FaceView Design Document

# Introduction

## Identification Information

November 19, 2015 (v0.1)

ViewCrobe Software

Thomas Albertine, Michael Phelps

## Change History

v0.1 – November 23, 2015

## Purpose

This document describes the design of the FaceView project. Different components are intended for different audiences, but all will be helpful for the developers creating the project. For stakeholders representing users, the UI and use case sections will be most helpful.

## Scope

Our goal for the FaceView project is to make it easy to visualize population data as faces. Specifically, such data would be microbial population data, but the same principles apply to many kinds of population data, so the project has applications outside that specific area. These kinds of visualizations may help scientists identify patterns between samples that would otherwise be missed when examined through existing visualization techniques.

## Overview

This document covers the design from different views. It begins with views more relevant to stakeholders representing users, and moves to additional views that might be more helpful to developers who would need to read the whole document anyway. Within this framework, focus moves from a high level downward when applicable.

In this document, we recognize two stakeholders. Firstly, there are our users. This category also includes the project proposer, as Dr. Lang is a microbiologist and therefore is representative of our primary users. We recognize the following concerns on behalf of our users.

* Loading data and file type support
* Interpreting data
* Viewing results
* Large (hundreds of samples) input support
* Ease of use

Our second stakeholder is the developers of the project. On behalf of our developers, we recognize the following additional concerns.

* Maintainability
* Interactions between components

Table of Contents

[Introduction 1](#_Toc436073348)

[Identification Information 1](#_Toc436073349)

[Change History 1](#_Toc436073350)

[Purpose 1](#_Toc436073351)

[Scope 1](#_Toc436073352)

[Overview 1](#_Toc436073353)

[User-oriented Views 2](#_Toc436073354)

[Use Cases 2](#_Toc436073355)

[Visualize Data 3](#_Toc436073356)

[Configure Parameters 3](#_Toc436073357)

[Load Data 3](#_Toc436073358)

# User-oriented Views

## Use Cases

The following use cases describe the operations that the FaceView software will support. Depending on the complexity of the use case, commentary may also be included. It behaves as a context viewpoint and addresses user related concerns regarding loading and interpreting data, viewing results, and working with large data sets. We believe that this viewpoint can best describe the general functionality of the product.



Figure : The four main use cases

### Visualize Data

The obvious goal when using the FaceView visualization software is to visualize the population data. To actually display the visualization, the software must generate a 3d model representing the sample and draw that to the screen. In order to examine all of the aspects of that model, it is necessary to allow users to manipulate that model, so simple transformations are necessary, such as camera rotation, zoom, and pan.

### Configure Parameters

Depending on what kinds of patterns the user is looking for, or where the user expects to find it, certain parameters may need to be explicitly set certain model features to correspond to certain organisms. For example, if a user is particularly interested in populations of a small set of bacteria, he or she might associate those bacteria with more obvious model features, like nose length, mouth size, and ear angle.

Additionally, the user might be interested in normalizing the data in different ways. If the user is interested in the populations as a fraction of the whole, the numbers might be normalized based on the total population of the sample. If the user is interested in the populations as a number relative to other samples, the numbers might be normalized based on the greatest value for that organism in the set of samples. Finally, if a user is interested in comparing the samples to a previously generated image, he or she has the option to normalize based on any chosen number, with the understanding that the results may be capped if necessary.

Finally, in the event that there are more organisms in a sample than there are supported features in the model, the user will have to select which organisms to compare, and which to ignore.

### Load Data

In order to generate models based on population data, the program must be capable of loading the data. As per our client’s request, the project will support, at minimum, the QIIME tab delineated format, as it is well established in the community. It is currently a stretch goal to expand support to other types, such as the BIOM format (which is JSON based) or to a plain CSV format.

Additionally, some formats, including QIIME tab delineated may include a metadata file that would allow for categorization. This would be loaded as well.

### Group Data

In order to handle large data sets, it may be useful to categorize certain selections of samples. As mentioned in the “Load Data” section, that initial category data may be loaded from a metadata file, or it may not contain any categories. In either case, categories can be modified later on.

## UI Mockup

The following is a wireframe mockup of the UI which will clarify how the user will interact with the program. It describes the process for loading a file, the thumbnail view, and the analysis view.

### Load Menu



Figure : A mockup of the UI for loading a file containing three organisms and 256 samples, to be normalized with a maximum value of 42

Figure 2 is a mockup of the UI for loading an input file and setting parameters. The first step in loading a data file is to select which files to load. The file selection boxes are in the upper left corner. Once files are loaded, the next step is to configure model generation parameters, so dialogs for those parameters are adjacent to the file selection boxes. Finally, the last step is to actually generate the models, so the “Generate” and “Cancel” buttons are in the lower right, farthest away from the file selection boxes. In this way, the order in which details are specified mimics the order in which the user might interact with other familiar systems, like reading a book.

On the left, the user can specify which model features correspond to which organisms in the data file. That way, users can assign the most interesting organisms to the most interesting features, or prevent uninteresting organisms from being visualized at all. Additionally, this allows users to resolve issues arising from the data file having more organisms than there are model features.

On the right, the user has normalization options. These are important because the range of a population in a data file could be very large, but the model generator only needs to know how pronounced a feature is. This range constraint restricts model parameters from 0, where the feature is nonexistent or minimal, to 1, where the feature is the most pronounced that it can be. In the mockup there are three options for normalization.

First, there is the option to normalize based on the greatest population of this organism in the file. This allows users to compare populations between samples in the file, which is helpful for drawing conclusions involving the total populations.

Second, there is the option to normalize based on the total population in the sample. This is good if the samples are of varying size, or if the user is more interested in the proportion of the population that is a given organism.

Finally, there is the option to normalize manually based on an arbitrary value. This is convenient if the user wants to compare features to an exported image of a previous sample, but does not want to add it to the existing file, or if the user does not have the data or parameters that were used to generate the existing visualization.

In the bottom right, there are the “Cancel” and “Generate” buttons, as well as a count of the number of samples. The two buttons are fairly obvious, but the count is particularly helpful because it allows users to confirm that FaceView was able to correctly read the data file by comparing that number with the expected number of samples. For additional confirmation, the user can compare the organisms found in the organism-model feature association dialog with the expected organisms in the file.

### Thumbnail View



Figure : A mockup of a Thumbnail View containing eight samples, currently view the group of all samples, and

Figure 3 is a mockup of the thumbnail view. This view allows users to modify groups, filter by groups, and view selected samples in detail. In the mockup, two sections are apparent. The left panel allows users to select, deselect, and preview samples, while the right panel allows users to perform operations on the selection.

The right panel allows the user to specify which group he or she would like to view, add or remove samples from any group, delete a group, or create a new group with the selected elements. There are a few special groups, for the user’s convenience.

The first special group is the “All” group. This group is available by default and always contains all samples. Therefore, it cannot be deleted or modified and it will not be selectable on the “Modify Group” drop down. The second special group is the “Selected” group, which includes only the samples that are currently selected. As viewing models in detail imposes certain constraints described later on, this view is helpful to deselect samples that violate those constraints.

Finally, there is a button to view the samples in detail. The main constraint to view samples in detail is that the user cannot view more than six samples at a time. This is because humans can’t keep more than seven plus or minus two things in working memory at any given time, so comparing more than seven samples is, in general, more detrimental to users’ ability to compare samples. We choose six as the limit, because decreasing the number would make the software slightly more easy to use, and because 7 (as a prime number) would result in wasted screen space. This is because it is impossible to arrange seven rectangular objects of the same shape in a 2 dimensional grid with more than 1 entry in both dimensions, without leaving empty space.

### Analysis View



Figure : The analysis view, where the selected samples were Sample 1, Sample 7, and Sample 6. If more samples were selected, the three empty windows would be populated.

The Analysis View allows user to compare their models more closely. In this view, samples are rendered in 3d in real time, so that users can manipulate them to better compare features. The supported manipulations are camera pan, zoom, and rotate. For ease of comparison, each window is manipulated together. In the event that the user loses the model, perhaps through excessive zooming or panning, there are buttons to return the camera to one of several common camera positions. On the right, next to the camera position buttons, there is a table reminding the user which attributes correspond to which organism. While the mockup only demonstrates three samples, the application supports up to six.

# Developer-oriented Views

## General Structure



Figure : A flowchart representing the general structure of the project.

Figure 5 summarizes the general structure of the project. It contains each entity in the project as well as their communications (and relationships). The main components of this diagram are the File Loaders, the Model Generator Translator, Data-Model Parameter Mapping, Frontend, Model Generator, and Model Loader.

The entities are presented in the order in which they are invoked in a typical use case, with the exception of the Frontend, which is invoked both at the beginning and the end of the process. These components are likely to be developed in the same order, again, with the exception of the Frontend, as it is easily mocked up for development purposes

### Local Frontend

The Frontend is where the user instigates any action of the rest of the program. It tells the File Loaders which files to load and when, and it stores the user’s Data-Model Parameter Mappings (which also includes normalization settings). When model generation is done, is receives the generated models and thumbnail images from the Model Loader and presents them to the user. It is also where Grouping information is managed, and therefore is where the File Loader sends default grouping information. For more information on the Local Frontend, see UI Mockup. As a stretch goal, we may also create a Web UI so that the tool can be used online. This component will likely be written last, as it depends on a large number of other components, and it can be mocked up and automated for development purposes relatively easily.

### File Loader

The File Loader is the component that reads files from the filesystem. It is composed of one file type specific loader for each file type (i.e. a BIOM loader) that is registered with a file extension. When a file is passed to the loader with that extension, the loader sends it to the appropriate specific loader, which loads the data. Each specific loader puts the data into a unified format for simplicity later on. This unified data is passed to the Model Generator Translator.

### Data-Model Parameter Mapping

The data-Model Parameter Mapping is a structure that contains associations between organisms present in the data and features present in the model. This is configurable by the user between initial file loading and model generation. These configurations are then used in the process of translating population data to model generation instructions. The advantage of this strategy is that, by passing these associations, users can force particular organisms to use more obvious model features. In addition to specifying a feature, there is an option to associate an organism with no feature, in which case it will be ignored, or with any feature, in which case an unallocated feature (or no feature, if none are available) will be chosen for it when it passes through the translator. Users are not guaranteed that the same organisms will be assigned the same features, but the Data-Model associations (including post translation) are listed in the Analysis View described in the UI Mockup section. Normalization configurations are stored in this component as well.

### Model Generator Translator

The Model Generator translator receives the population data and the Data-Model Parameter Mapping data and uses these to produce instructions for the Model Generator. Though the diagram shows this component interacting directly with the Model Generator, in reality these instructions are passed back to the UI before invoking the Model Generator so that the progress of the visualization process can be presented to the user. It is during this step that the Model Storage Folder is cleaned out, in order to minimize disk space usage.

### Model Generator

The Model Generator component receives the instructions generated by the Data-Model Parameter Mapping and uses them in conjunction with the MakeHuman Python API to generate 3d models of Humans in obj format. Thumbnail images are also generated at this time. Because each invocation of the Model Generator generates results for a single sample, it can be invoked several times in parallel. The produced models are stored in a Model Storage folder that was cleaned out before invoking this component. As this component will be invoked parallel to the main execution of the program, the main execution will use the return value to recognize when instances of the Model Generator are finished, and inform the user.

### Model Loader

Once the last instance of the Model Generator has terminated, the Model Loader loads the thumbnails on behalf of the Frontend. When the user has selected samples to examine more closely, the Model Loader also loads the obj file human models into a pyopengl-friendly structure. These structures are returned to the Frontend, where they are presented to the user.

## Inter-component Communications

Most components communicate with each other in a standard way. That is, the return value of the function implementing the component is passed to the next. However, communication between the Model Generator Translator and the Model Generator, communication between the Model Generator and the Model Loader, and communication of the Data-Model Parameter Mappings between the Frontend and the Model Generator Translator do not follow this pattern completely.

### The General Case

For most communications between components, the process is simple. When the component is invoked, it is contained within a function call. The communication is as simple as the arguments passed in and the return value.

### Model Generator Translator and Model Generator

The communication between the Model Generator Translator and the Model translator is special because the Model Generator should allow for parallelism. Additionally, it should be independent from the rest of the program, so that it can be easily reused for the Web UI stretch goal.

To that effect the Model Generator Translator outputs its results as a set of attribute-number pairs, where each attribute is a possible feature of the model, and the number is between zero and one, representing the value or the parameter in terms of its minimum and maximum values. Each sample corresponds to one invocation the Model Generator, which receives the message through stdin.

### Model Generator and Model Loader

Since the Model Generator is a special case, as described in the previous section, it communicates the results in a special way as well.

Since the Model Generator can run as a standalone program, it notifies the invoking program that it is finished just like any other child process. The generated models are saved to a folder along with a thumbnail image. Once the Model Generator processes terminate, the Loader can be invoked to load the tumbnails or models.

### Data-Model Parameter Mappings

As the Data-Model Parameter Mappings are a structure existing for configuration storage purposes, as opposed to a function implementing any functionality itself, it doesn’t communicate with anything per say. Entries are simply stored in a map, associating a key (the name of the organism) with a value (the name of the model parameter. The structure is passed to the Model Generator Translator along with the input data.