Homework 3: Parametric Modeling

Computational Fabrication / Advanced Computer Graphics
6.807/6.839
Assigned: 10/2/2020
Due: **10/15/2020 at 11:59 PM**

Before starting this homework, please read Codebase.pdf first to get familiar with the homework codebase and running environment.

Please update your codebase by executing following command in your codebase folder:

```
git add ./*.hpp ./*.cpp
git commit -m "My assignment 2 code."
git pull
```

The main entrance of this assignment is in `assignment3/main.cpp`. You can compile the code by first entering `build` folder and executing following command:

```
cmake ../ -DCMAKE_BUILD_TYPE=Release -DHW=3
make
```

You can run your solution by

```
./assignment3/run3
```

If your program outputs "Welcome to Assignment 3" and open a GUI window, you are done with updating your codebase.

1 Introduction

In this assignment, we'll exploring some techniques used in modern Computer-Aided Design - specifically, parametric modeling and interactive deformable modeling. You'll design CSG models in a domain-specific programming language called "OpenSCAD", and implement bounded biharmonic weights deformation in your codebase.

2 Assignment

This homework has two parts - Basic CSG, and bounded biharmonic weights deformation. For the first part, you'll need OpenSCAD. You can either install it through the website[1], or, if you are on Ubuntu or some other Linux distros, you can install it *via*

[1] `http://www.openscad.org/`

```
sudo apt install openscad
```

2.1 CSG Modeling (30 points) - Both 6.807 and 6.839

As described in class, constructive solid geometry (CSG) is a method of generating complex models through geometric primitives and Boolean operations. Its intuitive and useful nature has led to its inclusion in most modern CAD modeling software. Any geometry can be generated with sufficient complex CSG trees (though maybe not compactly); in other words, CSG spans the space of all geometric designs. Because it is represented as a tree, CSG is very similar to computer code. In this section, you will learn to model using OpenSCAD, which is a domain-specific language for specifying CSG.

To get started, we recommend working through some of the tutorials on the OpenSCAD website[2]. The full user manual can be found on the OpenSCAD website[3]. Finally, the OpenSCAD start-up splash screen features a tab with a few examples. After you've played around with the environment some, it's time to make a design.

First, we have included a few files, called `parametric*.stl` in `data/assignment3/CSG`. Take a good look at them in your favorite `.stl` viewer (say, Meshlab). These designs were all created by a single parametric OpenSCAD file with different parameter settings.

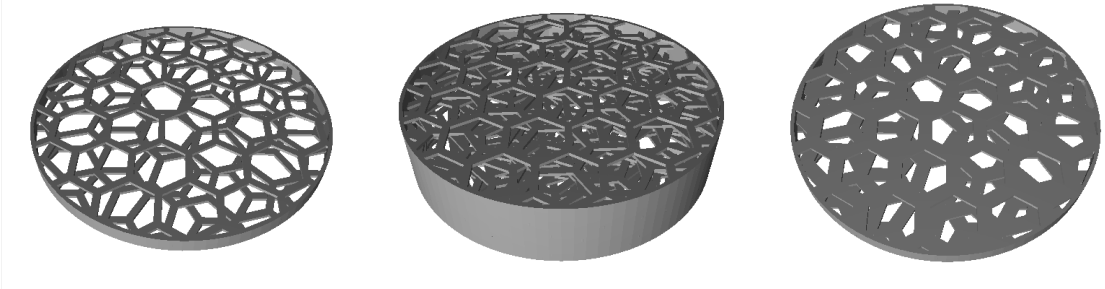


Figure 1: Three meshes created with the same parametric OpenSCAD file with different parameter settings.

First, we'd like you to reverse-engineer the generating OpenSCAD file. If you look at the provided meshes, you will notice plenty of repetition. Even though these designs could perhaps be designed brute-force by manually specifying unions, subtractions, and intersections, you should make use of the highly symmetric nature of the design in your code. Use for loops, recursion, and list comprehension as much as possible and recreate the part as closely as possible. Define all parameters as variables at the start of your file. In your writeup, be sure to specify what the parameters of your design are, what semantic features of the design they correspond to, and what values you set in order to generate meshes which match the

[2] `http://www.openscad.org/documentation.html`

[3] `https://en.wikibooks.org/wiki/OpenSCAD_User_Manual`

provided files. HINT 1: In our reference solution, there are three parameters in total. HINT 2: Try decomposition the geometry in smaller pieces to start, and use CSG operations to compose them.

After you've conquered victory over this reverse-engineering task, it's time to have a little fun. Create your own custom part using OpenSCAD - it can be whatever you want, but please try to make it interesting and represent some potential real-world object that could be fabricated. Your design must feature at least one use of a for loop and one use of recursion. Again, define all parameters as variables at the start of your file.

Your final designs can be exported to a .STL file by hitting F6, and then exporting it under the "File" tool. Generate a few versions of your design with different parameter settings. If the resolution of your mesh looks rather coarse you can refine the resolution using the "$fn" argument when generating your basic geometry[4].

Please do not import any other meshes in your design, as this defeats the purpose of trying to generate complex geometry using CSG. However, if you find you like OpenSCAD and would like to use it for your final project, do note that it has the ability to import files generated using other CAD software, which can then be manipulated.

Summarize your tasks in this section here:

1.1 (20 points) Reverse-engineer the generating OpenSCAD file for our provided models `data/assignments3/CSG/parametric*.stl` by fully parameterizing the models by a few parameters (e.g. 3 parameters in our solution).

1.2 (10 points) Implement the OpenSCAD code to generate your own interesting CAD models. Your design must feature at least one use of a for loop and one use of recursion.

2.2 Mesh Deformation (50 points) - Both 6.807 and 6.839

Deforming non-parametric meshes typically involves using mesh deformations via handles, skeletons or cages. And there are two classes of deformation algorithms, the variational methods and the direct methods.

This exercise is about implementing the direct deformation methods known as Linear weights deformation and Bounded Biharmonic Weights Deformation [1].

Consider an input mesh (tetrahedral mesh in our assignment) $\mathcal{M}$ and a set of handles $\mathcal{H}$. Let n denote the number of vertices in the mesh $\mathcal{M}$, m denote the number of tethedrons, and h denote the number of handles in $\mathcal{H}$. The bounded biharmonic weights method computes a weight function $\omega : p \in \Omega_M \rightarrow \mathcal{R}^h$. For each point p in the domain of the mesh, $\omega(x)$ consists h weights, so that the position of point p after deformation can be computed as a weighted combination:

[4] `https://en.wikibooks.org/wiki/OpenSCAD_User_Manual/Other_Language_Features#$fa,_$fs_and_$fn`

$$p' = \sum_{j=1}^{h} \omega_j(p) T_j p \tag{1}$$

where T_j is the transformation matrix of handle j.

2.2.1 Linear Weights

To warm up and get some comparison, you are going to implement the most naive linear weights for mesh deformation. In linear weights method, the weight is assigned by

$$\omega_j(p) = \frac{1}{\text{dist}(p, \mathcal{H}_j)} \tag{2}$$

In this part, you have only one task to do:

> 2.1 (10 points) Implement the `ComputeLinearSkinningWeights` function in `linear_weights.hpp`. You can test your implementation in the GUI by choosing linear weights method.

2.2.2 Bounded Biharmonic Weights

Then you are going to implement the bounded biharmonic weights for mesh deformation. We brifely summarize the algorithm here, and please refer to the lecture slides and the paper [1] for details.

Recall the lecture, the bounded biharmonic weights computes the weights by minimizing the laplacian energy:

$$\begin{aligned} \arg\min_{\omega_j, j=1,\dots,h} & \sum_{j=1}^{h} \frac{1}{2} \int_\Omega \|\Delta \omega_j\|^2 dV \\ \text{s.t. } & \omega_j|_{\mathcal{H}_k} = \delta_{j,k} \\ & \sum_{j=1}^{m} \omega_j(p) = 1, \qquad \forall p \in \Omega \\ & 0 \le \omega_j(p) \le 1, j = 1, ..., m, \qquad \forall p \in \Omega \end{aligned} \tag{3}$$

where the first constraint defines the boundary conditions of those handles. The second constraint normalize the sum of weights on each points to one, and the third constraint enforces the non-negativity of the weights.

This is the formula in continuous space, we need to discretize it in order to solve it numerically. We can discretize our weight functions by linear finite element approach. Suppose we have the weight value ω_j^i on each vertex v_i in our mesh for each handle $\mathcal{H}_j$. We can use linear interpolation to get the value of any points p in the mesh by $\omega_j(p) = \sum_{v_i \in T(p)} \phi_i(p) \omega_j^i$, where $\phi_i(p)$ is the basis coefficient of handle i on point p. Here, $T(p)$ is the tetrahedron which

contains point p and v_i is one of its four vertices. So now instead of compute a continuous function ω, we only need to compute $n \cdot h$ discrete values of ω.

Recall in the class, we show how to apply Galerkin finite element method to solve $\Delta u = f$ (this is the well known Poison equation). By applying a weak formulation and discretizing all the functions by linear finite elements, we got

$$Lu = Mf$$

where

$$L_{ij} = -\int_\Omega \nabla\phi_i \cdot \nabla\phi_j dV$$

it integrates over the whole domain with the dot product of the gradients of basis functions. And

$$M_{ij} = \int_\Omega \phi_i\phi_j dV$$

it integrates over the whole domain with the product of the basis function coefficients.

To derive the discrete biharmonic equations, we need a technique called mixed finite element, which is a little bit complicated to be explained here. We will rather give you some intuition to derive the laplacian formula for the weights. As we know in the class, the Poison equation is $\Delta u = f$, and we got $Lu = Mf$. Left multiplying M^{-1} on both sides can help us get f which is the laplacian of function u. Similarly, for our weights, we have

$$\Delta\omega_j \approx M^{-1}L\omega_j$$

Further discretizing the laplacian energy gives us

$$\begin{aligned}
\arg\min_\omega \sum_{j=1}^m \frac{1}{2}\int_\Omega \|\Delta\omega_j\|^2 dV &\approx \arg\min_\omega \sum_{j=1}^m \frac{1}{2}\int_\Omega (M^{-1}L\omega_j)^\top M(M^{-1}L\omega_j) \\
&= \arg\min_\omega \frac{1}{2}\sum_{j=1}^m \omega_j^\top (LM^{-1}L)\omega_j \\
&= \arg\min_\omega \frac{1}{2}\sum_{j=1}^m \omega_j^\top Q\omega_j \qquad (4)
\end{aligned}$$

Once we have the matrices L and M, the optimization problem becomes to a simple quadratic programming problem (quadratic energy with linear constraints), which can be efficiently solved.

To compute the matrices M and L for a tetrahedral mesh, we gave the formulas here. We highly recommend you to read the attached document *DiscreteLaplacian.pdf* for detailed derivation for discrete laplacian in tetrahedral mesh.

$$M_{ij} = \begin{cases} \sum_{T\in\mathcal{N}(v_i)} \frac{1}{4}V_T & i = j \\ 0 & i \neq j \end{cases} \qquad (5)$$

where $\mathcal{N}(v_i)$ is the tetrahedron set in the ith vertex's neighbourhood and V_T is the volume of that tetrahedron.

$$L_{ij} = \begin{cases} \sum_{e \in \mathcal{O}(ij)} \frac{1}{6} l_e \cot \theta_e & \text{if edge } ij \text{ exists} \\ -\sum_{i \neq j} L_{ij} & i = j \\ 0 & otherwise \end{cases} \tag{6}$$

here $e \in \mathcal{O}(ij)$ means the edge e is opposite to edge ij in a tetrahedron, l_e is the length of edge e and θ_e is the dihedral angle of the two faces around edge e in that tetrahedron.

In this part, we are going to implement the whole bounded biharmonic weights algorithm. We got you started. We implemented the overall algorithm structure for you. We also provided the tool function to compute the dihedral angles between two faces. Please read the code carefully to fully understand the code structure and the variables in the code. Your task is to complete this pipeline by constructing the cotangent laplacian matrix L and constructing the quadratic programming objectives (i.e. the Q matrix in the energy function). We have implemented the M matrix and incorporate an efficient quadratic programming solver.

Here's the list of the tasks you are going to solve to complete the bounded biharmonic weights algorithm's pipeline:

3.1 Read the derivation and the code structure carefully to fully understand the algorithm pipeline and the variables in the code.

3.2 (30 points) Implement the `cotangent_laplacian` function in `cotangent_laplacian.hpp`. Please see the comment in the code for more details.

3.3 (5 points) Complete the `bounded_biharmonic_weights` function in `bounded_biharmonic_weights.hpp`. Here you need to construct the Q matrix which is used for quadratic programming solver. Please see the comment in the code for more details.

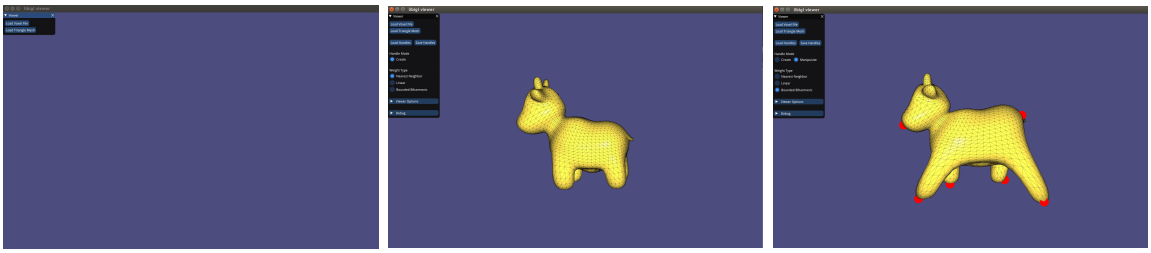
3.4 (5 points) Choose one of your favorite mesh models, voxelize it by the code of your assignment 2. Load the voxelization results (voxel_info.txt) into the GUI, and put your own handle points and manipulate it. Provide the screenshot of the shapes before and after manipulation in your report. HINT: reducing the voxelization resolution in assignment 2 is necessary, otherwise it takes too long to compute bbw weights for high-resolution mesh.

2.2.3 How to play with GUI

In this assignment, we developed a GUI for you to create handle points, drag to manipulate the mesh and visualize your deformation results. Once you compile your code successfully and run it, you are able to see a GUI like Figure 2a. You can click the "Load Voxel File" button to load a voxel_info file which is generated by your assignment 2 code, or click the "Load Triangle Mesh" button to load a triangle mesh into the system. Our code will automatically help convert the voxelization mesh of triangle mesh to tetrahedral mesh. Once you load your mesh succefully, you can see an object in your GUI as shown in figure 2b.

When you are in "Create" mode, you can create handles by left click on the mesh. You can also load predefined handles from file. When you are in "Manipulate" mode, you can choose which kind of deformation algorithm you want to use (nearest neighbour, linear, or bounded biharmonic). After the algorithm compute the weights, you can start drag your handle points to deform the model like figure 2c.

We've implemented the nearest neighbour weights for you to test. You can find several triangle meshes (`.stl`), a simple voxel mesh (`cube_voxel_info.txt`) and predefined handle points (`*_handles.txt`) in the directory `data/assignment3/BBW`. You can test your algorithm by loading the mesh and its corresponding handles. We highly recommend you to start from `cube_voxel_info.txt` first. You can save your weights by click `Debug -> weights`. We provide the weights from our solution in `data/assignment3/BBW/results/*_weights_std.txt` for comparison. (The stardard output is generated in ubuntu 16.04, different system may have different results).



(a) (b) (c)

Figure 2: GUI for deformation

3 High-level Understanding (20 points)

3.1 Non-negative weights (10 points)

In bounded biharmonics weights algorithm, we constrained the weight to the range $[0, 1]$ to avoid negative weights. Is that safe to remove this range constraints? If not, what problem it will cause. Please modify the algorithm to remove this constraints to validate your answer.

3.2 weights normalization (10 points)

In bounded biharmonics weights algorithm, we normalize the weight for each mesh vertex so that summing up its weights of all handles is one. Is that safe to remove this normalization operation? If not, what problem it will cause. Please modify the algorithm to remove this operation to validate your answer.

4 Submission Guidelines

All assignments must be submitted onto the Stellar submission system by the deadline. The submission should be in one single .zip file, including two parts.

1. All the code in `assignment3` folder and `Common` folder should be submitted.

2. The openscad code files in first part.

3. A short .pdf report describing how your code works and the results (table for execution time and screenshot images for results).

Late assignments will incur a 25% penalty for each late day (3 late days are permitted without penalty throughout the semester). No assignments will be accepted after the solutions are released (typically within a few days of the deadline; contact the TA for details).

4.1 Writeup Instructions

The writeup should be relatively short (around $1-2$ page). In your writeup, please tell us the following:

- At a high-level, what did you implement in order to complete the assignment?
- How does that code work, algorithmically?
- Some pictures to show your results.
- What did you like about the assignment?
- What did you not like about the assignment?
- Any other issues about which we should be aware?

Please submit the writeup as a .pdf. Also, feel free to include any relevant images (embedded in the .pdf).

Please submit the writeup as a .pdf. Also, include any relevant images (embedded in the .pdf).

References

[1] Alec Jacobson, Ilya Baran, Jovan Popović, and Olga Sorkine. Bounded biharmonic weights for real-time deformation. *ACM Trans. Graph.*, 30(4):78:1–78:8, July 2011.