The Many Faces of Microbial Communities

Senior Design
Winter Term

Thomas Albertine, Michael Phelps

Abstract

We intend for this project to provide an easier to use way to visualize microbial population data than some existing techniques, such as pie or bar charts, within the constraints provided by our client. We have had working backend components for loading population data files in the Quantitative Insights into Microbiology Ecology Operational Taxonometric Unit (QIIME OTU) format, generating 3D models based on the data in Wavefront obj format, and converting population counts into normalized values that can be used to generate 3D models of human faces. We also had a preliminary user interface (UI) for most of our project. Since then, we've improved the UI, connected the UI with backend functionality mentioned before, and done user testing to identify new improvements to be made. The remaining work is to fix the problems we've identified in our testing, continue testing to identify more problems, and possibly implement some of our stretch goals, such as expanding file format support.

CONTENTS

I. PURPOSES AND GOALS

The purpose of the project is to visualize microbial population data in a way that allows microbiologists secondarily other researchers to more easily compare complex population data. To that effect, our client has requested that we model it as human-esque faces, in order to take advantage of humans' ability to recognize and compare those faces. Resulting models need not look like visually appealing humans, but they should represent the data. For example, the user might associate Escherichia coli with the left ear rotation attribute, in which case a sample with a large population of Escherichia coli will correspond to a model where the left ear is tilted back more than other models.

To that effect, our project loads population data from files in supported formats and sample grouping data in corresponding metadata files. This data is converted to model attributes, values that determine how exaggerated some feature of the face (called a model parameter) is, based on associations provided by the user in the application. These attributes are passed into MakeHuman, an open source project that generates 3D models of humans. It generates 3D models in Wavefront obj format for each sample. Then the models are loaded and displayed to the user, who can view smaller sets of samples based on grouping data, or can choose a subset of no more than six samples to view and inspect in more detail.

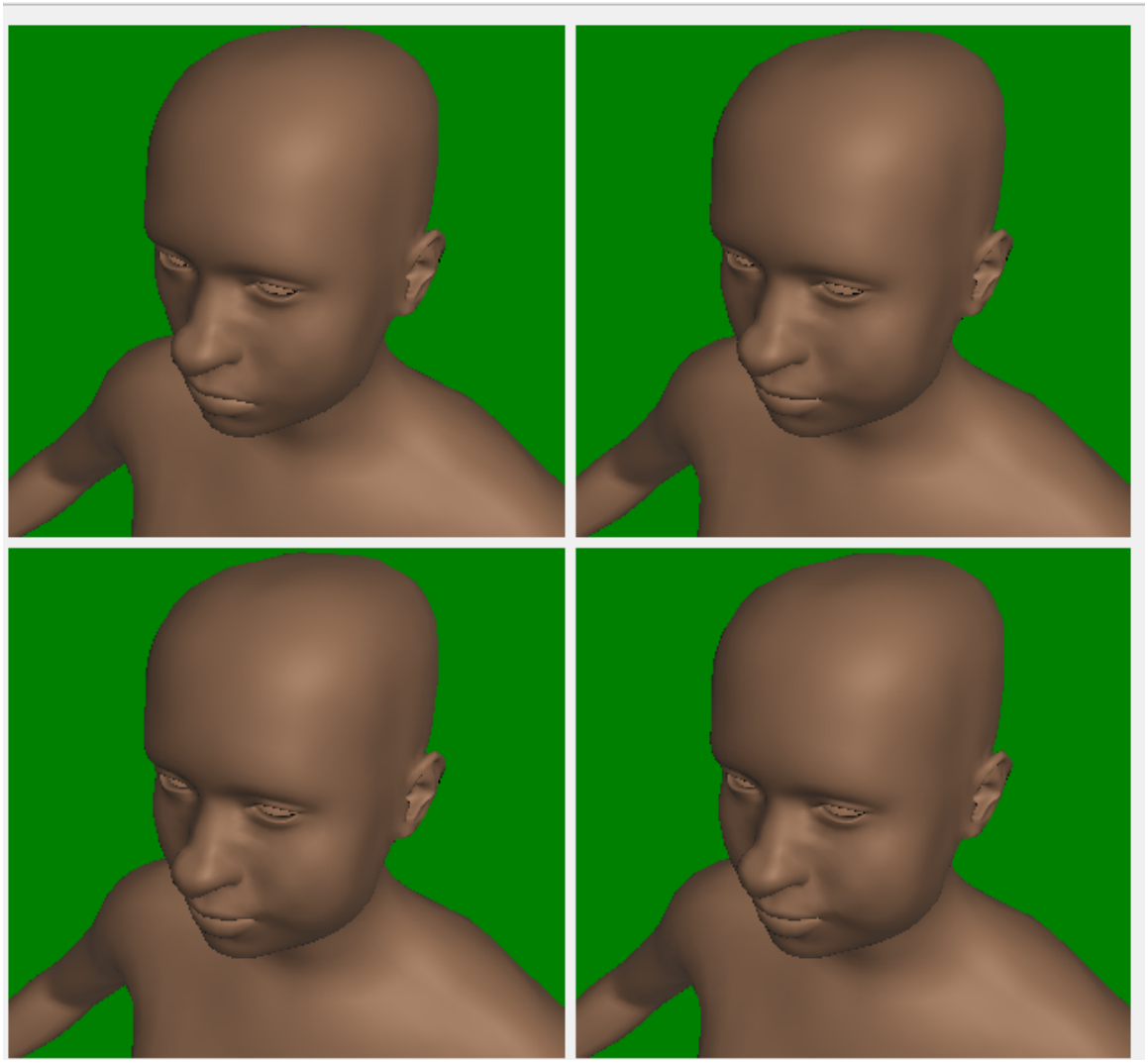

Fig. 1. Several models generated from a sample data file. This file had few enough organisms that only some features were modified. The rest were left at their default values to avoid distracting from the significant features of the data set. This image was generated using the obj viewer widget that Thomas wrote and the analysis view page that Michael wrote.

II. Completed Content

A. *Backend*

In the backend code of the project, we already wrote wrapper code around MakeHuman that can retrieve a list of valid model parameter names[1] as well as receive model parameter names and values in order to generate models. We have resolved a deficiency that prevented our models from generating with eyes. Figure 1 depicts a model generated from a data file.

[1]Our requirements document specifies that we support 150 model parameters, but we only have enough facial features to support 138. We can extend that number significantly however, if we use other model parameters besides facial features, but our client mainly wants to work with facial features alone. We are considering adding the others as optional parameters.

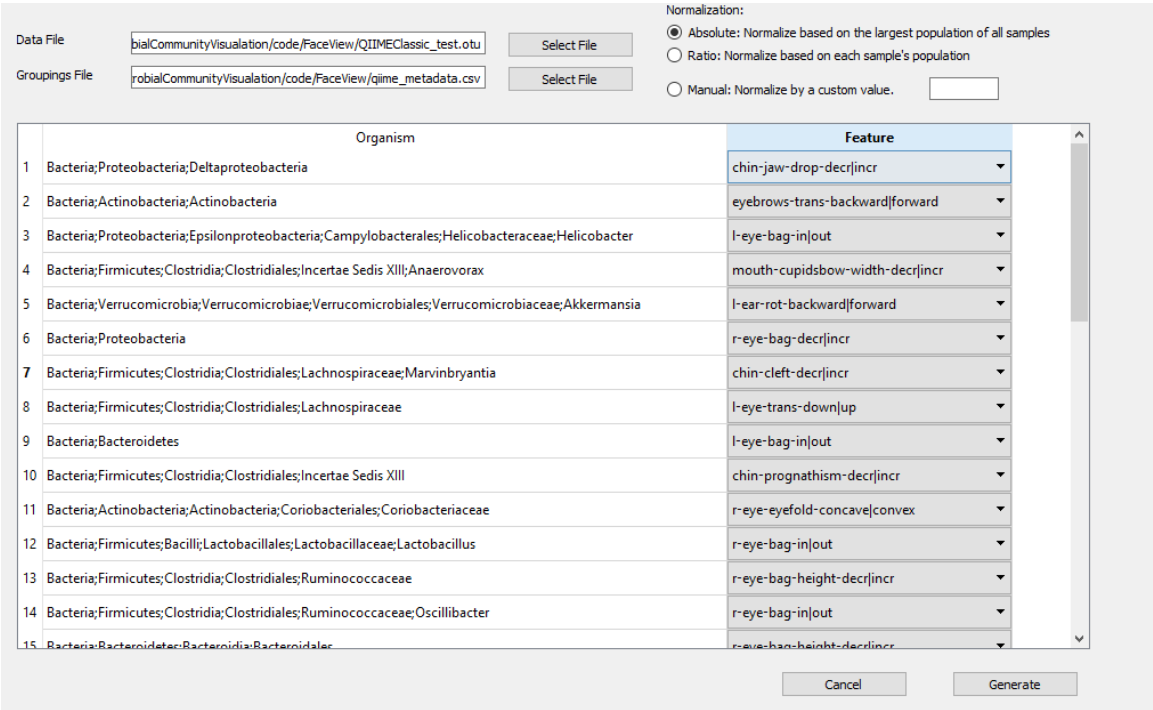

Fig. 2. The UI page that allows users to perform tasks related to loading data files.

We also wrote code to load a QIIME OTU data file, which the client specified as the minimum file support for data loading, as well as a framework to allow us to easily add support for other file types later on. This is described in more detail in section IV later in this document, but in short, we create a Python object that can parse a file format, and associate it with certain file extensions in another object that we have also created to act as a registry. When loading a file, simply pass the file path into the registry object, and it uses the extension from the file path to find the right parser, which is provided the file path. Each parser object is responsible for parsing the data file itself, as well as the metadata file that contains grouping information.

We also wrote code to normalize sample data based on the largest population in the sample (in order to compare ratios of populations between samples), based on the largest population in the file (in order to compare population sizes between samples), and based on a manual scaling value that can be set by the user, in order to compare between images of previously generated models, or models from separate files.

B. Frontend

We wrote the data loading page, the main selection page, and the detailed viewing page. Currently, the grouping functionality, model generation, and microbe-model parameter mapping, are all connected to the UI that supports viewing the generated models.

The Data Loading page (see Figure 2) allows users to select a data file and a groupings file. These files will be loaded, and then used to populate the table on the bottom, which allows users to associate model parameters with organisms in the sample. On the top right, a series of radio buttons can be used to specify which normalizing strategy to use. That is, whether the generated models should reflect ratio of each

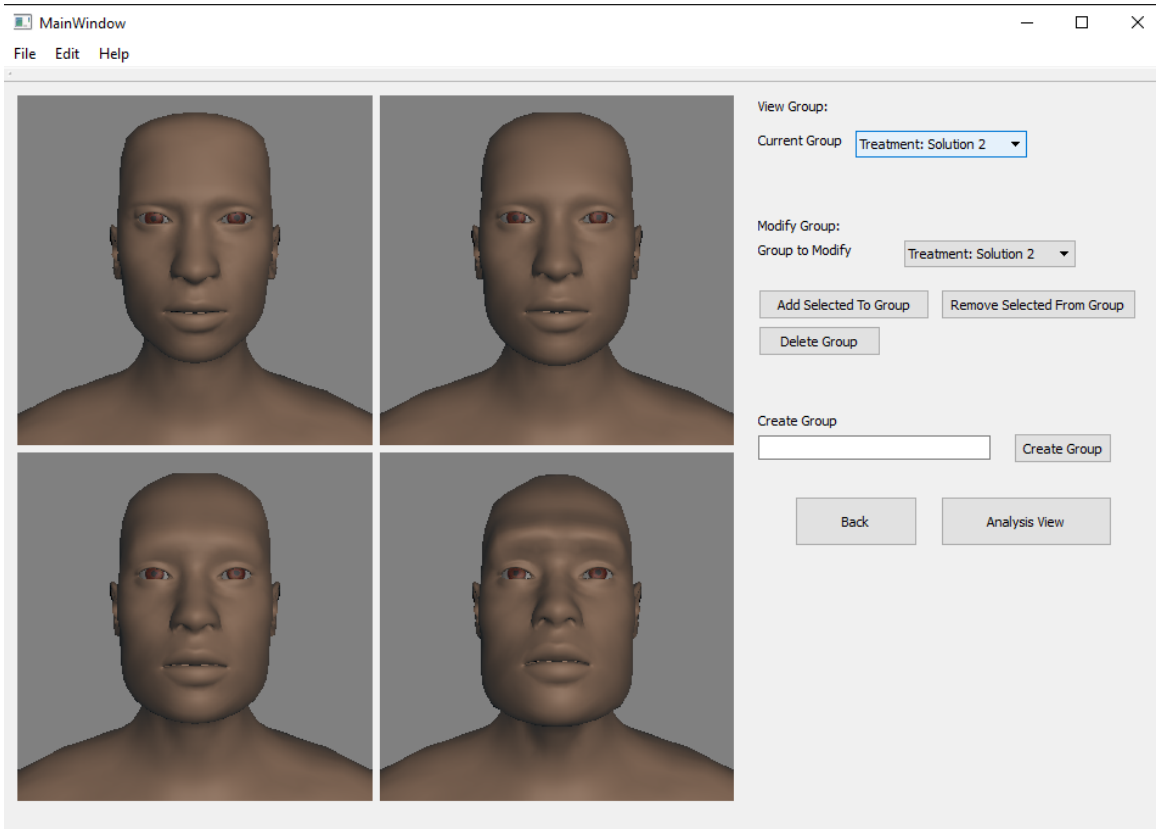

Fig. 3. The UI page that allows users to perform tasks related to selecting data files.

organism to the whole of the sample, or whether the generated models should reflect the ratio of each organism to the largest population in the file, or whether the models should be normalized in terms of some other number entirely. The first is good for identifying samples with similar ratios of an organism to the population in the sample. The second is good for identifying samples with similar total populations of an organism. The third is good for comparing models in a different file.

The Main Selection page (see Figure 3) will allow users to select a few samples out of the file to examine in more detail. The box on the left is where images of the samples' models will appear, while the elements on the right allow users to organize samples into groups, or view groups from the metadata file.

III. Significant Problems and Associated Solutions

One of the most difficult problem that we faced was trying to read and decypher the MakeHuman code enough to write the wrapper. Since Python is loosely typed, he couldn't match the types to see how things were being used. Instead, we walked through the code, jumping from function definition to function definition, searching through multiple files. Eventually, we remembered that Python objects have a dictionary of their attributes, so we were able to import MakeHuman modules and examine the the contents of objects returned from functions which sped up the process considerably. Additionally, we learned how to use grep-like functionality from within PowerShell, which made it much easier to look for function definitions.

Additionally, the obj viewer widget proved challenging because the the error policy for OpenGL is to fail silently and not draw anything. Most of this can be mitigated by occasional calls to $glGetError()$ Some mistakes however, are not technically errors, but still don't produce the intended output.

Another difficult problem was figuring out how to properly transition between the three interfaces and using PyQt in general, as we had never used any GUI libraries before outside of GLUT. The main difficulty came with Qt's poor functionality when trying to control multiple separate windows without having a main central window. Additionally, there was a great deal of difficulty in making sure all of the libraries used in the code were in sync between our two developers.

IV. Interesting Code

A. *fvParser*

The ParserRegistry is little more than a wrapper around a dictionary, mapping a file extension to a parser object, but with some additional functions added to it to make its usage more obvious. An instance of the Registry is created as a global, so that it can be referenced anywhere by importing the module.

```
#The parser registry
class parserRegistry:
        def __init__(self):
                self.mapping = {}

        def registerParser(self, parser, extension):
                self.mapping[extension] = parser

        def getParserForExtension(self, extension):
                if self.extensionIsSupported(extension):
                        return self.mapping[extension]
                else:
                        return None

        def extensionIsSupported(self, extension):
                return self.mapping.has_key(extension)

        def parse(self, filePath, metaPath):
                fileExt = os.path.splitext(os.path.normpath(filePath))
                    [1]
                parser = self.getParserForExtension(fileExt)
                if parser is not None:
                        return parser().parseFile(filePath, metaPath)
                else:
                        return None
```

Fig. 4. This is an excerpt of the source file fvParser.py containing the ParserRegistry class.

V. User Study

We have completed our first user study and gained some suggestions and insight to improve the project's usability.

A. Users

For our test subjects, we found college students of various disciplines and backgrounds. This was more than sufficient to provide suggestions about, and identify flaws in our UI, especially since it is relatively new and untested.

B. Methods

We used a set of 5 moderated, in-person tests. This was enough to identify several bugs and other opportunities for improvement, but not so many that it took an impractical amount of time to moderate it.

C. Tasks

Before each task, we asked users how confident they are that they can complete the task, and timed them while they performed the task, recording the time it took them. When we saw them struggling over a particular step in the task, we made a note of it. If the user seemed unable to make progress, we made a note of what step they were stuck on, politely interrupted them, and completed the step ourselves so that the user could see how it was done. When users requested that we repeat the task statement or some detail of it, we did. None requested clarification, but if they had, we would answer their questions and make a note of that as well, so that we could design a better test in the future.

Our tasks are as follows, starting immediately after opening the application for the first time, and given a sample data file:

1) Load the data file.
2) Associate Bacillus subtilis with head squareness[2].
3) Modify settings so that you can compare ratios of different populations between samples.
4) Generate the models.
5) Select some samples to compare in more detail.
6) Find the two samples with the most similar ratios of Bacillus subtilis to total population and view them in more detail.
7) Find the two samples with the most similar total population of bacillus subtilis and view them in more detail.

After the first few tests, we removed a task called "Manipulate Models" because users invariably started playing with that feature before we had a chance to time them or ask them anything about it.

The first five tasks are primarily to identify how intuitive the application is to a new user. The last two are to identify how easy the application is to use once someone has seen how it works.

D. Results

Figure 5 contains our data for the UI testing. In addition, we also have a series of observations that we made while watching our subjects, and comments that our subjects made when we asked them afterwards.

[2]This organism and this model parameter need not be the ones used in the test

Task	Subject					
	1	2	3	4	5	Average Time
1	“Yes” 0:31	“Yes” 0:44	“Medium” 0:20	“In the middle” 0:56	“Yes” 0:38	0:38
2	“No” 1:00	“Don’t Know” 0:12	“Very Confident” 0:01	“Medium” 2:00	“No” 0:11	0:41
3	“Yes” 0:02	“Yes” 0:03	“Confident” 0:01	“4/10” 2:27	“No” 0:22	0:35
4	“Yes” 0:10	“Yes” 0:12	“Confident” 0:10	“4/10” 0:10	“Yes” 0:10	0:10
5	“No” 1:20	“No” 0:09	“Confident” 1:00	“5/10” 0:09	“No” 0:11	0:34
6	“Yes” 1:26	“Yes” 0:53	“Maybe” 3:03	“6/10” 3:08	“No” 0:59	1:54
7	“Yes” 0:57	“Yes” 1:27	“Yes” 0:50	“Yes” 1:40	“Yes” 1:54	1:22

Fig. 5. This is the data from our UI testing. The middle cells contains the user’s description of his or her confidence before attempting each task as well as the time it took the user to complete the task

1) Subject 1: This subject is a junior majoring in biology. During task 5, she found that the blue and green colors we currently use to denote whether the models are not particularly intuitive[3]. During task 6, she spent longer than expected looking through the model parameter options, which delayed her completion time. Afterwards, she commented that the names in the model parameter dropdown were not particularly intuitive.

2) Subject 2: This subject is a freshman majoring in mechanical engineering, and who’s first language is not English. He discovered that not every selected sample appears in the analysis page when the analysis view button is clicked, a bug that has since been fixed. He also commented that the model parameter list is “detailed”, and that the samples do not show their sample name in the thumbnail view.

3) Subject 3: This subject is a freshman majoring in computer science. He showed signs of being competitive regarding completion times, but got distracted looking through the model parameter list. He commented that he was confused about how the data sets corresponded to the generated faces, but later said that the relationship was intuitive if the user understands what the input data contains.

4) Subject 4: This subject is a freshman majoring in business. He showed signs of being tired or not having slept enough recently, which may have affected his results. He made jokes about sleep deprived researchers which seems to confirm our observation. Initially, the subject had difficulty using functionality provided by Windows, accidentally minimizing the window, and then closing it while trying to unminimize it, but he did better once we reopened it for him.

During task 2, he commented that the tool was “very thorough”. During task 3, he was unable to find the radio button to specify that data be normalized as a percentage of the total sample population, requiring me to point it out to him. During task 4, while waiting for the models to generate, he suggested adding something to let the user know that the application is making progress.

Afterwards, the subject said that the tasks were difficult and that he would like to try again for a shorter completion time. He also mentioned that the relationship between the model parameters and their effect on the generated model was not particularly intuitive.

[3]We have since changed the colors to grey and green

5) Subject 5: This subject is a senior majoring in fish and wildlife who claims to be bad with computers. During task 7, he mentioned that not all model features are visible from the front view provided from the thumbnail view page, and suggested making the models rotate.