Homework 1: From Design to Machine Instructions

Computational Fabrication / Advanced Computer Graphics
6.807/6.839
Assigned: 9/3/2019
Due: **9/20/2019 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 pull
```

In this assignment you will implement a cutting-edge slicer (pun intended). This assignment has three main parts. In addition, we offer multiple opportunities for extra credit. The main entrance of this assignment is in `assignment1/main.cpp`. You can compile the code by first entering `build` folder and executing following command:

```
cmake ../ -DCMAKE_BUILD_TYPE=Release -DHW=1
make
```

You can run your solution by

```
./assignment1/run1
```

If your program outputs "Welcome to Assignment 1", you are done with updating your codebase. Also, in order to run scripts to validate your results, please ensure you have `Python` enviroment set up and `Numpy` library properly installed. To install `Numpy`, use following command

```
pip install numpy
```

1 Printing with a Virtual Printer (5 points)

In this assignment we will be using Repetier-Host as a virtual 3D printer. Download and install a current version of Repetier-Host for your platform from https://www.repetier.com/download-now/. Note that Retetier-Host has a slightly different interface layout on different platforms, but the functionality is identical.

1.1 Set Up a Virtual Printer

Open Config > Printer Settings Dialog. In the Connection Tab, select Virtual Printer in the Port drop-down, then click Apply. Then click the Connect button in the upper left corner to connect Repetier-Host to your new virtual printer.

1.2 Virtually Print an Object

In this part of the assignment you will slice and visualize an object using Repetier-Host. Later, you will write your own slicer and use Repetier-Host to visualize the results.

Thingiverse (https://www.thingiverse.com/) has a large selection of user-created, printable models. Find a model you like and download an STL file for it from thingiverse. Choose a model that has only one part to it (some models have multiple STL files for different pieces, or have single STL files with multiple disconnected shapes within them).

In Repetier-Host, click Add STL File in the Object Placement section (this may be a tab view on the right side of the screen, or in the top navigation bar, depending on your system). This should add a 3D object to the virtual printer bed.

Next, open the Slicer tab, and click Slice with Slic3r, using the default settings. This may take a few minutes. When it is complete, in the Print Preview or visualization tab (depending on your system), you can scrub through layer ranges to visualize the print. In the GCode tab you can read and edit the GCode that the virtual printer is using to print the object. **Please screenshot your slicing results and include it in your report.**

2 Brute Force Slicing (45 points)

In this section of the assignment, you will implement a brute-force slicer. We are calling it a brute-force slicer because you will not implement any sort of data-structure to help accelerate the process - that will be completed in the next part.

Recall from class that slicing has two parts. In the first part, planes at regular intervals are intersected with the triangles of the mesh; the resulting intersections provide a collection of edges for each layer with which to assemble a print contour. In the second part, these edges are stitched together in order to generate the contour.

We have provided some starter code to get you started. The `tri_mesh.hpp`, which can be found in `Common/Mesh` folder, provides some utilities for loading .STL files and pre-processing triangle meshes.

In this part, you need implement following stuff:

2.1 (10 points) The first step is to implement triangle-plane intersection. You will implement the function `IntersectPlane` in the `Triangle` class in `Geometry/BasicGeometry.hpp`.

2.2 (20 points) The second step is to implement a bruteforce slicing algorithm. You need to fill the `Slicing_bruteforce` method in `FabSlicer.hpp`. The result will be exported to `data/assignment1/results/slicing.ply`. To validate your result, go to `data/assignment1/results`, and run `Python compare_slice.py -i path_to_your_result -r reference_result`. We recommend testing `data/assignment1/triangle.stl`, which consists of just a single triangle. You can find reference results in `data/assignment1/results`.

2.3 (15 points) Finally, you will complete the `CreateContour` function in `FabSlicer.hpp`. This function stitches all edges in one layer together. You can implement either a bruteforce method or an accelerated method, which will automatically satisfy assignment part 3.2 for 6.839 students and extra credit 1 for 6.807 students. The result will be exported to `data/assignment1/results/contour.ply`. To validate your result, go to `data/assignment1/results`, and run `Python compare_contour.py -i path_to_your_result -r reference_result`. We recommend testing `data/assignment1/rect_hole.stl`, which is a rectangle with a hole in the center. For this mesh, at each layer, there should be two distinct contours. You can find reference results in `data/assignment1/results`.

Tips: You can find more details and tips both in the code and in section 6.

3 Accelerated Slicing (40 points) - 6.839 only

Note: This section needs only needs to be completed by students taking this course as 6.839. Students taking EECS 6.807 may optionally complete this section for extra credit.

By now you've probably noticed that slicing can be quite slow. This is because each of k planes attempts to intersect all m triangles in your mesh - a relatively expensive $\mathcal{O}(km)$ operation with large m. In this step, you'll implement an accelerated slicing algorithm. One option is **interval tree**[1] in which to store the set of triangles. Using this, we'll be able to efficiently find the entire set of triangles that intersects a particular plane in (roughly) $\mathcal{O}(n \log m)$ time, where n is the number of triangles per slice. There are also other options for that (some are even faster than $\mathcal{O}(n \log m)$), you can implement any of them.

Another bottleneck of the algorithm comes from the edge stitching step. One way to stitch the edges of a contour together might be to connect nearest endpoints; but such an algorithm, if implemented naively, will run in $\mathcal{O}(n^2)$ time, where n is the number of the triangles in the slice. Here, you should implement the edge stitching algorithm in a more

[1] `https://en.wikipedia.org/wiki/Interval_tree`

clever way. Think of how to find the nearest neighbors quickly or even directly find the stitching edge by some indexing technique. Please refer to section 7.2 for grading criterion.

Here are the steps you need to complete in this part of assignment:

3.1 (25 points) Implement an accelerated slicing algorithm. You will implement the function `Slicing_accelerated` method in `Fabslicer.hpp`. You can also add any necessary code in the codebase to help implement your algorithm. Exceptionally, if you choose to implement the interval tree, you can complete the functions `build` and `query` in `IntervalTree.hpp` and some other code to call those functions.

3.2 (15 points) Implement an accelerated edge stitching algorithm. You will implement the function `CreateContour` function in `FabSlicer.hpp`.

Tips: You can find more details and tips both in the code and in section 6.

4 High-level Understanding (10 points)

Question 1 You have implemented a simple slicer that takes in a triangle soup. This code will work well if you give it a "nice and clean" input, but is not robust. Describe one type of input that could cause your code to fail and then describe a method to

1. detect this type of input
2. handle this type of input.

5 Extensions and Extra Credit

We will award extra credit for (correct) implementations of each of the following.

5.1 Accelerated Edge Stitching (10 points) - 6.807 only

One way to stitch the edges of a contour together might be to connect nearest endpoints; but such an algorithm, if implemented naively, will run in $\mathcal{O}(n^2)$ time, where n is the number of the triangles in the slice. Here, you should implement the edge stitching algorithm in a more clever way. Think of how to find the nearest neighbors quickly or even directly find the stitching edge by some indexing technique. Please replace your original `CreateContour` function with your accelerated algorithm.

5.2 Generating the Infill Patterns (10 points)

Implement a simple cross-hatching print infill pattern by completing the method `Infill` in FabTranslation/FabSlicer. Each infill edge should be only contained in the interior of the mesh.

The method `Infill` receives the contour you generated in last step and the infill pattern parameters (*e.g.* spacing), and outputs an edge soup for each slice representing the infill pattern. The infill pattern should be a structure which can be built procedurally, which means the infill patterns on different slices should be overlapped in vertical direction so that the infill material will be always supported by other infill material under it. The simplest infill pattern is cross-hatching pattern. Cross-hatching pattern is a grid pattern formed by two sets of axis-parallel lines with equal spacing. For each slice, you need to find all intersection points between your generated contour and each line of cross-hatching pattern. Based on those intersection points, you need to tell which part of the cross-hatching pattern is inside the contour, and you should only generate that portion inside contour as your infill pattern. (If you are familiar with ray-tracing, you can think of this as an extremely simple 2D ray-tracing exercise). For cross-hatching pattern, you can use the default spacing in starter code.

5.3 Custom Infill Patterns (15 points)

The cross-hatch is an easy-to-implement, though mechanically and time-wise suboptimal infill pattern. Implement any of the infills in Figure 1, or even better, implement the Fermat spiral infill pattern [3].

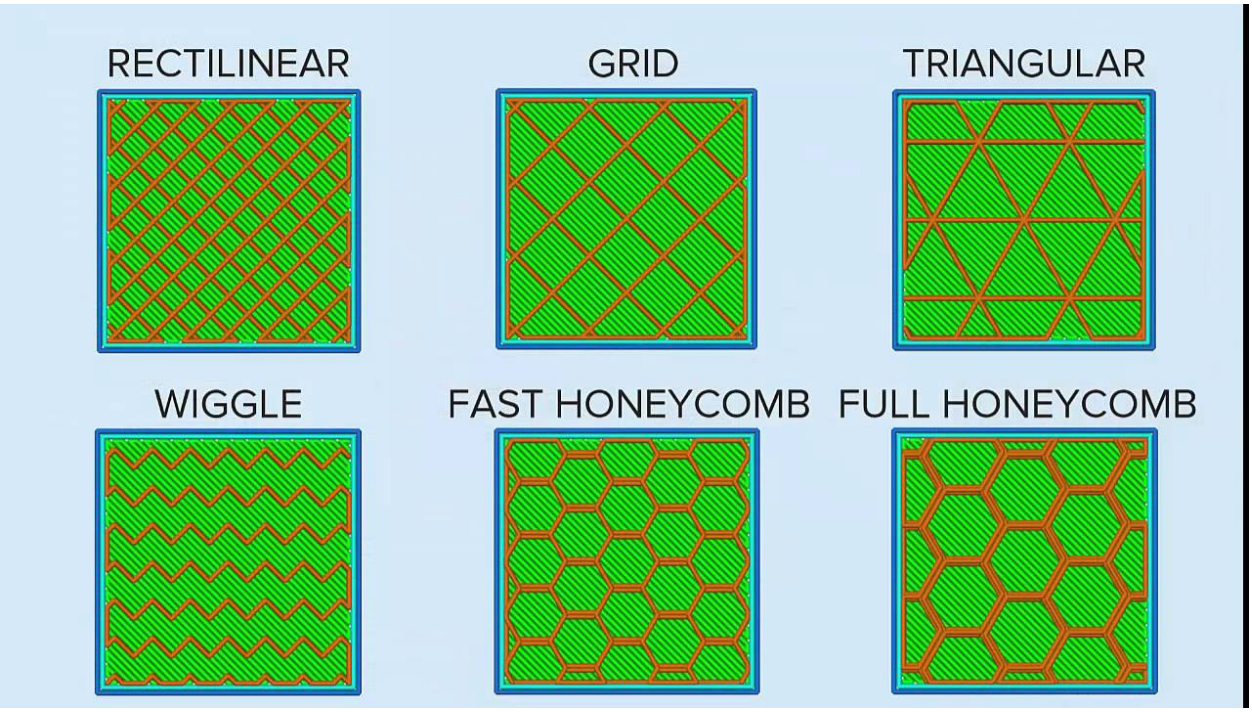

Figure 1: Various commercially used infill patterns.

5.4 Support Generation (15 points)

While our virtual printer has no gravity, a physical 3D printer will. For real prints, it's important to a) detect unsupported overhangs, and b) generate sacrificial print patterns to support these overhangs. This is tricky, since the supports must be supported themselves! Overhangs up to a certain length tolerance are okay (make this tolerance parameter in your code that we can play around with). There are many strategies to support generation; feel free to choose any and cite your sources. You can try a simple/naive method or go all out an do a more advanced algorithm. For a more advanced method, we recommend, in particular, [1], [2], or the MeshMixer algorithm described in class.

6 Tips

1. To run the code, please run `./assignment1/run1 bunny 1`. The first argument indicates the model you want to test (you can choose from bunny/armadillo/tyra/cube), and the second argument indicates if you want to run your bruteforce algorithm first. By default (`./assignment1/run1`), it will run bunny example with bruteforce algorithm first.

2. The slicer is run by using the `RunTranslation` method of FabSlicer. It takes three arguments. The first two are the contours and infill patterns generated by your slicer (output) The third argument, `bruteforce` indicates whether to use your bruteforce slicing implementation (input).

3. For the purposes of debugging you'll probably want to try out different models. A few have been provided for you in the `data` directory. This is also where the output of your slicing goes by default, as well as the timing results. If you would like to run a different model, call the `slice` method of `main.cpp` and pass in the path to a different .stl file. Only the first string argument needs to be an existing file, the other three are used to specify where the output files should go.

4. The function `ConvertToGCode` automatically generates gcode for your generated contour and infill pattern. The generated gcode can be visualized in Repetier-Host.

7 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 `assignment1` folder and `Common` folder should be submitted.

2. A short .pdf report describing how your code works and the High-level Understanding questions in section 4.

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).

7.1 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.
- Did you do any extra credit? If so, describe it as above.
- 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).

7.2 Grading

Three parts will be considered during grading:

1. **Correctness**: We will run your code locally in the environment specified in your report, and the code which does not compile in the environment specified cannot and will not be graded. We will compare the results generated by your solution with correct implementation and check its correctness. Specifically in this assignment, we will compare the .ply file and gcode generated for slicing, contour and infill patterns with the correct results. (Those .ply will be generated automatically with starter code).
2. **Speed**: For part 3 in this assignment, you are supposed to implement some accelerated algorithms for slicing and stitching contours. The given code will output the running time of each function. A code which can finish slicing in 0.5 seconds and finish stitching edges in 0.5 seconds for all three examples is considered to be correct. (Standard implementation finish both steps in 0.08 seconds.)
3. **Writeup**: The submitted report should be clear enough.

References

[1] Jérémie Dumas, Jean Hergel, and Sylvain Lefebvre. Bridging the gap: Automated steady scaffoldings for 3d printing. *ACM Trans. Graph.*, 33(4):98:1–98:10, July 2014.

[2] J. Vanek, J. A. G. Galicia, and B. Benes. Clever support: Efficient support structure generation for digital fabrication. *Comput. Graph. Forum*, 33(5):117–125, August 2014.

[3] Haisen Zhao, Fanglin Gu, Qi-Xing Huang, Jorge Garcia, Yong Chen, Changhe Tu, Bedrich Benes, Hao Zhang, Daniel Cohen-Or, and Baoquan Chen. Connected fermat spirals for layered fabrication. *ACM Trans. Graph.*, 35(4):100:1–100:10, July 2016.