WEB-BASED MODEL SLICING
FOR 3D PRINTERS

BY

MICHAEL U.B. MEDING
B.S., UNIVERSITY OF MASSACHUSETTS LOWELL (2015)

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF MASSACHUSETTS LOWELL

Author .....
April 29, 2016

Certified by .....
Fred G. Martin
Professor
Thesis Supervisor

Certified by .....
Jeff Brown
Associate Professor
Thesis Reader

Accepted by .....
Jie Wang
Department Chair

Web-Based Model Slicing for 3D Printers

by

Michael U.B. Meding

Abstract of a thesis submitted to the faculty of the
Department of Computer Science
in partial fulfillment of the requirements
for the degree of
Master of Science
University of Massachusetts Lowell
2016

Thesis Supervisor: Fred G. Martin
Title: Professor

Thesis Reader: Jeff Brown
Title: Associate Professor

Abstract

3D printing currently has a large gap between software and hardware. A hobbyist machine can now be purchased for less than $500 but having good software to drive it is hard to find. Currently the only competent and free slicing software available is Cura and Repetier Host. Currently, Cura has varied support on all platforms, and Repetier Host is intended only for Windows. Neither software have any web support nor will they be likely to have any support in the future as the slicing process requires a computer with a considerable amount of both graphical and computational power. The purpose of this research is to construct a web based slicing software and make it simple for users without any prior knowledge of 3D printing to take full advantage of their printer and as a result will make 3D printing much more approachable for users who are not computer savvy. Additionally, this opens up opportunities for educators in STEM programs to teach students about 3D printing in a simple and practical way.

Acknowledgments

Contents

1 Introduction **1**

1.1 The RepRap Idea . . . 3

1.2 Sudden Growth Of 3D Printing . . . 3

1.3 Purpose of This Research . . . 3

1.4 Research Objectives . . . 4

1.4.1 See a model, get a model . . . 4

1.4.2 Wrap CuraEngine . . . 4

1.4.3 Design intuitive web interface . . . 4

1.5 Existing Technology . . . 5

1.6 Thesis Map . . . 5

2 Libraries & Existing Code **6**

2.1 CuraEngine . . . 6

2.2 Client Side Libraries . . . 6

2.2.1 Bootstrap . . . 7

2.2.2 AngularJS . . . 7

2.3 JavaEE . . . 8

2.4 OctoPrint . . . 8

2.5 G-Code Visualizer . . . 9

3 Methodology **10**

3.1 Research Design . . . 10

3.2 Working procedure . . . 10

3.2.1 Web Interface . . . 10
3.2.2 Slicing Engine . . . 12
3.2.3 Web Tool Path Viewer . . . 12
3.2.4 Final Steps . . . 12
3.3 Review and Usability Testing . . . 12

4 Client Side **13**
4.1 AngularJS, a bit of background . . . 13
4.1.1 Controllers & Data Binding . . . 13
4.1.2 Factories & Services . . . 14
4.1.3 Directives . . . 14
4.2 WebSlicer AngularJS Structure . . . 14
4.2.1 app.js . . . 14
4.2.2 index.html . . . 15
4.2.3 Settings . . . 16
4.2.4 Gcode Visualizer . . . 18
4.3 Key Challenges . . . 19
4.3.1 Visualizer Integration . . . 19
4.3.2 Interpolating Settings . . . 19
4.4 Other Planned Integrations . . . 20
4.4.1 OctoPrint . . . 20
4.4.2 Thingiverse & YouMagine . . . 20
4.5 Issues & Known Bugs . . . 20

5 Server Side **22**
5.1 JavaEE Structure . . . 22
5.2 ProcessBuilder . . . 22
5.3 REST API . . . 24
5.4 Key Challenges . . . 25
5.4.1 ProcessBuilder Deadlock . . . 25
5.4.2 FileTracker Revamp . . . 26

5.5 Future Improvements 28
5.6 Issues & Known Bugs 28

6 Discussion **29**

6.1 Usability Testing 29
6.2 Data Gathering 29
6.3 Design Updates & Improvements 29
6.4 Future Work 29

List of Figures

1-1 (a) High level view of the normal 3D printing process. (b) High level view of proposed new process using WebSlicer. 2

3-1 High level view of how WebSlicer functions and how users will interact with it 11

4-1 Full AngularJS structure breakdown 15

4-2 The flow of data through the application from beginning to end of client side user interaction 17

5-1 The structure of the server side of WebSlicer 23

5-2 Diagram of a deadlock issue that took weeks to resolve 26

List of Tables

5.1 Documentation of all exposed endpoints of my RESTful API 25

Listings

4.1 A sample from a static settings file in JSON format. 16
4.2 An example of a ng-repeat looping construct in HTML5. 16
5.1 An example of running CuraEngine C++ executable directly from the command line. 24
5.2 WebSlicer's underlying file structure supported by FileTracker. 27

Chapter 1

Introduction

3D printing, over the past few years, has become immensely popular in the home hobbyist space because of its new found availability. A hobbyist can now go and buy a do-it-yourself 3D printer kit for less than $500. Even though the hardware to build a 3D printer is easily available, the software support leaves much to be desired. Most of the big name companies that used to hold all of the patents for 3D printers have retained the software patents but not the hardware patents. This creates a gap in knowledge between building the printer and actually running it. This proposed research is to find out what software already exists in the open source world and then try to expose this on the web in a simple and easy to use manner. This effectively will democratize the world of 3D printing, much in the same way that Google has democratized the way that we search the web. In an article written by Harvard Business review they theorize that this rise in the popularity of 3D printing will spur an industrial revolution as manufacturing becomes more personalized and decentralized. D'Aveni (2015)

To print something on a 3D printer, there is a multi step process that can be daunting to many first time users. The first step in the process is to either create or download a model. Creating a model can be done with any standard 3D CAD software, such as AutoCAD or SolidWorks. Downloading pre-existing models from an online repository can be done from websites such as Thingiverse or YouMagine. Once

a model file is obtained, it is time to slice the model. Slicing is the act of taking this model file and splitting it up into many thin layers that the 3D printer can understand. This process can only be done by a dedicated slicing software which can often be complicated to use and difficult to install. Once the file has been sliced, the resulting file is a G-code file which is simply a set of movement instructions that the printer head must follow. This file is then loaded to an SD card or sent via a print server similar to the way that a normal 2D printer is networked. Once the G-code file has been loaded all that is left is to hit print either manually using the printers interface for the SD card or by hitting print on the network interface for the file that was uploaded.

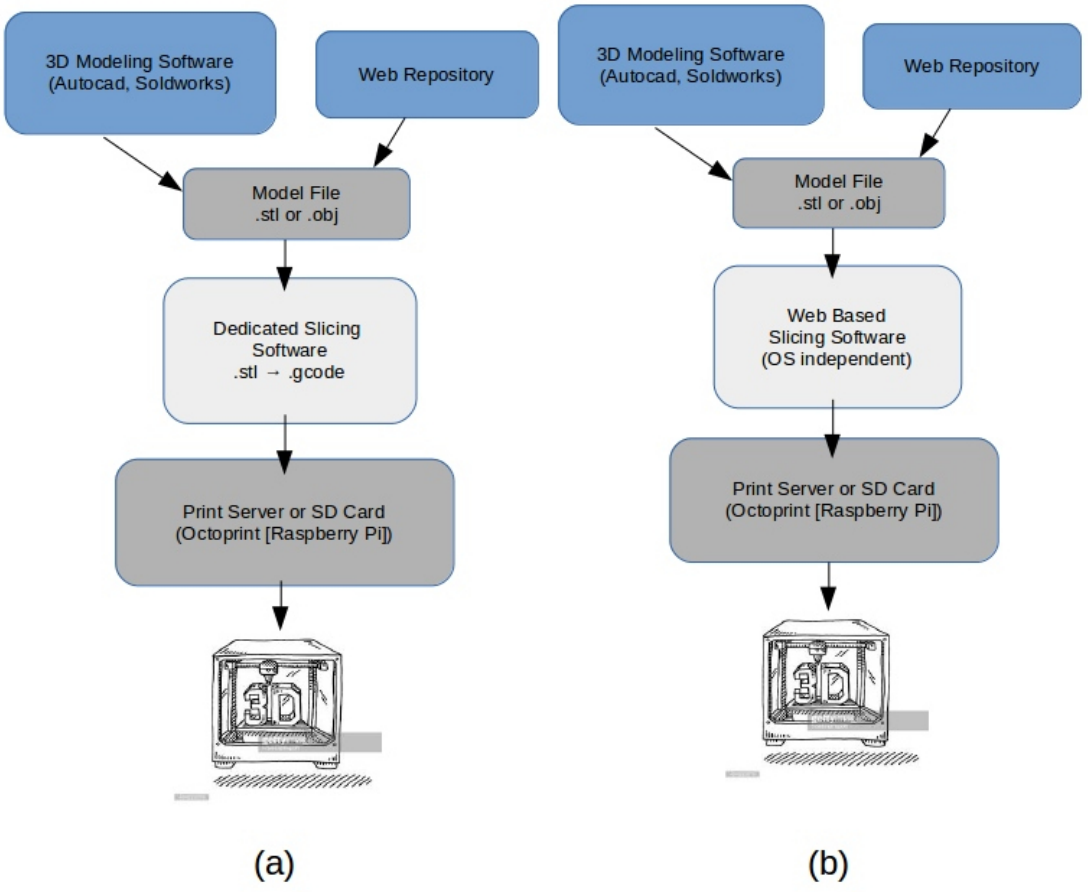

Figure 1-1: (a) High level view of the normal 3D printing process. (b) High level view of proposed new process using WebSlicer.

1.1 The RepRap Idea

The Replication Rapid-Prototyper Project (RepRap) is a movement with the goal of providing Open-source, diy 3D printers at low cost. RepRap printers are 3D printers with the additional ability to produce most of the parts necessary to assemble another identical printer. This idea also extends outside of hardware but to the software as well. Much of the software available for RepRap style printers are open source projects put together by the community.

1.2 Sudden Growth Of 3D Printing

In an article written by Forbes they did an analysis of the current market trend for 3D printing and were surprised to find that it is becoming one of the fastest growing emerging markets. "According to Wohlers Report 2014, the worldwide 3D printing industry is now expected to grow from $3.07 billion in revenue in 2013 to $12.8 billion by 2018, and exceed $21 billion in worldwide revenue by 2020". Columbus (2015) One reason that 3D printing has been reaching more people is the availability of the RepRap designs. Additionally, to build a 3D printer from a kit only requires basic hand tools and basic electronics skills which means that it has been opened up to a much broader audience.

1.3 Purpose of This Research

The purpose of this research is to construct a web based slicer and make it simple for a user, who knows hardly anything about 3D printing, to slice models and run their 3D printer. This will make 3D printing much more approachable for occasional users who are not prepared to spend thousands of dollars on a professional machine and software. Additionally, this opens up opportunities for educators in STEM programs to teach students about 3D printing in a simple and practical way. Under normal circumstances this would not be a feasible project for a year long Masters thesis. However, many of the technologies required to complete this project exist in varying

states of completeness. Putting them together will be the subject of this study.

1.4 Research Objectives

This research has several milestones which must be met for this thesis to be considered complete. These milestones are in no particular order. However, several of them must be completed before allowing other milestones to be completed as detailed below.

1.4.1 See a model, get a model

Getting a model directly from a repository, such as Thingiverse, would be ideal for this web interface because it would not require any local storage for a model file. A file stored in local storage could be quite large and cause memory problems for the user.

1.4.2 Wrap CuraEngine

CuraEngine is an open-source slicing engine designed to take model files in .stl file format and convert them into G-code for 3D printing. For this project, wrapping this code and making it callable from the web would be the heart of this application.

1.4.3 Design intuitive web interface

This project requires both an intuitive and easy to way to slice a model file for 3D printing. This would be done using Bootstrap and AngularJS because they are both scalable and flexible for almost any web design. These technologies also allow for small scale and mobile use.

1.5 Existing Technology

AstroPrint is an all included cloud operating system for 3D printers. It attempts to encompass the entire package of 3D printing to a dedicated Raspberry Pi, or similar computer system. It is also one of the only forays I have found into a cloud based slicing software. Unfortunately, their software tries to accomplish too many tasks at once and has become somewhat like a Swiss army knife as it is capable of many tasks but can only do a few tasks well. Additionally, their cloud based slicing software, while being reasonably fast, lacks any support for reviewing the sliced model. This review stage is critical for anyone who is printing something that will take more than a few hours to complete.

1.6 Thesis Map

- Chapter 2, Existing libraries and why I chose them.
- Chapter 3, Software architecture.
- Chapter 4, Client side in detail.
- Chapter 5, Server side in detail.
- Chapter 6, Discussion about usability testing and further improvements.

Chapter 2

Libraries & Existing Code

2.1 CuraEngine

CuraEngine is the back end of a larger application called Cura. Cura is an open source 3D print slicer designed by Ultimaker and is part of their software suite for their Ultimaker line of 3D printers. It will be talked about extensively in latter portions of this paper as it is the main portion of the back end of WebSlicer.

The reason for choosing to use CuraEngine as my main slicing engine is that it has the most clear separation between application and interface. Another option was to use Slicer which is much older and has better documentation but is unfortunately written on windows and would require a large amount of extra effort to get working properly for my needs. CuraEngine is also platform agnostic as it is written in C++ and uses one library called protobuf which is its main interface library for the Cura application.

2.2 Client Side Libraries

When structuring a website for optimal layout and scalability there are few better options than using the combination of Bootstrap and AngularJS. Bootstrap is a client side CSS library which includes most commonly used CSS options such as

buttons and input fields and styles them all accordingly. AngularJS creates a client side environment to support higher level language constructs and features that are normally reserved for complex server side applications.

2.2.1 Bootstrap

Bootstrap is one of the most popular CSS and JS frameworks for developing responsive and mobile projects on the web. Being responsive means that Bootstrap is capable of being dynamically displayed on many different size screens. When the screen size changes Bootstrap is capable of moving and resizing elements on the page without loosing any content or overlapping items. In addition it speeds up the process of developing web based applications as it removes the need to write heavy amounts of CSS to make an application look and preform nicely. Bootstrap also includes many standard features which most web developers use every day further simplifying the process of building a website.

2.2.2 AngularJS

AngularJS is a framework which allows for easy development of dynamic web pages. It contains many ways to logically separate your code similar to popular object oriented programming languages like Java or C++. **?** states, "The goal of AngularJS is to bring the tools and capabilities that have been available only for server-side development to the web client and, in doing so, make it easier to develop, test, and maintain rich and complex web applications" (p. 48). Therefore, the combination of Bootstrap for the unified look, responsiveness, and mobile support combined with the power of AngularJS as a framework make it the easy choice for a web based application such as WebSlicer.

2.3 JavaEE

JavaEE is a layer built on top of JavaSE the standard Java development environment. This additional layer provides many important architectural interfaces such as being able to put code into EJB containers or web containers for deployment of large applications which may have many of such containers. Pilgrim (2013) JavaEE also provides frameworks for working with RESTful web services which streamlines the creation of a useful application API.

There are many server side frameworks which have support for RESTful web services but most lack any kind of real scalability like that provided in JavaEE. Furthermore, JavaEE provides the ability to easily distribute your computing as more resources are needed. As this research may eventually scale to supporting many users it was logical to choose a framework which allowed for this kind of growth. Another reason for choosing JavaEE as a framework is that it provides the full platform cross compatibility which is standard in Java. Allowing for cross platform compatability would make it simple to link many computers that are running on different operating systems together further simplifying the prospect of scaling.

To run a JavaEE based application it must run in a web capable application container. This container must be placed on an application server which exposes it to the web under some context. For WebSlicer the application server WildFly was chosen for this task as it is well known for its reliability.

2.4 OctoPrint

OctoPrint is a small server that connects a Raspberry Pi to a 3D printer and serves printer info and files to the printer remotely. The Raspberry Pi serves a small webpage which allows you to easily keep track of the temperature of the printer and the current print progress. This interface allows for a web based connection between your remote g-code files and the local printer. Horvath (2014) OctoPrint also contains its own API

which makes for easy integration into other web applications.

The fact that OctoPrint is so easily integrated into other web based applications made it a logical choice for integration with WebSlicer. Integrating with OctoPrint would allow for one touch printing by allowing me to send g-code files directly to a connected OctoPrint server. The only interaction that is required of the user is to connect their OctoPrint server which is done by simply indicating the address at which the server can be reached at. Additionally, users are required to include an API key to allow for other applications to directly access the servers API.

2.5 G-Code Visualizer

When building the g-code visualizer it seemed logical to find one which already existed and worked well to include in my project. I came across an open source project called gCodeViewer originally written by Nils Hitze which would work perfectly for my needs. Hitze (2015) This project was written in pure JavaScript and would be seemingly easy to integrate with WebSlicer.

Unfortunately this could not have been farther from the truth as many of the libraries that were needed to make this code needed to be updated. Additionally, I wished to integrate this code into my existing AngularJS framework which turned out to be much more challenging than anticipated. This meant that what remained from the existing open source project was very small when all was said and done.

Chapter 3

Methodology

3.1 Research Design

This thesis is a mixture of both research and design implementation. The research portion of this project focused on linking an existing C++ application (CuraEngine) into a larger JavaEE based project. In this research we also had a small group of people run though some beta testing of the application and logged their observations.

3.2 Working procedure

As shown in Figure 3-1, the application will have 3 major components that all need to work together in a cycle until the user decides that the output is what they desire.

3.2.1 Web Interface

I anticipate that it will include a set of forms for collecting the users settings for their printer and account tracking info so that they may retain certain settings. Additionally, I would like to include a gcode editor which will allow users to edit specific portions of their gcode and see the resulting output in the viewer. This does not include the actual slicing engine which must be driven and accessed independently.

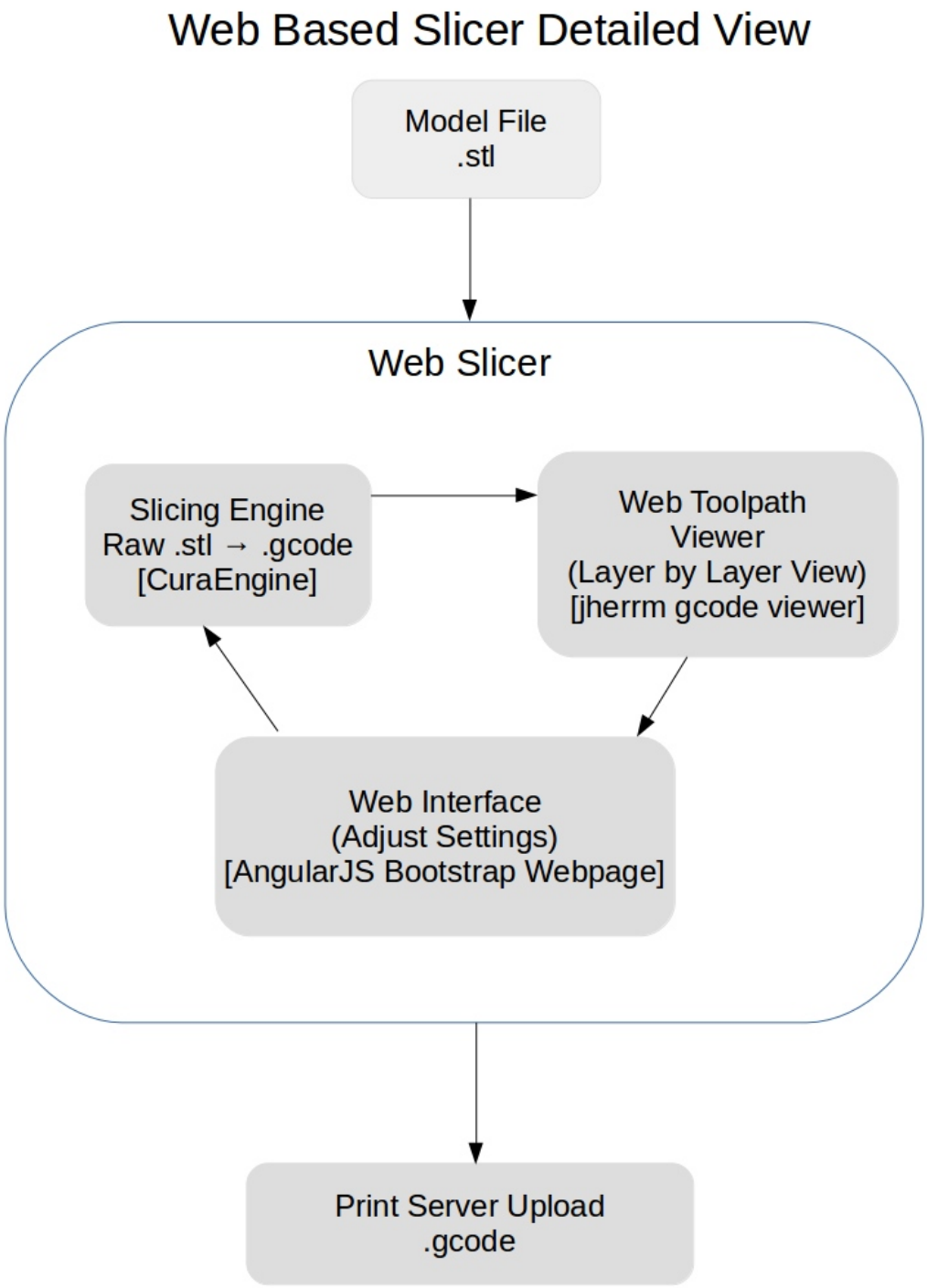

Figure 3-1: High level view of how WebSlicer functions and how users will interact with it

3.2.2 Slicing Engine

The slicing engine is at the heart of this project. It will include taking uploaded .stl model files by the user and convert them into raw G-code. The engine that carries out the raw geometry calculations is CuraEngine and is written in pure C++, (it is not web friendly). Thus, this portion of the project will require deploying CuraEngine at a cloud provider and then creating a RESTful API to interface with it.

3.2.3 Web Tool Path Viewer

After configuring and generating the G-code representation for a 3D model, there still must be a way to review visually how the slicing engine actually will split up the model.

3.2.4 Final Steps

The final step in the process of this application will be creating links to the finished files where the user can either choose to download the completed G-code file. If the user has access to a print server such as OctoPrint, they may opt to copy/paste the link to it where it will be uploaded automatically for printing using the OctoPrint API.

3.3 Review and Usability Testing

After running through all steps of the working procedure it was then necessary to test WebSlicer by running a small beta test. This beta test consisted of a small group of users with varying familiarity with 3D printing to test how easy WebSlicer is to use through a series of simple tasks.

Chapter 4

Client Side

4.1 AngularJS, a bit of background

To understand the structure of WebSlicer a few things must be known about how AngularJS applications are structured and a bit about how AngularJS itself works. **?**, has more detailed explanations of each angular componenet than those that follow should the reader need more detail than that provided.

4.1.1 Controllers & Data Binding

The controller structure in AngularJS sits in between the view and the JavaScript world acts like the glue between the two. From a controller you are allowed access to "scope" variables which are like normal JavaScript variables but have a special two way data binding property. Two way data binding is one of the most interesting features of AngularJS as it gives the user real time updates as the user interacts with a view. In traditional JavaScript a developer has to listen for key events that occur in the browser as the trigger for a sequence of events. Angular on the other hand also has a sequence of events but no trigger is required as the sequence of events is triggered automatically. This action is known as a digest cycle. The secret to this is that AngularJS has a watch for each variable attached to its scope and when any changes to this variable occur they propagate that change to the functions that are

associated with that variable.

4.1.2 Factories & Services

Plain JavaScript has many pitfalls but one of the most frustrating has to be the lack of any kind of larger design pattern support such as OOP (Object Oriented Programming). The current trend with web based applications is to make single page applications with the same functionality as prior designs with many pages. AngularJS is the answer to this with Factories and Services which mimic the design of a POJO (Plain Old Java Object) and Singleton objects in Java.

4.1.3 Directives

A directive is one of the most powerful structures in AngularJS. It allows the programmer the power to write their own HTML5 tag with its own parameters and rules. Directives also contain support for templates which are just short HTML snippets that replace the main tag when the code is loaded. Combine this with the use of the aforementioned design patterns and it creates a very powerful tool to create and organize dynamic content.

4.2 WebSlicer AngularJS Structure

4.2.1 app.js

The full graphical AngularJS structure of WebSlicer is shown in Figure 4-1. Flow through this diagram starts with the app.js node which represents the main controller of the application. This can be thought of as a main function in C++. The main control variables are also inside of app.js which are similar to global variables. Variables in this controller are used to store the current settings, file pointer, and output gcode.

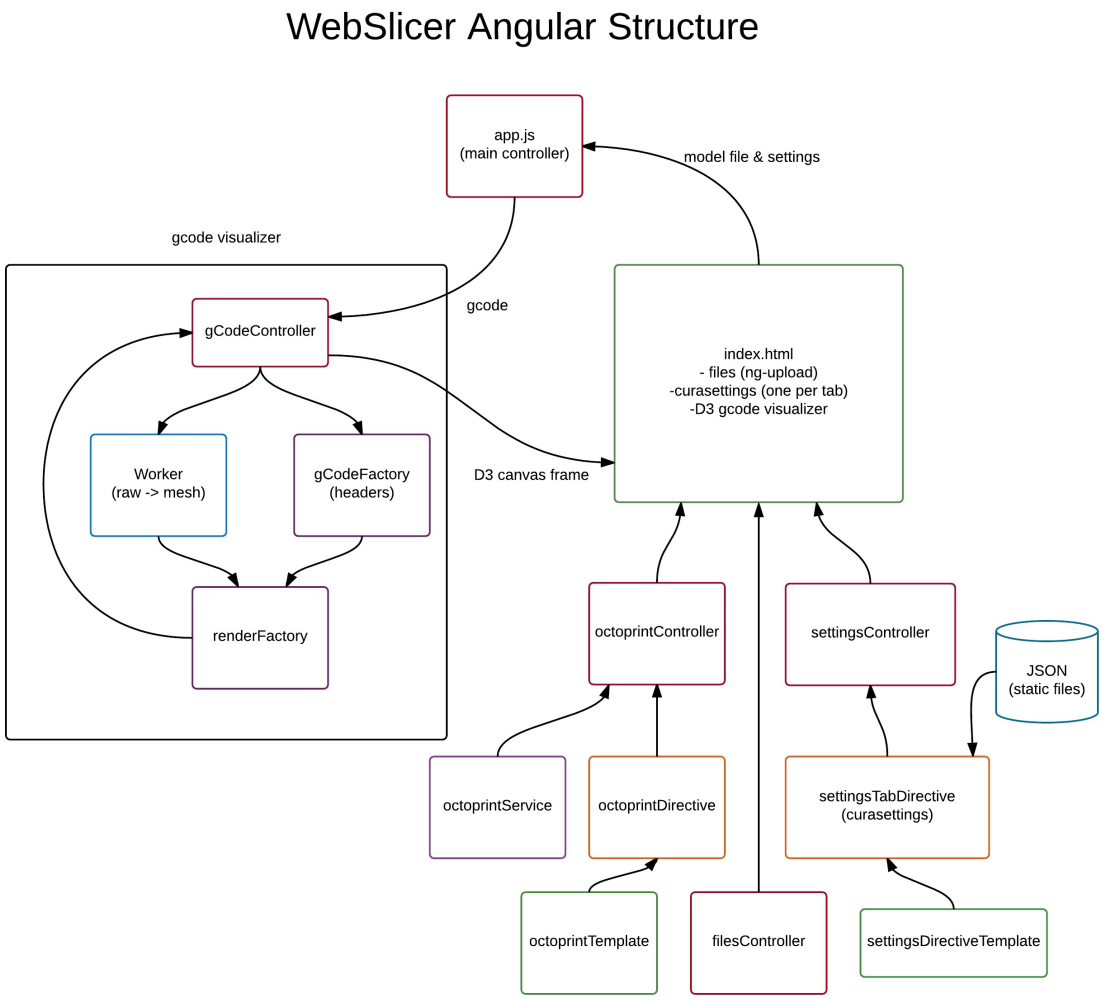

Figure 4-1: Full AngularJS structure breakdown

4.2.2 index.html

Figure 4-1 also shows that index.html is large hub and as WebSlicer is a single page web application this is the only static HTML file. Index.html has several other functions

such as bringing in all libraries and including the custom directives.

4.2.3 Settings

As shown in Figure 4-2, settings have a long path that they must travel behind the scenes before they are submitted to be used while slicing. This data flow starts with loading a static JSON (JavaScript Object Notation) file which describes the settings in a pattern as shown in Listing 4.1.

Listing 4.1: A sample from a static settings file in JSON format.

```
{
    "setting": "layer_height",
    "default": 0.1,
    "type": "float",
    "category": "Quality",
    "label": "Layer Height (mm)",
    "description": "Layer height in millimeters. This
       is the most important setting to determine the
       quality of your print. Normal quality prints are
        0.1mm, high quality is 0.06mm. You can go up to
        0.25mm."
}
```

Listing 4.2: An example of a ng-repeat looping construct in HTML5.

```
<table class="table">
  <tr><td>Action</td><td>Done</td></tr>
  <tr ng-repeat="item in todos">
    <td>{{item.action}}</td>
    <td>{{item.done}}</td>
  </tr>
</table>
```

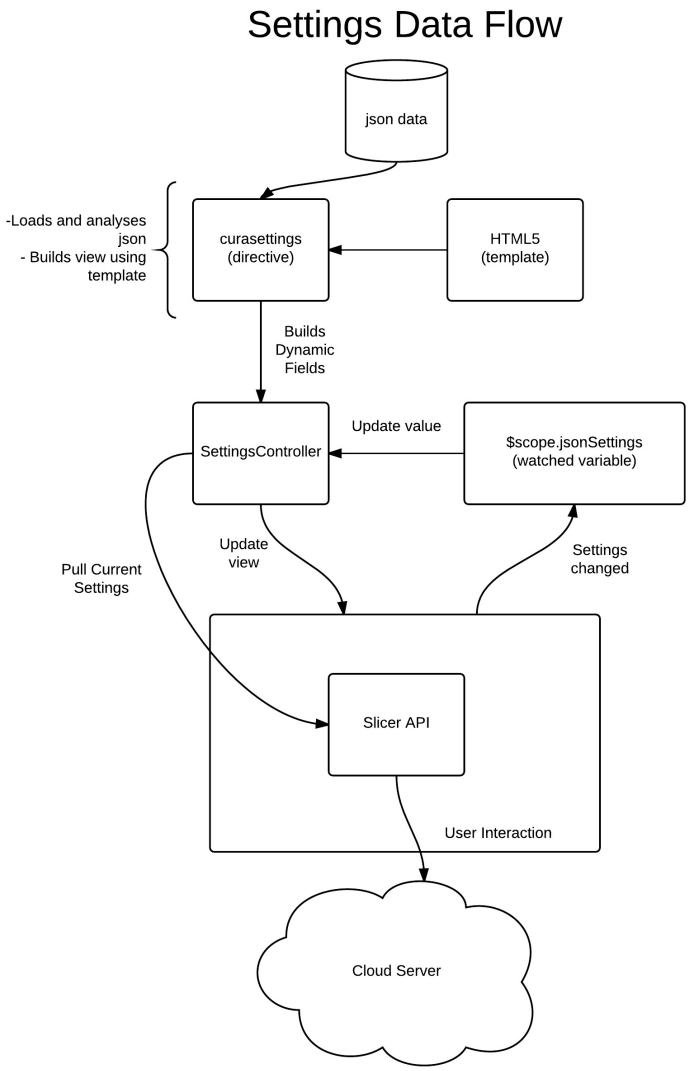

Figure 4-2: The flow of data through the application from beginning to end of client side user interaction

A directive called curasettings takes this static JSON file and splits it up so that for each setting object an input field exists in the template. AngularJS provides this functionality through the use of an ng-repeat which is written in a similar fashion to that of a for loop in Python which is shown in Listing 4.2. Using a template for a directive in this case also meant that I could have control over many different kinds of fields such as drop downs and number specific inputs. The curaengine directive also gave me the logical separation of one static JSON file per tab of settings in the UI which made it easy to find and modify the settings as needed.

The interesting part about how settings work in WebSlicer is how they are tracked after being loaded to the client. In most applications such as this there would be an individual watch on each field or a submit button which would trigger grabbing all the fields. WebSlicer however uses a single object to track all of the settings for the application and uses a dynamic method of AngularJS to map from the input field to a field of a single object. Thus when the settings are submitted there is no more interpolation needed as the settings object already has the current state as shown in the Figure 4-2 as "Pull Current Settings".

4.2.4 Gcode Visualizer

The gcode visualizer for WebSlicer is written using a combination of D3 and a JavaScript web worker. From Figure 4-1 it can be seen that the controller for the visualizer is sent a gcode file which it splits up to two separate services. The controller itself does an initial parse of the file which places each one of the lines into its own entry in an array before passing it off to the gCodeFactory and Worker. Both of these files parse through this entire array but do so at roughly the same time to speed along the process of visualizing. The worker takes the array of lines and ignores the header to just focus on converting the raw movement commands into D3 lines so that they may be rendered. The gCodeFactory takes the headers from the array and uses them to do analytics of the gcode file such as total print time.

The final steps in the process of visualizing the gcode require splitting the gcode file into layers for rendering. A task which is made easy by simply following the meta tags in the gcode that signal a layer change and marking them in our array. Once all of the heavyweight tasks of parsing and splitting up the gcode file are done it is simply a matter of returning layer requests with canvas frames. Each time a layer is requested a progress for that layer is also sent. The controller then just goes to the layer height in the array and renders a frame with the number of lines that are described by the current progress. As array indexing is nearly instantaneous the visualizer once parsed and loaded runs very quickly to display layers.

4.3 Key Challenges

4.3.1 Visualizer Integration

By far one of the most difficult challenges overcome while writing WebSlicer was the visualizer integration. The starter code for this was originally written by Nils Hitze as an open source project which had many other features Hitze (2015). This code however required a lot of work to integrate properly with the rest of the application. AngularJS despite its many features does not mingle well with other projects. When all was said and done the only code which remained from the original was the JavaScript web worker and some of the parser code.

4.3.2 Interpolating Settings

Another difficult task when building this application was designing a method to handle a lot of input fields. It would have been simple to just create a series of fields each with their own variable and submit methods and triggers but when settings change format and type it would have been an unmanageable mess very quickly. So spending the extra time to design an intelligent method of handing large amounts of input data seemed logical. This however ended up taking much more time and effort than anticipated. At one point this even required getting in touch with the

original developers of CuraEngine to ask them about what some of their settings meant. Documentation for many of the settings was quite bad and in many instances was non existent which further slowed down development.

4.4 Other Planned Integrations

4.4.1 OctoPrint

A feature which was removed at a late state in the process of building this application was an integration with OctoPrint. OctoPrint has a good API to allow for external applications to integrate easily with it making it an ideal choice for this application. The idea of this integration was to allow a user who was running an OctoPrint server to be able to send files directly to their server. This would eliminate the need of having to download the gcode from WebSlicer only to be uploaded to the print server seconds later. It was decided at the last moment that this feature need not be in the minimum viable product and that time was best allocated to finishing more crucial features of the application.

4.4.2 Thingiverse & YouMagine

Another planned integration was the ability to import from a web based repository such as Thingiverse or YouMagine. These repositories are public sites where users can upload their 3D designs so that others can 3D print them. Thingiverse in particular has a nice API for grabbing models from their site which would make it an easy integration for a web based slicing software. However, this feature was given the axe early on as it would have required too much unnecessary development time to finish.

4.5 Issues & Known Bugs

As mentioned in prior sections WebSlicer was designed with a minimum viable product in mind. Developing a working 3D print slicer for the web was the primary

task and all other features needed to support this or extend this functionality. For this reason there is no login or user database which would normally be the first item to be developed for an application such as this. There is also no way to view any of the models in 3D which for most users makes the software significantly harder to use.

Chapter 5

Server Side

5.1 JavaEE Structure

The server side of WebSlicer was written in JavaEE, the structure for which is shown in Figure 5-1. JavaEE was the optimal choice for this application as it allowed for the easiest deployment and was also the easiest to scale. Additionally, JavaEE has a good code packaging mechanism for web and non web based applications alike. The web container which is in use for this application exposes a RESTful API on a privately hosted server.

To further simplify the development process Maven was also used. Maven is a build tool for Java and has support for deploying complex applications such as those in JavaEE. This means that when a build was completed it was automatically deployed and ready for testing.

5.2 ProcessBuilder

At the core of the server side application is an executable called CuraEngine. It is the main executable which is compiled from the open source slicing platform Cura which is written in C++. This presented a problem as all of my server side code is written in Java. ProcessBuilder was the solution to this problem as it is capable

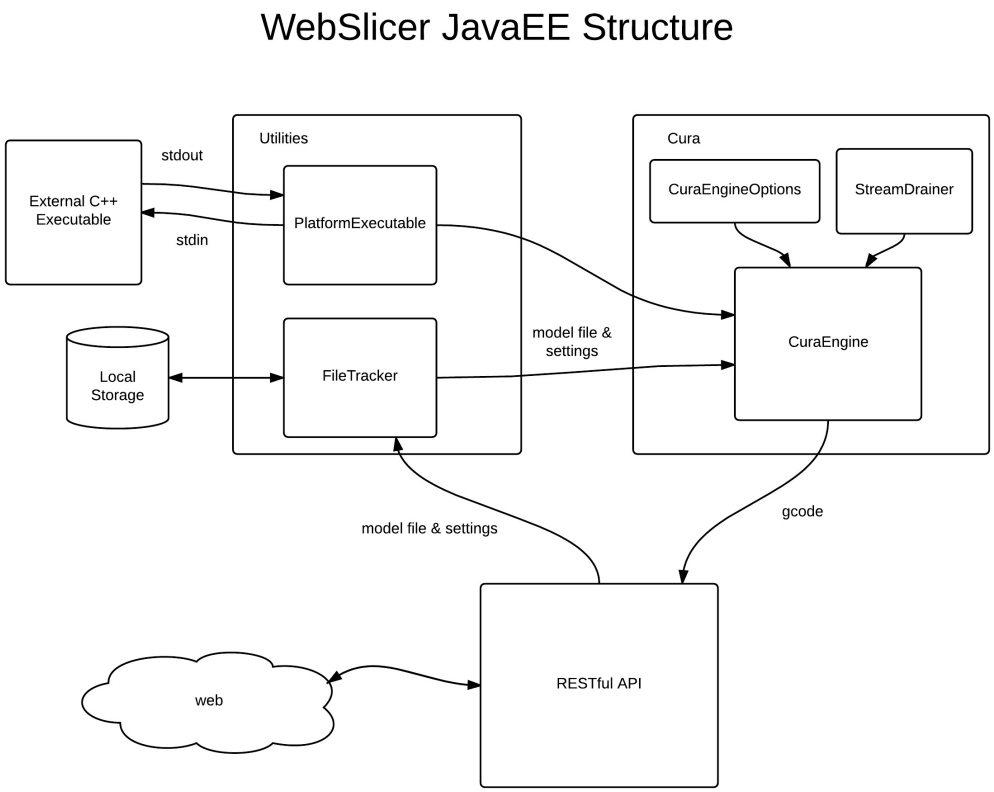

Figure 5-1: The structure of the server side of WebSlicer

of redirecting the input and output streams of a local executable process into my Java server application. CuraEngine from Figure 5-1 uses a ProcessBuilder and the

PlatformExecutable to create a runnable Java method that is capable of executing like a C++ executable. CuraOptions feeds the CuraEngine class with all of the parameters that it needs from the API. It gathers the path to the appropriate settings file and includes all of the parameters needed to run the CuraEngine executable.

Listing 5.1: An example of running CuraEngine C++ executable directly from the command line.

```
1 CuraEngine slice -v -j {settings.json} -g -e -o {
    output.gcode} -l {model-file.stl}
```

An example of running the CuraEngine C++ executable from the command line is shown in Listing 5.1. When the ProcessBuilder class of WebSlicer recieves a slice command from the API it gathers the arguments listed in brackets and sends them to PlatformExecutable. PlatformExecutable then spawns a native process and pipes its input and output streams into the respective Java streams. At the same time StreamDrainer spawns a new thread and waits for the output stream that was created by PlatformExecutable. StreamDrainer's task is to take the unneeded output from stdout and pipe it into a log file for debugging.

After CuraEngine has finished slicing the current file and PlatformExecutable has returned the REST API, which has been waiting, unblocks and starts reading the output gcode file. This file is then packaged and sent back to the client as the response of the "/slice/{clientId}/{modelId}" command as shown in Table 5.1.

5.3 REST API

Type	Address	Description
GET	/ping	A simple ping endpoint used for testing.
POST	/setupClient	Sets aside all needed files for a new client and return its unique ID.
POST	/importStl/{clientId}	Takes a MIME type file stream and imports the file to the clientId specificed in the URL. It also returns a unique identifier for the file.
POST	/importSettings/{clientId}	Similar to importStl this endpoint takes a settings JSON file and imports it to the specificed clientId
POST	/slice/{clientId}/{modelId}	This is the main slice function of the API. It combines all of the parameters specified by the calls before and returns a gcode file to the user.
POST	/testSlice	A test endpoint that requires no parameters and simply returns some arbitary gcode to the user.
GET	/getFiles/{clientId}	Returns all the model file names and their tracking ID's that are associated with a clientId.

Table 5.1: Documentation of all exposed endpoints of my RESTful API

5.4 Key Challenges

5.4.1 ProcessBuilder Deadlock

One of the biggest bugs encountered while developing this project was a thread deadlock issue. The server side code uses javas ProcessBuilder which builds a system native call to an executable and then pipes the input and output into the input and output pipes of javas stdio as shown in Figure 5-2. This works very well for small platform executables with limited I/O but can become problematic when complex native calls such as the curaengine are used.

ProcessBuilder does its normal writes to stdout and the drainer just pipes them into a file. However the drainer has to wait for a file pointer using the fp.available() function. This is a non blocking function which only estimates the buffer size that it has for the file. The check for file pointer availablity was checking if this function

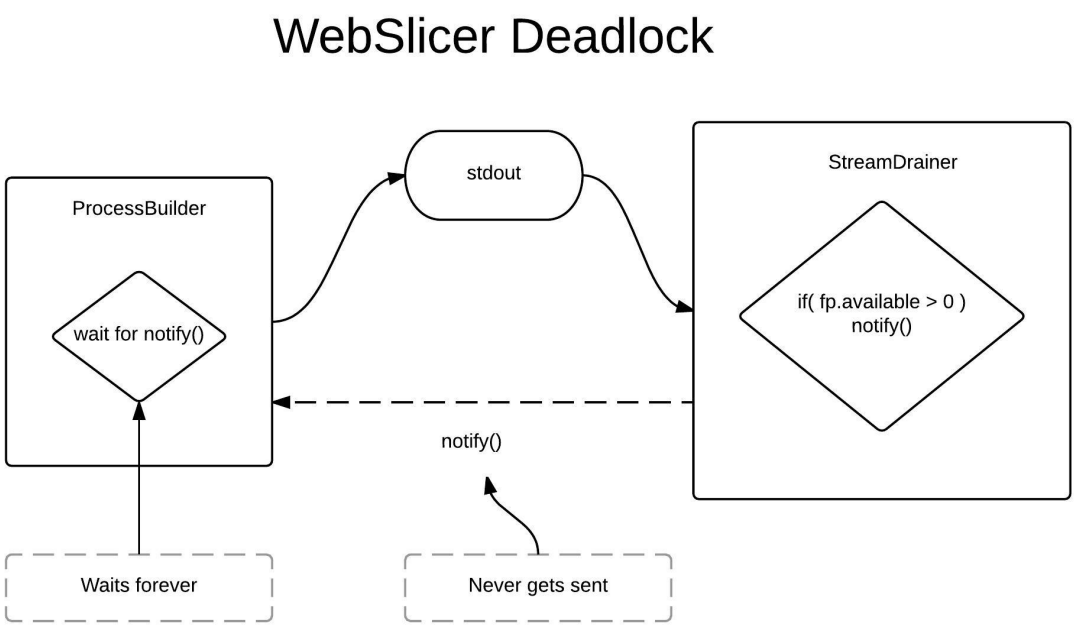

Figure 5-2: Diagram of a deadlock issue that took weeks to resolve

returned something greater than 0 as an estimate before notifying the ProcessBuilder that it was ready. However the buffer size would often start as zero before allocation and as this check was not part of a loop it would stay stuck forever as the notify was missed.

This problem was solved by using the correct blocking file pointer available check. Occasionally the buffer size was larger than 0 and the application ran fine but with some models it would consistently fail as the buffer had not been allocated yet. This solution is seemingly obvious yet this solution still took many days to find and correct as the application did not fail consistently.

5.4.2 FileTracker Revamp

The first iteration of FileTracker was a bit crude and not well planned out. It tracked two hashmaps, one for model files and the other for settings files with no mind for the client who actually needed to have access to those files. This worked fine for

testing but had many pitfalls including the inablility to reuse files that already existed. As soon as the client died those files were lost which is a major inefficiency.

The fdmprinter.json file within the unique client folder is symbolically linked to the fdmprinter.json file within common. The CuraEngine executable requires that all of the settings files rest within the same directory when preforming a slice. This is somewhat problematic with the potential of having this file copied for many clients. Thus, symbolically linking the file to the rescue.

The output.gcode and settings.json files are dynamically overwritten for every iteration so their existence here is merely to please CuraEngine as it requires these files as arguments when preforming a slice. The user has no grasp of these files and is only able to access their content through the web interface which parses in and out of files.

Listing 5.2: WebSlicer's underlying file structure supported by FileTracker.

```
1  webslicer/
2  - b1a2a69e-5893-4d7c-aa1f-d639fa3b4ed1/
3    - fdmprinter.json -> /tmp/webslicer/common/
       fdmprinter.json
4    - models/
5      - balanced_die_version_2.stl
6      - raldrich_planetary.stl
7    - output.gcode
8    - settings.json
9  - common/
10   - fdmprinter.json
11   - presets/
12     - prusa_i3.json
13     - ultimaker2.json
```

5.5 Future Improvements

Currently FileTracker does not take advantage of the presets within the common/presets/ folder as described by Listing 5.2. These files contain the default settings for the corresponding printer which right now are only the ultimaker2 and a basic configuration of a prusa i3 variant. Optimally, the user would select from one of these starting presets and then modify and save their own. This would allow users an optimal starting point and lowering the amount of starting knowledge and increasing the usability of WebSlicer.

This new file structure also allowed for an easy client index. In the future the unique folder ID will become the client's identification number which will be tied to their login. Additionally, simplifying the login process with googles oauth 2.0 system was also planned.

5.6 Issues & Known Bugs

Currently there is no way for the server to import existing user files into its structure. This means that when the server is restarted for any reason that the entire supporting file structure with all user files is lost. Resolving this is just a matter of writing an initial import function that indexes all of the existing files. It was left out of the initial version due to time constraints.

Chapter 6

Discussion

6.1 Usability Testing

6.2 Data Gathering

6.3 Design Updates & Improvements

6.4 Future Work

References

L. Columbus. "2015 Roundup Of 3D Printing Market Forecasts And Estimates." *Forbes*, 2015.

R. D'Aveni. "The 3-D Printing Revolution." *Harvard Business Review*, 2015.

N. Hitze. "gCodeViewer." January 2015.

J. Horvath. *Mastering 3D Printing*. Apress, 2014. doi: 10.1007/978-1-4842-0025-4_6.

P. A. Pilgrim. *Java EE 7 Developer Handbook*. Packt Publishing, 2013.