Homework 2: Solid Modeling

Computational Fabrication / Advanced Computer Graphics
6.807/6.839
Assigned: 9/16/2020
Due: **9/30/2020 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 1 code."
git pull
```

If you get `CONFLICT` in `Common/Geometry/BasicGeometry.hpp` file, you should keep both of your code and the incoming changes.

The main entrance of this assignment is in `assignment2/main.cpp`. You can compile the code by first entering `build` folder and executing following command:

```
cmake ../ -DCMAKE_BUILD_TYPE=Release -DHW=2
make
```

You can run your solution by

```
./assignment2/run2
```

If your program outputs "Welcome to Assignment 2", you are done with updating your codebase.

1 Introduction

In this assignment, we will be going from mesh to voxels and back again. Voxelization has two important uses. First, if you want to voxel print (say, using a polyjet, SLS printer, or similar process), converting your representation to voxels (or working directly with voxels from the outset) is a key operation. Second, voxels, or hex meshes, are representation with high simulation accuracy.

Meanwhile, after a design has been simulated, you may want to still post-process the geometry (or re-edit it) in some way. To that end, meshing (and perhaps remeshing) is an important tool in our geometric toolbox.

1.1 How to run the code and grade

You can run all the tests by

```
./assignment2/run2
```

It runs all the test cases, generate the resultant meshes, and print the running time. Below is a table of running time from your TA's solution, test on Ubuntu 16.04 with 1 Intel(R) Core(TM) i7-7600U CPU @2.80GHz and 8G memory. Your implementation doesn't need to match these numbers exactly but should not be significantly slower:

Test case	Time (s)	Test case	Time (s)	Test case	Time (s)
Bunny (basic)	26.8	Bunny (fast)	0.85	Bunny_with_hole	2.3
Fandisk (basic)	35.1	Fandisk (fast)	2.75	Spot_with_hole	1.1
Spot (basic)	2.4	Spot (fast)	0.55		
Dragon (basic)	284.9	Dragon (fast)	5.72		

We have also provided you mesh results generated from your TA's solution. Your goal is to generate meshes that are reasonably close to the reference mesh (details in each section later) within the amount of time comparable to or faster than your TA's reference solution. Your results are located at `data/assignment2/results` and TA's reference solutions are located at `data/assignment2/std`.

To run each individual test, please read the comment at the top of `main` function in `main.cpp`.

To compare your results with TA's results, you can run `grade.py` in data/assignment2 folder by:

```
python grade.py
```

The python script will output the similarity of your voxelization results with TA's. A result is regarded as correct if the similarity is greater than 95%.

2 Assignment

This homework has two parts - voxelization and marching cubes. We suggest you skim through `assignment2/main.cpp` before coding so that you can get a sense of the overall structure of the assignment. You don't need to change anything in this file for this assignment.

2.1 Basic Voxelization (30 points) - Both 6.807 and 6.839

The idea behind voxelization is pretty simple: given a watertight triangle mesh, we divide its bounding box into regular grids. For each grid, we determine whether it should be classified as being inside or outside the mesh, typically by checking if the center of this grid is inside or outside. In this assignment, we have set up the regular grids for you, and

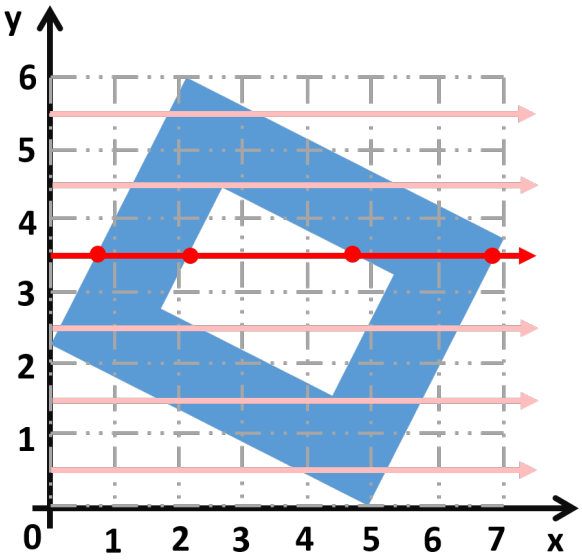

Figure 1: The 2D illustration of the basic voxelization algorithm.

your task is to figure out the correct label (0 or 1) for each grid. You will implement a basic voxelization algorithm based on the idea of shooting parallel rays from one face of the bounding box, which we briefly illustrate in 2D below (Figure 1):

In this 2D example, blue shows the input mesh and gray is the regular grid. To figure out the sign of each grid, we shoot 6 parallel rays (light and dark red arrows) from the center of each grid at `x=0` and process them one by one. As an example, consider the ray at $y = 3.5$ (dark red arrow in the figure). We loop over all the edges of the mesh (all the triangles if it's 3D) and compute the intersection of that ray and each edge. At the end of this loop, we have a list of intersections along that ray (4 dark red dots). We can now sort them and use their coordinates to determine the signs of all the grids along this ray. For example, if we let x_1, x_2, x_3, x_4 be the x-coordinates of the 4 dots in the figure from left to right, for all the grids along $y = 3.5$, we can call it inside if x_c, the x-coordinates of its center, satisfies $x_c \in [x_1, x_2] \cup [x_3, x_4]$.

You should figure out how to implement this algorithm for 3D meshes by yourself. In this part, you need implement following stuff:

1.1 (10 points) Implement the `IntersectRay` function defined in the `Triangle` class in `Geometry/BasicGeometry.hpp`.

1.2 (20 points) Finish the `BasicVoxelization` function in `voxelizer.hpp`, in which you assign the correct `true` or `false` value to each element in the `_voxels` array.

Tips: Take a look at the `Voxelizer` class defined in `voxelizer.hpp`. We have set up the regular grid for you inside this class.

After you finish it, you should be able to run the first four commands specified in `main.cpp` and generate voxelized meshes (almost) identical to the reference meshes we gave

in the `data/assignment2` folder.

2.2 Fast Voxelization (30 points) - Both 6.807 and 6.839

Let n_x, n_y, n_z be the number of grids along each direction (so you have $n_x n_y n_z$ grids in total) and let m be the number of triangles in the mesh. Assuming we shoot rays along the z-axis, the algorithm we just described runs in $O(n_x n_y m)$ time because we test intersections of each ray (there are $n_x n_y$ of them) and each triangle (there are m triangles in total). In this section, you will implement a different algorithm that will be much faster for high-resolution voxelization. Here we first illustrate the idea in 2D again (Figure 2): in this new algorithm, instead of looping over all the rays, we loop over all the edges (green lines in the figure above). For each edge e, if we loop over all the rays and compute the intersections of e and each ray, then it is equivalent to the previous algorithm. The trick here is to project e on the y-axis and see how many grids are covered. For example, for the edge on the left figure, the projected green edge occupies 4 grids: $y = 2.5, 3.5, 4.5, 5.5$. Now for this edge, we compute the intersections of this edge and the 4 rays from the covered grids only. We then move on to process the second edge (right figure) and collect more intersection points.

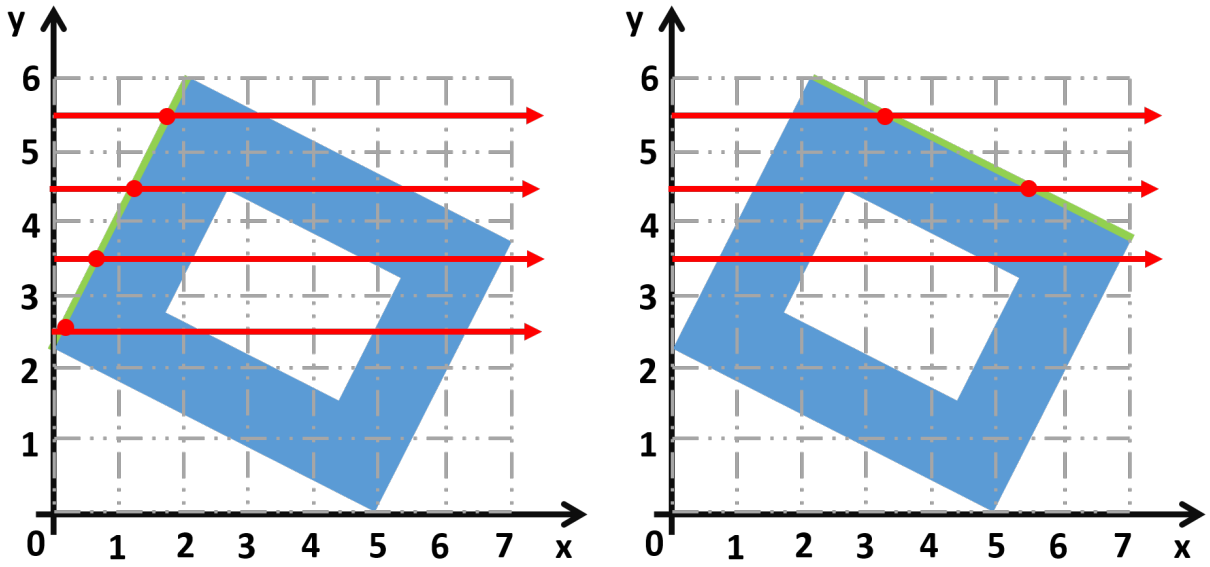

Figure 2: The 2D illustration of the fast voxelization algorithm.

Your task is

> 2.1 (30 points) Implement the 3D version of this algorithm in the `AdvancedVoxelization` function defined in the `Voxelizer` class.

Once it is done, you should be able to run another four test cases in `main.cpp` and generate the same voxelized meshes within seconds. You can then compare your results to the reference meshes in the `data/assignment2` folder and they should be almost identical. Specifically,

the output meshes generated by your implementation for Section 2.1 and Section 2.2 should be completely identical.

2.3 Voxelization for Non-watertight Meshes (30 points) - 6.839 only

In the case of non-watertight meshes (e.g., meshes with holes), whether a point is inside or outside the mesh is ill-defined. As a result, there is no correct definition of the sign of each grid, and we have to make reasonable guesses for them. In this section, you will randomize the algorithm in Section 2.2 by changing the plane you project the triangles on. The idea is best described in Figure 3: In this 2D example, dark blue lines are the edges of this non-watertight 2D shape. First, we project all the edges on the y-axis and run the old algorithm in Section 2.2. This assigns labels for each grid (Here we use light blue to indicate grids that are considered to be "inside" this shape). Second, we run the algorithm again by projecting edges on $y = 6$ (right figure), which is equivalent to shooting vertical rays from $y = 6$ to $y = 0$. This will label all the grids in a potentially different way. For example, the grid $(0, 0)$ was considered to be outside on the left but inside on the right. In general, we randomly generate k different ray directions and run the old algorithm k times. Now each grid collects k labels, and we use the majority to decide the final label of this grid.

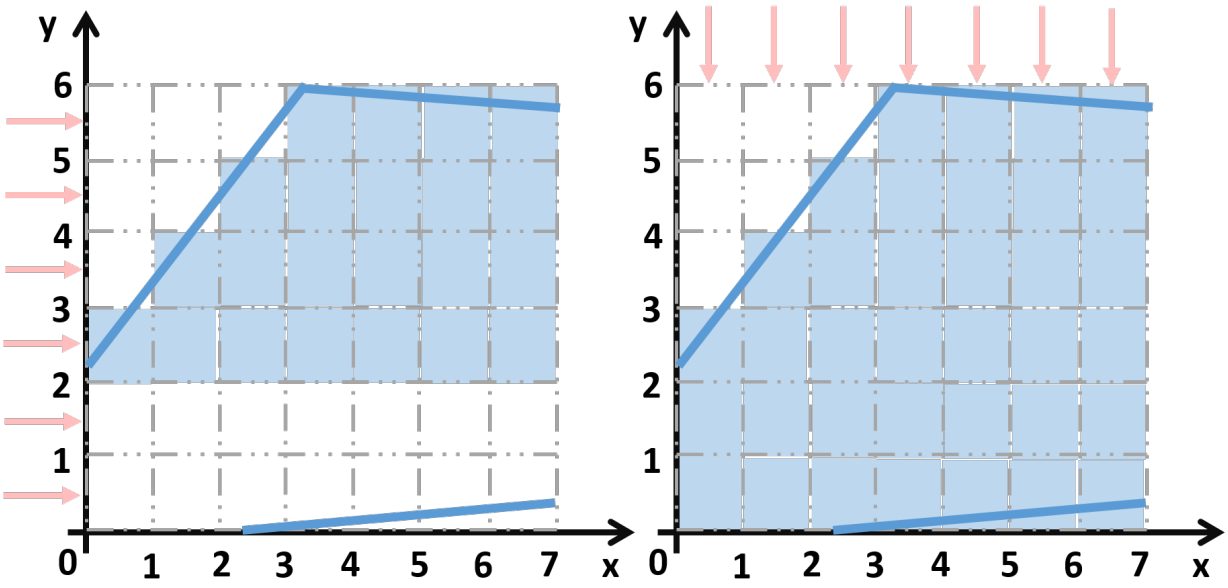

Figure 3: The 2D illustration of the voxelization algorithm for non-watertight meshes.

Your task is to implement this algorithm for 3D cases. In particular, we require the random ray directions you pick should **not** be axis-aligned. You need to think about how to apply the algorithm in Section 2.2 in the case of a general ray direction. The general ray direction also brings you another challenge: the regular grids for different ray directions are no longer the same. For example, in the 2D case above, if you shoot rays from the direction $(1, 1)$, you are sampling intersection points in a regular grid rotated by 45 degrees. You

will need to come up with a way, e.g., some interpolation methods, to transfer information between these grids.

Once you figure out all the details of the algorithm, you are supposed to

> 3.1 (30 points) Implementing the `AdvancedVoxelizationWithApproximation` method in the `Voxelizer` class. Feel free to add any data members, data structures, or methods in this class as you see fit. If T is the time of running the algorithm in Section 2.2 once and k is the number of random ray directions, your implementation should finish in $O(kT)$ time. We used $k = 11$ in our solution.

After finishing this part, you will be able to run the two test cases `bunny_with_hole` and `spot_with_hole` and generate results similar to the reference meshes we have provided in the `data` folder. Besides that, you also need to test your implementation on one more non-watertight mesh you find on the Internet and provide the result in your report.

2.4 Marching cubes (10 points) - Both 6.807 and 6.839

Now that we have the Mesh to Voxel direction complete, it's time to introduce the Voxel to Mesh direction. For this, you'll run the Marching Cubes algorithm [2], perhaps the most widely used computer graphics algorithm to date. **Note that no coding is needed in this section**.

Recall that voxelization gives you a regular grid defined on the bounding box of the mesh. Let's assume we have also pre-computed the inside/outside flag for each corner of each grid. The marching cube algorithm loops over all the grids and creates triangles based on the signs of the 8 corners. There are $2^8 = 256$ cases, but most of them are equivalent after rotations. It turns out that there are only 15 unique cases (Figure 4), and the marching cube algorithm processes each grid and adds triangles by simply referring to this lookup table.

We have already implemented the marching cube algorithm for you. Your task here is to run the last 4 commands and provide screenshots of the output meshes.

2.5 Extra Credit

2.5.1 Perfect Information Maintenance (10 Points)

If your algorithm is in fact able to perfectly retain all of its information as geometry changes representations, then you will be awarded bonus points. This will probably require making changes to our naive Marching Cubes implementation.

Demonstrate this capability of your code by adding a section at the end of your main.cpp's main method in which you transfer the file between solid and surface representation twice and write the intermediate meshes to files. If you've done it right, each voxel file and each triangle mesh should be the same.

2.5.2 Adaptive Distance Fields (ADFs) - (20 Points) - 6.839 Only

The voxelization techniques we have developed thus far can tell you whether or not a point is inside or outside a shape - and for many, if not most applications, this is both sufficient

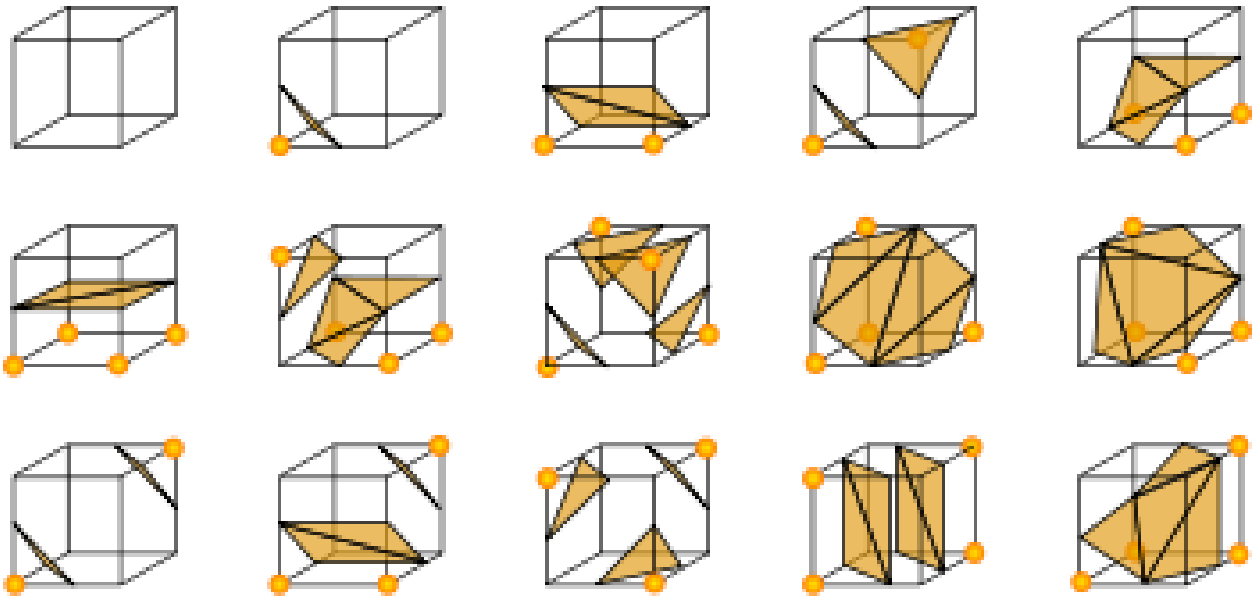

Figure 4: The marching cubes "lookup" table.

and extremely powerful. In some applications, however, it is necessary to have a fast way of knowing *how far* inside or outside a shape a point is. ADFs are useful for precisely that purpose. A successful implementation and demonstration of Adaptive Distance Fields will be awarded 20 points. For details on the algorithm and its implementation, please refer to the original ADF paper [1].

3 Submission Guidelines

All assignments must be submitted to the Canvas by the deadline. 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)

3.1 Code Guidelines

Good style and comments are expected. Code which does not compile in the environment specified cannot and will not be graded. If you add code outside of the sections we recommend editing, please tell us where specifically to look in your writeup. Although we have written our code as a header-only library, you are not compelled to do the same.

We will grade your assignment based on both your implementation, your output meshes and the reported running time.

3.2 Writeup Instructions

The writeup should be relatively short. In your writeup, please tell us the following:

- At a high-level, what did you implement in order to complete the assignment?
- (6.839 only) Briefly describe your voxelization algorithm for non-watertight meshes.
- 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).

References

[1] Sarah F Frisken, Ronald N Perry, Alyn P Rockwood, and Thouis R Jones. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 249–254. ACM Press/Addison-Wesley Publishing Co., 2000.

[2] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *ACM siggraph computer graphics*, volume 21, pages 163–169. ACM, 1987.