Intro

The GUI is a key element of the Formagine prototyping platform – it is responsible for translating computer models into files readable by the embedded system on the robot itself. The software itself is fairly similar to standard 3D printing software packages, thus requirements will seem familiar to those who have used such software.

GUI requirements

For one, the program will accept STL models from the computer. Given that STL files are the standard for 3D printing, select that file type for this adjacent prototyping system is reasonable. As with 3D printing software, the GUI will need to support rotation and translation of a given STL model in the X, Y and Z axes. This is so that the model can be effectively positioned within the "printable" bed area. Clearly, visualization of the model within the bed will need to be implemented, as will file selection of STL files. The GUI will also need to support the setting of heating time, a key adjustable factor for the vacuum forming process. Finally, the GUI needs to export data in a format understandable by the robot. Various decisions needed to be made for this.

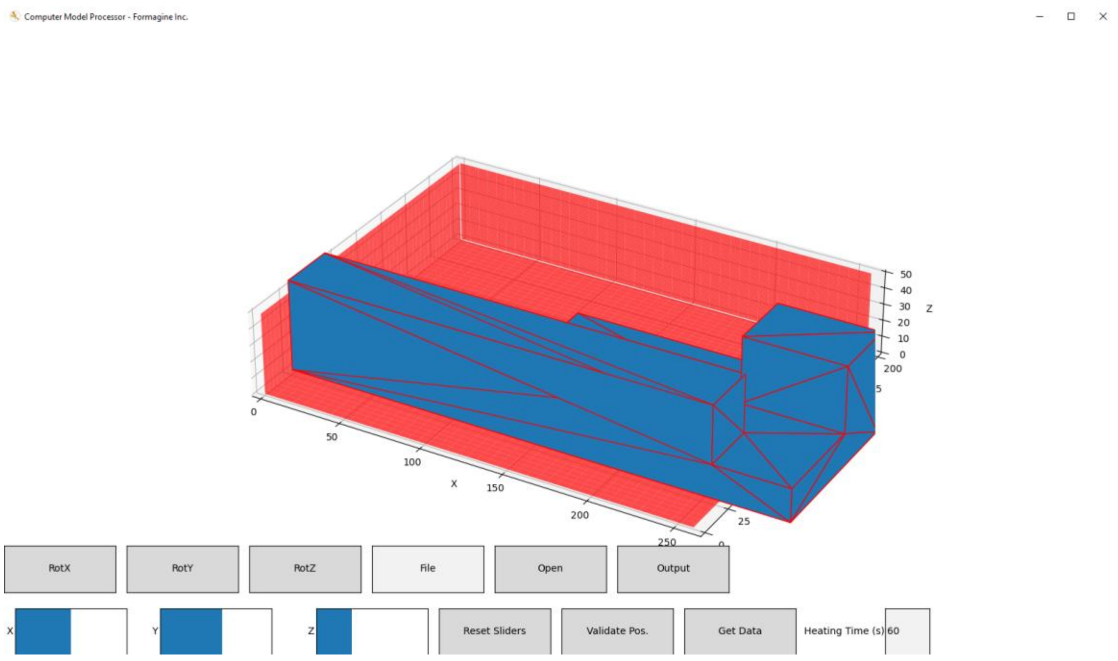
File formatting

The first step of data exporting is selecting the file format, and JSON is selected for this task. JSON files are easy to create, and have relatively low memory overhead, and are easy to process (a characteristic useful for embedded system programming). The data which needs to be translated to the system is the pin heights, and the heating time. The latter can easily be set in a "heating_time_sec" variable. However, transmitting the pin heights is less than obvious. While it would be possible to export X, Y, and Z tuples for each pin, that is not memory efficient, and is not a particularly useful format for the embedded system to interpret. Therefore, a 2D array is filled with height data. Row numbers correspond to given rows on the system, and columns clearly correspond to real pin bed column. Uniquely, rows have either 49 or 50 pins (due to the pin bed's hex grid), therefore the non-existent pins have their heights set to -1. Variables are set to assist with mapping row and column numbers to physical positions on the pin bed. Although the initial complexity in configuring such a data formatting system was somewhat high, these JSON files allow for easy pin height and heating time interpretation by the embedded system.

Implementation 1

Python was selected as the language of choice for implementing the GUI. This is due to its relative simplicity when it comes to developing large software, and its large collection of available libraries, key to decreasing implementation time and supporting more complicated behaviour. The selected library for supporting STL files was numpy-stl (https://pypi.org/project/numpy-stl/), and the selected visualizer was mplot3d (https://matplotlib.org/2.2.2/mpl_toolkits/mplot3d/index.html), a matplotlib toolkit. Matplotlib (https://matplotlib.org/) is a powerful and easy to use library for scientific calculations and visualization, thus seemed to be perfect for the task.

A functional program was developed quickly, and it can be seen below in operation. Files can be easily selected and visualized, and repositioned. A button to re-zero all translation sliders was introduced, as models would sometimes end up positioned far outside the bed when rotations were applied. This allowed for translation sliders with smaller ranges (useful for accurate manual positioning) to "bring back" a model after such rotations occurred. Heights at given positions were obtained manually in a rather unique method.



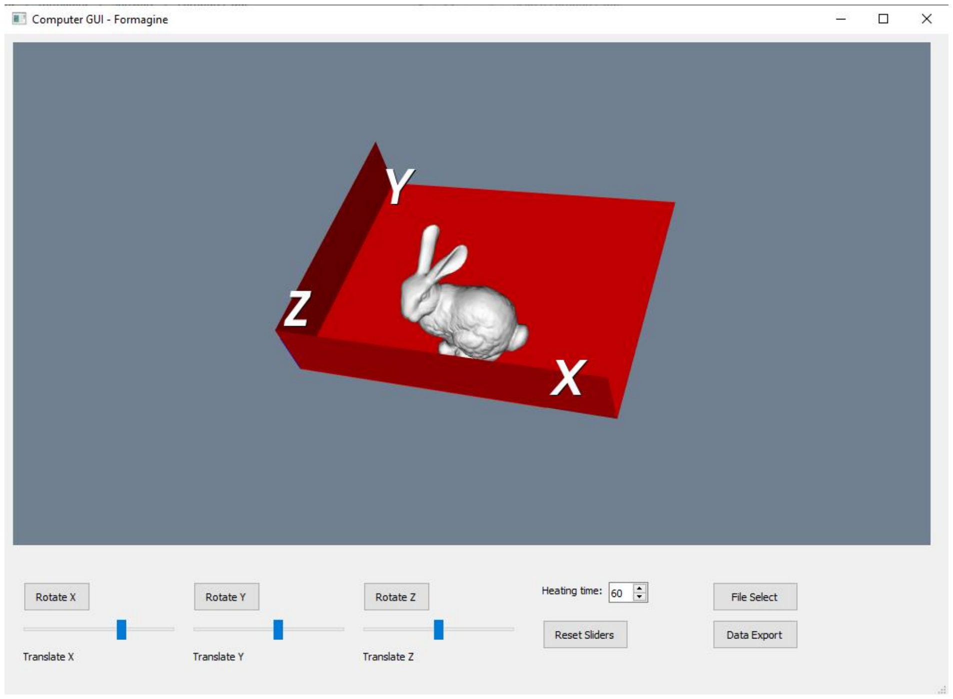
For every pin, every tuple of X, Y, Z points (this forms a triangle, which is fundamental to how STL models are stored) would be analyzed. First, a 2D triangle (in X and Y) would be developed from the tuple, and an algorithm obtained from (https://stackoverflow.com/questions/2049582/how-to-determine-if-a-point-is-in-a-2d-triangle) would determine if the pin was positioned within the tuple's X, Y range. If so, an equation of the tuple's plane (equation from https://kitchingroup.cheme.cmu.edu/blog/2015/01/18/Equation-of-a-plane-through-three-points/) would be created and an interpolation height would be calculated using the pin's X, Y position. The maximum height from all tuples would be stored as the pin's height. This ensured that if multiple planes existed above a given pin, the model would represent the top one. Due to physical limitations, heights were limited between 0 and 50 mm.

While the algorithm worked, it was extremely slow on anything but the most basic of STL models. Worse, the visualized itself was effectively unable to show "real" (ie. not basic models developed for testing purposes) STL files without effectively crashing due to its inability to handle, and visualize, the sheer amount of data in more complex models. Since this limitation was due to the library selected, and there were no clear steps to improve fundamental library performance, the GUI had to be effectively re-implemented based on a new library.

Implementation 2

VTK was selected as the renderer for the new GUI (https://pypi.org/project/vtk/). It was selected primarily due to its built-in ray casting support (to be discussed later), however its native STL support was a nice touch. A program was quickly developed to select and load an STL file, and performance was markedly improved. The program showed no performance issues loading and viewing models which caused the previous implementation to effectively crash immediately.

Unfortunately, widgets available in VTK did not seem usable / sufficient for our needs. Therefore, it was decided to implement the VTK rendering window inside a Qt user interface (https://www.qt.io/qt-for-python). Various guides exist explaining how to do so. Design of the Qt interface through code was daunting, therefore Qt Designer was used to implement the interface, including all buttons and widgets (https://doc.qt.io/qt-5/qtdesigner-manual.html). The finalized UI was a significant improvement over the previous matplotlib UI, and the improvements in the renderer made for a much more visually pleasing user experience. A screenshot of the program in operation can be seen below.



Positional interpolation followed a similar strategy as with the previous GUI, where analysis would be done for each pin and the highest point for any position, capped between 0 and 50 mm, would be stored in an array for exporting to JSON. However, instead of a manual interpolation algorithm, the new program took advantage of ray casting. In short, a line would be drawn from the bottom X-Y plane to well above the anticipated maximum height of any model (in this case, 1000 mm). Both (https://pyscience.wordpress.com/2014/09/21/ray-casting-with-python-and-vtk-intersecting-linesrays-with-surface-meshes/) and (https://stackoverflow.com/questions/57058089/how-to-use-vtkobbtree-and-intersectwithline-to-find-the-intersection-of-a-line-a) were extremely helpful in implementing this functionality. The algorithm would identify points where the line intersected with the model, and the maximum height of these points (again, capped to 50 mm), would be stored in the array of heights. This algorithm was substantially faster, allowing for analysis of significantly larger models within much shorter timespans than the previous program took to analyze much simpler models.

Both VTK and Qt are significantly more complex software packages to work with, and re-implementation was more challenging than initially anticipated. However, the new GUI has a significantly more professional appearance, vastly improved performance, and allowed for analysis of models which were impossible for the previous program to handle.

Validation

Note that it would have been effectively impossible to validate program output by running files on the robot itself, as the time spent running a file would be much too slow to help for debugging purposes. Further, the GUI was not developed in the same physical location as the robot, and development was often in parallel, further complicating matters. Therefore, a means to validate the output from the GUI was required.

It was decided to export X,Y,Z tuples from the GUI to .xyz files, which could be interpreted by LiDAR software for visual confirmation that output matched anticipated results. Since the X, Y, Z data came from the program itself, which also created the JSON files, it was decided that results from this test should be reflective of the data in the JSON files (which LiDAR data visualizers could not have rendered directly). This method worked quite well, and sample output can be seen below.