CG_4_Tutte Parameterzation Report

PB21010479 Caoliwen Wang

March 24, 2024

1 Problem Statement

First, read the Framework3D framework configuration instructions and configure the framework. Currently, there is no need to delve into the implementation of the framework. Simply refer to the documentation and examples to understand how to write a single node .cpp file.

Second, Read about mesh data structures to learn the basic representation of meshes, as well as the OpenMesh library we will use. An example using the homework framework and half-edge structure to compute and visualize curvature of triangular meshes is provided. Besides, Learn how to traverse vertices, faces, edges, and access their neighborhoods using the half-edge structure. The structure is shown in Figure 1.

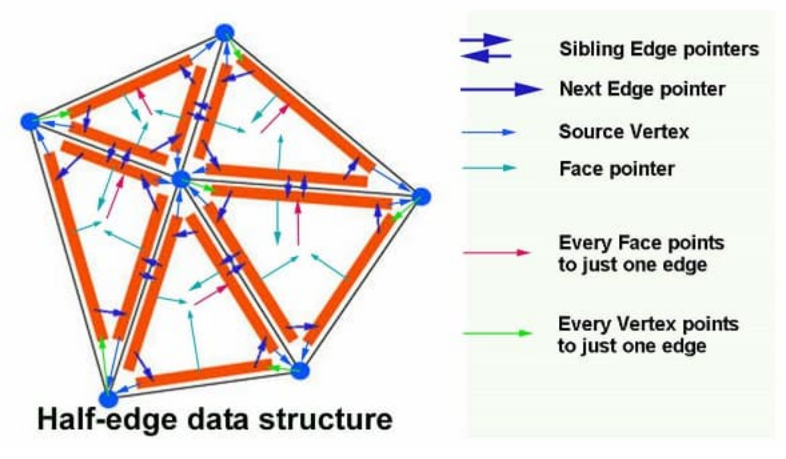


Figure 1: half-edge structure

Then, refer to papers and assignment requirements to implement the Tutte surface parameterization algorithm. We can add node implementations for the algorithm in the specified directory and test the correctness of the algorithm in the framework, as well as the impact of different variable selections on experimental results. We can complete this assignment following the steps below:

1. Implement the establishment and solution of the Laplace equation on the mesh (with uniform weights). The boundary conditions are still chosen as the original spatial point

positions, and you can obtain a "minimal surface" with fixed boundaries by solving it, completing the minimal surface node;

2. Under the same coefficients, modify the boundary conditions to map the mesh boundary to the boundary of a convex area on the plane (square, circle), completing the boundary mapping node;
3. Solve for (uniformly weighted) Tutte parameterization (connecting the above two nodes!), then use the texture mapping node to visualize parameter coordinates;
4. Try and compare different weight selections, such as Cotangent weights (to be implemented) and shape-preserving weights (Floater weights).

2 Propose Algorithms

First, we need to define vertex differentials:

$$\boldsymbol{\delta}_i = \boldsymbol{v_i} - \sum_{j \in N(i)} w_j \boldsymbol{v_j}, \tag{2.1}$$

In equation 2.1, $N(i)$ represents the 1-neighborhood of i. w_i represents the weight of i. We have two definitions of w_i:

1. Uniform weights:: $w_j = 1$
2. Cotangent weights: $w_j = \cot \alpha_{ij} + \cot \beta_{ij}$, the definitions of α_{ij} and β_{ij} are shown in Figure 2;

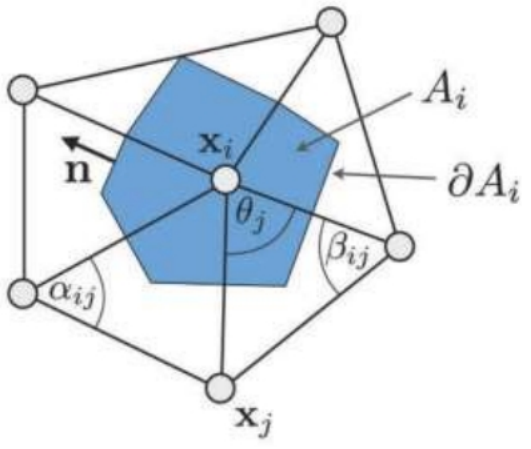


Figure 2: α_{ij} and β_{ij}

However, we need to notice that the coefficients should be normalized:

$$w_j = \frac{w_j}{\sum\limits_k w_k}. \tag{2.2}$$

Then, we can calculate the minimal surface by constructing linear equations. Let $\boldsymbol{\delta}_i = \mathbf{0}$, we know that

$$\boldsymbol{v_i} - \sum_{j \in N(i)} w_j \boldsymbol{v_j} = \mathbf{0}. \tag{2.3}$$

The equation 2.3 gives the restriction to all interior points, to simplify the construction of linear equation, the objective of establishing the equation can be modified to include all types of points. For the boundary points, the restriction is

$$\boldsymbol{v_i} = \boldsymbol{v_i}. \tag{2.4}$$

The right side of equation is given. This way of construction linear equations is much easier. It also does not affect computational efficiency, as boundary points are always sparse compared to the overall set of points.

Then, we can also realize the boundary mapping. First, we need to find the boundary. I write *halfedge_handle_start* to find handle on the boundary, and use this handle to find all the boundary points.

Two different kinds of boundaries are provided.

1. Circle in [0,1]×[0,1];
2. Square [0,1]×[0,1].

The most important problem to be solved is to determine distribution of boundary points. A simple approach is to evenly distribute boundary points along the circumference, but it's obviously not realistic. My approach is to define initial points and then calculate the arc length distance from each point to the starting point in sequence, storing it in *bp_distance*, and then distribute them according to the proportion of this distance to the total arc length. To be more specific, the relationship between coordinates and distances is as follows:

1. Circle: the starting point is $(1,\frac{1}{2})$.

$$x = \frac{1}{2} + \frac{1}{2}\cos(\frac{dis}{sum} * 2\pi), \quad y = \frac{1}{2} + \frac{1}{2}\sin(\frac{dis}{sum} * 2\pi) \tag{2.5}$$

2. Square: the starting point is (0,0).

$$x = \begin{cases} 4 * \frac{dis}{sum}, \ \frac{dis}{sum} \in [0, \frac{1}{4}) \\ 1, \ \frac{dis}{sum} \in [\frac{1}{4}, \frac{1}{2}) \\ 3 - 4 * \frac{dis}{sum}, \ \frac{dis}{sum} \in [\frac{1}{2}, \frac{3}{4}) \\ 0. \ \frac{dis}{sum} \in [\frac{3}{4}, 1) \end{cases} \qquad y = \begin{cases} 0, \ \frac{dis}{sum} \in [0, \frac{1}{4}) \\ 4 * \frac{dis}{sum} - 1, \ \frac{dis}{sum} \in [\frac{1}{4}, \frac{1}{2}) \\ 1, \ \frac{dis}{sum} \in [\frac{1}{2}, \frac{3}{4}) \\ 4 - 4 * \frac{dis}{sum}. \ \frac{dis}{sum} \in [\frac{3}{4}, 1) \end{cases} \tag{2.6}$$

We can assume the boundary points are all in x-y plain, so the z-coordinates are zero. We just use the new boundary coordinates to replace the right side of equation 2.4, then get the construction of boundary mapping.

Finally, we use SparseLU to solve the equation and *set_point* function to reconstruct the 3D object.

3 Programming

3.1 Halfedge Structure

It is shown in Figure 1. Then OpenMesh has provided us many functions to read the information from the mesh structure.

```
for (const auto& vertex_handle : halfedge_mesh->vertices())
```

This loop can obtain vertex-related information.

```
for (const auto& halfedge_handle : vertex_handle.outgoing_halfedges())
```

This loop can obtain half-edge information related to a vertex. Some functions used are shown as follows:

1. halfedge_handle.to(): use the halfedge to find the end point;
2. halfedge_handle.prev(): find the last handle;
3. halfedge_handle.opp(): find the opposite halfedge handle;
4. is_boundary(): judge whether the halfedge and vertex is on the boundary;
5. .idx(): find the index of vertex.

3.2 Node programming

Another interesting task is completing node programming. Node programming refers to the process of writing code or scripts that operate on individual nodes within a system or network. In the context of software development or computer science, nodes typically represent individual elements or entities within a larger system, such as data structures, computational units, or network devices.

Benefits of Node Programming:

1. Modularity: Node programming allows for the development of modular and scalable systems, where individual nodes can be easily added, removed, or modified without affecting the entire system.
2. Concurrent Processing: Nodes can often operate concurrently, allowing for parallel execution of tasks and improved performance in multi-core or distributed systems.
3. Flexibility: Node programming provides flexibility in system design and implementation, as developers can choose the appropriate level of granularity for nodes based on the requirements of the application.
4. Fault Isolation: By encapsulating functionality within individual nodes, node programming can help isolate faults and minimize the impact of failures on the overall system.

Drawbacks of Node Programming:

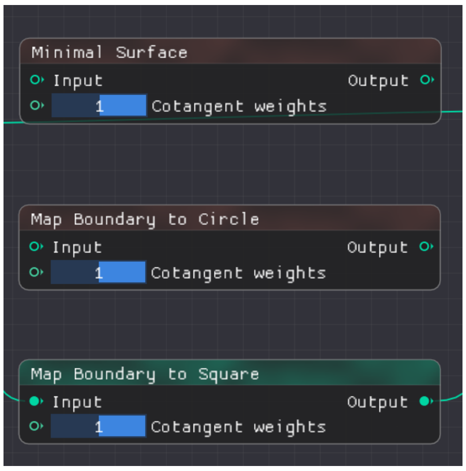
1. Complexity: Managing a large number of nodes and their interactions can introduce complexity into the system, leading to potential challenges in debugging, testing, and maintenance.

2. Coordination Overhead: In distributed systems, coordinating the behavior of multiple nodes may introduce overhead and latency, especially when nodes need to communicate and synchronize with each other.

3. Resource Consumption: Each node consumes system resources such as memory, processing power, and network bandwidth. Managing resource usage across a large number of nodes can be challenging and may impact overall system performance.

4. Scalability Challenges: While node programming can support scalability, achieving efficient scaling in practice may require careful design and optimization to avoid bottlenecks and resource contention.

To be more specific, in this project, I added a dialog box, where the value 0 represents the use of uniform weights, and the value 1 represents the use of cotangent weights, which is shown in Figure 3.



1 Cotangent weights

1 Cotangent weights

1 Cotangent weights

Figure 3: cotangent weights

The final implemented nodes are shown in the Figure 4. The initial state already implements texture embedding.

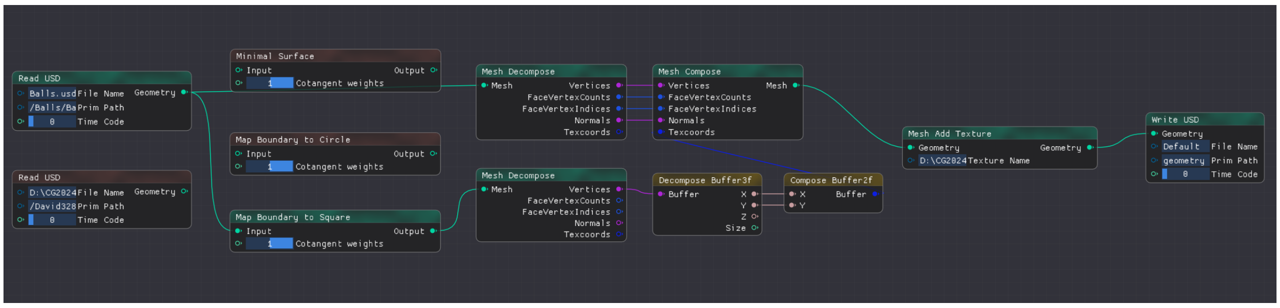


Figure 4: Node Programming

4 Experimental Results

I think the results of this algorithm are fantastic.

First, we use the balls to test the minimal surface, the results are in Figure 5 and 6.

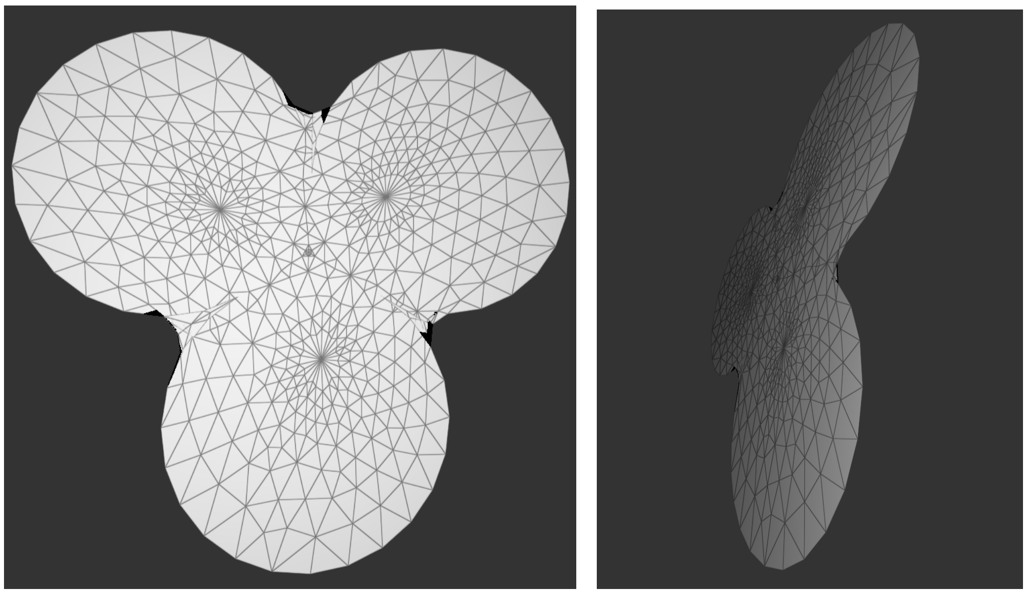
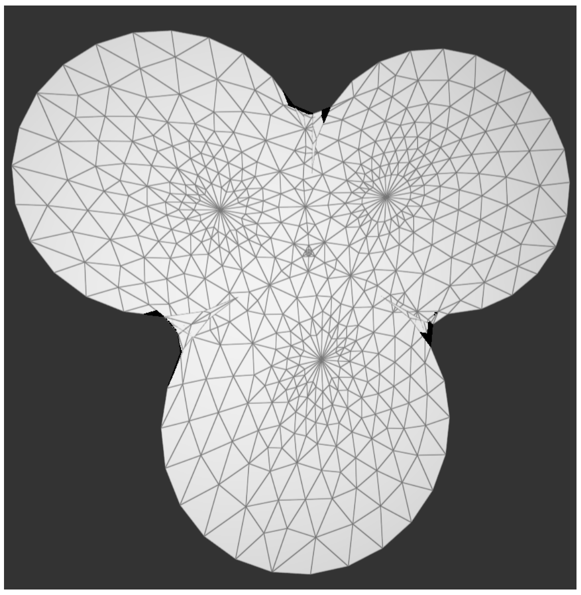

(a) Main view (b) Side view

Figure 5: Balls: Minimal Surface(Uniform Weights)

We can find that the mesh is obviously divided into about three parts, responding to the source figure.


(a) Main view

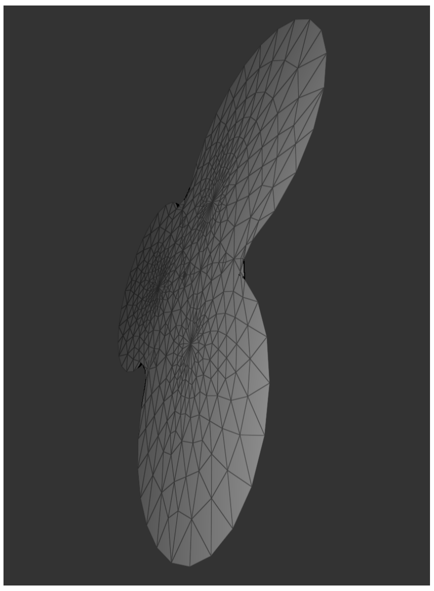
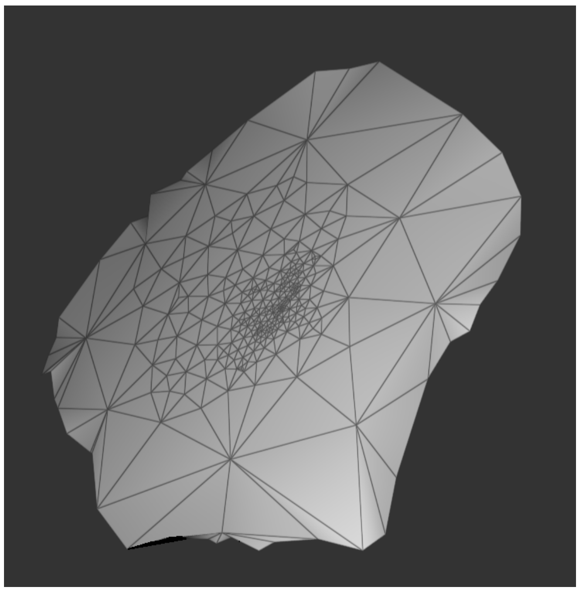

(b) Side view

Figure 6: Balls: Minimal Surface(Cotangent Weights)

We also use David to test. The boundary points of David are generated near the shoulders. The results are shown in Figure 7 and 8.


(a) Main view

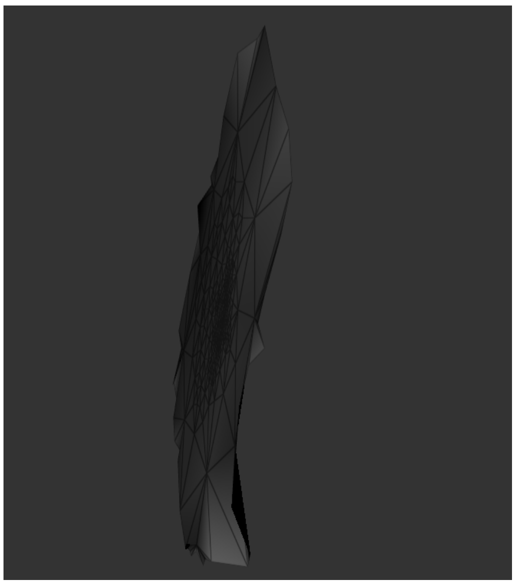

(b) Side view

Figure 7: David: Minimal Surface(Uniform Weights)

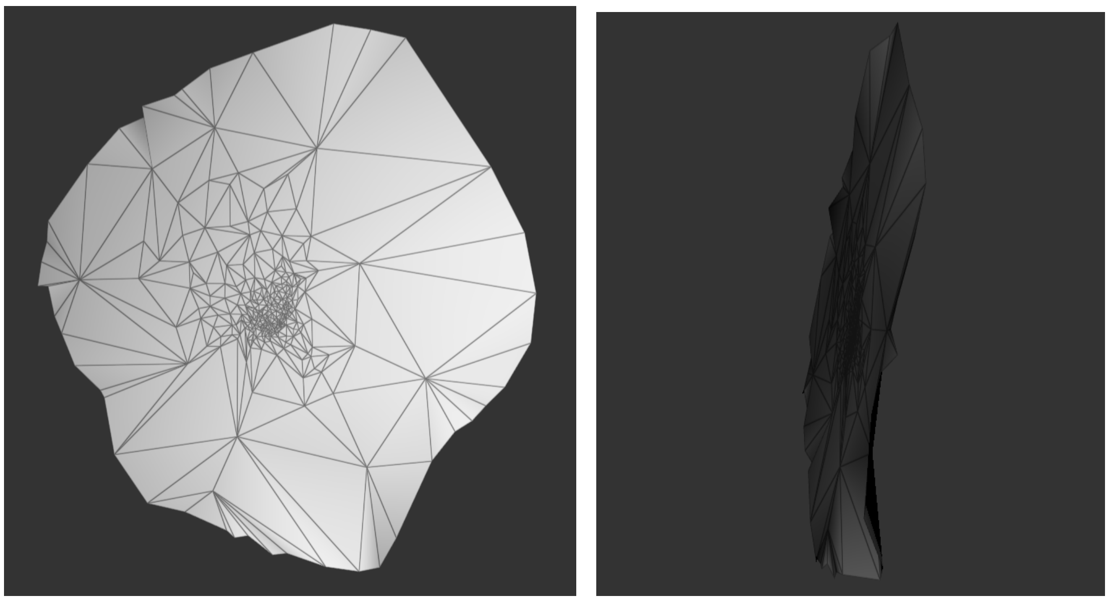

(a) Main view (b) Side view

Figure 8: David: Minimal Surface(Cotangent Weights)

Second, we use the Balls to test the boundary mapping. The results are shown in Figure 9 and 10.

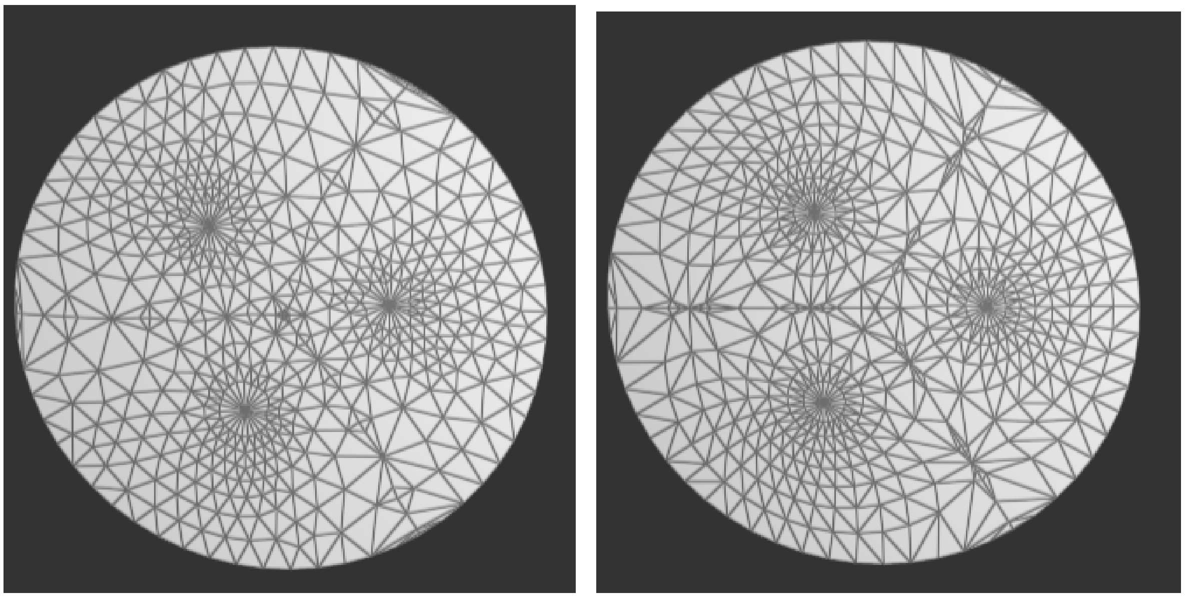

(a) Uniform Weights (b) Cotangent Weights

Figure 9: Balls: Circle Boundary

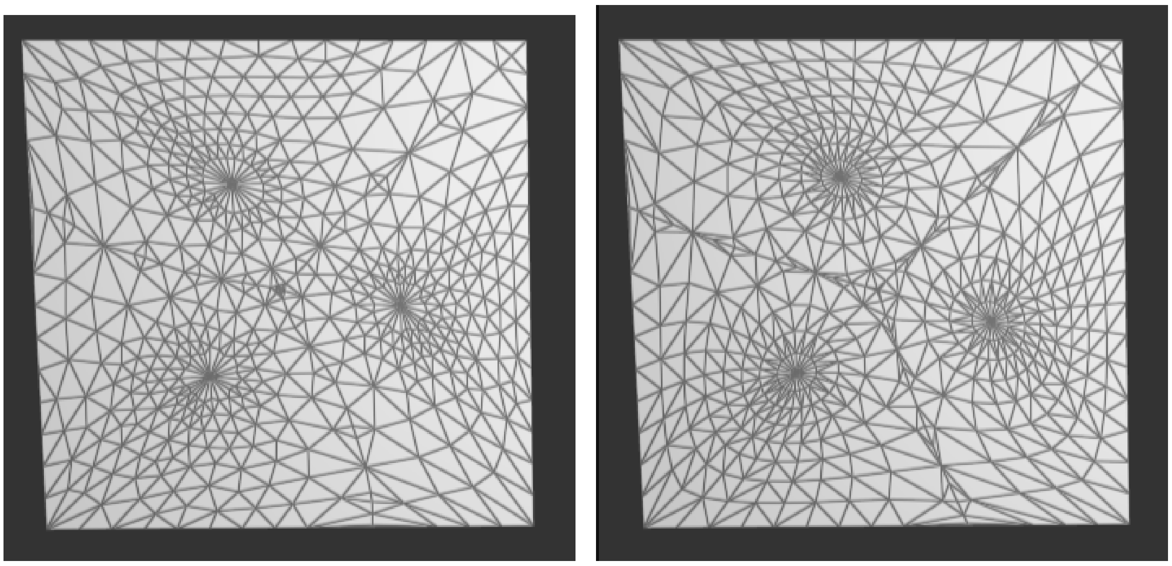

(a) Uniform Weights (b) Cotangent Weights

Figure 10: Balls: Square Boundary

Finally, we use Balls to test the texture embedding. The results are shown in Figure 11 and 12. We claim that we map the boundary to square.

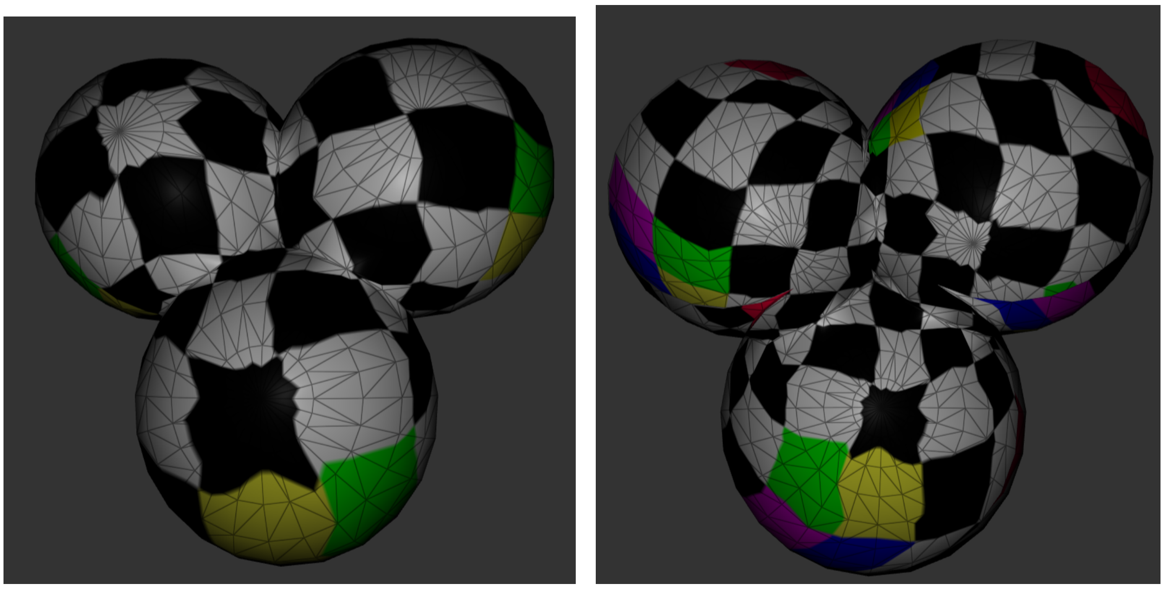

(a) Main View (b) Rear View

Figure 11: Balls: Texture Embedding(Uniform Weights)

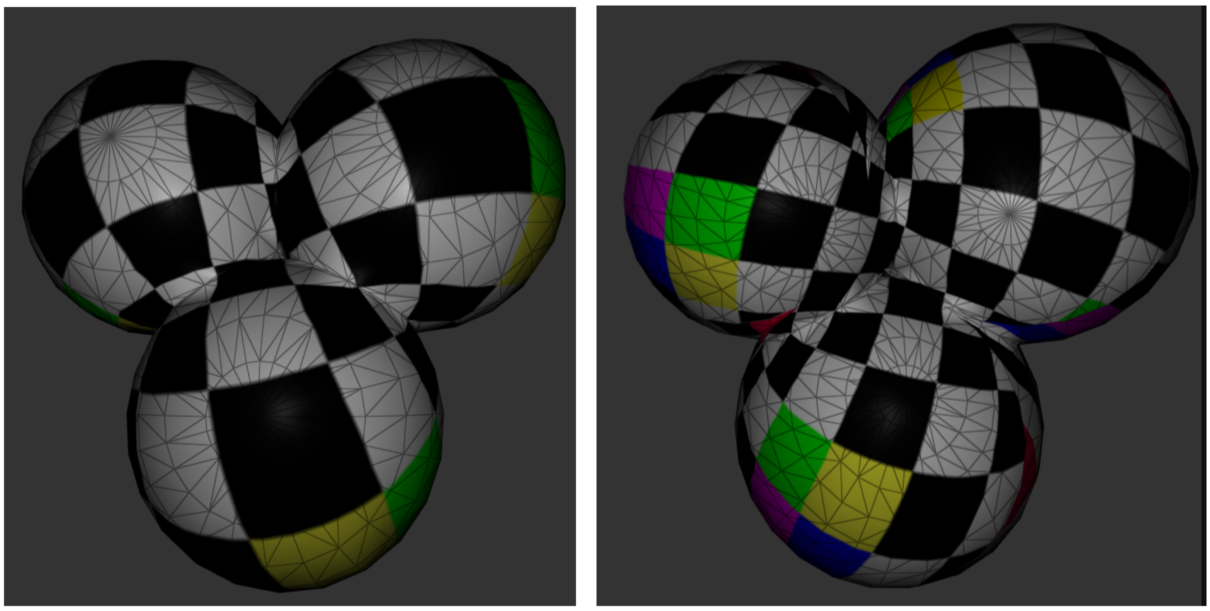

(a) Main View (b) Rear View

Figure 12: Balls: Texture Embedding(Cotangent Weights)

We can find that under the uniform weights, the distribution has a severe deformation.

5 Summary

In this experiment, we have achieved the generation of minimal surfaces with given boundaries, boundary mapping, and texture embedding. Most of the methods used are as follows: solving sparse matrices by formulating equations based on central coordinates and different types of weights, and reconstructing three-dimensional surfaces. In addition to this, we have also learned about half-edge data structures and node programming. However, there are several issues with the algorithms we implemented:

1. For surfaces with higher genus, our method can only detect one boundary. Due to the presence of multiple boundaries, distinguishing between the inside and outside is a problem worth investigating.

2. In fact, the cotangent weights used here may not necessarily be optimal. Exploring how to create better meshes is also a worthwhile topic for discussion.

3. For the generation of minimal surfaces, there may be some confusion at the boundaries..