by eliminating unnecessary searches. Then we geared towards working on general surfaces in 3D space. However, due to some issues with the paper we're referring to and time limit, we haven't really reached our goal of solving the problem for a general surface in 3D. However, we do solve the basic cases such as spherical surface and Gaussian surface, and have several directions to go in the future.

Reflection

Student: Xinran(Joyy) Wan

Throughout this summer, I've learned a lot both through the research itself and some out-of-research group gatherings. My mentor and I also participated in the Physics Department program where students and mentors meet for Friday lunch where we discuss our researches and projects. It is great experience to connect with students whose fields are similar or related to mine and also just to know them personally. The poster and the talk session at the end of the program are especially helpful, because it gave me a very specific sense of the timeline to prepare a poster presentation and a talk; it also gave me more confidence to do the presentation because originally I viewed it as a very challenging task, but after practicing with my mentor and my friends, and repeating/revising the process over and over again, it turned out to be manageable and I'm glad with the final outcome. As for the PUR community, I feel less connected to them as I am with the students in physics department program, and it is probably because of the large number of students involved and the limited time we spent with each other weekly. However, it's still good to be able to meet other students on campus and chat over the Wednesday ice-cream socials.

As for the research itself, I've gained skills such as how to read an academic paper (eg. which parts need more dedication and which parts can be briefly skimmed in the first round of reading), learned on my own the programming language C++ and wrote scripts based on our goal, as well as how to make good reports and presentations such that the audience can understand the topic without having to know more technical details. The weekly meetings are also very useful in a non-research-related way, as it taught me how to divide a task into different parts and set goals and deadlines for each part. The mentor-mentee relationship also gave me a chance to learn much more than the way to conduct academic research, and I would undoubtedly benefit from this experience in the future on my road to pursue further studies. I also learned about my own strengths and weaknesses. In this program, I feel that I can easily apply skills that I already have even though it's been a while when I used them last time, such as coding and making posters; however, it takes time to get used to new skills such as paper-reading because sometimes I would just forget and go back to my old habits of reading it word-by-word. It also takes longer for me to enter into the state of doing non-coding tasks

than it takes to begin coding, and I think I should work on this because in the future my work would most likely include more than just coding, and sometimes finding the direction and setting the goal are much more important than writing the program itself.

Finally, I'd really want to thank PUR for the grant and summer housing coverage, as it saved me from financial concerns and made sure that I can dedicate to the program itself wholeheartedly.