CSM Hildreth: Box Designer

Final Report

June 10, 2020

Advisor: Donna Bodeau
Client: Dr. Owen Hildreth, Colorado School of Mines Professor
Team: Grace Clark, Lucas Emerson, Charles Gougenheim

Table of Contents

Introduction

The client, Professor Owen Hildreth, is an Assistant Professor in the Department of Mechanical Engineering at Colorado School of Mines. He oversees the Hildreth Research Group which focuses on additive manufacturing phenomena at the nanometer- to centimeter-scale. Professor Hildreth has a variety of small and delicate equipment that requires special storage. He also owns and operates a Glowforge Laser Cutter which he uses to cut out custom-designed boxes for this equipment.

Designing these custom boxes can be difficult. Some applications are currently available and commonly used for designing laser-cut box designs. These include full-fledged CAD packages such as Fusion360, web applications such as Makercase, and a variety of github projects, such as project laser-cutter (for links to these applications and projects, see Appendix B). These projects each carry their own disadvantages. Complex Cad packages are great for high level projects, but require more effort than is necessary to create box templates. Many github projects are challenging to work with; many github programs require comfort with command line code or lack a graphical user interface (GUI). Makercase software resolves some of the previously discussed issues. It has a simple GUI and does not require the user to understand code. However, Makercase falls short because the user lacks flexibility in increasing the complexity of the box and lacks the ability to save boxes for future editing.

The goal of this project was to design a simple GUI-based application tailored to drafting box templates for a laser-cutter. This application incorporates a GUI to provide real-time previews of the box model and provides saving capabilities. The user can simply run the program and input values without any understanding of code. For developers, the code structure is designed to allow ease in updating or extending the features to meet future needs.

Requirements

I. Functional Requirements

GUI:

-display box
-user can zoom in/out
-user can rotate the box
-user can pan the box
-display potential tools for manipulating the box
-change box model and box display when tools are used

BOX MANIPULATION

-user can specify overall box dimensions in either outer dimensions or inner dimensions
-user can specify wall thickness
-user can specify locations for added features
-user can add and delete sides of the box
-user can add additional interior walls
-user can add, modify, and delete kerf (i.e., notches)
-user can change the width of the tabs used for wall joins

TOOLBAR:

-user can save a box model
-user can open a previous box model to continue editing
-user can save a model as a pdf for input to laser cutter

II. Stretch Goals

-user can select default orthographic, isometric views in GUI
-user can drag features with snapping options for common snaps
-user can add a tabbed lid
-user can add a sliding lid
-user can add, modify, and delete cut outs for holes
-user can add, modify, and delete text
-user can distinguish between cutting and engraving
-user can switch between inches and millimeters
-user can change the margin (distance between edge of cutting bed and first cut) on the PDF
-user can change the padding (distance between individual pieces) on the PDF
-user can add corner compensation on the PDF
-PDF is efficiently laid out by default

III. Non-functional Requirements

-program must be suitable for macOSX use
-program must be completed in Swift
-program must be reasonably responsive to user input
-program must be easy-to-use, yet powerful

System Architecture

I. UML Diagrams

The system architecture plan went through several iterations as the team gained better understanding of design patterns, structures, and architectures particular to systems that rely heavily on GUIs. As time progressed, a better understanding of existing Swift libraries aided in creation, manipulation, and display of the box model. The final architecture follows *Figure 1*.

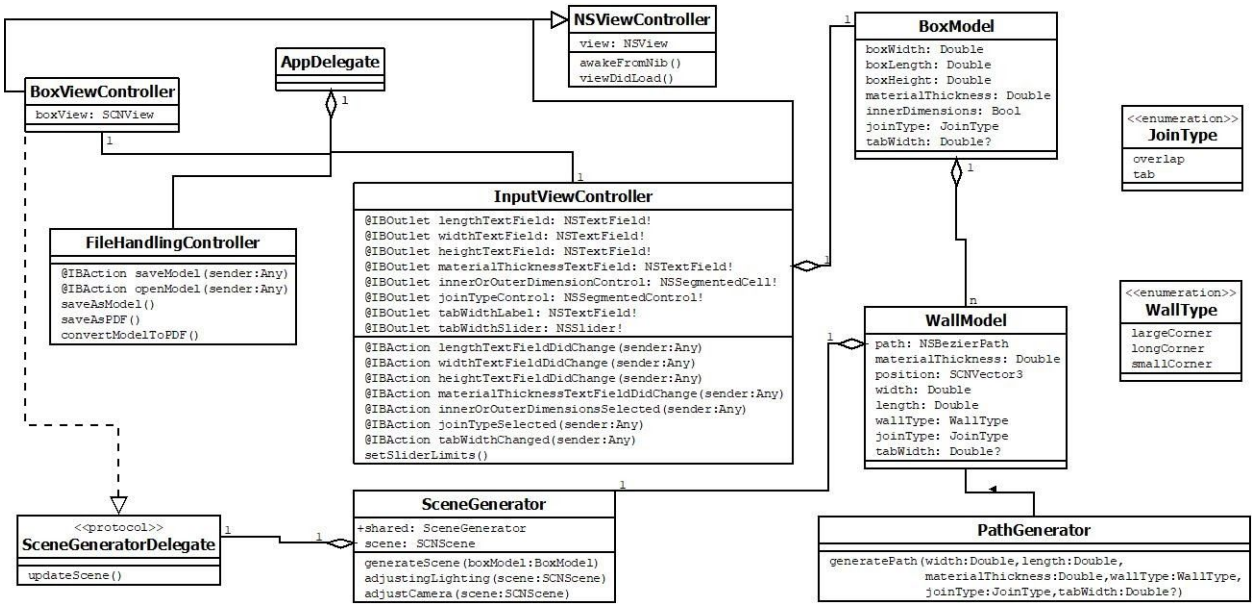

Figure 1: an overview of the system architecture

Overall system architecture can be broken down into a series of parts. Typical of iOS and macOS application development are the AppDelegate class and ViewController classes. The AppDelegate is provided by XCode and is used to control the main application states. It also contains the ViewControllers, children of NSViewController, which are used to display distinct views within the application and control data flow. The application includes three ViewControllers, seen in *Figure 2*. The BoxViewController contains the scene which displays the box model. The InputViewController contains the text fields, buttons, and sliders which allow manipulation of the box model. The FileHandlingController responds to requests to save and open files and provides file system panels for these purposes.

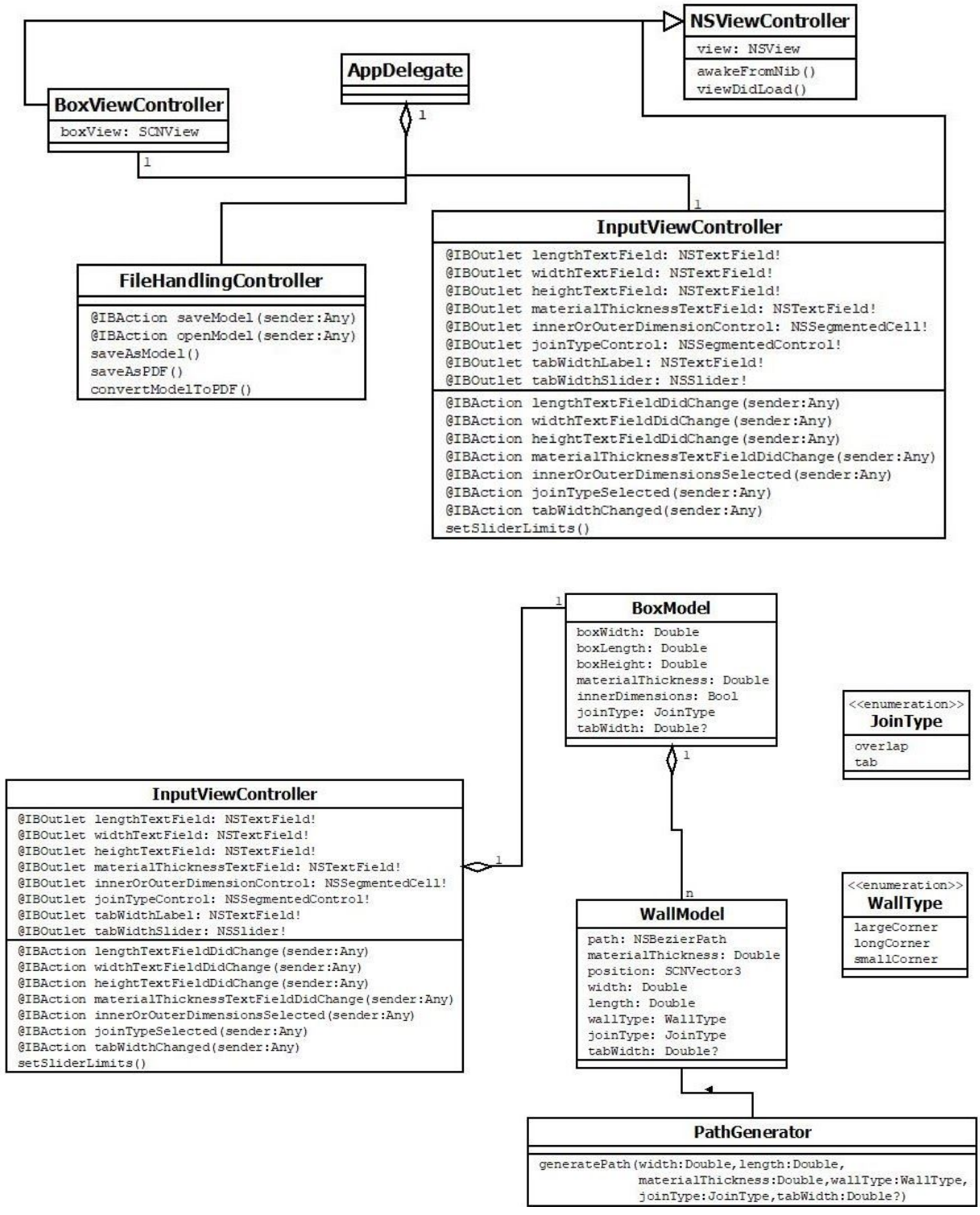

Figure 2 (above): relationship between AppDelegate and ViewControllers
Figure 3 (below): box model and its underlying wall model. Changes caught by InputViewController eventually cause PathGenerator.generatePath() to be called.

As seen in *Figure 3*, the InputViewController controls changes to the box model. The InputViewController receives a notification of any actions that take place with respect to the various input methods it contains. Upon receiving these notifications, it sends the relevant changes to BoxModel. BoxModel then sends the relevant changes to its individual WallModels, and each WallModel calls PathGenerator.generatePath() to update their path variable.

After BoxModel has been updated, the scene displayed in BoxViewController must be updated. SceneGenerator is a singleton class that BoxViewController and BoxModel both have a copy of. BoxViewController is also a delegate of SceneGenerator, seen in *Figure 4*, allowing it to respond to internal changes of the SceneGenerator class. When the box model finishes updating, it notifies SceneGenerator to update its scene to correspond to the new model. After this scene is updated, BoxViewController responds to the change and updates its own scene to this new scene.

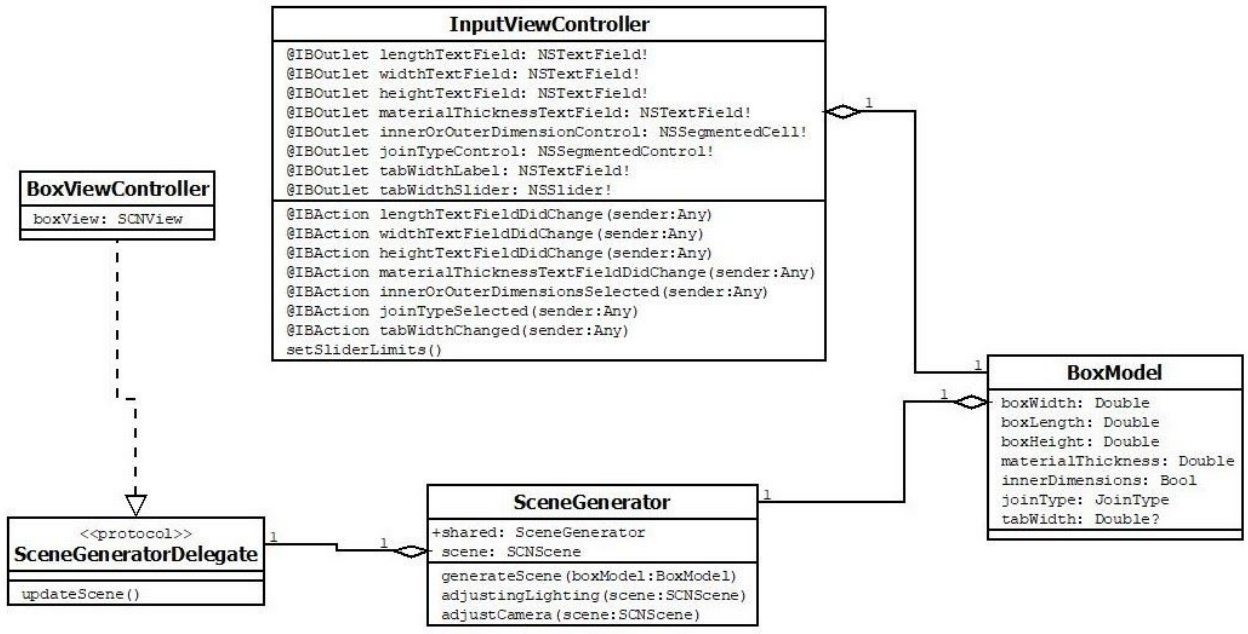

Figure 4: flow of communication to update the box display after changes are made

II. Application Wireframe

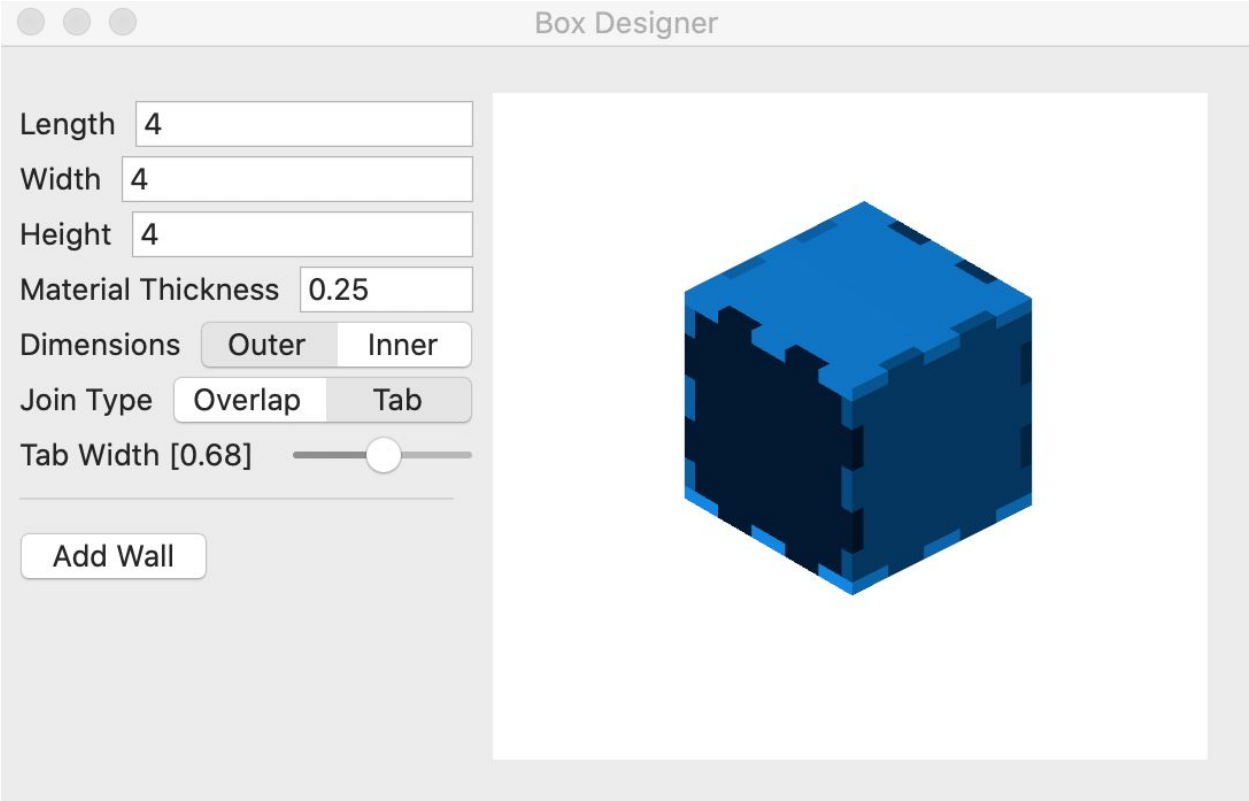

Figure 5: sample application display

As part of the development of system architecture, the team planned wireframes for the expected layout of the application. *Figure 5* shows the final layout of the application. This layout features a SCNView (through Swift library SceneKit) on the right and a series of inputs on the left. The SCNView corresponds to the BoxViewController. This view displays the box model, allows zoom, rotation, and panning, and updates to user input. The inputs on the left correspond to the InputViewController and communicate user input to the box model. The FileViewController, not shown here, corresponds to the menu bar constantly displayed at the top of the screen for computers running any macOSX.

Technical Design

I. Wall Type

Figure 6 shows the three different wall types that were implemented in the creation of the boxes. The three types of walls created are the large corner type, small corner type, and long corner type. When the team was originally developing the program, the wall types were simply case statements followed by a block of bezier path creation. This caused confusion among the team members as it was difficult to determine which block created which bezier path and which path corresponded to each wall type. This problem compounded until the team created an enumeration for the three wall types. This enumeration greatly helped the team in ease of use and readability when creating boxes. These three types drive the creation of the boxes.

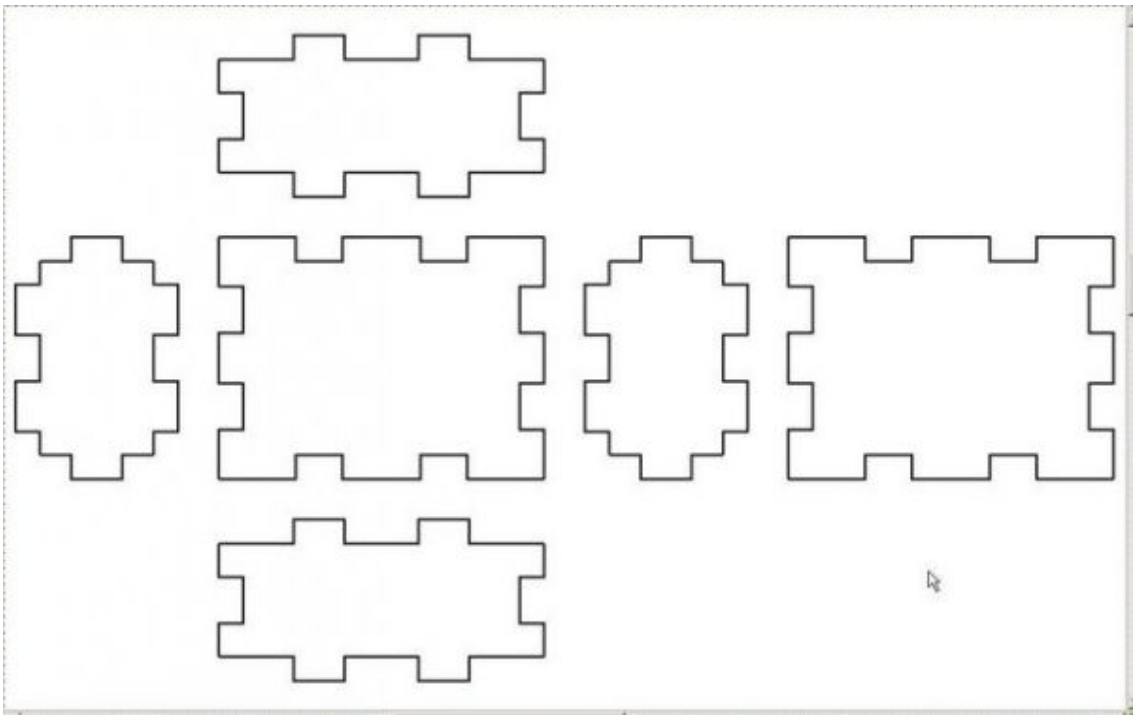

Figure 6: display of the three different wall types. The leftmost wall is a small corner type; the topmost wall is a long corner type; the rightmost wall is a large corner type.

The three wall types follow a general layout pattern in a three dimensional plane: the large corner type walls are parallel to each other in the x-z plane with outward tabs on all four sides, the small corner walls are parallel to each other in the x-y plane with inward tabs on all four sides, and the long corner wall types are parallel to each other in the y-z plane and have two sides with inward tabs and two sides with outward tabs.

II. Path Generator

A path generator was created to allow selection of different wall types. Paths were created using bezier paths provided by the Swift Foundation library. Swift uses bezier paths to create straight lines and curved paths. The Paths created can be used to create a three-dimensional object in the box model view. Only four lines are needed to create overlapping style joins, which makes them relatively simple to draft. Tabbed walls have many more lines to draw, which makes tabbed walls significantly more difficult to create. Path generation was simplified for tabbed walls by using a constant number of tabs for each path.

Different functions were created to generate paths of different wall and join type settings. The primary method of the Path Generator, shown in *Figure 7*, sorts the given information and calls the correct function to generate the path. This organization makes the code cleaner, easier to change in the future, and simpler to expand upon.

```
static func generatePath(_ width: Double, _ length: Double, _ materialThickness: Double,
    _ wallType: WallType, _ joinType: JoinType, tabWidth internalTabWidth: Double?) ->
    NSBezierPath {
    var path = NSBezierPath()
    switch (joinType) {
    case JoinType.overlap:
        path = generateOverlapPath(width, length, materialThickness, wallType)
    case JoinType.tab:
        if let tabWidth = internalTabWidth {
            path = generateTabPath(width, length, materialThickness, wallType, tabWidth:
                tabWidth)
        } else {
            /*
             This path is reached when the JoinType: .tab is originally chosen;
             at this moment, no tabWidth has been set yet.
             */
            path = generateOverlapPath(width, length, materialThickness, wallType)
        }
    }
    return path
}
```

Figure 7: snapshot of the overarching generatePath function

Quality Assurance

I. Unit Testing

There were initial plans to incorporate unit testing to confirm correct geometric calculations. The intent was to test proper calculation of tab widths to ensure box dimensions adjusted appropriately, as well as PDFs produced output with proper dimensions. These tests did not come to fruition. Box display dimensions were instead verified visually to adjust to changes by referencing GUI. PDF dimensions were also cross referenced via integration testing by observing the resulting final product of the laser cutter. The team did not implement variable tab widths and instead verified that hard-coded tabs lined up correctly through the GUI.

II. User Interface, Integration, and Acceptance testing

Proper user interface response was observed visually, which included:

- Testing zoom, pan, and rotate features responded properly
- Observing walls appear correctly and did not clip into each other
- Observing walls update after user input:
    - Typing into text boxes for length, width, height, and material thickness
    - Selecting one of two control options for inner or outer dimensions
    - Selecting one of two control options for overlapping or tabbed join
- Observing a walls correctly add after incrementing interior wall plus button
- Clicking the Open menu item opens an NSOpenPanel
- Ensuring that the selected file for opening was correctly loaded into the application
- Clicking the Save menu item opened an NSSavePanel
- Model was saved with the provided name and file extension
- PDF output generally corresponds to the expected shape of the walls in the model

Verifying PDF output is correct requires integration testing with the GlowForge Laser Cutter:

- Generated PDF is accepted by the laser cutter
- Cut out pieces fit together properly
- Dimensions of the final product correspond to those expected from the model

III. Code Quality

Fundamental coding standards were used to ensure good code readability. An emphasis was placed on maintaining object-oriented programming standards to allow for easy class or method add ons in the future. Refactoring took place when files became to robus and no longer conformed to the Single Responsibility Principle.

Results

I. Features

Most of the required features were successfully implemented, but the following features did not make the cut:

- User can specify tab width (tab width was hard-coded)
- User can add, modify, and delete kerfs (feature not implemented)

Additionally, the team did not accomplish any of the stretch goals for this project.

II. Recommendations

Stretch goals were low in priority and not implemented due to time constraints. It is recommended that teams in the future implement stretch goal features such as:

- Convert walls into lids
- Add kerfs/notches
- Add holes
- Add text
- Specify engraved features
- Select “snap points”

Other features that can be created or improved upon include:

- Allowing the user to specify tab widths. The Path Generator class includes many extra functions that are not called within the program. These functions were part of a bug-ridden attempt to allow the user to specify the tab width. These functions have been left for reference for future developers. The team recommends exploring inputting a path:NSBezierPath component as an in-out parameter, instead of returning and appending a new path for each function called.
- Implementing some amount of rendering where walls can be more transparent and clearly defined from each other. Walls currently have no outlines and joints between them are only distinguishable within the scene if they are each set to different colors. Application of varying rendering techniques is recommended.
- Allowing the user to export the model as additional file types, such as SVG, JPG, and PNG.

Conclusion

Field Session provided significant learning opportunities for improving team work, understanding Agile and Scrum methodologies, as well as learning a new programming language. During this five week period, students from different backgrounds and personalities combined to achieve a common goal.

Solidifying specific roles within the group early on is ideal. Although the team designated Product Owner and Scrum Master early on, there was some confusion about what duties these roles, particularly Scrum Master, actually fulfilled. Solidifying understanding and delineation of these roles early on would significantly benefit team dynamics.

Similarly, establishing methods and standards of communication early on is exceedingly important. The team had to adapt to certain aspects of working over Zoom due to Covid-19 and had to become comfortable communicating consistently in purely online environments. Establishing this communication and becoming comfortable with it early on would significantly help with the virtual working process. If there is ever to be another online only Field Session, it is recommended that Slack or other business communication platforms not be the sole line of communication. Frequently update communication streams by utilizing cell phone or email.

Pair programming and consistent code reviews should also be emphasized. The team originally split tasks between members but had no form of code review in place to ensure that each member understood the others' code. Further along in the development of the application, refactoring changes by one member complicated and confused other members but were never satisfactorily explained. After this point, the team maintained pair programming as a standard to ensure that changes were made and understood by the group. Enforcing pair programming and code reviews early on would have decreased the confusion and frustration between team members with respect to each others' code.

Revisiting architecture is another improvement that can be made. As the application progressed and the team gained better understanding of architecture paradigms particular to app development, expectations for the architecture necessarily needed to change. However, architecture was not revisited until significant frustrations - leading to equally frustrating refactoring - mounted. Continually referencing and updating architecture expectations in the future would lead to smoother program development.

Appendices

Appendix A: Using the PDF

The PDF output by this program has a size of 1100 units by 1900 units; this corresponds to the size of the cutting bed of the GlowForge Laser Cutter (11 inches x 19 inches). In the PDF, a line of 2 inches of length has 200 units, a line of 1 inch length has 100 units, a line of ½ inch length has 50 units, and so on. For the laser cutter to cut each wall with the correct dimensions, the PDF should be blown up in setup so that it fills the entire cutting bed; i.e., the 1100 unit sides should line up with and completely fill the 11 inch side of the bed, and the 1900 unit sides should line up with and completely fill the 19 inch side of the bed.

Appendix B: Links to Applications and Projects for Making Laser Cut Box Templates

https://www.makercase.com/#/basicbox

https://github.com/florianfesti/boxes

https://github.com/kigster/laser-cutter

https://github.com/LaserWeb/LaserWeb4-Binaries

https://github.com/mbugert/laserscad

https://github.com/Obrary/Laser-Cutter-Engrave-Calibration-Plate

https://github.com/rahulbot/box-designer-website