FOLDLINGS: an Interface for Interactive Pop-Up Card Design

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Master of Science
in
Computer Science with a Concentration in Digital Arts

by

Nook Harquail

in Conjunction with Marissa Allen

Department of Computer Science

DARTMOUTH COLLEGE

Hanover, New Hampshire

September 1, 2015

Examining Committee:

Advisor _____
Emily Whiting

Member _____
Lorie Loeb

Member _____
Jodie Mack

_____
F. Jon Kull, Ph.D.
Dean of Graduate Studies

ABSTRACT

Crafting a 3D paper pop-up can be a lot of fun, and can help develop spatial reasoning skills. However, designing the cuts and folds is often a frustrating trial-and-error process. Foldlings is an iPad application that assists in this exploratory process. Our tool-based approach allows users of all skill levels to create complex cards with ease, by separating folding geometries into logical units. This thesis focusses on the user interface and algorithms used in creating the two-dimensional view of the popup card from user input. Our approach uses a modular set of tools, which can be combined to create complex designs. We guarantee the validity of the folded card at all stages of design, and our interface provides an intuitive set of visual aids to help users visualize the relationship between 2D patterns and 3D geometry.

Acknowledgements

Thanks to Marissa Allen, my collaborator on this project, and co-author of Chapter 1: Introduction and Chapter 5, section III: Potential Applications.

Special thanks to Tim Tregubov, for working with us to develop the initial prototype for Foldlings, and for being a costant source of advice and technical help.

Special thanks to Kevin Baron and the Thayer Machine Shop at Dartmouth for assistance with laser cutting.

Thanks also to our advisors: Jodie Mack, Lorie Loeb, and Emily Whiting. Particularly to Emily, whose guidance and comments on the software and thesis have been transformative.

Thanks to our first user: Rukmini Goswami. Her feedback (and patience with our buggy software) were invaluable to us.

And thanks to Kiko Lam and Luke Zirngibl for their feedback throughout the process.

We use a number of open-source packages in Foldlings:

- **ANPathIntersection**: https://github.com/unixpickle/PathIntersection
- **CGPoint+Vector**: https://github.com/koher/CGPoint-Vector
- **ExSwift**: https://github.com/pNre/ExSwift
- **PerformanceBezier**: https://github.com/adamwulf/PerformanceBezier
- **SVGh**: https://github.com/GenerallyHelpfulSoftware/SVGgh
- **UIBezierPath+Interpolation**: https://github.com/jnfisher/ios-curve-interpolation
- **UIImage+AnimatedGif**: https://github.com/mayoff/uiimage-from-animated-gif

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

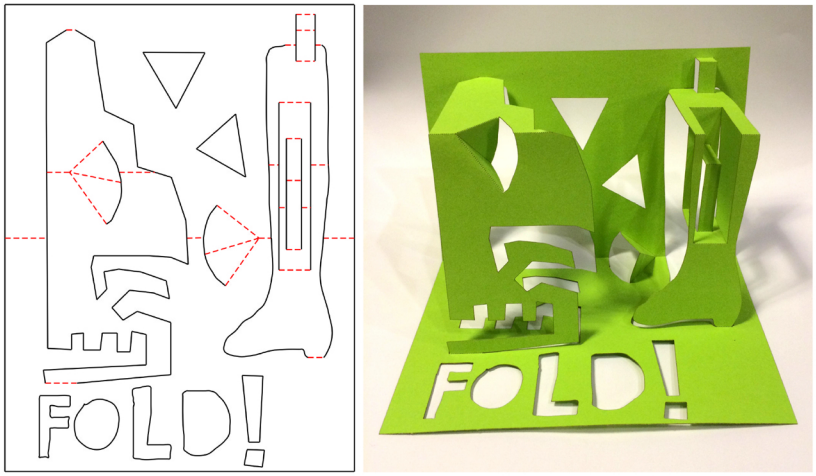


Figure 1.1: A complex design created with our software.

I. Background

This section is co-authored with Marissa Allen

Kirigami is the art of papercraft originating from 17th century Japan (Temko and Takahama [1978]). Kirigami structures vary widely in form and scale — from sub-microscopic creations to large sculptures (Grosso and Mele [2015] to Andrews et al.). Kirigami can represent a wide range of 2D and 3D constructions. For example, a design can be a 2D

lace-like pattern, or a large architectural structure; as long as the paper design does not require glue or other attachments, it is kirigami. Further constraints define subsets of kirigami — such as origami, which only allows folds[1]. In particular, Foldlings is concerned with 90-degree pop-up cards, which have the additional constraint that orthogonal relationships exist between planes. We do not use glue or other attachments in building pop-up cards; all designs are cut from a single piece of paper, in the tradition of kirigami (Temko and Takahama [1978]). Thus, Foldlings creates orthogonal, kirigami popup cards.

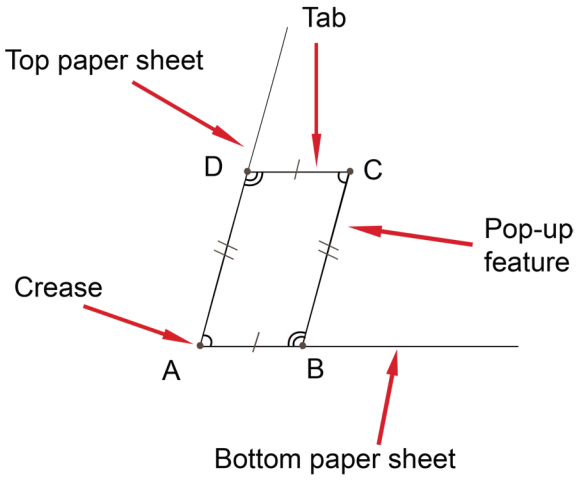


Figure 1.2: Cross-section of a popup card. Figure modified from https://en.wikipedia.org/wiki/File:Popup-diagram.svg.

Typically, users create popup cards manually. For example, a user might sketch out shape on a card in pencil, and then measure with a ruler to determine where to place folds. Or, they might fold the paper while cutting, discovering correct fold positions experimentally. The second method works well for simple designs, but becomes difficult with complex and nested geometries[2]. Constructing popup cards manually is difficult for several reasons:

1. **2D to 3D visualization is difficult**. Users have difficulty understanding how the card will fold based on a 2D design.

[1]Purist origami adds further restrictions, including the requirement that the starting piece of paper be square (Burczyk).

[2]These are behaviors we observed by watching users create pop-up cards.

2. **Geometric constraints**. Pop-up cards present strict constraints, which are often unintuitive to novice designers.
3. **No "undo"**. Since pop-up card design is often a trial-and-error process, designers must sometimes make many versions of their card to test their design.
4. **Physical constraints**. In addition to the geometric constraints, the paper medium presents physical limitations on where edges can be placed.

The 90-degree popup card presents a tightly-constrained problem, with opportunities for both interface design and algorithm innovations. Our work spans human-computer interaction, graph-based algorithms, and algorithms for manipulating bezier paths and pop-up card structures. Our culminating product is an iPad application.

We present a system for designing popup cards, whose audience is deliberately broad. That is, our tool aims to make the design process easier and more fun for users with all degrees of popup card design experience.

II. Motivation

This section is co-authored with Marissa Allen

Figure 1.3: A sample of cards created using Foldlings. For larger images and fold patterns, see Appendix D: Sample Cards on page 107.

We set out to create a tool that would help people design 3D pop-up cards. We were driven by the desire to create original kirigami designs without a template. As we progressed in our project, we began to focus on developing a tool that would provide a way to create pop-ups more intuitively, without explicitly understanding how to create a valid 90-degree pop-up card. Our tool allows users to design, iterate, and preview an original pop-up card before they even pick up a pair of scissors.

We outline some user stories — potential use cases for our software:

- George is an avid scrapbooker. He uses Foldlings to create small pop-up elements to liven up his scrapbooks. Sometimes he, creates cards to commemorate moments — other times, he pastes photos or other media onto a pop-up card structure created with our app.

- Sally designs cards for a commercial greeting card company. She uses Foldlings to create rough prototypes and concept sketches for pop-up cards. After getting feedback from the rest of her team, she brings the exported SVG file into Adobe Illustrator to refine the final card designs.
- Jim forgot to make a card for his wife's anniversary present, and she's coming home in an hour! He downloads Foldlings from the Apple App Store, and is able to quickly design and create a beautiful card.
- Alex is a creative college student who wants to continue making things for her friends and family. She wants to explore pop-up cards, but doesn't like any of the templates she's found online and can't pay for pop-up design books. She finds Foldlings in the Apple App Store and tests her creativity with our tool.
- Kate, a middle school math teacher, is teaching a module on pop-up card geometry. She uses Foldlings to teach the students about the parallelogram constraints between planes as cards fold. By using our app, her students gain an intuitive understanding of the geometric constraints.

III. Technical Overview

This section is co-authored with Marissa Allen

Our algorithmic simulation and validity detection is based on a tight set of constraints to the pop-up card problem. We require the card to have a central main valley fold — from there we can determine the orientation of planes, as alternating folds fold in opposite orientations. The card then folds 180 degrees; this implies parallelogram constraints between all the edges in a sideways cross-section (with the exception of v-fold features).

We capture touch input, translating touches into cuts and folds depending on the geometry of the fold feature. We validate fold features before they are added to the sketch, ensuring that the design can fold in 3D. At this point, we also perform bezier path opera-

tions to modify existing edges in the sketch based on the new design element. When a fold feature is added to the sketch, we re-calculate planes (areas enclosed by cuts and folds) by traversing a directed graph of edges in the sketch. Based on these 2D planes, we create 3D planes, which are oriented and translated based on their relationship to other planes in the sketch. In 3D, we animate planes in response to user input. Each of the planes is translated in relation to the plane oriented above the fold and rotated in relation to the main driving joint.

Our approach constructs a tree representation of the planes based on fold adjacency and uses this for determining the parent child relationships in the simulated 3D view. We also use a tree-based structure to store associations between logical geometric units.

(a) Development Process

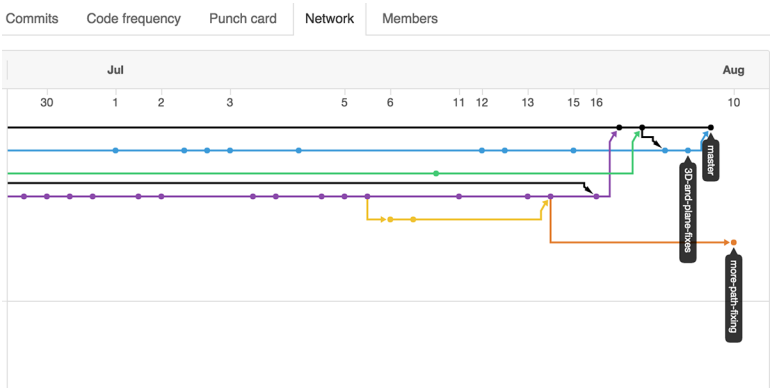


Figure 1.4: Branches in our github.com repository.

We designed and developed the software interactively, frequently testing prototypes with users. We used github to build Foldlings collaboratively. Our workflow involved creating new code branches for each feature, and reviewing the changes before merging back into the master branch of the codebase. The full source for our software is available at `http://github.com/harquail/foldlings/`.

IV. Pipeline Overview

This section is co-authored with Marissa Allen

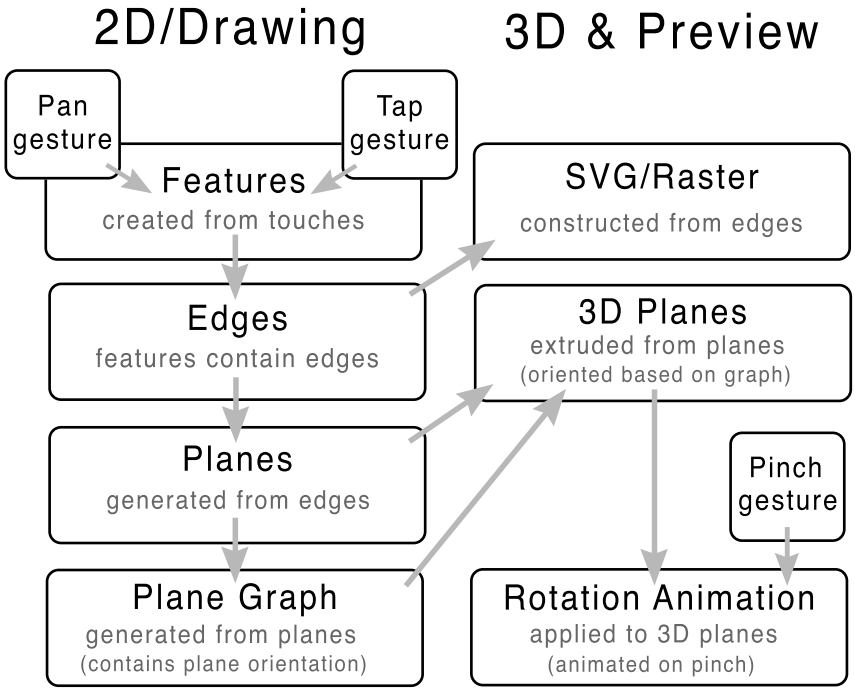


Figure 1.5: Overview of data flow between 2D and 3D systems.

To begin, a user draws a design using the fold feature tools: box fold, polygon, freeform and v-fold. These tools create a pattern of cuts and folds, displaying an interactive preview of the design as the user creates it. The cuts and folds created with these tools remain associated with each other, and can be modified or deleted as a unit.

Each time a new shape is added to the design, it is evaluated for validity: whether it can fold to 90 degrees and be parsed into individual planes. The planes are then linked together in an acyclic graph based on the planes' abutting top edges. This acyclic graph allows us to shade planes based on orientation in 2D and simulate the design in 3D. Each feature can also be modified or deleted, by tapping on the feature an selecting an entry from the list of available options.

This process continues until the user previews the design in 3D. The 3D preview dis-

plays a simulation of how the design will fold, which can be manipulated using a pinch gesture. The user is free to return to the 2D drawing interface and continue editing his/her design or save the design as either a raster file or SVG vector file. After this step, the user can print and cut the raster file or open the SVG file on a laser cutter or other cutting tool. We automatically save designs locally when leaving the design workspace, so users can restore their work.

Figure 1.6 shows the full pipeline for designing cards. It shows a user's design process, starting with concept sketches, and moving through iterations of the sketch using the 3D preview to test the design. Finally, the user exports the design as a fold pattern, and cuts and folds the pop-up card.

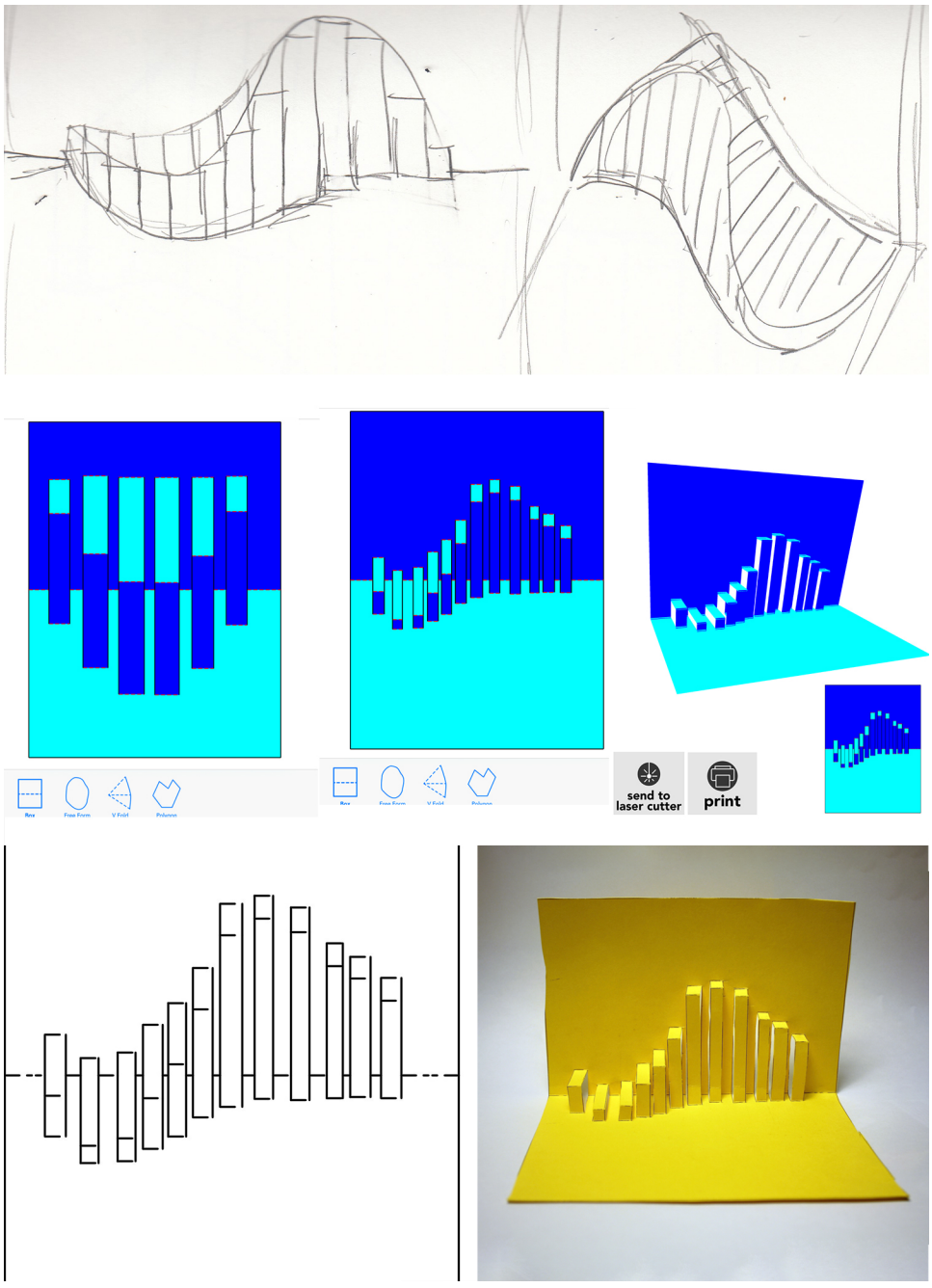


Figure 1.6: A full outline of the design process for Foldlings, from initial concept sketches to full realization of a paper pop-up card.

V. Related Work

This section is co-authored with Marissa Allen

Software that creates 3D geometry from 2D sketches is an active and vibrant area of HCI and graphics research. For example, Igarashi et al. [2007] present a sketch-based interface that creates 3D geometry in real time based on 2D silhouettes sketched by users. Patrick et al. [2002] use a silhouette-based approach in a tool for searching 3D models, and Wang et al. [2003] use 2D sketching to design 3D garments. We solve a far more tightly-constrained problem, in that we are concerned with foldable pop-up kirigami, rather than arbitrary 3D meshes.

Specifically, there has been some previous work in pop-up card design software. The seminal work on popup-card design software, Glassner [1998], uses glue to connect components, and so does not meet our criteria for a kirigami pop-up card. Still, we take inspiration from Glassner's work, and borrow his calculation for resolving the v-fold angle constraint. Our approach to pop-up creation does not require glue or 3D modeling experience, but is strict kirigami, as described in Background on page 1. Glue-based design presents different affordances than kirigami, but requires an assembly stage that disconnects the initial 2D pattern from the final pop-up geometry.

Others approach the pop-up design problem from the opposite direction: creating designs by modeling in 3D space Ruiz et al. [2014]. For example, Li et al. [2010] develop an algorithm that transforms user-defined models into paper architectures that are stable and rigid. Because their algorithm modifies the input geometry given by the user, the end result does not necessarily preserve the user's design intent. Other attempts to create pop-up cards such as Abel et al. [2013] can only take in simple polygonal meshes and require the user to fold and glue additional pieces of paper together, whereas our approach creates a pop-up card with arbitrary shapes from a single piece of paper. These methods all require the end user to know how to use traditional 3D-modeling software, and thus are not accessible to novice users. Another drawback of these is relying on 3D model as input allows for less

exploration on the part of the user. Because constructing designs in 3D requires significantly more effort than creating a 2D sketch, users are less likely to explore many design possibilities, and therefore less likely to gain an intuitive understanding of the geometric constraints imposed by paper.

Way et al. [2013] also create pop-ups from 3D models. The tool segments a 3D model and then uses shape recognition to create paper models in pop-up cards. However, their method of simulation is similar enough to our's to merit special mention. The authors create an acyclic graph from paper segments and use the nodes to drive a simulation of the pieces based on opening and closing the card. We also create an acyclic graph of nodes, and rotate planes during simulation based on traversal of the graph.

Others approach a slightly-different pop-up card design problem. Li et al. [2011] present algorithms for creating v-style pop-ups. Not to be confused with v-folds, they create pop-ups composed of "patches falling into four parallel groups," and present algorithms for constructing and analyzing v-style popup books. V-style planes have different angle constraints than our orthogonal pop-ups, allowing for a completely different set of designs fulfilling geometric constraints. Their designs require multiple sheets of paper — attached via glue or with a paper hinge mechanism. Okamura and Igarashi [2009] describe a program to design elaborate 180^{o} pop-ups that can be applied to both cards and books strictly within a 3D environment. They implement collision testing between multiple components inside their pop-up program. However, their program does not incorporate the card's primary fold in designing pop-ups, it is simply a mechanical driver for the rest of the design, which is attached using glue. Therefore, their designs are not kirigami.

Xu et al. [2007] describe algorithms for creating 2D cut patterns from input images, allowing designs to be composed from multiple composited shapes. Although their result is strictly two-dimensional, we share their interest in producing paper art through software. Similarly, Johnson et al. [2012] present an interface for creating precise laser-cut designs by interpreting and smoothing user sketches. Although not directly related to the pop-up

card design problem, we use a similar method for smoothing user input.

In addition to pop-up card design, several authors create tools for origami design. That is, designs created using only folds. Fastag [2009] describe "eGami," an interactive system for folding flat (zero-thickness) origami models. While they solve a very different problem than Foldlings, they take a similar approach in terms of providing a toolset of common operations and displaying an interactive preview. Kasem and Ida [2008] present a web-based tool for origami design, built on the work of Zamiatina [1994]. Ju et al. [2002] take an unusual approach to origami design software, in that the user physically folds the card, while projectors and a sensing system provide feedback on how closely the user's actions match a target design. While this kind of feedback is useful in reproducing an existing design, it does not facilitate the creation of novel designs.

Hendrix et al. [2006] describe a pop-up design interface with goals very similar to those of Foldlings — they aim to help users create valid designs and visualize their constructions in 3D. However, this tool was built as an introduction to the engineering sciences and not as a tool to aid artistic design. Perhaps as a result of an engineering-centric design methodology, the pop-up tool has a more restrictive UI than ours. Their program is designed to only create the simple folding mechanism of the pop-up card. Non-rectilinear cuts must be added manually, outside the software. In comparison, our tool allows for more creativity, by allowing the user far more control over the design and the ability to preview the entirety of the design, not just the folding structure. Additionally, while their software includes feature-based tools, their approach does not allow for the modular modification of design elements after creation.

Chapter 2

Design

I. Design Philosophy

Our design philosophy is simple: follow the user (Bell [2008]). As often as possible, we presented our interface concepts to potential users, and allowed their feedback to guide the design process through the final prototype.

Roughly following the Agile Methodology, we designed and developed the application collaboratively (Martin [2003]). Our process was driven by user experience design, rather than graphic design. That is, we spent relatively little time polishing aesthetic interface details, and instead focussed on the core interactions of the application.

Throughout the design process, we explored the conflict between creativity and rigid geometric constraints. That is, interfaces that give the user more room for creativity generally tend to make it more difficult to create designs that will fold correctly in 3D. We aimed for an interface that provides a great deal of flexibility and creativity, while guaranteeing that all popup card designs created with it are valid. Two primary systems maintain this balance.

1) Tool-based feature creation allows for complex geometry while solving geometric constraints transparently.

2) The validity system disallows geometry that would interfere with existing design elements.

The iPad (and tablets in general) presents unique affordances[1]. We chose to design for the iPad as a way of forcing the interface to remain minimal and intuitive. With a simple, gesture-based interface, we could not rely on complex interactions and instead focussed on distilling the essential design elements. One benefit of designing for a touch-based interface is that the interactions are "natural" — that is, intuitive and easy to learn; however, gestural interfaces present a unique set of design problems[2] (Norman and Nielsen [2010]). Additionally, the iPad lends itself well to casual, fun experiences, which aligns with our goal of creating a usable popup-card design interface for a general audience (Johansen [2013]).

We strive to design an interface that is **modular**, **friendly**, and **delightful**. **Modularity** stems from the conception of the pop-up card as a collection of discrete units that can be acted on individually. Modularity allows users to think in terms of shape constructions, without concern for individual cuts and folds. Users can modify, add, and delete individual geometric units, without affecting the majority of their design. **Friendliness** is seen in the careful structuring of our experience to make getting started as painless as possible. For example, we structure our tutorial not as a step that must be completed before using the app, but as a series of brief videos that appear when using a tool for the first time. **Delight** comes from small, unexpected details that enhance the user experience. For example, our color scheme for planes is inspired by the colors of construction paper. Through this color scheme, we hope to evoke the spirit of fun and exploration associated with casual paper-craft.

[1] Affordances being possible interactions. The term was popularized by Norman [2013].

[2] For example, gestural interfaces have relatively few inputs: compared to a mouse and keyboard, which have more than 100 unique input options, Apple's UIGestureRecognizer class recognizes fewer than 12 gestures.

II. Interface Iteration

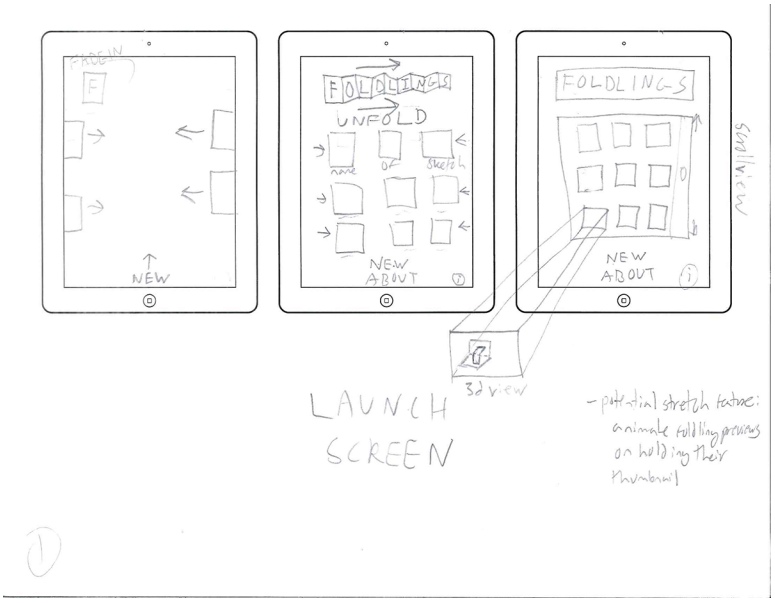


Figure 2.1: Initial mockups showing cards for saved sketches on the main screen.

We started development in Fall 2014, as the final project for a course on Computational Fabrication at Dartmouth. By December 2014, we had a functional prototype of our popup card design software: users could create cuts and folds and fabricate them using a laser cutter. However, in many ways our software was no better than manually creating cards. In order to create a valid and beautiful design, users needed some prior knowledge of the geometric constraints described in Chapter 3, section VI. (a), Geometric Constraints, on page 60. Performance problems also hampered users' ability to create designs. Our goal was to improve both of these aspects of Foldlings dramatically, to arrive at an intuitive and functional tool. Paper mockups were an integral part of our design process. We used these paper mockups to drive interface development, and also to quickly get feedback from users before investing development resources[3].

In arriving at our final design, we iterated through several potential designs, each time getting feedback through informal user studies. We made many changes to the toolset and experience based on feedback from users. The final approach is "feature-based" — in other

[3]See Appendix X: for samples of paper mockups we created during the design process

words, the user creates multiple cuts and folds in a single action, rather than individually. These discrete logical units allow the user to design more quickly, and to combine multiple different feature types to create complex designs. A key advantage to this design over a less structured approach is modularity. This is both an algorithmic and an interface advantage: our algorithms benefit from collecting cuts and folds into discrete logical units, and the user benefits from a faster and more structured design process.

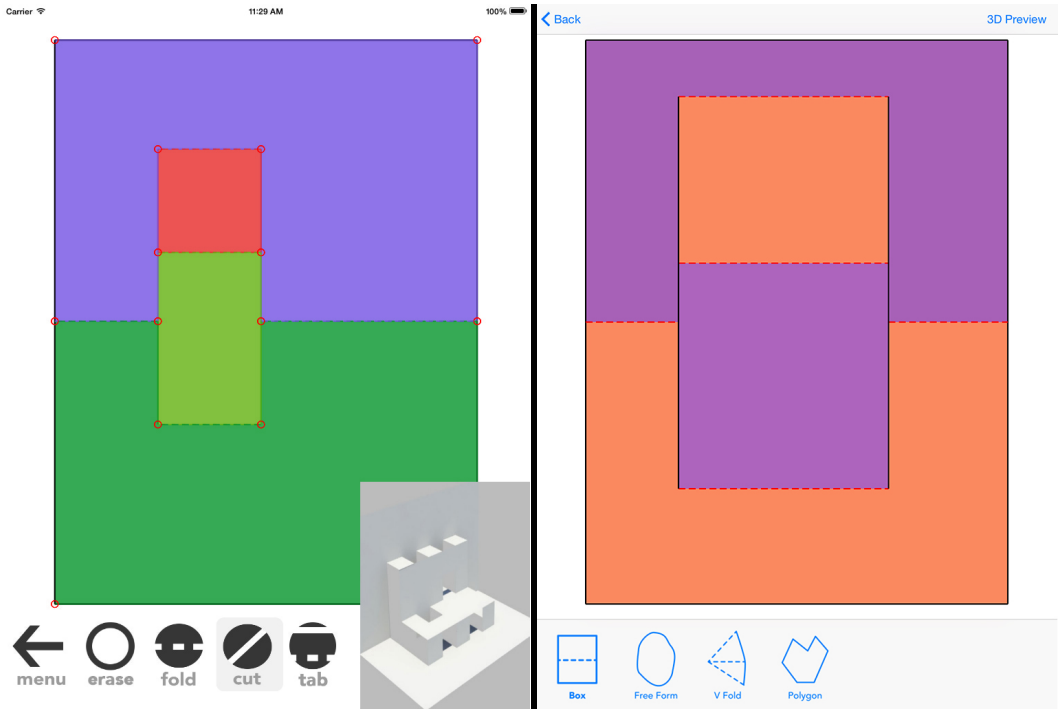
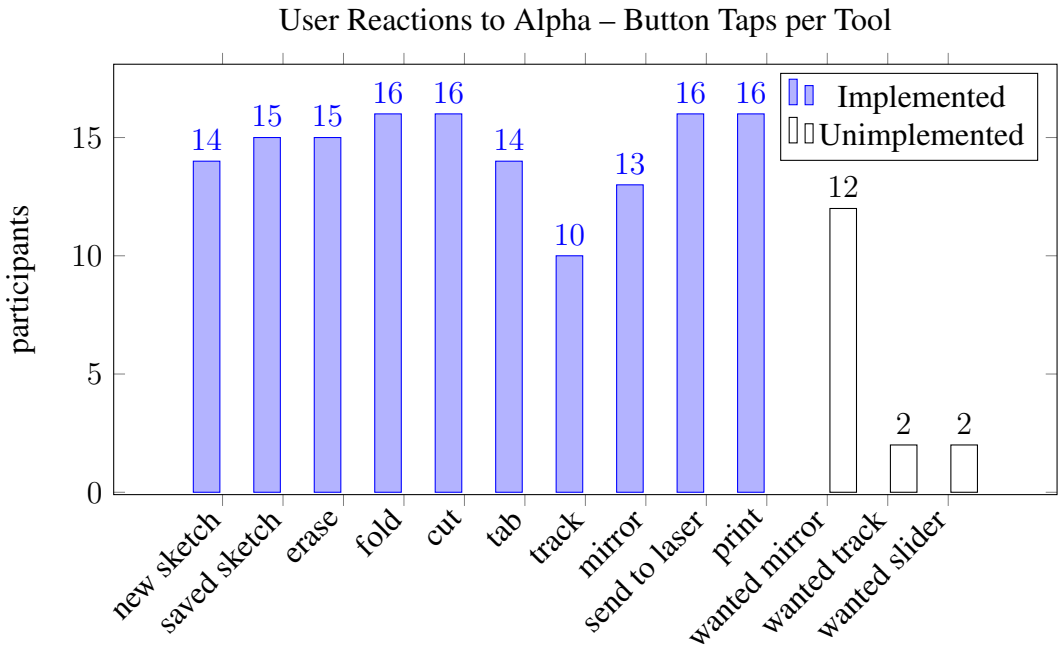


Figure 2.2: Left: drawing interface as of December 2014. Users directly draw cuts and folds. Right: drawing interface as of August 2015. Modular, feature-based interface.

Throughout the development process, we collected feedback through informal user tests. One test we performed involved presenting a partially-implemented version of our software to users. The majority of buttons were functional — erase, cut, fold, and tab, but our interface also contained buttons for unimplemented features. The primary goals were to test whether our existing tools were useful and to collect feedback on potential new features for Foldlings. When a user tapped a button that we had not yet implemented, we asked them to describe how they thought the tool would work, and talked with them about the behavior the button represented.



In the graph above, the first ten labels indicate the number of participants who tapped a button. The last three labels — starting with "wanted mirror," show responses to unimplemented tools. To calculate these numbers, we asked first users to describe what they thought the button would do, and then gauged their reaction as we described our vision for that tool. The tally represents a qualitative measure of whether the user was enthusiastic about the feature or had a more negative response. Some negative responses include confusion about the tool's purpose and statements such as "I don't think I would want to use that."

The unimplemented tools were:

1. Mirror: a tool that would take an existing group of edges and reflect them across a fold in the sketch
2. Track: a tool that would create a cut in the sketch, and a second shape that would move within the cut, with folded tabs on the opposite side of the card. This tool required multiple pieces of paper.
3. Slider: a tool that would create a parallel set of cuts, making a "pocket" that a second shape would slide through. This tool required multiple pieces of paper.

Each of these tools was inspired by features often found in commercially-designed popup cards and popup books (Birmingham [1997] and Valenta [1997]).

From this test, we learned that the track and slider tools were confusing, and that novice users were generally not interested in creating features that require multiple pieces of paper.

As users created sketches using our software, we also took notes on their experience and collected suggestions for improvements. Although only a small fraction of the features requested by users are implemented in the final application, the feedback from these early user tests set us on the path toward feature-based design. A common theme among the observations in Table 2.1 is the difficulty in creating valid sketches and confusion about the proposed track and slider tools.

Table 2.1: Observations of behavior from first user test.

Observations
made a cake using cuts & tab
track and slider will need explanation; wanted to use non-horizontal folds
erased master fold, causing preview to fail
made a cat with cuts
momentary confusion getting back to sketch from 3D preview
confused about concept of a laser cutter
very frustrated by tools that aren't implemented yet
confused by track & slider
needed heavy guidance; completely confused by track/slider
fairly self-sufficient after tools were explained, made a house, moved slowly, waiting for planes to calculate

A common theme among the feature requests was a desire for a more complete tutorial and explanation of tools. Features only appear once in Table 2.2, even if they were

requested by multiple people. The most-requested feature was an interactive tutorial, requested by 4 (of 16) users. In the final version of our software, we display short example videos when users use a tool for the first time. See Chapter 2, section III. (c) Tutorial on page 25 for more discussion. These tutorials allow users to get started quickly, with minimal interruption. In addition, a commonality between our observations of user behavior and the feature requests is a desire for more guidance in creating valid cards. This indicates that the geometric constraints of popup cards were not sufficiently intuitive or discoverable in this version of Foldlings.

Table 2.2: Feedback from first user test.

Feature Requests
option to reverse all folds
draw over existing fold with tab tool
non-horizontal folds
fold by pinching
rename sketches
delete sketches
orthographic views
long press to view information about tool
reorder sketches on main screen
interactive tutorial
more snapping/validity guidance

As a result of this and other informal user tests, we modified our software based on user feedback. One of the key findings from this early test was that our toolset was very difficult for amateur users to use. Due to the difficulty users had creating valid designs, we completely overhauled our interface to focus on keeping the user's design in a valid

state. In our final interface, users cannot add an element to a sketch that will cause it to become invalid. Along with a system that validates cuts and folds before adding them to the sketch, our modular tools help ensure validity. Rather than individual cuts and folds, our tools create modular folding units, which solve geometric constraints transparently and display an interactive preview of edges during feature creation. Although we considered a system based on automatically correcting the user's arbitrary folds and cuts, we believe our tools-based system allows for a similar level of creativity, while guaranteeing a valid pop-up card.

III. Tool Interactions

Through the iterations described in the previous chapter, we arrived at Foldlings' tool-based system for card design. Each tool creates a specific type of feature — a group of cuts and folds that define planes that will fold together in 3D[4]. The four feature types: box fold, freeform, polygon, and v-fold, help users design complex cards while maintaining the card's foldability at all stages of design. The core interaction is as follows:

1. User selects a tool
2. User drags on the screen to define a feature
    a. (Some features require more than one touch to define)
3. The feature is added to the sketch on releasing the drag

(a) Feature Interactions

Some interactions are common to all features. To add a feature, you select the tool that creates features of that type. Each feature type[5] has a corresponding button in the toolbar at the bottom of the sketch. In general, all features are defined by dragging in the drawing area. Features are generally completed by releasing the drag. As long as you remain in that tool, you can continue creating features of that type by dragging. Consistent tool interactions help reduce the burden of learning new tools, and allow for a scaffolded user experience (Wood [2001]).

We can infer that the user has completed a sketch when a touch completes the feature. Completion conditions are different depending on the feature, but the completion state is never ambiguous. Two feature types are always defined with a single touch: Box Fold and Free Form. The multi-step tools: Polygon and V-Fold — require more than one touch to define.

[4]Described in more detail in section IV., Interface Data Structures, on page 30.
[5](Except for the master card)

The fold features are described in section IV., Interface Data Structures, on page 30. What follows is a description of the interaction to create each type of fold feature.

Box Fold Interactions

A box fold is created by dragging to define the bounds of the box. Box folds are only valid if they span a driving fold.

Free Form Interactions

Free-form shapes are created by dragging a closed shape. Free-form shapes that cross a fold are truncated[6], and a center fold is automatically added at the correct height. If a free-form shape does not cross a fold[7] it is considered a hole, and no folds are added. Initially, we considered having a separate tool for creating holes. However, through informal user tests we discovered that users intuitively understood that free-form shape that do not cross a fold will become holes — and we were therefore able to combine the two functions into a single tool.

Polygon Interactions

Polygons are created one vertex at a time. Points are added by tapping or dragging on the sketch, adding edges between successive points with each tap. Once a vertex is (nearly) coincident with the initial point, the feature is complete. Users can also drag existing points in the polygon to modify the shape.

Initially, tapping was they only way to add points to a polygon — we added dragging points to make the interaction more consistent with other feature types.

Polygons are the only feature that can be created by tapping rather than dragging. This creates a conflict with tap options, which are also accessed using a tap. If the user is in the

[6] See Chapter 3 section II. on page 48 for a description of the truncation process.

[7] More formally, if a feature does not span a fold, as determined by the featureSpansFold function described in Chapter 3 section II. on page 46.

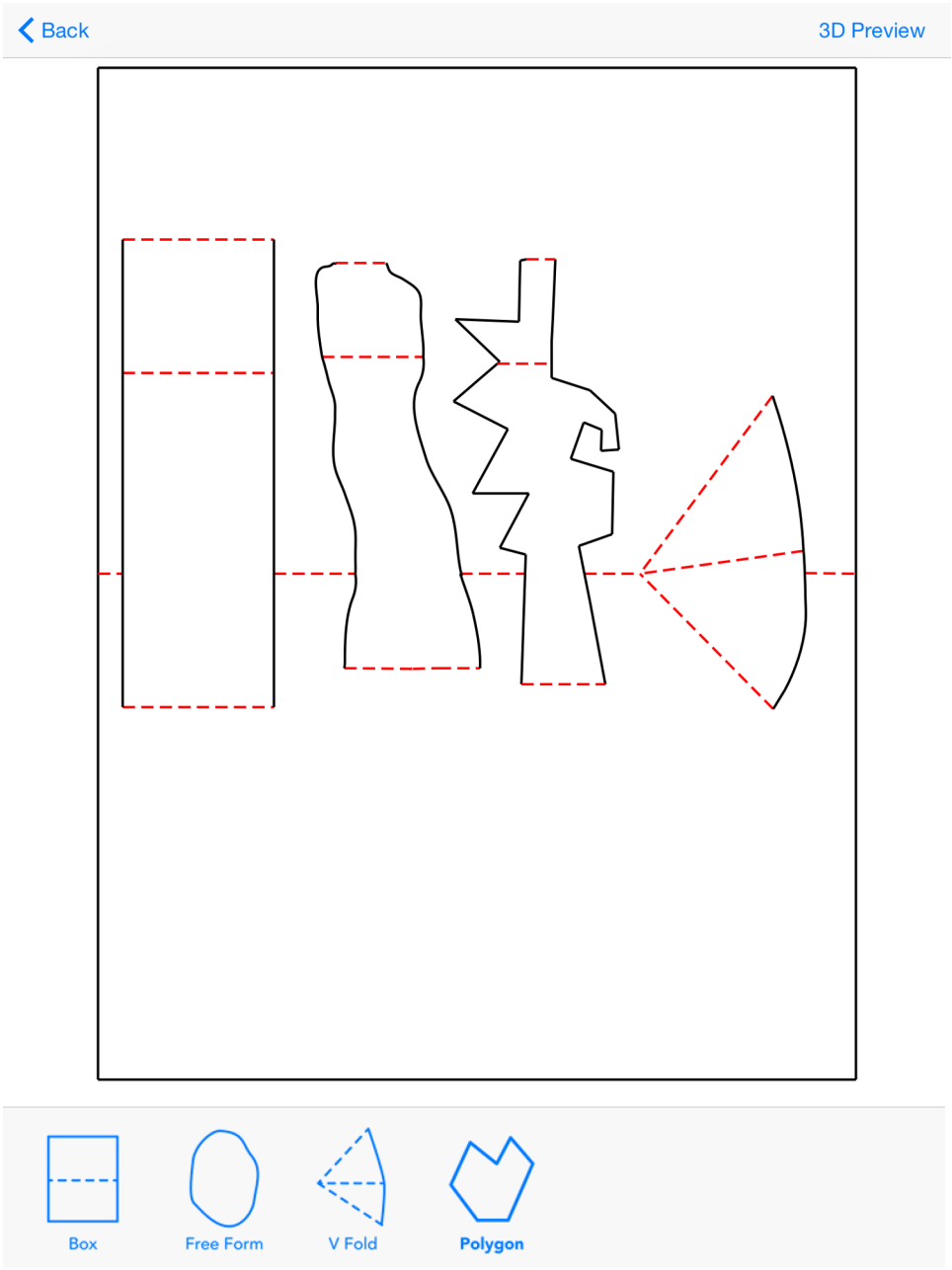


Figure 2.3: Examples of the four fold features created by the tools. Left to right: box fold, freeform, polygon, v-fold.

polygon tool and taps inside an existing feature, it is ambiguous whether they want to start a new polygon or select a tap option. To resolve this conflict, if the first tap of a polygon is inside another feature (other than the master card), we display the tap options rather than creating a polygon. Users can either start polygon within the master card or use a drag to add the first point, if they wish to construct a polygon inside another fold feature.

V-Fold Interactions

V-Folds require two touches to complete. The first touch creates a "vertical cut" that crosses a fold, the second defines the point on that fold from which diagonal folds are constructed.

(b) Tap Options

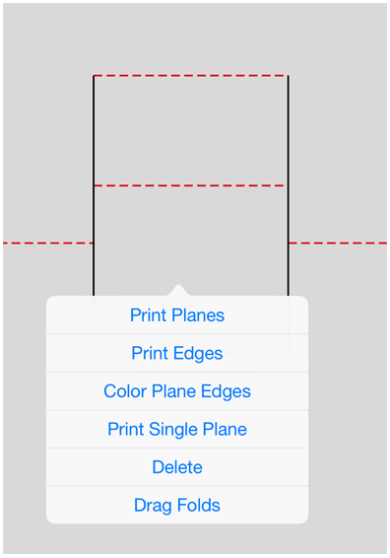


Figure 2.4: Options presented when tapping a box fold feature.

Tap options are actions that can be performed on a feature. These options allow the user to modify or delete features in the sketch. Different options are presented based on the feature type and state. For example, currently only leaf nodes in the feature tree have the "Drag Folds" option. That is, you can only move folds within a feature that has no children. Since moving folds within a feature with children would require modifying the folds of all

of their children[8], implementing fold dragging in features with children is future work.

(c) Tutorial

We eschewed detailed drawing instructions or a separate tutorial mode, in favor of short video tutorials that appear the first time each tool is used. These tutorials can also be accessed by tapping the feature icons on the about page.

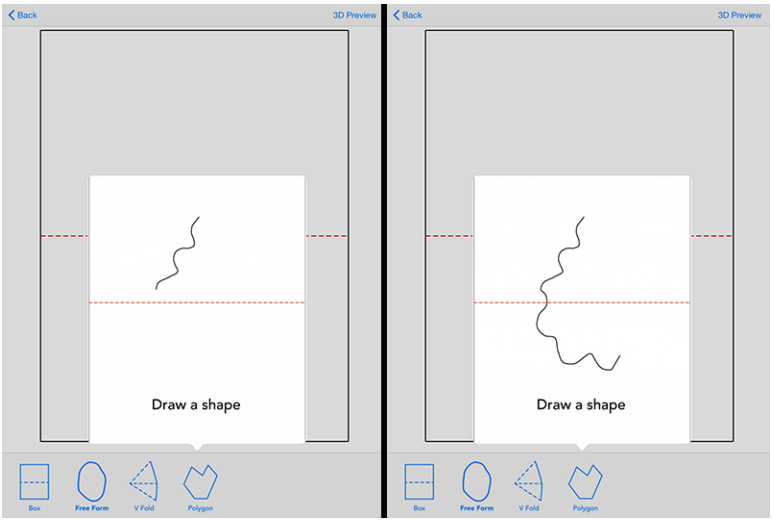


Figure 2.5: Free-form shape tutorial video.

We also show helpful tips between screens — for example, when moving to 3D preview and restoring from a saved sketch.

(d) Warnings and Errors

We display warnings and errors as bright-red banners above the sketch work area. These warnings are displayed in response to failing the validity checks performed when adding a feature to the sketch[9].

The goal of these warnings is to give users descriptive feedback when errors occur, and to give them an intuitive sense of which actions create invalid features.

[8]Dragging folds in a parent feature can also cause child features to become invalid, depending on their fold positions.

[9]The system for validating fold features is described section Chapter 3 section V., Validity on page 58

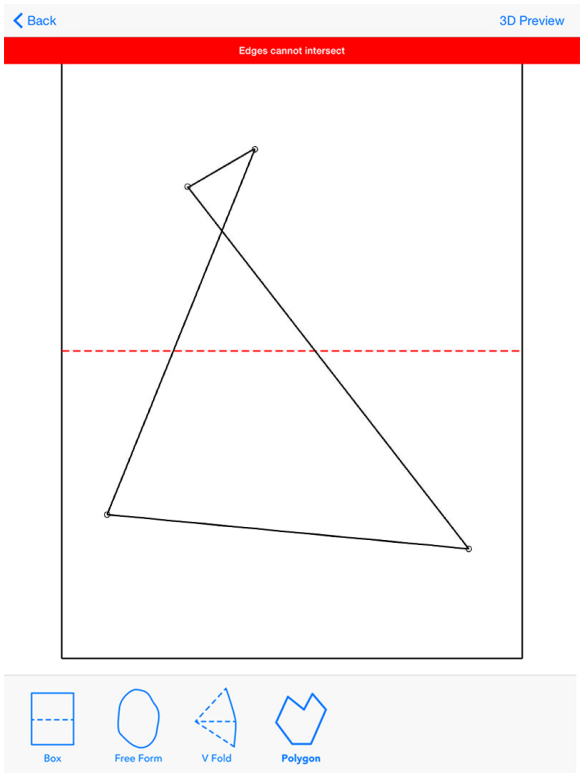


Figure 2.6: An error message shown when rejecting a polygon with intersecting edges.

(e) Intersecting Features

Some features can be drawn over cuts and folds of existing features. When a new feature intersects a previously-drawn feature, it occludes existing cuts and folds — creating the new feature on top of existing features. The implementation of these intersections is described in Chapter 3, section IV. on page 55.

(f) Send to Laser Cutter

Users can tap the "send to laser cutter" option; this sends the user an email with an attached SVG file. This file can be fed to a laser cutter or paper cutting machine, and can be opened in a vector graphics editor to make further changes.

The sketches are bound by physical constraints, as described in Chapter 3 section VI. (b), Physical Constraints on page 64. One constraint is the precision of the cutting tool, which limits how closely cuts and folds can be drawn to each other. We take these physical

constraints into account during the sketching process, so the user's design is foldable.

(g) Print

In addition to sharing an SVG file for laser cutting, users can press "print". This option provides what is essentially a screenshot of the 2D sketch. This image can be printed, emailed, or shared via social media. Typically, this is the option a user would choose to cut and fold their design by hand.

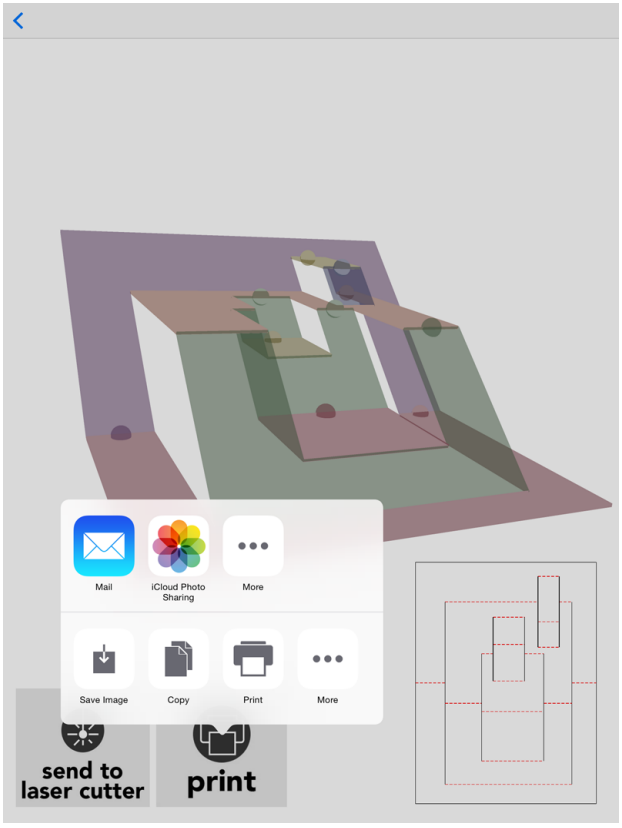


Figure 2.7: Options for sharing a fold pattern from the 3D preview.

(h) Visual Aids

Foldlings utility relies on the user understanding how their design will fold while they are designing. We use visual aids to help the user. Using data from the user study described in Chapter 3, section II., on page 69, we adjusted our interface to include more cues to help

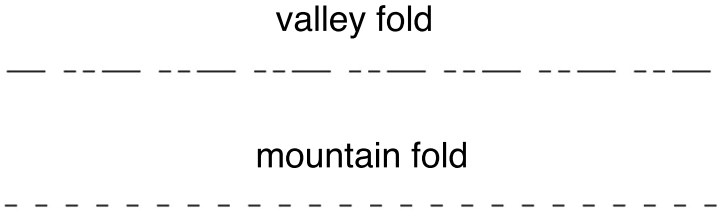


Figure 2.8: Line patterns in SVG export

users visualize the design. The primary visual aid is the 3D preview, which displays an interactive preview of the folded card. Users can interact with this card through an intuitive "pinch" gesture. In addition, we shade planes based on orientation (cool colors for planes that will be vertical when the card is halfway folded and warm colors for horizontal planes). In the SVG file (accessed via "Send to Laser Cutter"), we adjust line dash patterns for folds based on orientation: dotted lines are mountains, creased away from the main fold — dot-dash lines are valleys, creased in the same direction as the main fold. From the visual aids user study, we have data suggesting that displaying fold orientation is very helpful to users when folding pop-up cards.

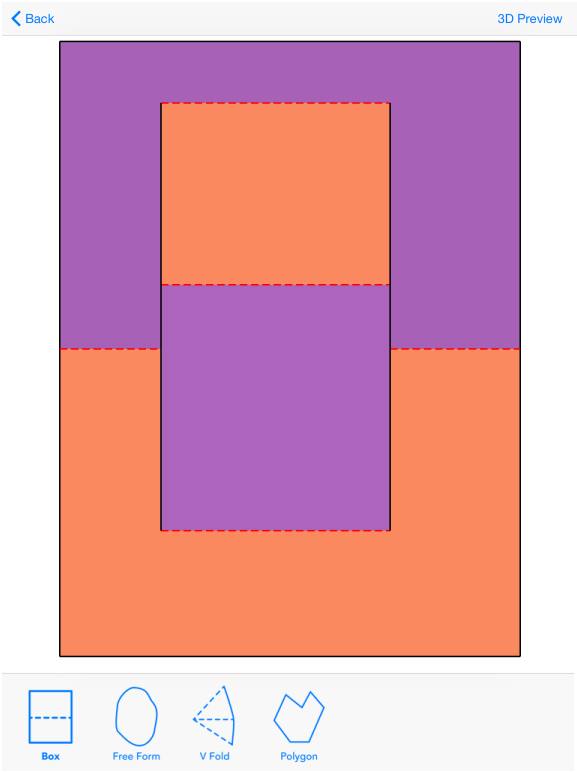


Figure 2.9: Plane shading is one of many visual aids in Foldlings.

IV. Interface Data Structures

We will refer to several data structures throughout the discussion of user interface design and implementation. These are the primary means of storing user input, and are processed by our algorithms to draw designs in 2D and simulate them 3D[10]. For a discussion of the algorithms that act on these features to form planes and the 3D simulation, see Allen [2015].

(a) Edges

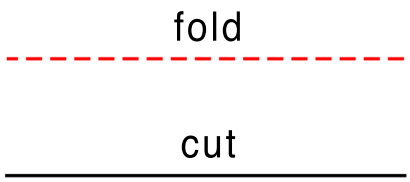


Figure 2.10: In Foldlings, cuts are displayed as solid black lines. Folds are displayed as dotted red lines. This convention is familiar to those who use traditional instructional books (Berenson [1972],3).

An edge represents a cut or fold. Edges are the basic building block of planes, and an integral element of all fold features. An edge is minimally defined by a start point, end point, and a a type (either cut or fold). This minimal definition represents a straight edge between two points. In addition, an edge can contain further information: the bezier path drawn to create it (for non-straight edges), and a reference to the plane or feature it is part of. Additionally, each edge contains a reference to its "twin" edge.

Twin Edges

Although it is often simplest to think of edges as cuts and folds created by the user, the reality in Foldlings is slightly more complicated. For each edge that the user creates using

[10]This section only describes the primary data structures necessary for constructing fold and patterns from user input, detecting planes, and determining the relationships between features — not the systems for drawing features in 2D or 3D. For a discussion of 2D drawing, see Chapter 3, section I., Interface Implementation, on page 41

a tool, two edges are created. We create edges with direction, such that there is an edge from the start point to the end point of the edge, and another edge starts at the endpoint and has the reverse path of the original edge. This distinction is necessary for the plane detection algorithm described in Allen [2015], but must also be taken into account whenever edges are processed. For example, when rendering the 2D view of a sketch, we skip drawing the twins of edges already drawn, which reduces drawing work by half.

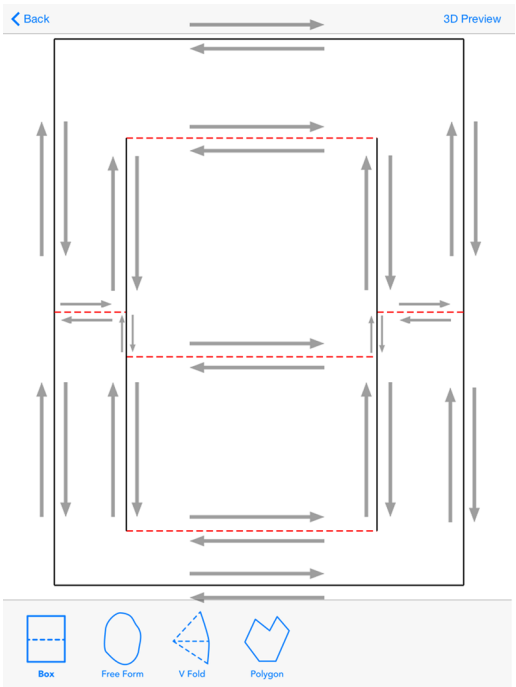


Figure 2.11: This sketch contains 34 edges, with orientations shown by the overlaid gray arrows.

Driving Folds

A driving fold is not a special type of edge, but rather a relationship between an edge in one feature and a feature "spanning" that edge. A feature is said to span a fold when it is drawn on top of an existing fold, so that it has horizontal folds both above and below the fold it spans — as shown in figure 2.12.

An edge can be the driving fold for more than one feature, but each feature has only one driving fold (if there are multiple potential driving edges at the same height, the left-

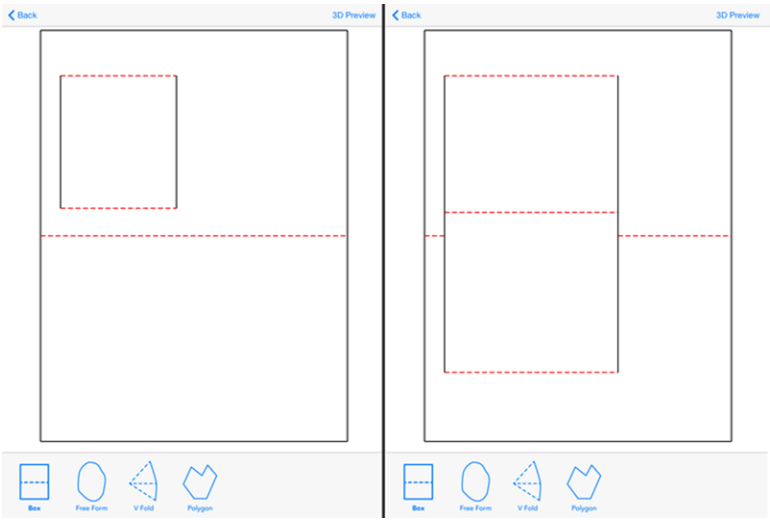


Figure 2.12: Left: a box fold mid-drag. The feature does not have a driving fold. Right: a box-fold after the user has released the touch. The feature's driving fold is the master card's middle horizontal fold.

most edge is selected). That is, there is a many-to-one relationship between children and their parent feature. The driving fold is important for calculating parent-child relationships between features: a feature's parent is the feature that contains it's driving fold[11]. These parent-child relationships are described in more detail in the section Hierarchy on page 38 and in Allen [2015].

Fold Orientation

Traditionally, kirigami patterns indicate fold direction: "mountain/hill" or "valley". These folds form angles in opposite directions — mountain folds are pinched away from the paper surface, while valley folds are pinched into the surface (Chatani [1986]). In Foldlings, edge orientations are determined by traversing the plane tree structure described by Allen [2015].

(b) Planes

Planes are an enclosed shape, bounded by edges. Plane are detected from edges by traversing the directed edge graph, as described in Allen [2015]. They are drawn as colored areas

[11]The exception to this rule is holes — a hole's parent is the plane that contains it.

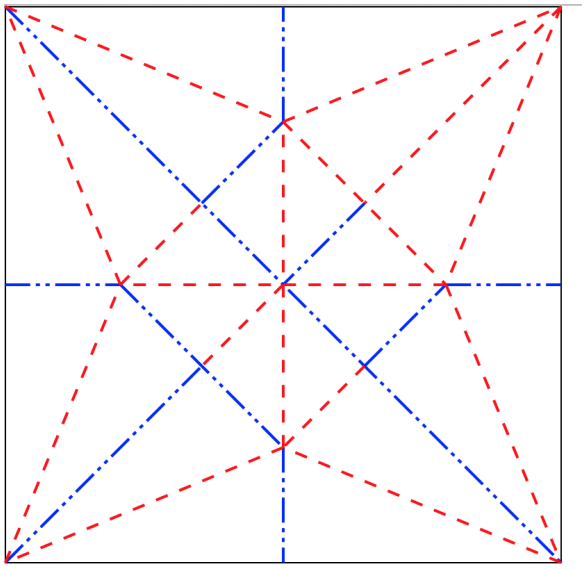


Figure 2.13: Kirigami fold pattern (mae [2015]).

in the two-dimensional sketch, and simulated in the 3D preview as extruded shapes that rotate about a pivot point. In order to simulate the planes in 3D, we construct parent-child relationships between the planes, which determine how they move during simulation. Each plane's parent is the plane that contains its topmost edge, as shown in figure 2.15.

(c) Fold Features

The central data structure of Foldlings is the fold feature: a representation of a shape drawn by the user that folds in 3D. Each fold feature is a single design element — and can be individually created, modified, and deleted. There are five subclasses of FoldFeature: MasterCard, BoxFold, FreeForm, Polygon, and V-Fold, representing differences in drawing behavior, geometry, and appearance (the differences are described in detail below). Each of these features is a subclass of the base FoldFeature class.

All fold features have functionality in common:

- Each feature contains a list of edges in the feature — cuts and folds, including twins.
- Each feature has a driving fold — in the case of unconnected features (i.e the master card and holes), the driving fold is nil.

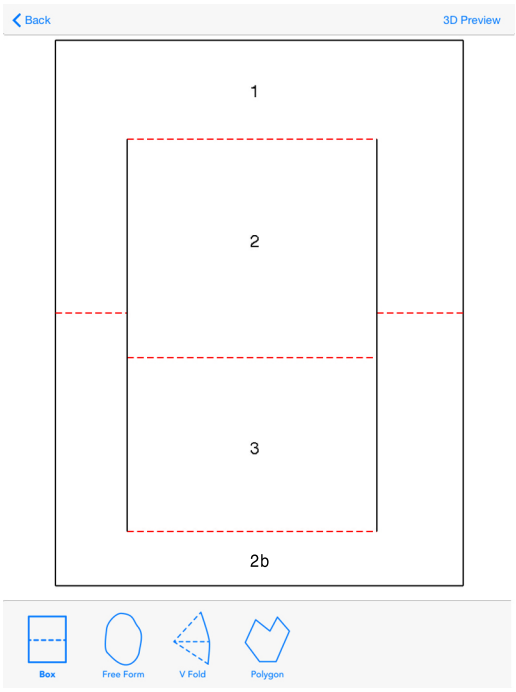


Figure 2.14: Planes in a simple sketch, numbered by ancestry. Starting at the root plane 1, each successive plane is the child of the previous numbered plane. 2b is the child of plane 1.

- Each feature can be deleted from the sketch, "healing" the sketch by closing gaps left in any existing cuts and folds.
- Features implement Apple's NSEncoding protocol, allowing them to be serialized to a file on the device and restored from the saved file.
- Each feature can provide a list of current "tap options" — actions that can be performed on the feature given its state.
- Each feature can perform hit-testing: given a point, it can determine whether that point is inside or outside the feature.

Master Card

Each sketch always contains a single master feature, which is the ancestor of all other features. The master feature consists of two planes with a valley fold between them. Users do not create features of this type — each sketch begins with one. All of the edges in

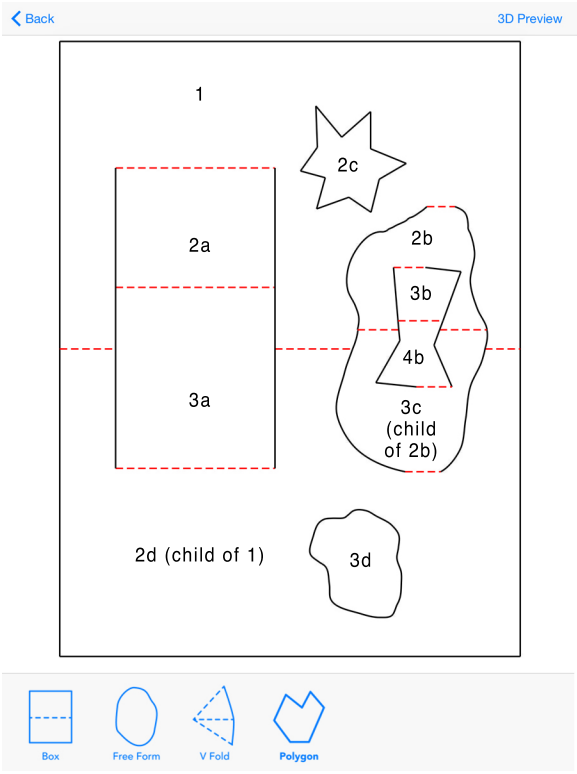


Figure 2.15: Planes in a more complex sketch, numbered by ancestry. Starting at the root plane 1, each successive plane is the child of the previous numbered plane. Letters indicate branches within the plane tree (I.e. 3a is the child of 2a).

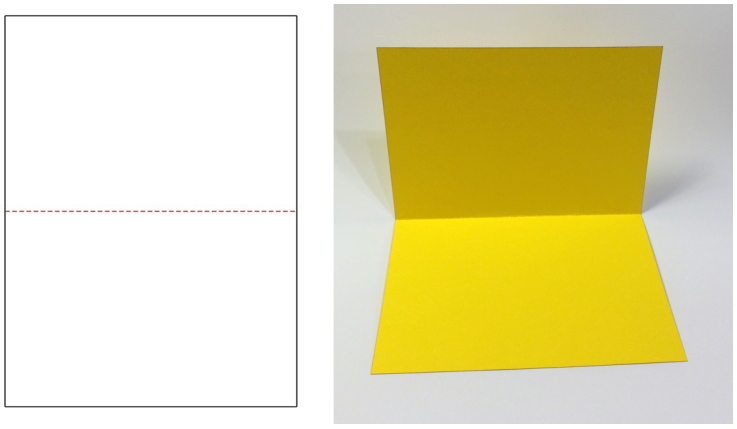


Figure 2.16: Left: fold & cut pattern of the master feature. Right: laser-cut model of the same.

the master feature are marked with a flag indicating that they belong to the master feature, because master edges and planes are sometimes treated differently than normal edges. For example, the parent-child relationships between planes are constructed by starting at the top plane in the master feature, determined by edge type and height.

Box Fold

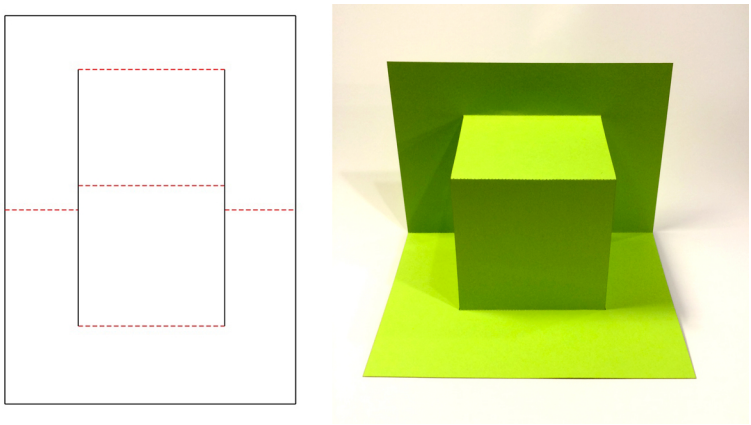


Figure 2.17: Left: fold & cut pattern of a box fold feaure. Right: laser-cut model of the same.

A box fold consists entirely of straight edges, and can be constructed from two points: the top left point, and the bottom right point. The middle fold position is determined by the position of the driving fold, as described in Chapter 3, section VI. , Constraints on Fold Features on page 60. Box folds are only valid if they span a driving fold.

Free Form

Freeform shapes are defined by a single, closed path. When the feature is completed (by releasing the touch), the shape is truncated, horizontal folds are added, and the path is split into multiple edges (assuming the shape spans a fold). The curved path is defined by a set of "interpolation points" — points captured by sampling touch positions while a user draws a shape on the screen. A path is interpolated between these points using the Catmull-Rom algorithm (Catmull and Rom [1974]).

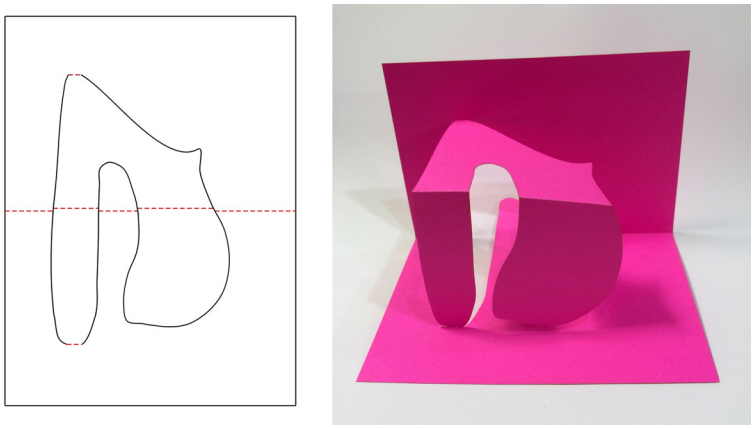


Figure 2.18: Left: fold & cut pattern of a freeform feature. Right: laser-cut model of the same.

Holes are a special case of FreeForm shapes, and are cut out from the final design, rather than simulated as a separate plane. FreeForm shapes that do not cross a fold are considered holes — drawn in white in the 2D sketch and rendered as subtractions from planes in the 3D view.

Polygon

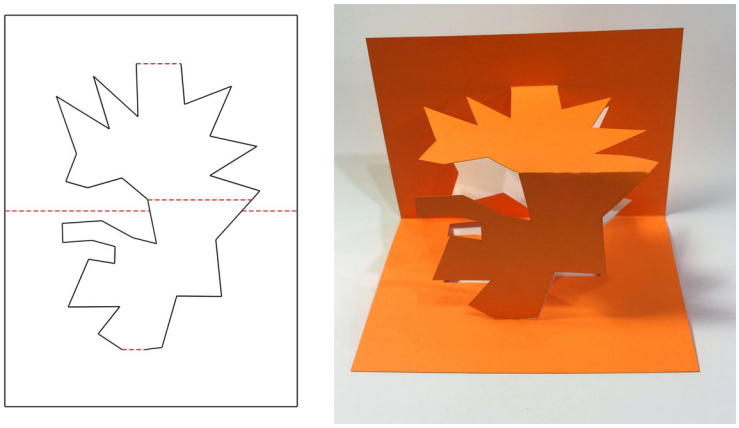


Figure 2.19: Left: fold & cut pattern of a polygon feature. Right: laser-cut model of the same.

Polygons are created from a list of "tap points" constructed from user input. As in free-form shapes, these points are connected with a bezier path, and are truncated if they have a driving fold. For polygons, this path consists only of straight line segments. Unlike

interpolation points in freeform features, tap points in polygons can be moved at any time during the drawing process.

Like freeform shapes, polygons that do not have a driving fold are considered holes.

V-Fold

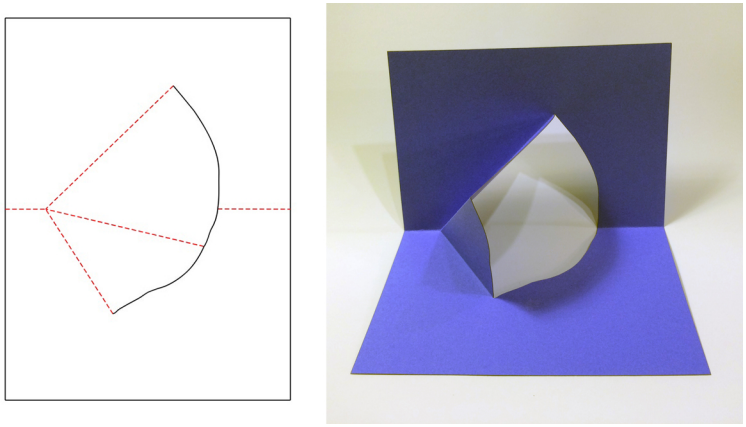


Figure 2.20: Left: fold & cut pattern of a v-fold feature. Right: laser-cut model of the same.

V-folds are partially defined by a path that crosses the driving fold, called a "vertical cut." This path can be any arbitrary shape that crosses the driving fold once. They are fully-defined by adding a point on the driving fold. From this point, we construct three diagonal folds, two to the top and bottom of the vertical cut, and one to a point that intersects with the vertical cut at a point calculated to make a valid 90-degree feature[12].

V-folds are only valid if they have a driving fold, and their vertical cut intersects the driving fold exactly once.

Hierarchy

Fold features have two types of hierarchy. The first is feature hierarchy: each feature can have other features as children. When a feature is drawn spanning a fold, its driving fold is set as the fold it spans, and its parent is the feature that contains this driving fold. Feature

[12] See Chapter 3, section VI., Constraints on Fold Features on page 60.

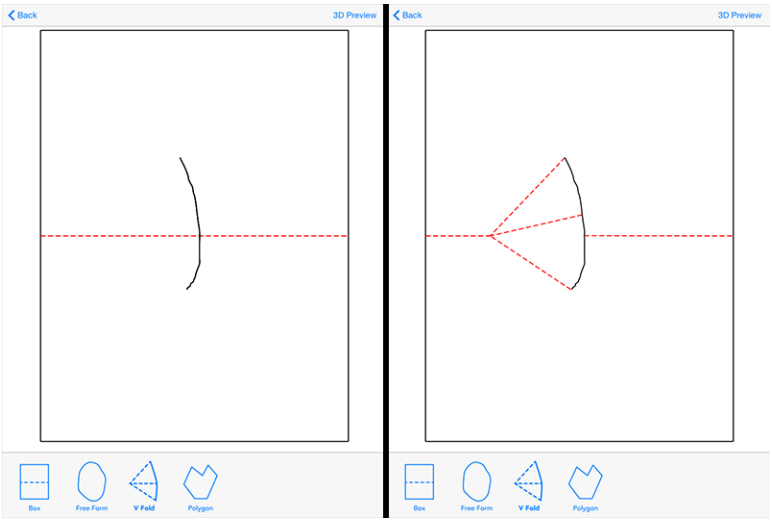


Figure 2.21: Left: an unfinished v-fold, consisting only of a vertical cut. Right: a v-fold after defining the point on the driving fold to create diagonal cuts.

hierarchy allows us to consider each feature individually or as a chain of features. For actions that only affect a single feature, we need only consider the edges in the feature and it's driving fold. We can also traverse the tree to perform actions on a chain of features. The second type of hierarchy is parent-child relationships between planes. Each plane has one or more children, forming a branching tree that starts at the top plane of the master card and ends with the master card's bottom plane as one of its leaf nodes. This hierarchy is described in Allen [2015], and is most important for rendering the scene in 3D.

(d) Sketches

A sketch is the representation of the user's drawing — its primary role is as a collection of features. It also contains information about the current drawing state, and the state of user interaction (for example, which features are currently being modified, and which tool is selected).

V. Saving

Each time users leave a sketch, Foldlings archives the sketch to a file, from which the session can later be restored. Users restore a sketch by tapping on one of the cards on the main screen. These cards can be deleted through a long press on the sketch, which presents an option to remove the saved file. This convention is familiar to iOS users.

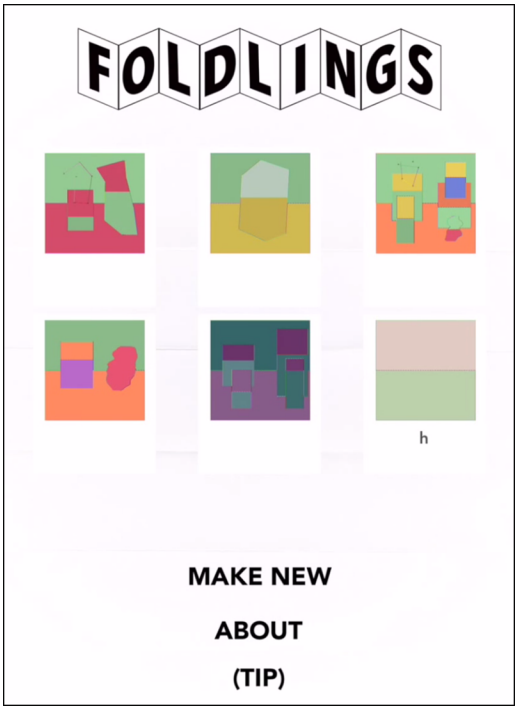


Figure 2.22: Saved sketches displayed on the main screen.

In addition, we also save fold patterns for user designs to Amazon S3. Saving these files allows us to debug user problems remotely and see the kinds of cards users attempt to make with our software. Along with user testing, these captured sketches have been instrumental to our design process.

Chapter 3

Algorithms and Implementation

I. Interface Implementation

One of the core components of Foldlings is the interface. To create features, we capture touch input, display a preview of fold features as the user creates them, and add created features to the fold pattern. This system is outlined in Figure 3.1.

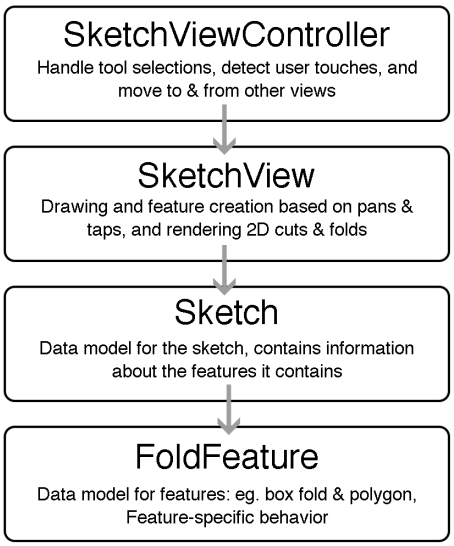


Figure 3.1: Relationship between interface classes: a SketchViewController manages a SketchView that contains a Sketch that contains FoldFeatures.

Although this structure borrows heavily from the Model-View-Controller design pattern, we do not follow the pattern strictly. The MVC paradigm often becomes muddy, with much information shared between the separate modules (Veit and Herrmann [2003]). In our case, we separate modules not purely based on MVC encapsulation, but based on functionality. For example, rather that strictly separating all touch handling into the view controller (or all drawing into the view), these responsibilities are shared between the classes as needed to perform their roles. See section (a), below, for more information.

(a) Touch Handling

In order to create features, we first need to capture touch input. Foldings handles two types of touches: pan gestures and tap gestures. Apple describes gesture recognizers in the documentation for the UIGestureRecognizer class:

> Gesture recognizers convert low-level event handling code into higher-level actions. They are objects that you attach to a view, which allows the view to respond to actions the way a control does. Gesture recognizers interpret touches to determine whether they correspond to a specific gesture, such as a swipe, pinch, or rotation. If they recognize their assigned gesture, they send an action message to a target object. The target object is typically the view's view controller, which responds to the gesture... (Apple [2015]).

In Foldlings, all gestures are captured by the SketchViewController , which passes unhandled gestures on to classes lower down the chain. In general, all taps are handled at the view controller level; the only exception to this rule is the Polygon feature, described in Chapter 2, section III. (a) Feature Interactions on page 21.

(b) Tool Selection

Tool state is maintained by the view controller. The SketchViewController class handles taps on tool buttons and features, and passes pan gestures to the SketchView. The function called on the SketchView in response to pan gestures depends on what feature is selected. The function below in SketchView captures pan gestures, and calls the appropriate function depending on the tool selected and the drawing state.

```
func handlePan(sender: AnyObject) {
    if(sketch.tappedFeature != nil){
        switch(sketch.tappedFeature!.activeOption!){
        case .MoveFolds:
            handleMoveFoldPan(sender)
        default: break
        }
    }
    else{
        switch (sketchMode) {
        case .BoxFold:
            handleBoxFoldPan(sender)
        case .FreeForm:
            handleFreeFormPan(sender)
        case .VFold:
            handleVFoldPan(sender)
        case .Polygon:
            handlePolygonPan(sender)
        default:
            break
        }
    }
}
```

Within the specific function in SketchView, pan gestures are converted into feature edges by listening to $touchesBegan$, $touchesMoved$, and $touchesEnded$. When a pan begins, we create a new feature of the appropriate type and set it as the currently active drawing feature. When the pan is updated, we add and/or modify edges in the active feature. When the pan ends, we make final modifications to edges, validate the feature, and add it to the sketch.

Of course, the implementation of pan delegate methods varies widely between features.

For example, a diagonal pan with the box fold tool selected would create folds and cuts to form a box between the start and end point, whereas the same touch in the freeform tool would create an invalid feature consisting of a line.

(c) Fold Feature Preview

While the user is drawing, the SketchView displays a preview of the feature in progress, to give feedback on the drawing. Meanwhile, the SketchView also displays the previously drawn features that have been added to the sketch. Although the display of cuts and folds for the current feature is visually similar to those of the final feature, the method of drawing a preview of the currently active feature varies from the generation of final feature edges.

We store the currently active feature separately; it is not added to the list of features until it is completed and passes validation. Thus, its edges are drawn by the function that draws all edges currently in the sketch. For performance reasons, we display a preview of the active feature on top of all features in the sketch. Modifications to existing features, and computationally expensive operations such as truncation for freeform shapes, are performed only when the feature is added to the sketch. During drawing, we display the set of preview edges stored in the feature, which sometimes differ from the final feature edges.

For box folds, we display a preview of all the edges, but do not occlude the middle fold until the feature is completed. For freeform features, we do not perform truncation until the feature is completed (and spans a driving fold)[1]. As a preview, the SketchView shows the user's touch path as a cut. For polygons, we display control circles at vertices. These circles indicate that the vertices are draggable, modifying edge endpoints dynamically. For v-folds, we display a dynamic preview of the vertical cut and top and bottom diagonal folds. The middle diagonal angle is not calculated until the feature is added to the sketch.

[1] A description of truncation can be found in Chapter 3 section II. Tool Implementation on page 46

(d) Feature Creation

When the user completes a feature, we add it to the sketch[2]. Adding a feature to the sketch requires modifying existing features in the sketch, constructing parent-child relationships, and recalculating planes from edges. How these tasks are achieved depends on the specific fold feature type.

The specific implementations of the FoldFeature superclass are described in section II. Tool Implementation on page 46. The Sketch class contains methods for adding and removing features from the sketch. It also contains lower-level functions for adding, removing, and replacing edges. These methods are typically called by feature-specific methods that modify the sketch, such as splitFoldByOcclusion.

In addition to adding a feature to the sketch, we also calculate the planes as described by Allen [2015]. This allows us to shade planes based on orientation. After calculating planes from edges, relationships between planes are stored in the plane tree within the sketch. Within getPlanes, plane orientations are set by traversing the plane tree, alternating between vertical and horizontal.

[2] Assuming the feature is valid. See Chapter 3, section V. Validity on page 58.

II. Tool Implementation

Below is the definition of the FoldFeature superclass — all features created using Foldlings can override these methods to provide specific functionality. This structure holds properties common to all features, such as a list of edges and parent-child relationships between features. It also contains several functions that allow the other classes to modify the feature and sketch. For further discussion of the FoldFeature data structure, see Chapter 2 section IV., Interface Data Structures, on page 30.

```
var horizontalFolds:[Edge] = [] //list of horizontal folds
var featureEdges:[Edge]?        //edges in a feature
var children:[FoldFeature] = [] // children of feature
var drivingFold:Edge? // driving fold of feature
var parent:FoldFeature? // parent of feature
var startPoint:CGPoint?
var endPoint:CGPoint? // start and end touch points

/// splits an edge around the current feature
func splitFoldByOcclusion(edge:Edge) -> [Edge]
{
//by default, return edge whole
return [edge]
}
/// features are leaves if they don't have children
func isLeaf() -> Bool
{
return children.count == 0
}
/// options or modifications that can be made to the current feature
func tapOptions() -> [FeatureOption]?
{
  return [FeatureOption.PrintPlanes, FeatureOption.PrintEdges,
  FeatureOption.ColorPlaneEdges, FeatureOption.PrintSinglePlane]
}
/// whether a feature is drawn over a fold, determines whether
/// a fold can be the driving fold for a feature
  func featureSpansFold(fold:Edge)->Bool
{
  return false
}
/// returns and calculates planes in a feature
func getFeaturePlanes()-> [Plane]{
  return featurePlanes
```

```
}
/// whether a feature contains a point
/// needs to be overridden by subclasses
func containsPoint(point:CGPoint) -> Bool{
  return self.boundingBox()?.contains(point) ?? false
}
```

Of these functions, the most complex are featureSpansFold and splitFoldByOcclusion. FeatureSpansFold is the test to find the driving for a feature — i.e. the fold over which a feature is drawn. The implementation of this function varies by feature, but the general algorithm is as follow:

1) Test for intersections between the shape's outer path and the given fold.

   (a) If an even number of intersection points are found, return true, otherwise, return false. Fold features with multiple potential driving folds are consider invalid.

SplitFoldByOcclusion takes as a fold as its input, typically the driving fold for the feature. The function splits the given fold into multiple pieces, removing sections of the edge that would lie inside the bounds of the feature on which the method is called.

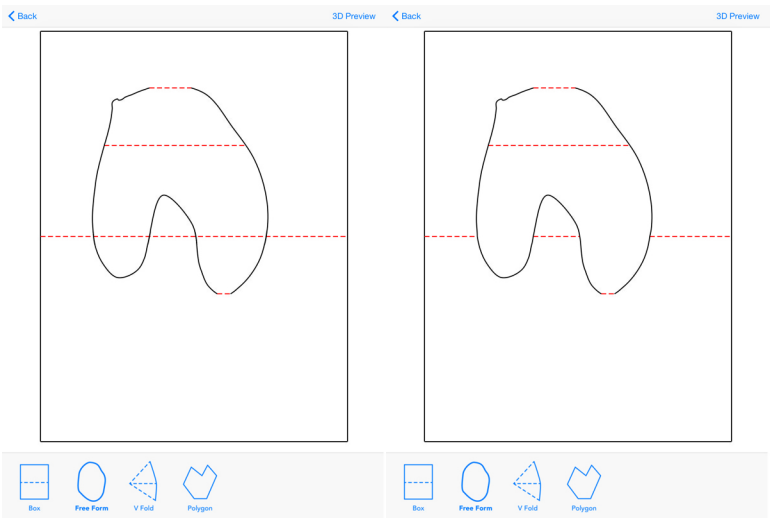


Figure 3.2: Left: a freeform shape before performing SplitFoldByOcclusion. Right: a freeform feature after performing SplitFoldByOcclusion.

(a) Box Fold

We can analytically determine intersection points for box folds, which makes testing for fold spanning and other operations very fast. We dynamically update edges for this feature (including the middle fold), as the user drags out the box. Box folds are the only feature for which we can display a live preview of the middle fold, as a result of the simple intersection tests.

(b) FreeForm

A freeform shape consists of a series of interpolation points — through which we construct a curve using the Catmull-Rom algorithm (Catmull and Rom [1974]). We capture interpolation points as a function of touch velocity. That is, when the user draws more quickly, we capture more interpolation points closer together. This allows us to capture the entire drawing with a similar level of detail throughout, and correct for the gesture recognizer sending relatively more frequent updates when the touch is moving more slowly.

However, the Catmull-Rom algorithm only draws a full path when the start and end points of the curve are coincident, so we manually construct straight line segments to and from the touch start and end points for unclosed freeform shapes. We use an alpha value of 1.0, which we found to be the closest to the intended touch shape through informal user studies.

Truncation

Truncation is the process of converting a closed free-form shape into a valid feature with a driving fold. To truncate a shape, we first find locations for top and bottom folds that will yield folds longer than the minimum edge length. Then, we split the existing path at those intersection points, and remove paths that lie outside the top and bottom folds of the new shape. Thus, we "truncate" the shape, adding horizontal folds and trimming edges that lie

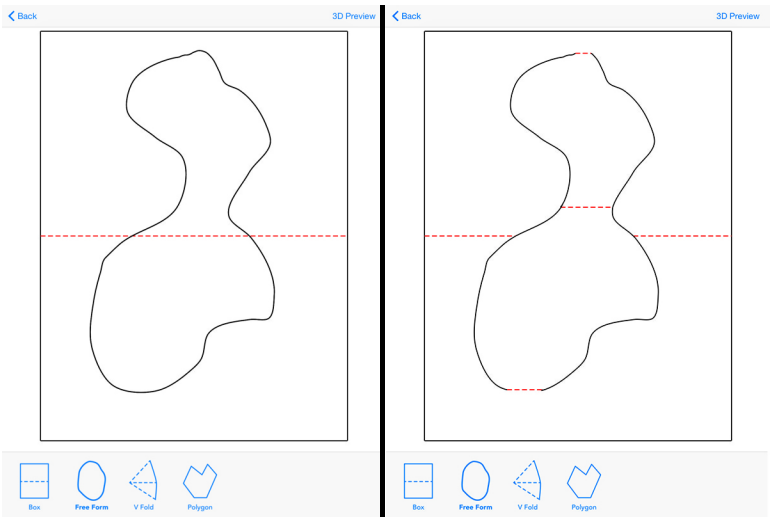


Figure 3.3: Left: A freeform feature before truncation. Right: A freeform feature after truncation

outside the new shape. Finally, we place the middle fold as described by the geometric constraints in Chapter 3, section VI., Geometric Constraints, on page 60. If the user desires

different truncation positions, they can drag folds after the feature is added to the sketch.

Data: *path*, the bezier path for the freeform shape

Result: edges for truncated shape

create *scanline* at top of bounding box for *feature*

while *scanline* above bottom of *feature* **do**
    intercepts ← intersection points between *scanline* and *path*
    if *intercepts* not nil and distance between *intercepts* > min edge length **then**
        fragments ← *path* split by *intercepts*
        foreach *fragment* in *fragments* **do**
            if *fragment* center is above top fold or below bottom fold **then**
                remove *fragment* from *fragments*
            end
        end
        break
    end
    translate *scanline* down
end

repeat *scanline* operation from bottom to find bottom fold

calculate middle fold position

folds ← top, middle, and bottom folds

return *folds* + *fragments*

Algorithm 1: Truncation

This algorithm depends on the ability to calculate intersections between bezier paths. We use a bitmap approximation of the intersection point between bezier paths, because calculating intersections between arbitrary curves is computationally more expensive. We use the ANPathIntersection library for intersections, which draws the two paths into a buffer, and then finds points where both paths have pixels filled (ANP [2015]). This fast approximation allows us to perform the many intersection tests required for truncation quickly, minimizing the delay after creating a feature.

After finding the intersection points, we split the existing path at the intersection points.

The freeform shape's path is composed of many cubic bezier curves output by the Catmull-Rom algorithm. In order to split the path, we must first find the closest cubic bezier curve to the intersection point, and then find a value, t, that gives interpolated position at the intersection point. First, we iterate through all bezier segments in the path, comparing representative points to find the closest curved segment to the intersection point. Within that segment, we recursively subdivide the curve to find a t value very close to the intersection point (Phillips [1997]). For example, we start with t value of 0.0 (at the beginning of the curve), 0.5 (at the middle of the curve), and 1.0 (at the end of the curve). We evaluate the curve to find the point at each t value. Next, we measure the distance between the interpolated points and our intersection point. Of these three points, we then take the two closest points to the intersection point, and repeat the process to find the new search area. For example, t at 0.5, 0.75, and 1.0, and then t at 0.5, 0.625, and 0.75 would approach an intersection point found at t = 0.6. After performing this process several times, we reach a point very close to the interpolated point (arbitrarily, we stop when the distance between the intersection point and any interpolated point is less than five pixels).

(c) Polygon

Polygons are very similar to freeform shapes. The main difference between polygon and freeform shapes is that the intersection tests for polygons are much cheaper. To calculate intersections between polygons and folds, we can use a simple system of equations, rather than the bitmap intersection technique described above.

The interpolation points are vertices of the polygon, and can be modified by the user at any time. As the user drags points, we delete and recreate edges dynamically to match the new vertex positions.

(d) V-Fold

V-Folds are constrained by the angle restriction described in Chapter 3, section VI., Geometric Constraints, on page 60. To satisfy this constraint, we first construct a line along one of the diagonal folds. We then rotate the line about the point on the v-fold's driving fold, and perform the intersection tests described in section (b), Truncation, on page 48 to find the intersection point for the middle fold. Lastly, we perform path splitting (also as described above) and add the feature to the sketch.

III. Self-intersecting Paths

In order to be rendered by SceneKit in 3D, paths cannot have self intersections. Thus, we attempt to repair self-intersecting paths when adding features to the sketch. Self intersections occur in two ways: the user creates a self-intersecting path, or paths self-intersect as a result of imprecision in performing intersections or capturing touch interpolation points.

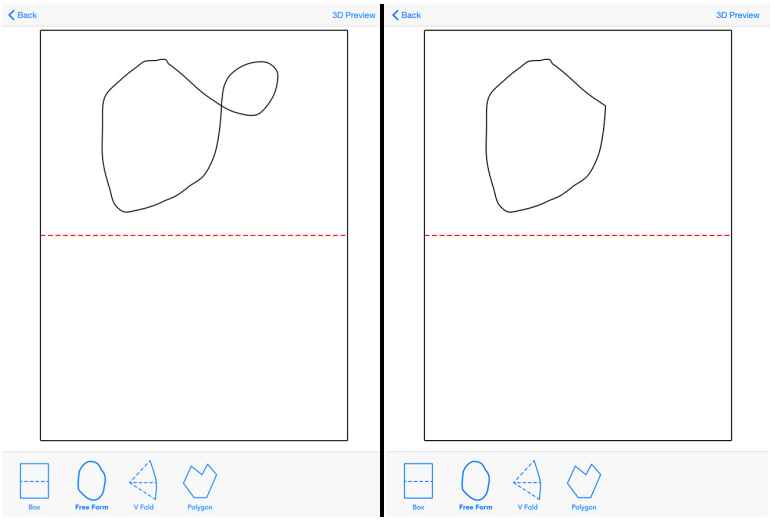


Figure 3.4: Left: a feature with a self-intersection before processing. Right: the feature after the self-intersection has been resolved.

```
segments ← bezier path discretized into straight line segments using adaptive
subdivision
sanitizedSegments ← empty array
for i ← 0; i < segments.length; i++ do
    //compare with all segments after current segment
    for j ← i; j < segments.length; j++ do
        if segments[i] intersects segments[j] then
            remove segments[i..j] from path
            i ← j //skip segments inbetween intersecting segments, thereby repairing
            the "loop"
        end
        sanitizedSegments.append(segments[i])
    end
end
return sanitizedSegments
```

Algorithm 2: Self-intersecting path repair

A convoluted design with many overlapping self intersections can fail to resolve to a valid shape[3]. In cases where our algorithm fails, we display an error and do not add the feature to the sketch.

[3]I.e. a valid shape being one that does not intersect with itself and has more than zero enclosed area.

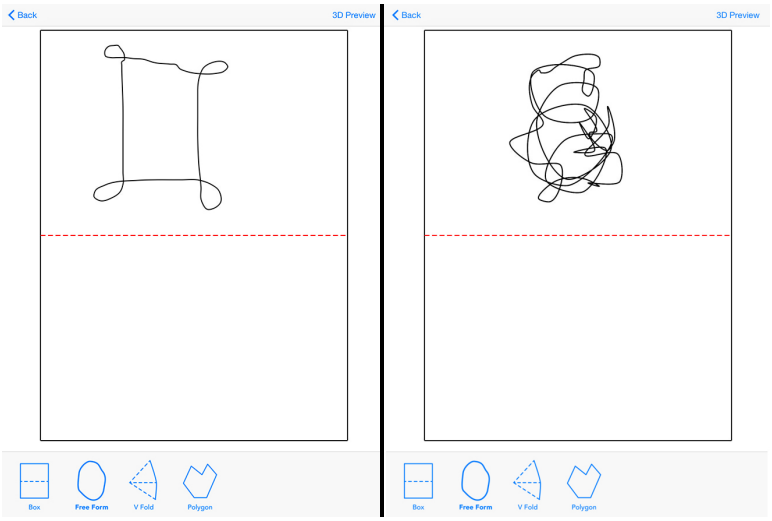


Figure 3.5: Left: a feature with multiple self-intersecting loops, which our algorithm can repair. Right: a feature with multiple overlapping intersections, which our algorithm fails to repair.

IV. Intersections Between Features

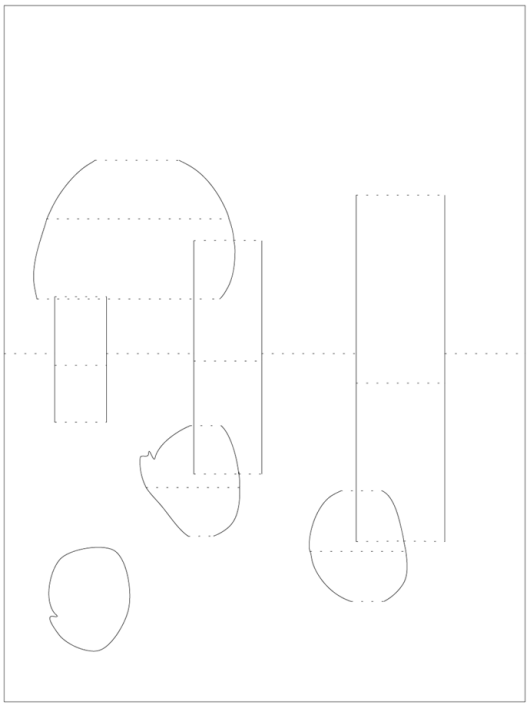


Figure 3.6: An attempted sketch with overlapping features.

As we observed at the Digital Arts Exhibition[4], users often attempt to draw features that overlap with existing features in the sketch. To allow users to perform these kinds of operations as expected, we allow for feature intersections.

Our intersection algorithm attempts to resolve overlapping features, by occluding existing edges with the new feature and removing interior edges. The new feature is drawn "on top" of existent features in the sketch. This is a complex problem, and is only partly implemented[5] due to the complexity involved in adding intersections to our system. Currently, we display an error message if the intersected feature will have any folds less than the minimum edge length after intersection. A more complete implementation would analyze the

[4]See Chapter 4, section I. on page 66.

[5]For example, we do not currently support intersections between all types of fold feature. In order to do feature-feature intersections in Foldlings, we transform all intersecting features into the same feature type, to simplify the process. At the current time, only freeform shapes and polygons are supported.

resultant geometry for potential problems.

```
foreach feature in sketch do
    if feature intersects with current drawing feature then
        foreach edge in feature do
            intercepts ← all points of intersection between feature and current
            drawing feature
            if intercepts not nil then
                fragments ← edge split by intercepts
                foreach fragment in fragments do
                    if center of fragment inside bounds of current drawing feature
                    then
                        remove fragment from sketch
                    end
                end
            end
        end
        reassign fragments as edges of feature
    end
end
recalculate planes for sketch
```

Algorithm 3: Feature Intersections

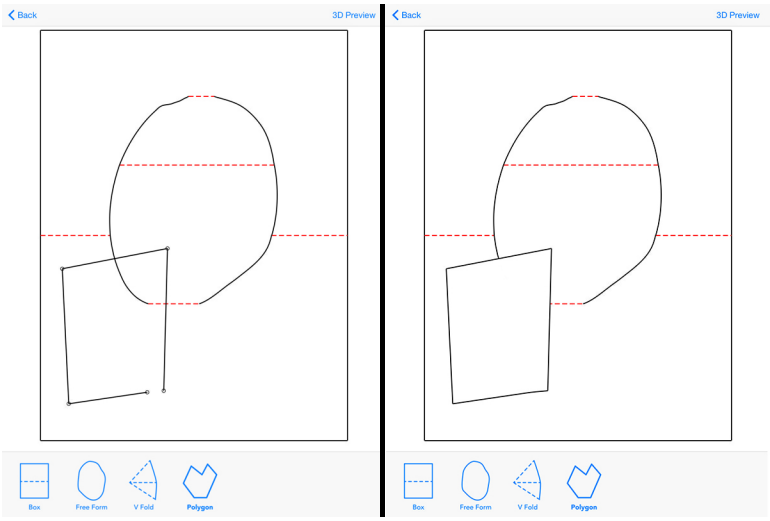


Figure 3.7: New polygon feature intersecting with an existing freeform shape.

V. Validity

One of the primary goals of our software is to keep the user's sketch in a valid state. Since our algorithms for plane detection and 3D simulation only succeed if the edges form valid shapes, it is essential to prevent invalid features from being added to the sketch. Therefore, each feature has a function with the following signature, that validates edges in a feature:

```
func validate() -> (passed: Bool, error: String)
```

The tool/template-based system is the primary means of insuring that user input is valid. The validate function is a secondary system, which attempts to fix errors in user input, and then returns an error message if the feature could not be validated. In case of errors, the feature is removed from the feature and we display a message using the warning system. For example, the $validate()$ function of v-fold features reads as follows:

```
override func validate() -> (passed: Bool, error: String) {
 let validity = super.validate()
 if(!validity.passed){
    return validity
 }
 // clever test for concave paths: close the vertical cut's
 // path and test whether vfold end point is inside it
 var testPath = UIBezierPath(CGPath: verticalCut.path.CGPath)
 testPath.closePath()
 if(testPath.containsPoint(diagonalFolds[0].end)){
    return (false,"Angle too shallow")
 }
 if(!tooShortEdges().filter({
    \$0.kind == Edge.Kind.Fold
    }).isEmpty){
        return (false,"Edges too short")
 }
    return (true,"")
 }
```

Here, we first validate using the superclass, performing checks that apply to all features. Then, we perform checks specific to v-fold features. V-folds are invalid if they form a

concave angle. To test for this, we construct a straight line between the start and end of the vertical cut, and then test whether the intersection point with the driving fold lies within the closed shape. This feature type is also invalid if any of its folds are shorter than a minimum edge length.

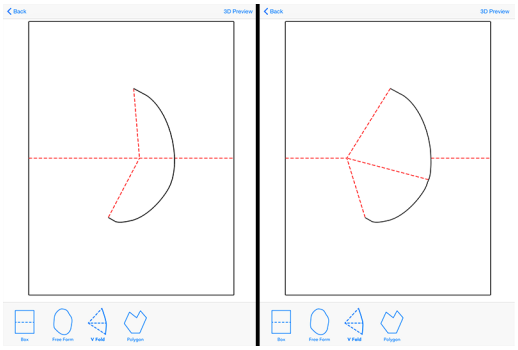


Figure 3.8: Left: an invalid, concave v-fold. Right: a valid, convex v-fold.

VI. Constraints on Fold Features

Constraints are central to the implementation of Foldlings' algorithms, which help our software provide an accurate preview of sketches, and ensure that designs created with Foldlings will fold correctly. As an educational tool, one of the primary benefits of our software might be the development of an intuitive understanding of the limits of paper.

(a) Geometric Constraints

Several geometric constraints drive Foldling's algorithms. These constraints are the core reason for the difficulty of creating designs manually; a key advantage of our system is that these constraints are resolved automatically.

Box Fold

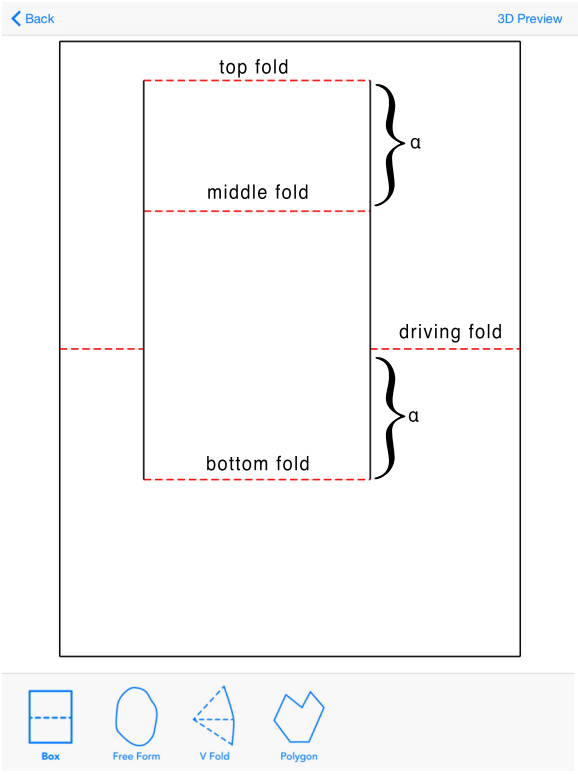


Figure 3.9: Geometric constraints for box fold features

A box fold is constrained by a relationship between its folds. Given a top fold and bottom fold at fixed height, with a driving fold with a height between the other two folds, the vertical distance between the bottom fold and the driving fold must be equal to the distance between the top fold and the middle fold of the feature (see figure 3.9).

This 90-degree angle constraint applies equally to freeform and polygon features (at least, those that span a fold).

Freeform

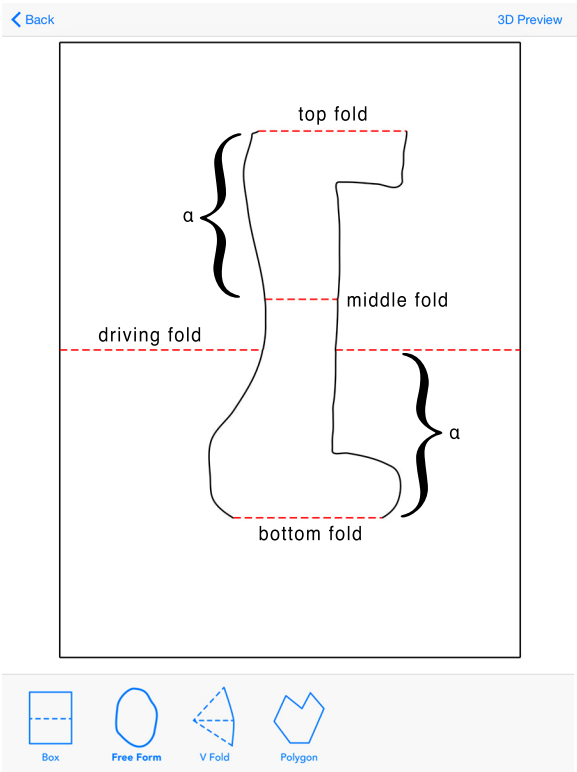


Figure 3.10: Geometric constraints for freeform features

Freeform fold heights are calculated similarly to those in a box fold. After performing truncation[6] to place the top and bottom folds of the freeform shape, we apply the 90-degree constraint to place the middle fold in the feature. Of course, holes are not bound by this constraint, because they do not have a driving fold (see figure 3.10).

[6]Described in section II., Tool Implementation, on page 46.

As with all features, validity constraints are separate from geometric constraints. Freeform shapes that intersect themselves do not have a place for the middle fold, but by performing validity checks before solving for geometric constraints, we avoid many problems and edge cases that would otherwise occur.

Polygon

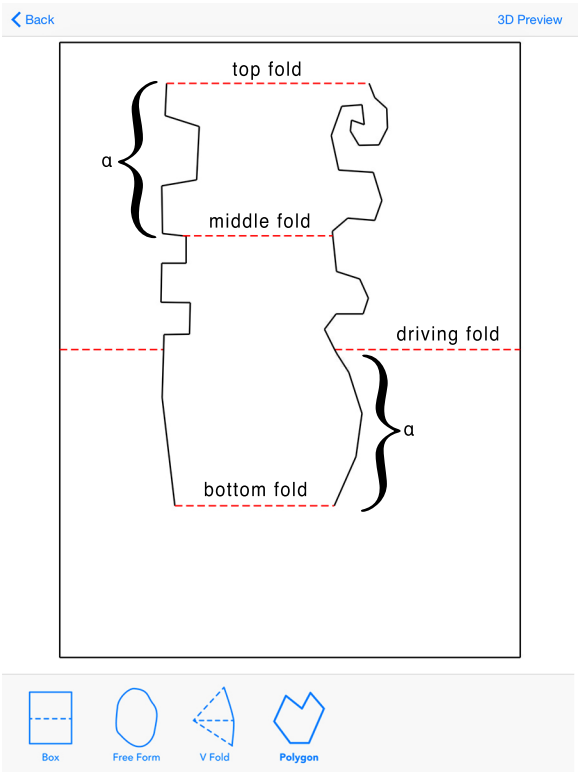


Figure 3.11: Geometric constraints for polygon features

The same constraints that apply to freeform shape apply to polygons. Although the edges are constructed differently, polygons are essentially a subset of freeform shapes, composed only of straight lines (see figure 3.11).

V-Fold

Although a v-fold also folds from zero to 180 degrees with the card, its planes move at non-orthogonal angles. The constraint on v-fold features is therefore based on angle rather

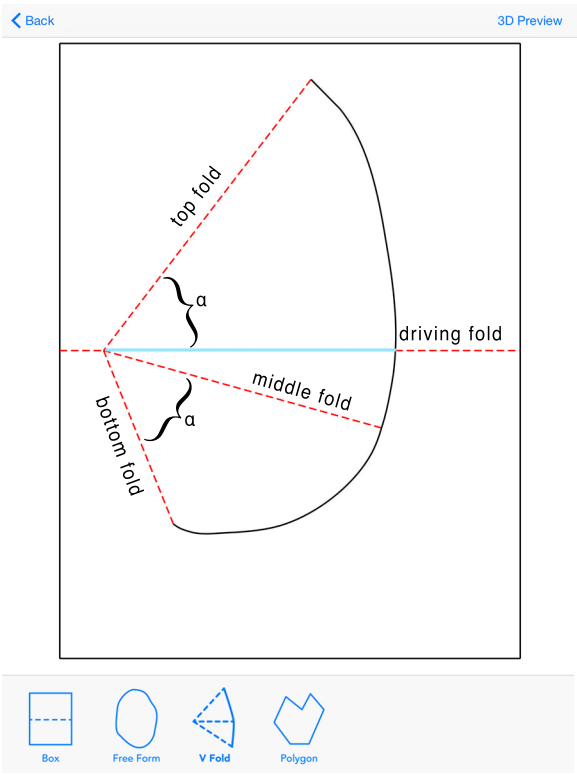


Figure 3.12: Geometric constraints for freeform features

than height.

The simplest case of v-folds — and the one most popularly constructed by our user testers using manual methods — is a symmetric v-fold. In this case, the top and bottom angles of the fold are equal. The shape approximates an isosceles triangle.

In the more complex case, the angles between the two diagonal folds and the driving fold differ. Andrew Glassner demonstrates that for any "single slit mechanism," the angle from the driving fold to either the top or bottom fold must be equal to the angle between the opposite fold and the middle fold (Glassner [1998], 3). This constraint is only solvable if the sum of the angles between each diagonal fold and driving fold is less than 180 degrees. Intuitively, this constraint is not dissimilar from the constraint for box folds — the position of the middle fold is constrained by the driving fold's relationship to the top and bottom folds (see figure 3.12).

(b) Physical Constraints

In addition, the physicality of paper places constraints on where cuts and folds can be placed. Depending on the manufacture method, there is some minimum line length that can be cut or folded, and some minimum distance folds and cuts must be apart. These depend on a number of variables ranging from paper thickness to manufacture method (for example, laser cutters have a higher tolerance for closely-drawn lines). Even within a specific technology, there is also a wide variation in cutting precision.

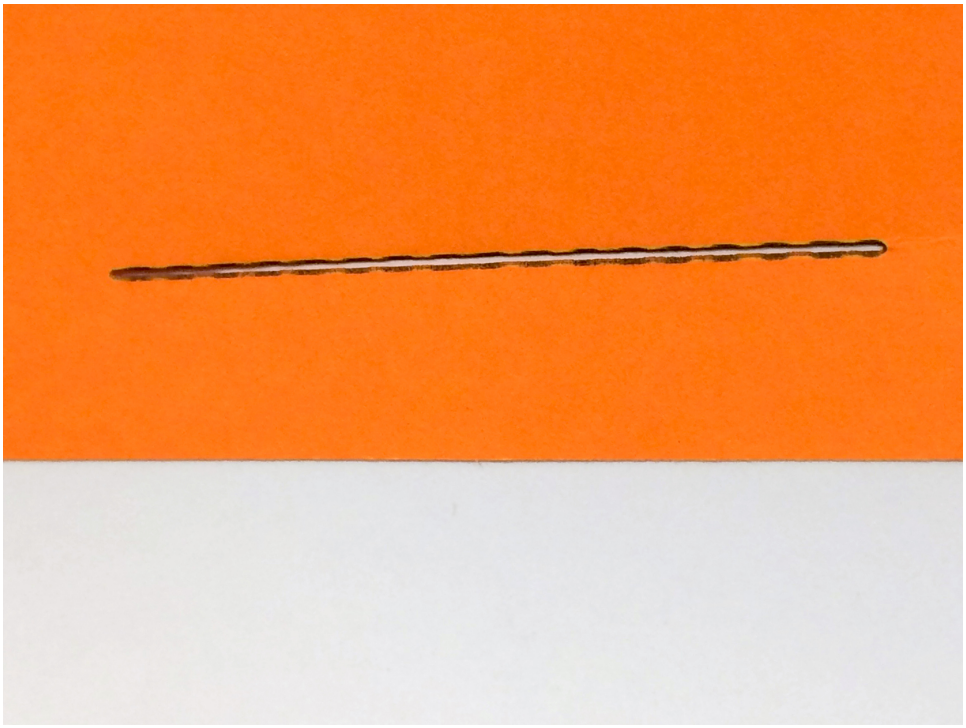


Figure 3.13: A malformed laser-cut fold. This is supposed to be a dotted line, but because the dots were too close together, they have merged to become a single cut.

In our software, we take this into account by incorporating a minimum edge length constraint. The minimum edge length is the minimum length for all edges in a feature. If the user creates a feature with edges less than this length, a warning will appear as described in section see Chapter III., section (d), Warnings and Errors, on page 25, and the feature will not be added to the sketch. This length constraint solves a variety of potential problems: it prevents users from creating features that are too small to tap, and ensures that edges can be

easily cut by hand or using automated methods. We use the same length as the minimum edge distance. In order to be valid, all edges in a feature must be at least this distance from all other edges. In addition, we adjust the line pattern for folds in the SVG output to prevent the problem shown in figure 3.13. Because the fold line pattern must be easy to crease while maintaining the card's integrity, there is a tradeoff between ease of folding (dots close together) and durability (dots further apart). We adjusted the line pattern to consist of short dots spaced out slightly further than the minimum edge distance. Although we discovered the line pattern through experimentation, it performs well on multiple laser cutters, paper types, and manual cutting. This pattern should work well for most common paper cutting methods.

Chapter 4

User Studies

While we performed frequent informal tests of our software with users, these often consisted of a single user. The user tests described in this chapter were larger-scale tests of key pieces of our software, and were a major driving force in our design process.

I. User Test at the Digital Arts Exhibition

Figure 4.1: Foldlings at the Digital Arts Exhibition (Gervase [2015]).

On April 28, 2015, we tested our system with attendees of the Digital Arts Exhibition at Dartmouth. After a brief demonstration of how to create folds and preview their design, users designed cards using Foldlings. Users drew sketches, and then sent an email containing an SVG file to the computer connected to the laser cutter. Finally, they placed a piece of paper in the laser cutter, and watched as the laser beam cut out their design. Over the course of two hours, users cut and folded 31 popup cards.

The system we demonstrated at the exhibition was incomplete — it contained the basic box fold and freeform shape tools, but did not include some advanced features of the final software, such as dragging folds or shading based on plane orientation. The alpha software also contained several bugs that disrupted the experience. However, the system was usable enough for people to create cards, and observing user behavior was invaluable in designing our final product.

Because users were new to our system — and constrained by the pressure of other users waiting to design cards — designs were relatively simple. Sketches generally contained between two and five fold features in addition to the base card — the most complex design contained ten fold features. Despite their simplicity, sketches showed a wide range of designs, ranging from abstract shapes to representational scenes — users sketched symbols, Chinese characters, and geometric forms. Most of the sketches utilized both freeform and box fold features, mixing the two element types to create a composition. One of the most popular design elements was the user's name: five of the cards contained names or initials. Roughly one third of the designs took advantage of nesting — constructing fold features inside each other.

Because users were able to quickly design and fabricate their design, people generally left satisfied. People typically spent around 20 minutes at our booth, leaving with a popup card they had created. However, the experience was not frictionless. Users were frustrated by crashes: touching the screen with more than one finger or drawing while calculating planes were the most common reasons for failure. Other common complaints were the lack

of a delete/undo button and that the UI did not show which tool was currently selected.

Folding the fabricated design also presented difficulties. Although they were able to see a 3D preview of their design while creating it, users had often relinquished the iPad by the time they folded their design. They were often unsure how to fold their card, and struggled to discover the correct fold orientations. In some cases, it took longer for users to fold their creation than to design it.

We observed several unexpected behaviors. A few users rotated the screen to design a card in a landscape view, rather than the portrait orientation implied by the orientation of the buttons and 3D preview. They used this orientation to design cards that folded medially rather than laterally. Several users also constructed overlapping features by drawing on top of existing features. These features did not simulate correctly, as they intersected with existing edges. However, this behavior demonstrated a desire to construct more complex geometry. In the final software we implement unions for fold features — the most recently-drawn feature occludes features underneath it, modifying their edges.

Users also relied on the 3D preview to differing degrees. Some users viewed the preview after every operation, while others only switched to the preview occasionally. Many users relied on the 3D preview as a reference to how to fold their popup card. We were surprised by this, and conducted further user studies to determine the effectiveness of methods of displaying 3D information.

II. Visual Aids User Study

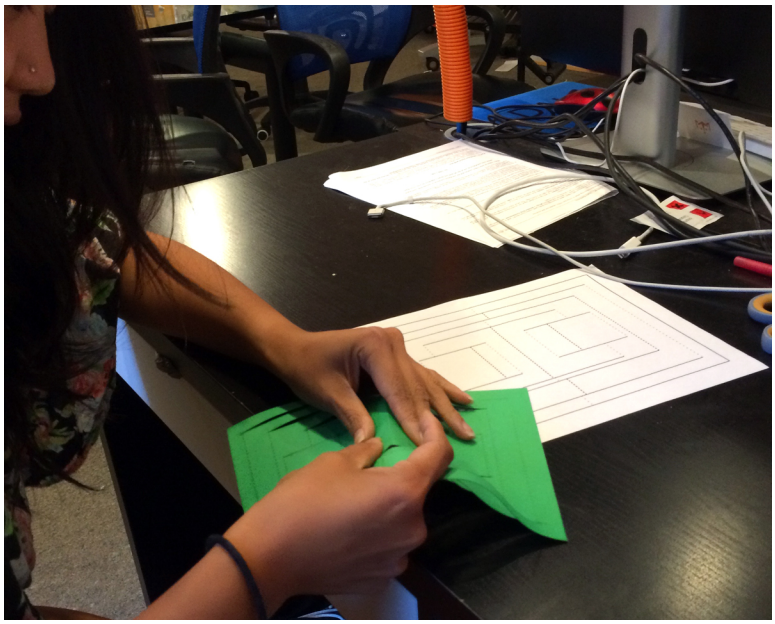


Figure 4.2: A Participant folding a card in our study on 2D to 3D visual aids for popup cards.

The goal of this study was to determine whether users understand the mapping of 2D fold patterns to 3D, and test the degree to which plane shading, edge patterning, and a 3D preview help users understand how a popup card will fold.

(a) Method

We performed this study with 22 participants, who spent an average of 18 minutes with us. Each subject received a set of five laser cut cards, and we recorded the time for them to successfully fold the card. Although there was some variety in age and background, the largest demographic was undergraduate Dartmouth students. 10 of the participants were male, 12 were female.

For each card, each subject was randomly given one of of the following five aids:

1) A two-dimensional design, showing planes shaded by whether they will be horizontal or vertical when folded.

2) A two-dimensional design, with edges patterned based on whether they are “hills” or “valleys” — whether they fold towards or away from the card.

3) A video showing a simulation of the card folding in three dimensions.

4) A still image of the card folded in dimension.

5) No visual aid.

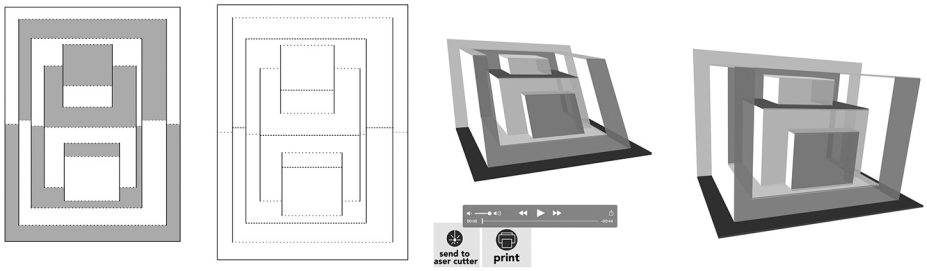
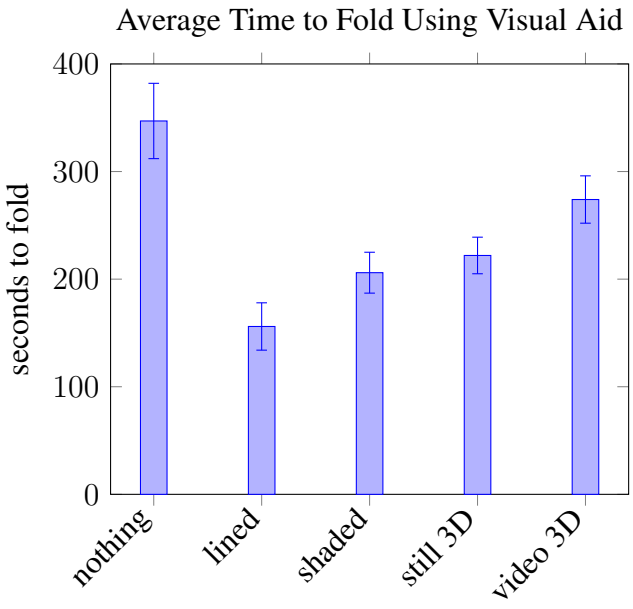


Figure 4.3: The four visual aids. Left to right: planes shaded by orientation, edges patterned based on orientation, video simulation, still image.

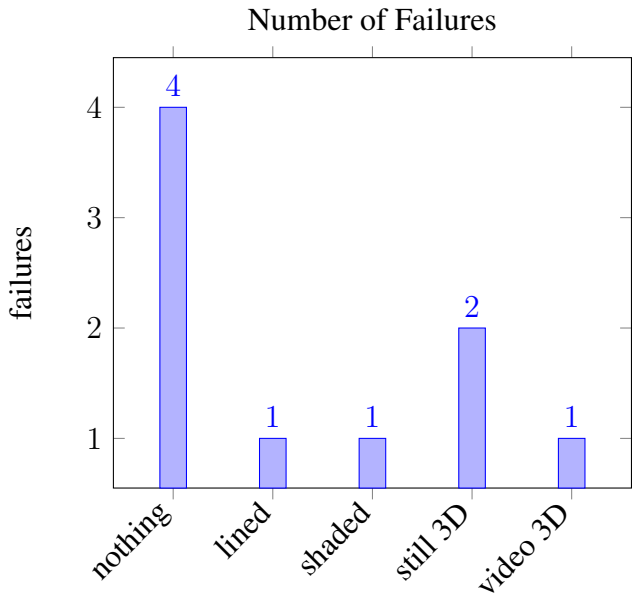
The order of aids and cards was shuffled, and then balanced to ensure an equal distribution of orderings. I.e each visual aid has an equal chance of being the first aid presented to a user and the last aid presented.

Finally, we asked subjects to rank the visual aids in order of most to least helpful. For materials used in the study, see Appendix A.

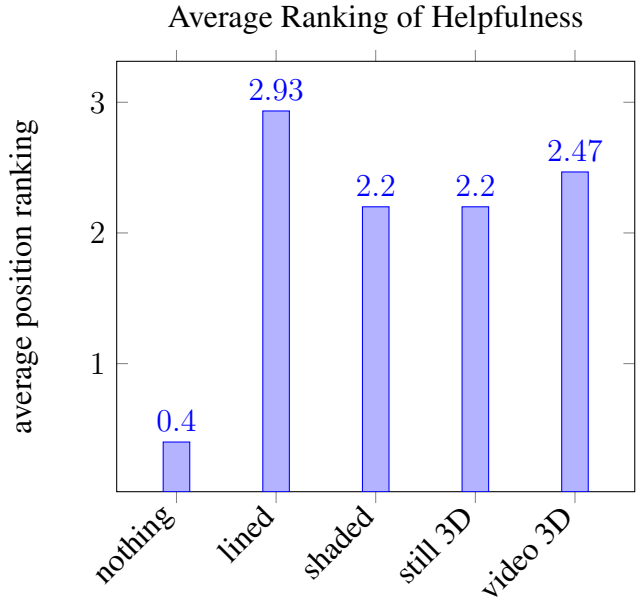
(b) Results and Discussion



The time to fold varied significantly depending on the visual aid used to fold the card. Error bars indicate a 95% confidence interval around the mean, showing a relatively narrow variance within each visual aid data set. From this, we can conclude that trials with no visual aid were slower than all trails with visual aids, and that the lined pattern showing fold orientation was more helpful that any other preview method. While the shaded visual aid and still 3D preview were indistinguishable, video 3D was slightly worse than all other visual aids.



Some trials were not completed successfully. In some cases, users folded the design incorrectly — in others, they refused to fold the card, feeling lost without a visual aid. Unsurprisingly, users were far more likely to fail to fold the card when they did not have a visual aid. The times for these failures were not recorded in the Number of Failures graph above.



We asked subjects to rank the visual aids from most to least helpful. In this graph, the data is averaged and inverted (5 being the most helpful), to show the relatively helpfulness

of each aid. Participants rated the lined aid as the most helpful — with small differences among the other aids. The lack of visual aid rated very poorly.

While this study gives persuasive evidence for the relative helpfulness of lines showing fold orientation, it is possible that other aids are more helpful in completing other 2D to 3D visualization tasks. For example, it would be interesting to compare the results if we asked users to simply label horizontal and vertical planes instead of folding a card. Perhaps the lined visual aid was most valuable to this task because it was closest to the action participants performed: fold orientation is very closely related to the task of folding a card.

Another limitation of this study is that we do not fully test the effectiveness of our interactive 3D preview in helping users visualize the folded card. We chose to limit the 3D video aid to a non-interactive aid, to make it more directly comparable to the other visual aids. However, a key benefit of the 3D preview in our software is that users can directly manipulate and rotate the simulated card, which was not possible in this study.

In Foldlings, we implement many of these visual aids. In the 2D sketch, we shade planes based on orientation, and display a 3D preview that users can manipulate in the 3D preview. As a result of our findings in this study, we also implemented line patterning based on fold orientation for the SVG export.

Chapter 5

Conclusions

We plan to release Foldlings in the Apple App Store in September. In many ways, the primary test of our software will be the extent to which real users are able to achieve their design goals. However, we can make several conclusions about the success of our tool. In general, users find the process of designing popup cards more intuitive with the current version of Foldlings than with previous versions of our software or manual methods. Additionally, we have qualitative evidence that suggest that users can create a wide range of complex cards, faster and with more precision than using manual methods. That said, there is much work to be done on this and related popup-card design problems.

I. User Interface Future Work

(a) Modifications to the Master Card

Currently, our software only allows for one size and type of master card feature. That is, a greeting-card sized piece of paper, with a single driving fold in the center. Allowing modifications to the driving fold might allow users to change paper size, rotate the card so that the middle fold is vertical, or construct a diagonal fold for the master card.

(b) Multiple Cards

Currently, our software simulates cuts and folds performed on a single piece of paper. In order to support combinations of interlocking sketches, we would need to create an interface that allow for connecting pop-up card elements in 3D. Although very complex to implement, this interface could ultimately allow for a far more complex arrangement of features that our software currently affords (Hart et al. [2007]).

(c) Safe Area Guides

Often, users wish construct a fully-contained popup card. That is, a card that can close fully, with no portions of internal fold features visible when the card is closed. In order to achieve this design in a symmetrical card, all features with a driving fold must be created such that their planes will not extend beyond the bound of the master card when fully folded. We could add "safe area" guides and warnings to indicate this area to users that want to add that additional constraint to their design.

II. Algorithms & Implementation Future Work

(a) Feature Intersections

Feature intersections are only partially implemented, and do not always succeed. To fully-implement feature intersections, we would need to refactor our FoldFeature class to add feature intersections as a primary component. This would replace the current method of intersecting features with folds — $splitFoldByOcclusion$ — and would allow for more generalizable intersections between features.

(b) Concurrency

A key limitation of Foldlings is that all functions currently run on a single thread. As a consequence, the user is sometimes blocked by operations that could be performed in the background. For example, when completing a feature, our app ignores touch input until the feature is added to the sketch and planes are calculated. This can cause a slight but noticeable delay between actions. Restructuring our algorithms to perform computationally-heavy operations in the background would reduce lag between actions, allowing users to design more quickly and fluidly.

III. Potential Applications

This section is co-authored with Marissa Allen

As a general-purpose design tool for cuts and folds, Foldlings has a wide variety of potential applications. For example, Melina Blees et al present a graphene transistor that is constructed through kirigami methods (Blees et al. [2014]). Simple, usable interfaces for designing complex kirigami structures are needed to advance similar research.

One exciting application of Foldlings is as a tool for developing advanced spatial reasoning skills. Taylor et al present a curriculum that uses popup card design as a tool for building mathematics and spatial reasoning skills (Taylor and Hutton [2013])[1]. A tool like Foldlings would likely increase the effectiveness of such a program, since much of the experimentation could happen more quickly in software than using manual methods. Because our code is open source, advanced students could even modify our software to develop new fold features and interactions.

[1] See also: Olson [2004]

Chapter 6

Appendix

I. Appendix A: User Interface Mockups

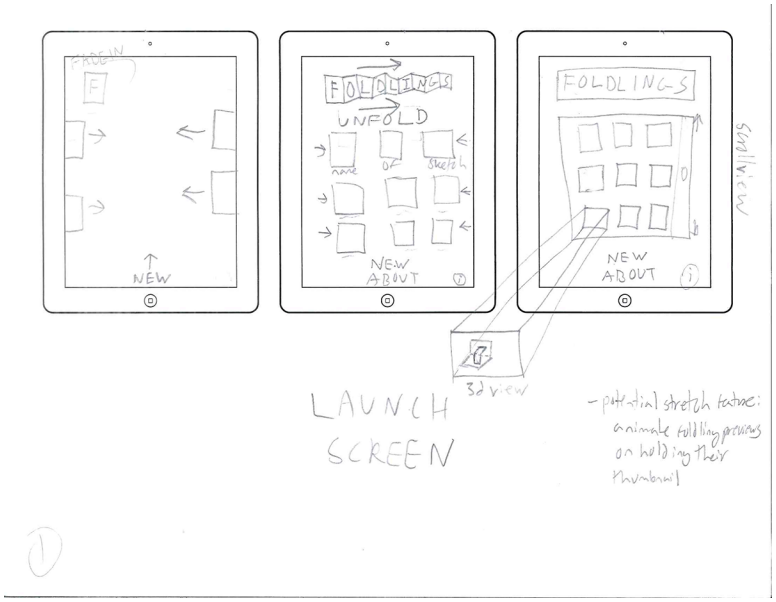


Figure 6.1

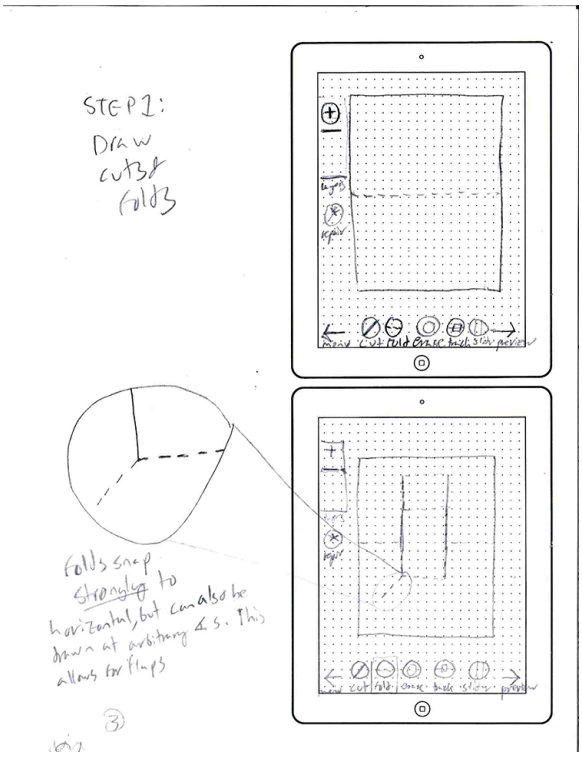


Figure 6.2

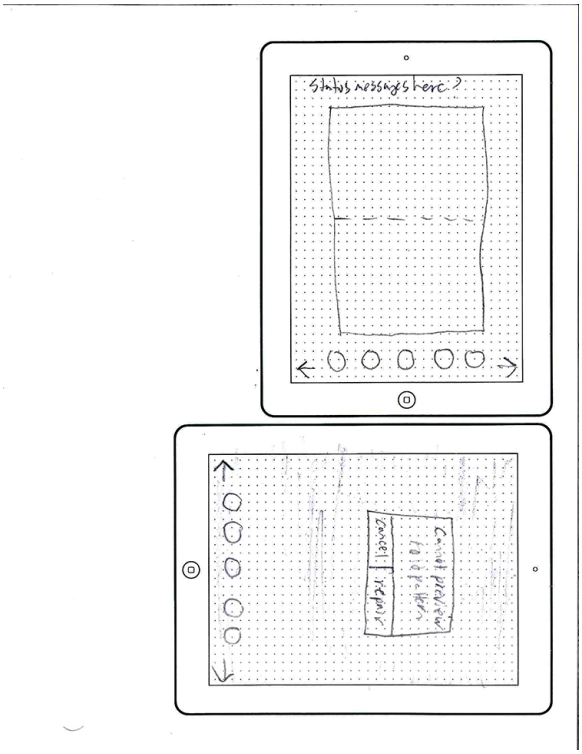


Figure 6.3

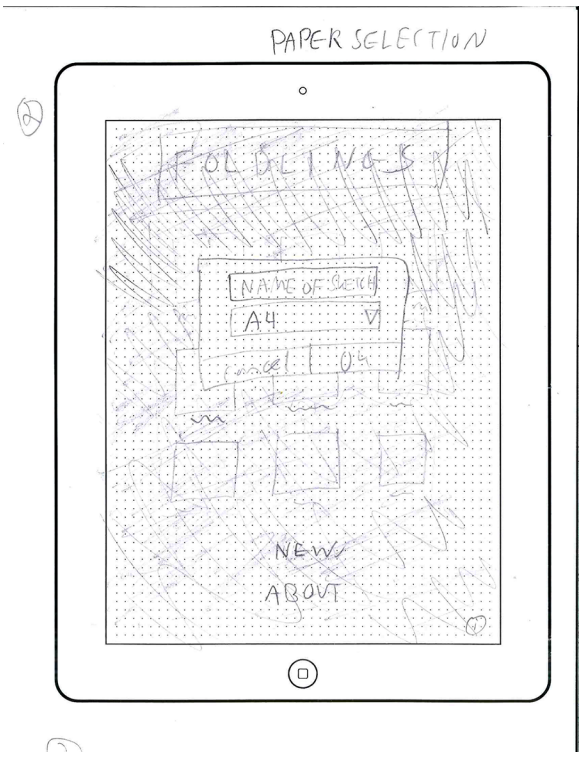


Figure 6.4

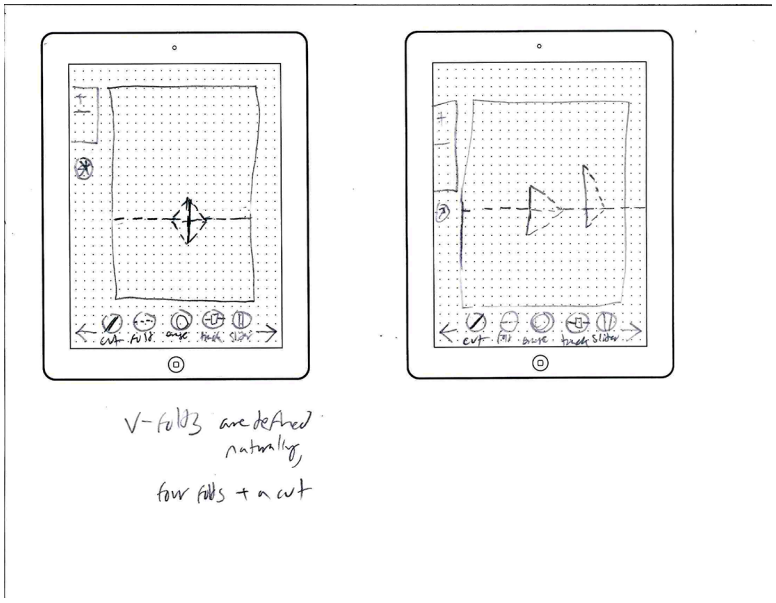


Figure 6.5

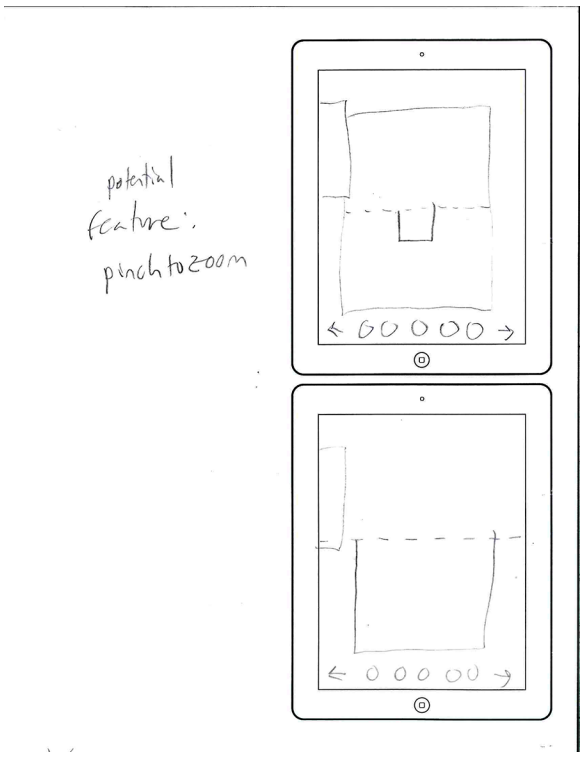


Figure 6.6

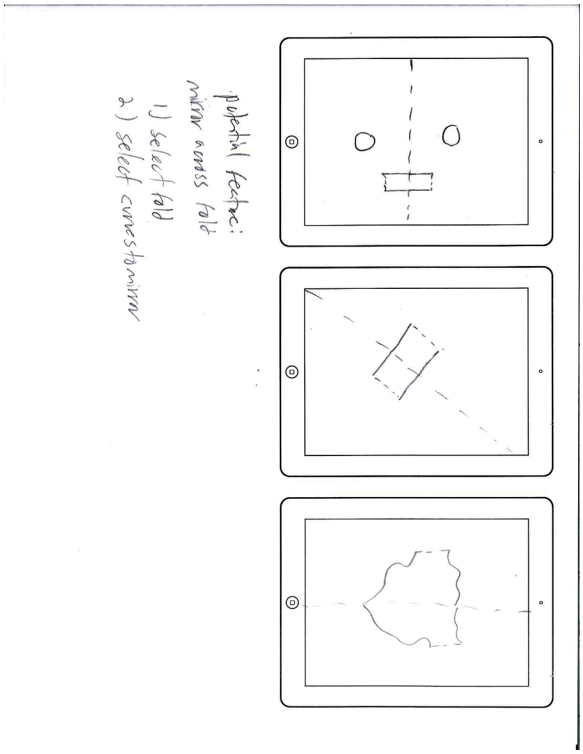


Figure 6.7

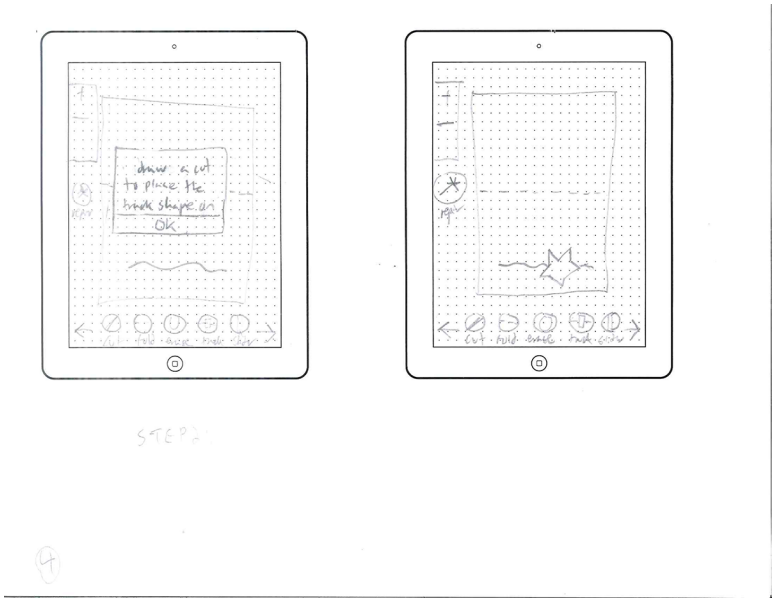


Figure 6.8

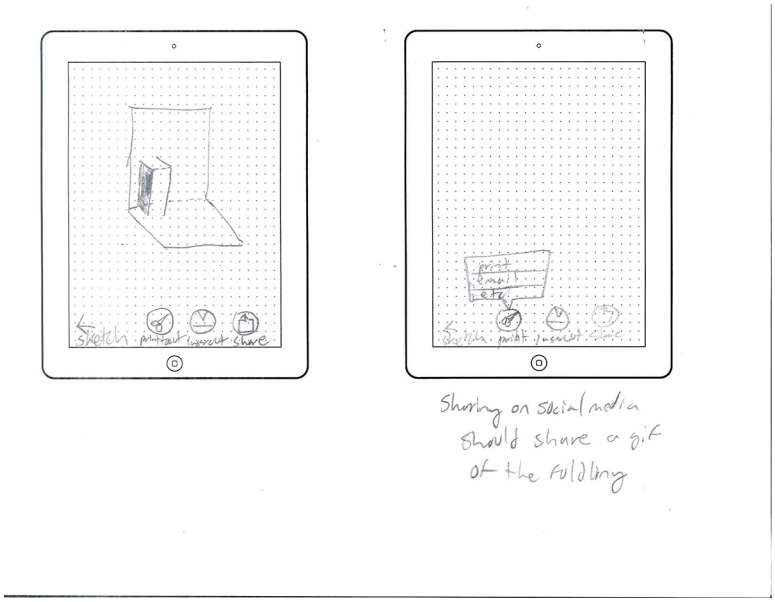


Figure 6.9

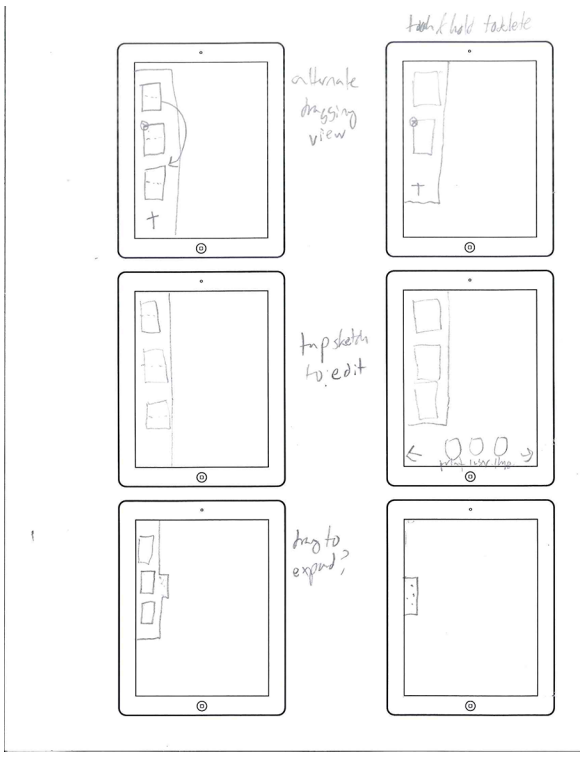


Figure 6.10

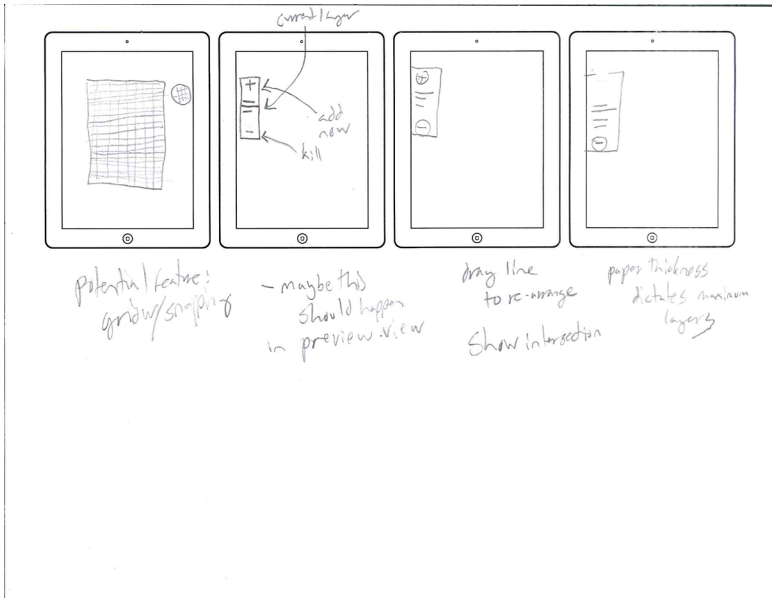


Figure 6.11

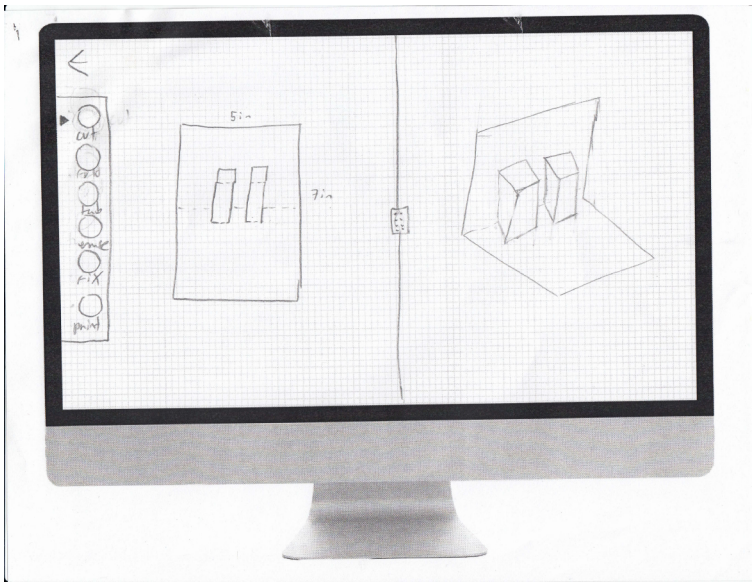


Figure 6.12

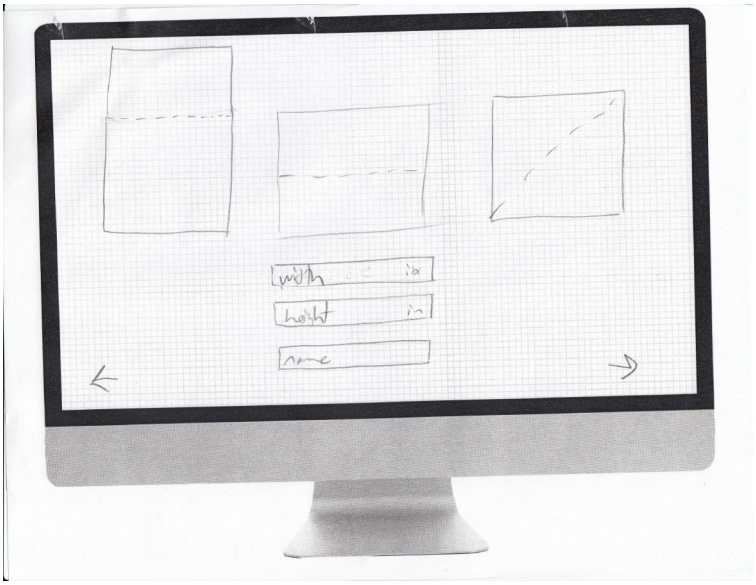


Figure 6.13

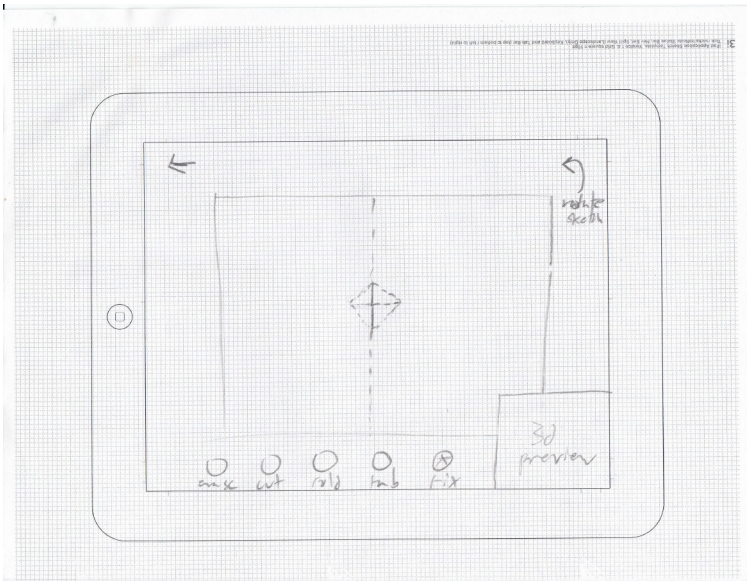


Figure 6.14

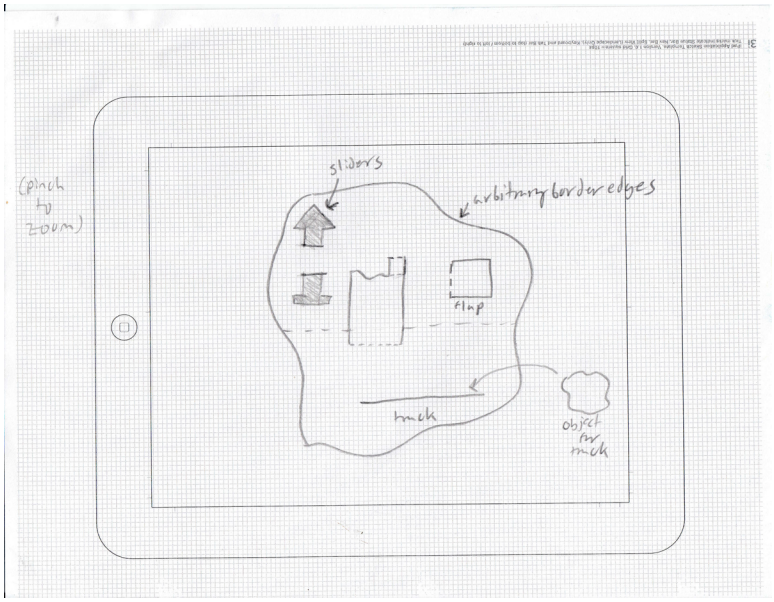


Figure 6.15

II. Appendix B: Designs Created at DAX

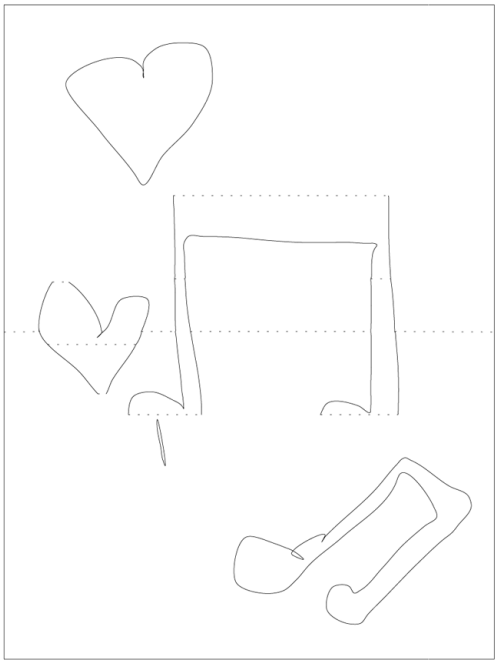


Figure 6.16

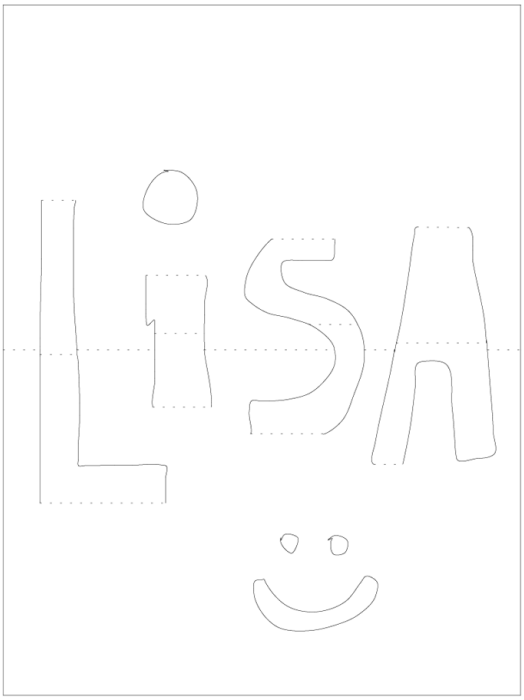


Figure 6.17

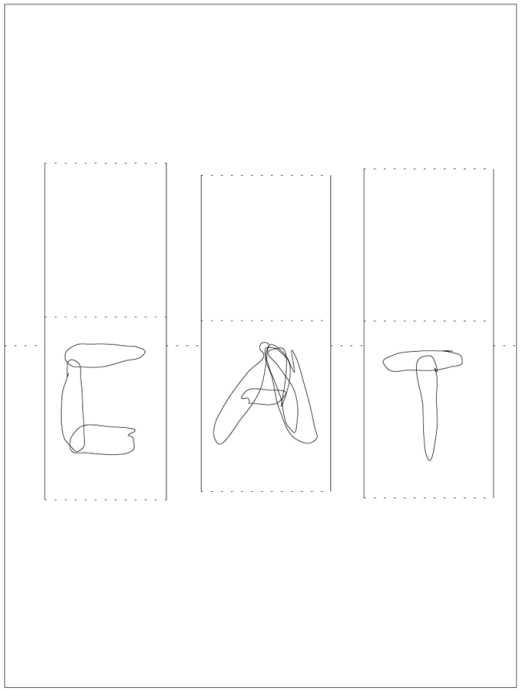


Figure 6.18

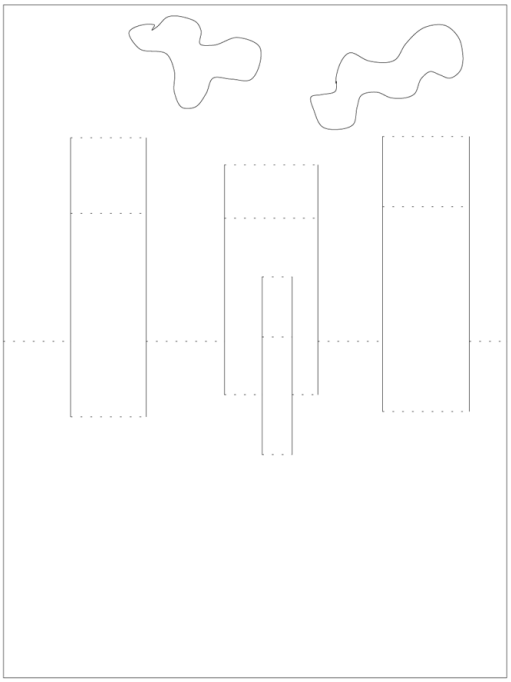


Figure 6.19

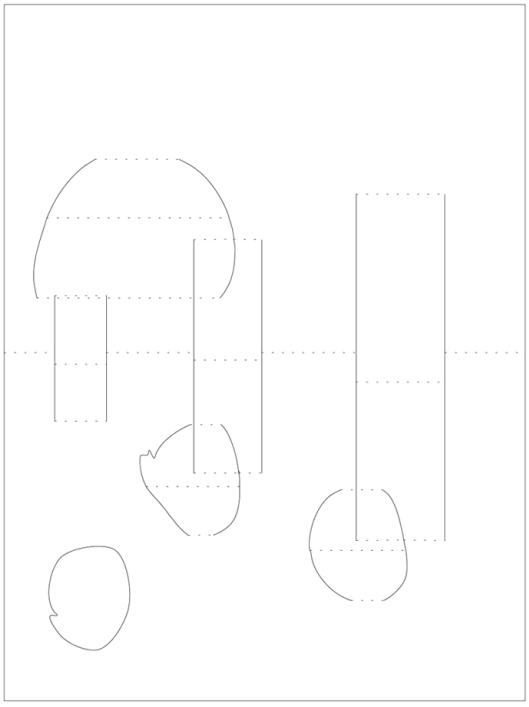


Figure 6.20

III. Appendix C: Visual Aids User Study Materials

(a) Lined Visual Aids

Lined cards indicating fold orientation — bold for hills, normal emphasis for valleys.

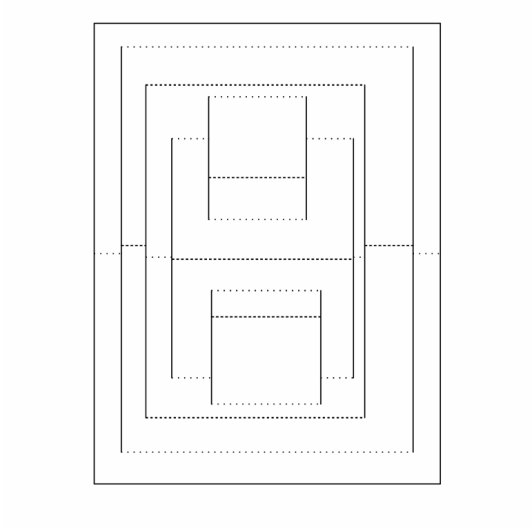


Figure 6.21

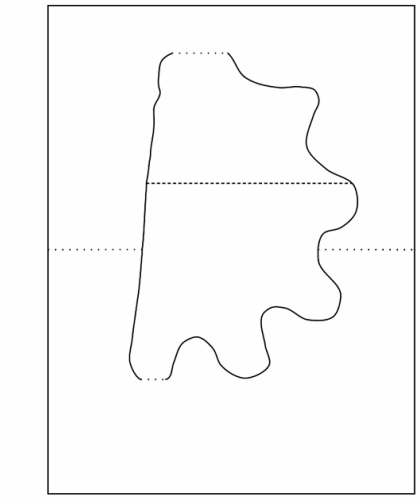


Figure 6.22

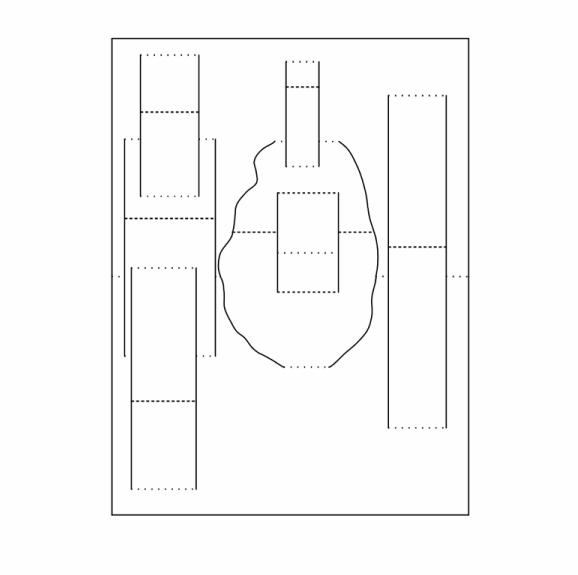


Figure 6.23

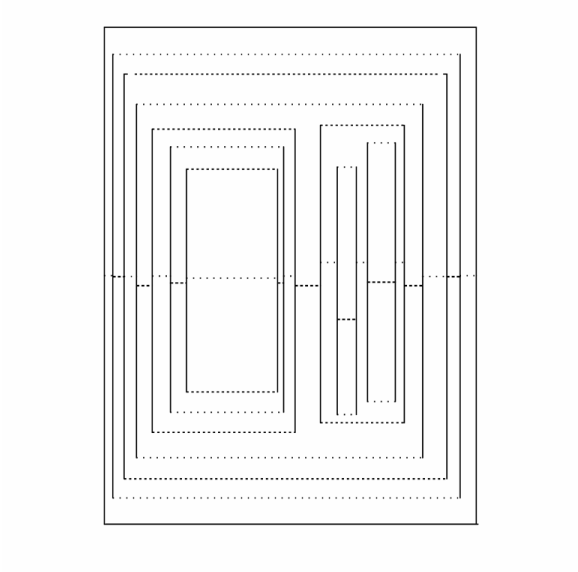


Figure 6.24

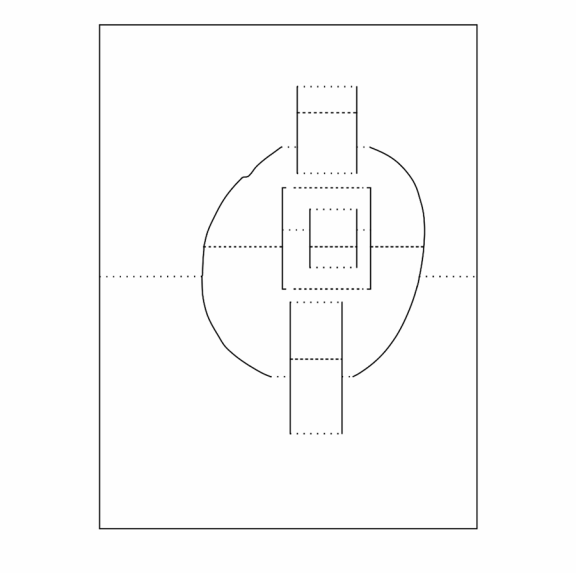


Figure 6.25

(b) Shaded Visual Aids

Shaded cards indicating plane orientation — gray for horizontal, white for vertical planes.

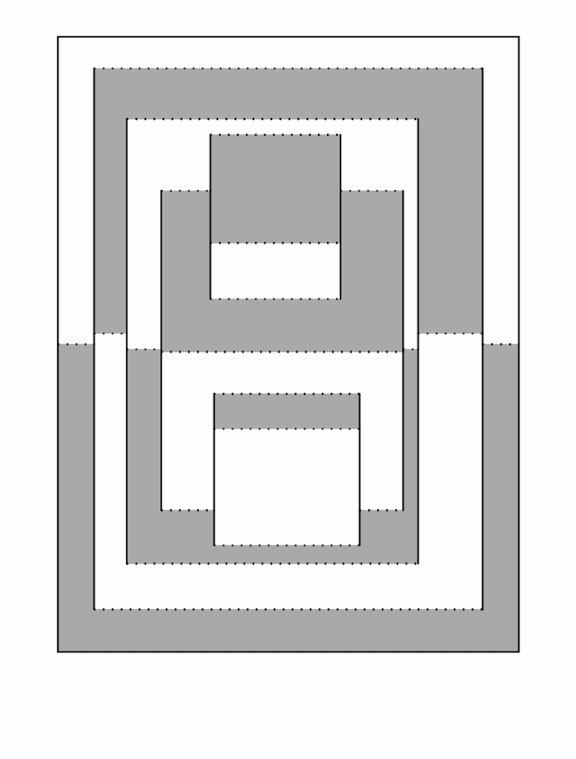


Figure 6.26

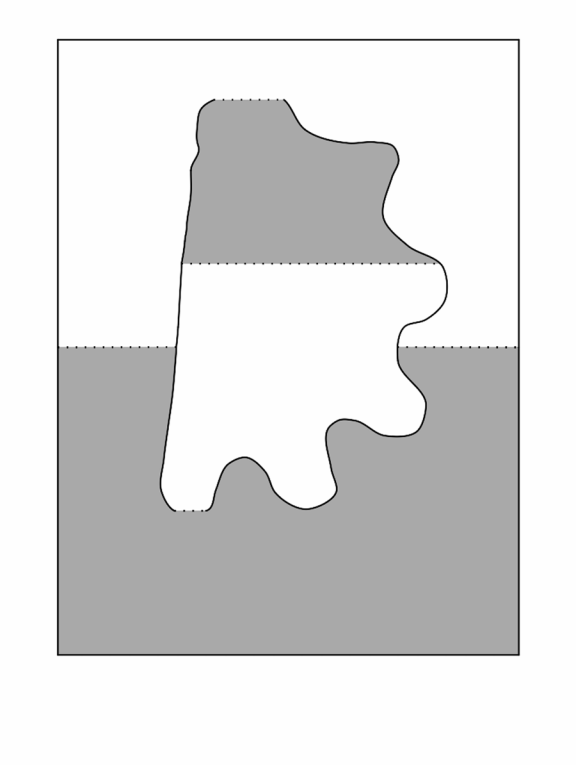


Figure 6.27

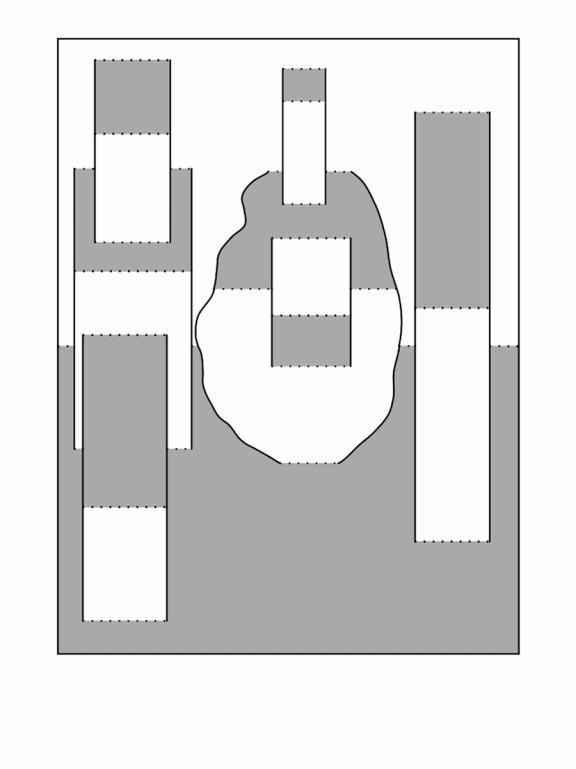


Figure 6.28

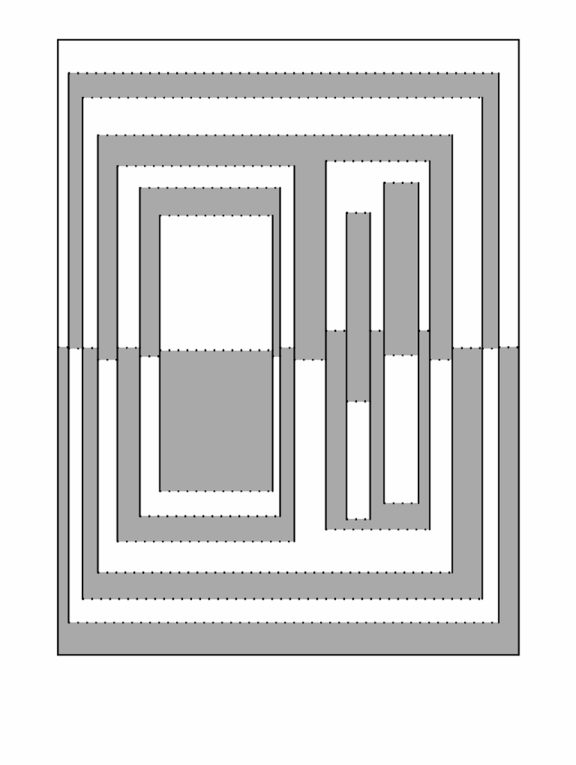


Figure 6.29

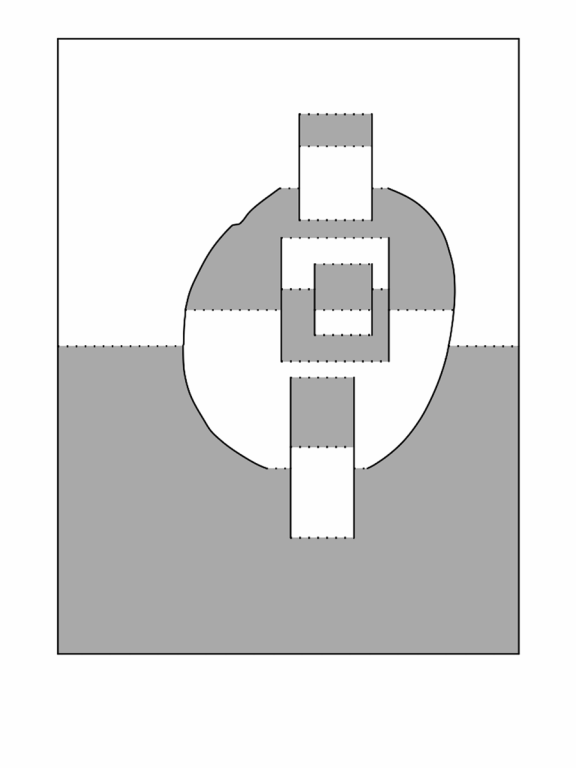


Figure 6.30

(c) Still 3D Visual Aids

Still 3d previews of the folded card.

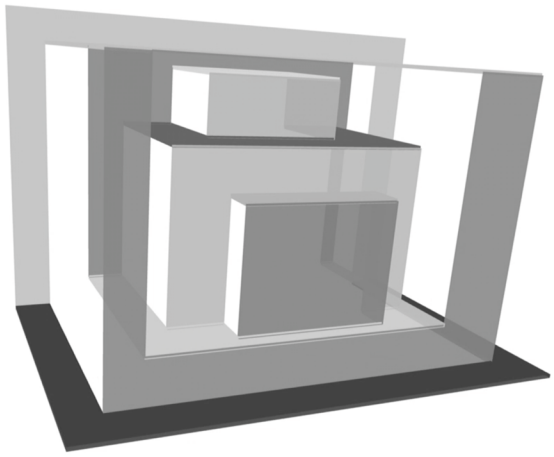


Figure 6.31

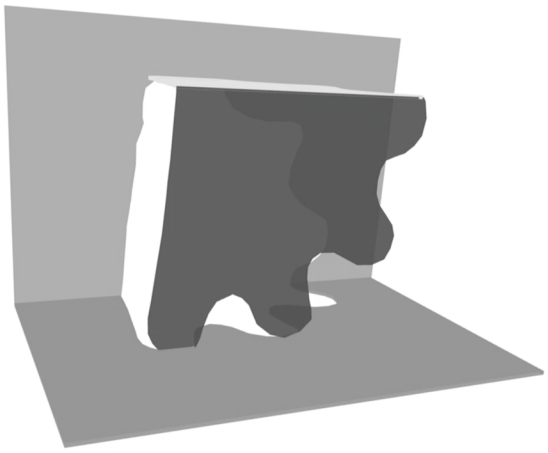


Figure 6.32

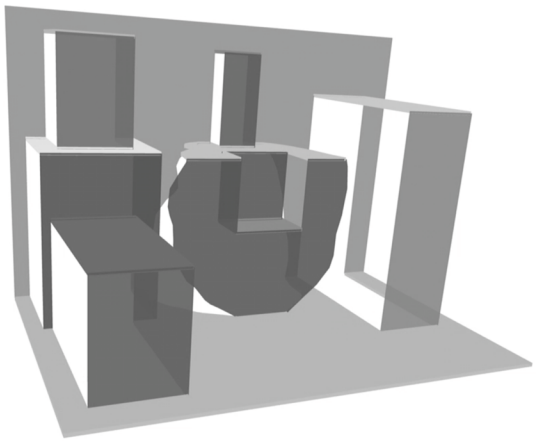


Figure 6.33

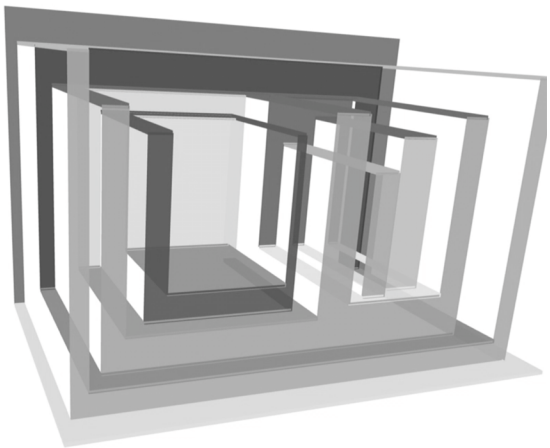


Figure 6.34

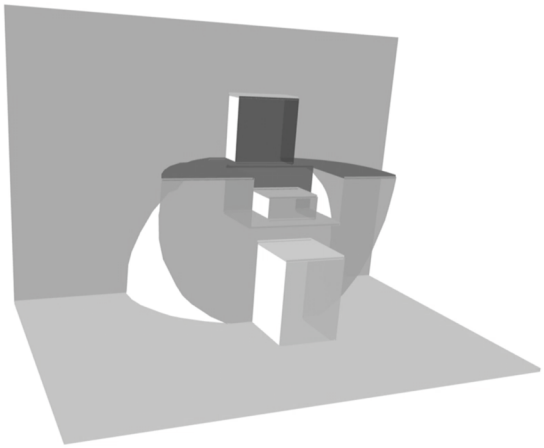


Figure 6.35

(d) Video Visual Aids

Note: these were displayed on a laptop screen as a video; only screenshots are shown here.

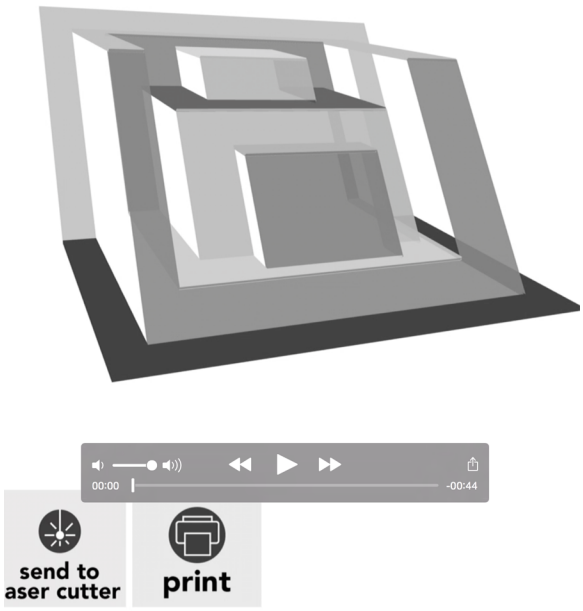


Figure 6.36

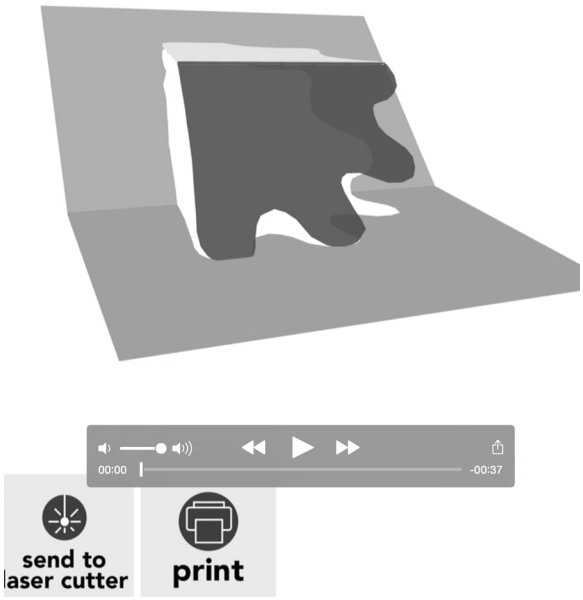


Figure 6.37

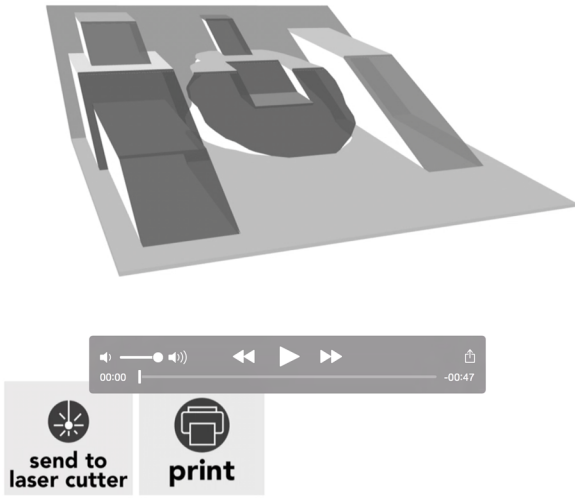


Figure 6.38

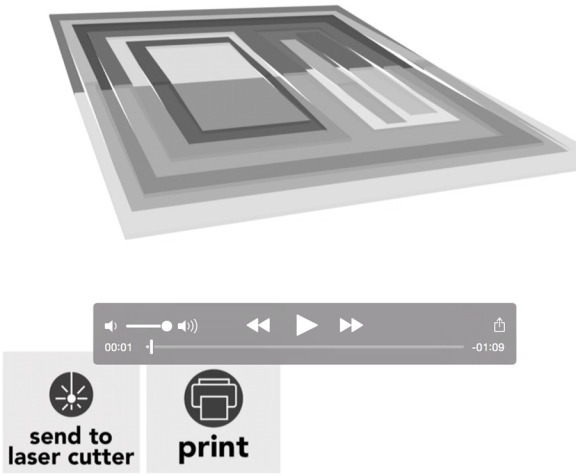


Figure 6.39

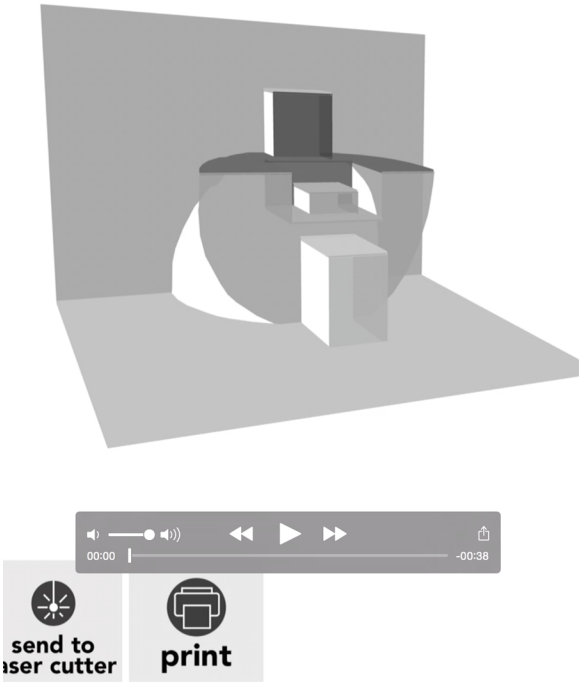


Figure 6.40

(e) Completed Cards

Cards folded by participants in the study.

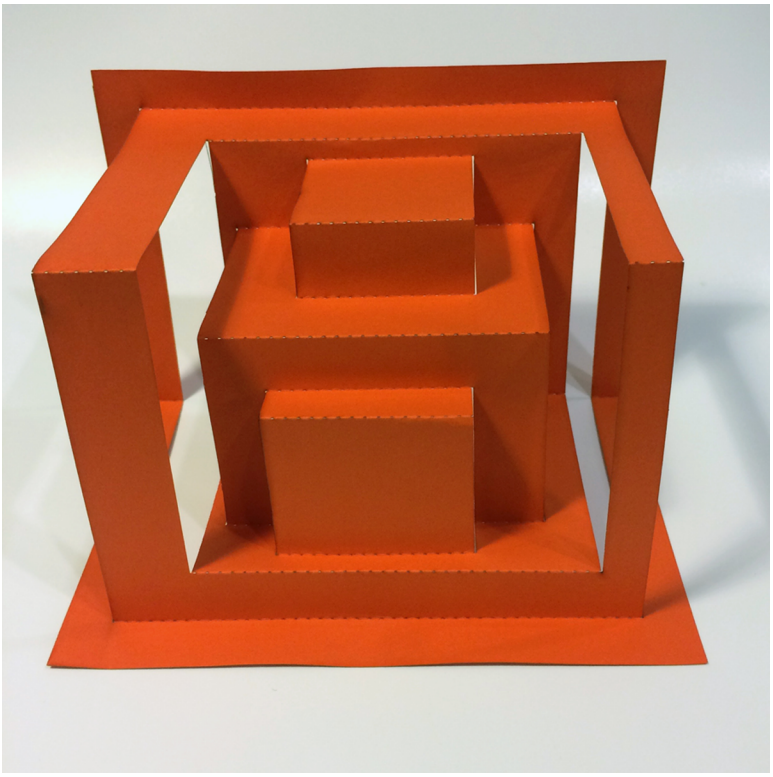


Figure 6.41

Figure 6.42

Figure 6.43

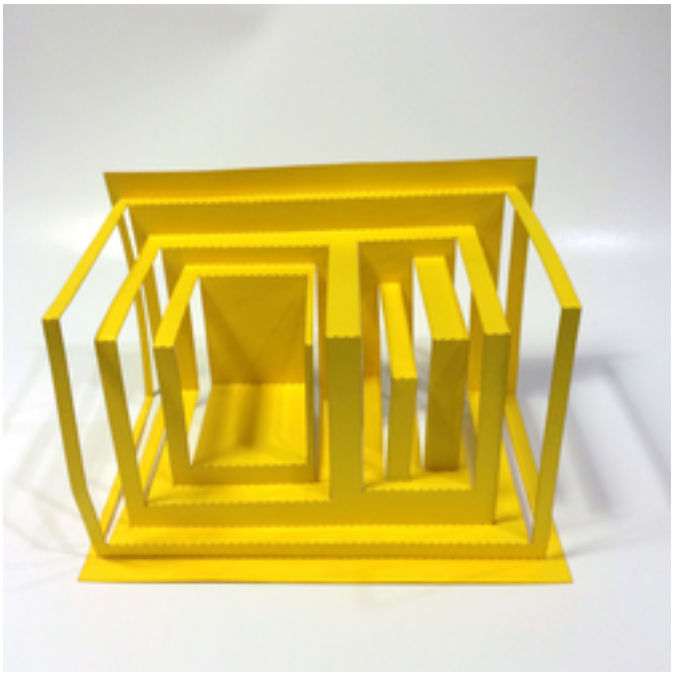


Figure 6.44

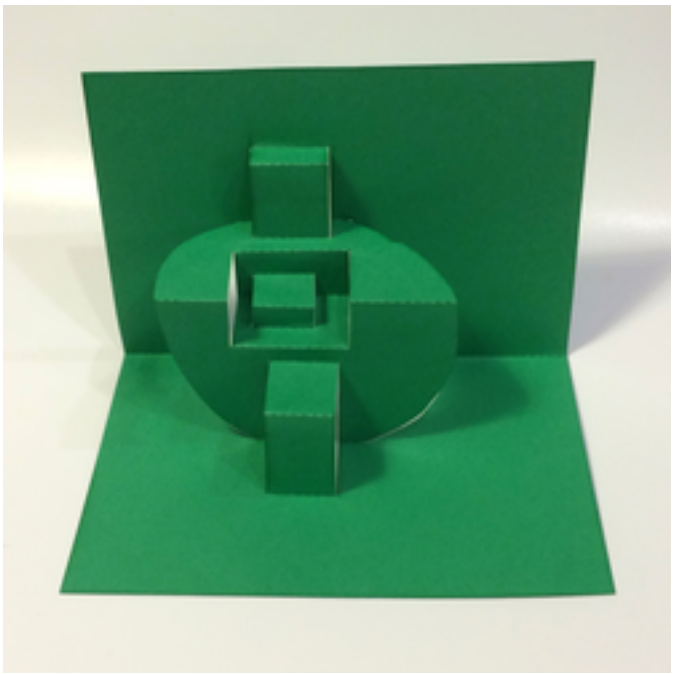


Figure 6.45

IV. Appendix D: Sample Cards

This appendix contains examples of cards designed using our software and manufactured by hand.

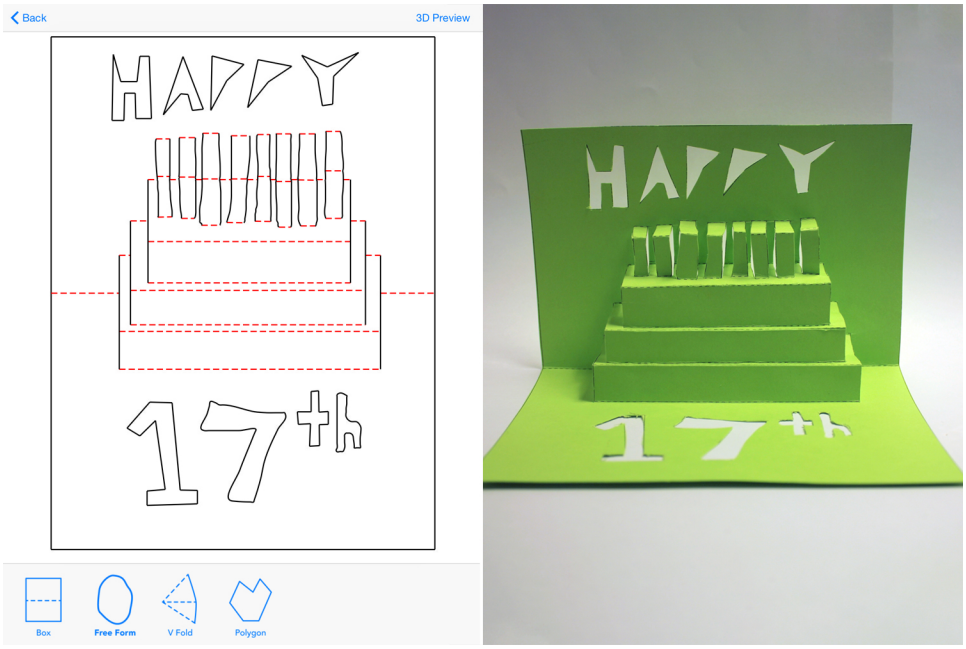


Figure 6.46

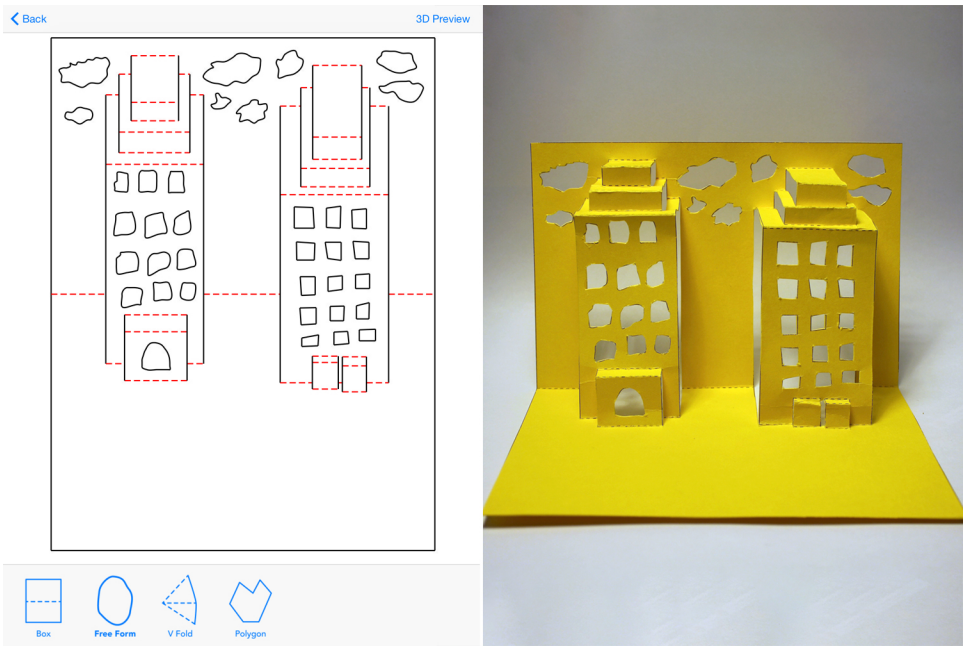


Figure 6.47

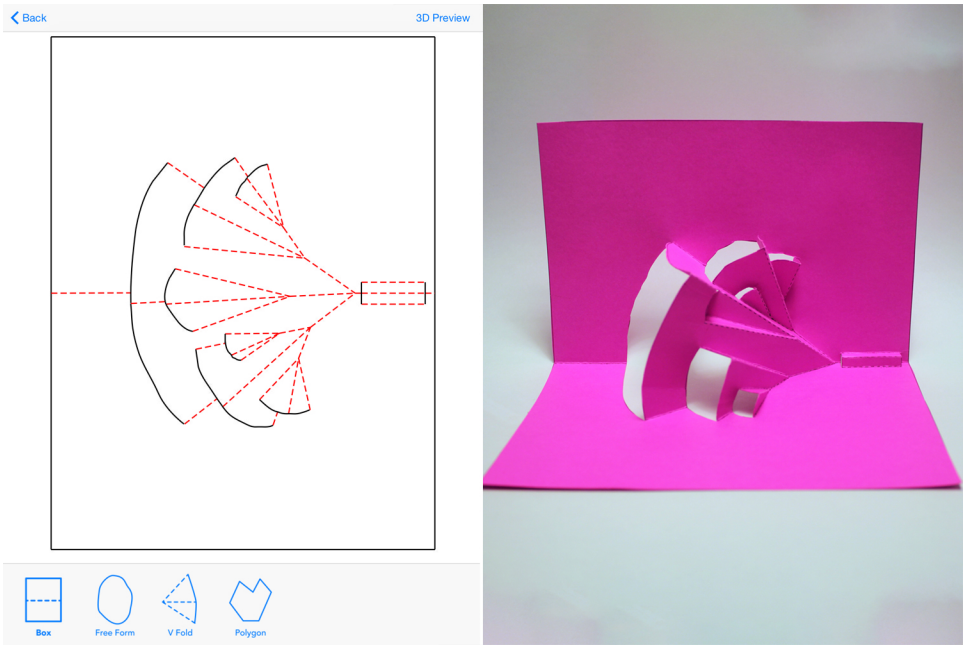


Figure 6.48

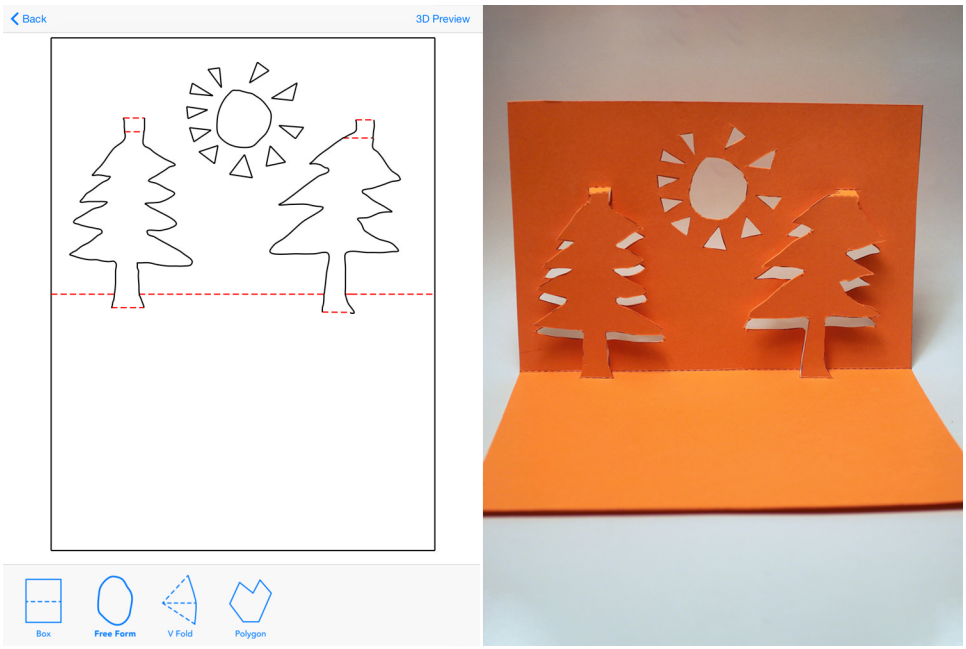


Figure 6.49

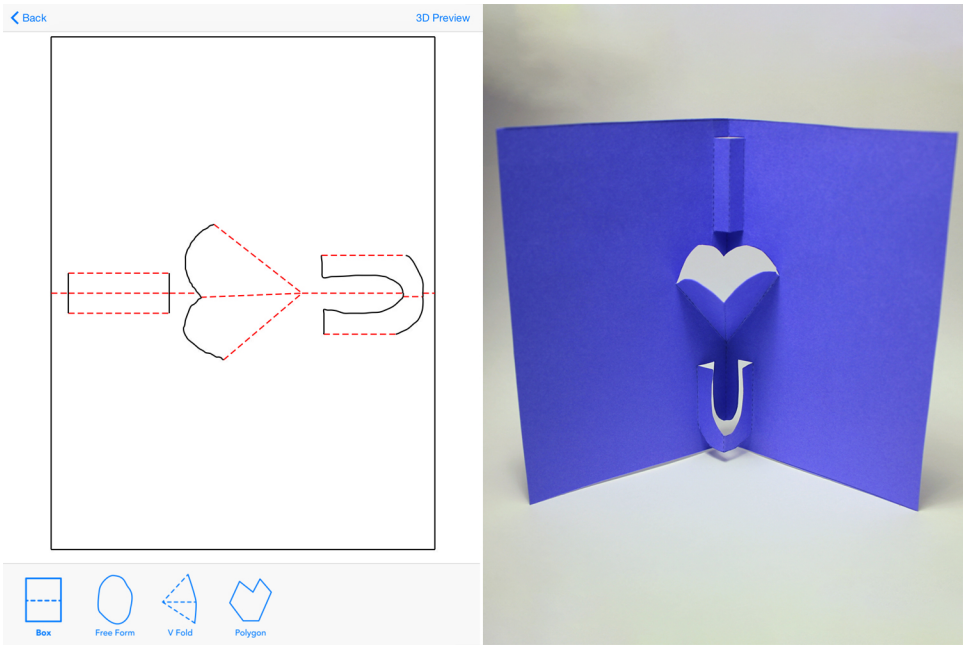


Figure 6.50

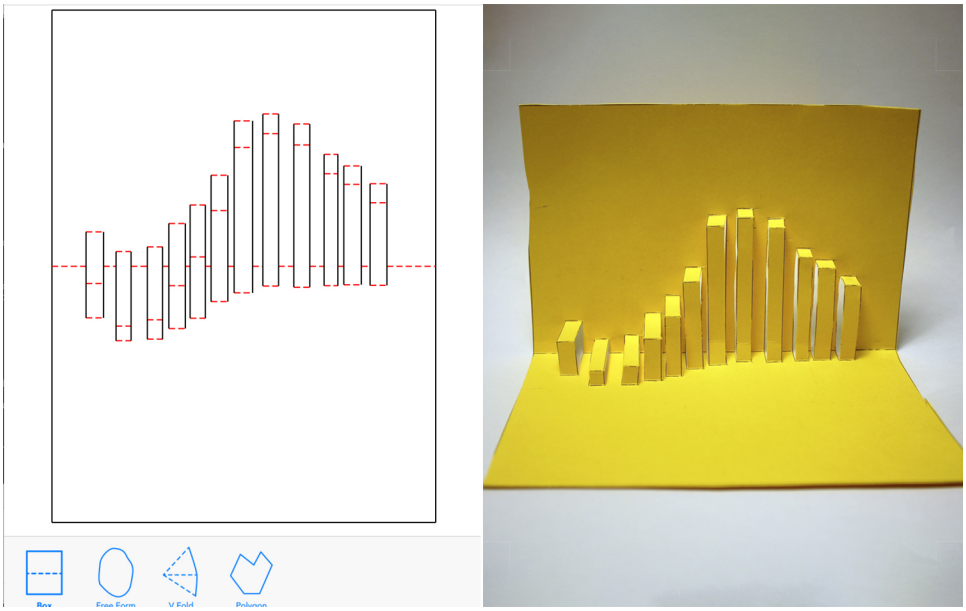


Figure 6.51

V. Appendix E: Separation of Work

This project is a collaboration between Marissa Allen and Nook Harquail. Although we worked collaboratively on the software, each of us presents an individual thesis paper. Our responsibilities on the software were as follows: Nook concentrated on tools and interactions, including performing user studies and processing user input, while Marissa concentrated on implementing the 3D simulation and data structures, including methods to define and render planes in 2D and 3D.

Bibliography

Sarah R Berenson. Kirigami polygon constructions for the general mathematics secondary school classroom. 1972.

Maekawas theorem, August 2015. URL `https://commons.wikimedia.org/wiki/File:Maekawas_Theorem.svg`.

Julietta Gervase. Dax photo. Photo published at https://www.facebook.com/daxdartmouth, May 2015.

Florence Temko and Toshie Takahama. The Magic of Kirigami: Happenings with Paper and Scissors by Florance Temko and Toshie Takahama. Japan Publications, Incorporated, 1978.

Bastien F Grosso and EJ Mele. Bending rules for nano-kirigami. arXiv preprint arXiv:1507.01805, 2015.

Ben Andrews, Daniel Felix, Kunal Joshi, Kumar Mehta, Edson Novinyo, Lorraine Higgins, Ingrid Shockey, OP Singh, and Viswanath Balakrishnan. Creating a kirigami shelter prototype for migratory populations of himachal pradesh, india.

Krystyna Burczyk. ul. konwaliowa 22 32-080 zabierzów, poland e-mail: burczyk@ mail. zetosa. com. pl.

Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: a sketching interface for 3d freeform design. In Acm siggraph 2007 courses, page 21. ACM, 2007.

Min Patrick, Chen Joyce, and Funkhouser Thomas. A 2d sketch interface for a 3d model search engine. In Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH, 2002.

Charlie CL Wang, Yu Wang, and Matthew MF Yuen. Feature based 3d garment design through 2d sketches. Computer-Aided Design, 35(7):659–672, 2003.

Andrew Glassner. Interactive pop-up card design. Technical report, Miscrosoft Technical Report, 1998.

Conrado R Ruiz, Sang N Le, Jinze Yu, and Kok-Lim Low. Multi-style paper pop-up designs from 3d models. In Computer Graphics Forum, volume 33, pages 487–496. Wiley Online Library, 2014.

Xian-Ying Li, Chao-Hui Shen, Shi-Sheng Huang, Tao Ju, and Shi-Min Hu. Popup: automatic paper architectures from 3d models. ACM Transactions on Graphics-TOG, 29(4): 111, 2010.

Zachary Ryan Abel, Erik D Demaine, Martin L Demaine, Sarah Charmian Eisenstat, Anna Lubiw, André Schulz, Diane L Souvaine, Giovanni Viglietta, and Andrew Winslow. Algorithms for designing pop-up cards. 2013.

Der-Lor Way, Yong-Ning Hu, and Zen-Chung Shih. The creation of v-fold animal pop-up cards from 3d models using a directed acyclic graph. In Advances in Intelligent Systems and Applications-Volume 2, pages 465–475. Springer, 2013.

Xian-Ying Li, Tao Ju, Yan Gu, and Shi-Min Hu. A geometric study of v-style pop-ups: theories and algorithms. In ACM Transactions on Graphics (TOG), volume 30, page 98. ACM, 2011.

Sosuke Okamura and Takeo Igarashi. An interface for assisting the design and production of pop-up card. In Smart Graphics, pages 68–78. Springer, 2009.

Jie Xu, Craig S Kaplan, and Xiaofeng Mi. Computer-generated papercutting. In Computer Graphics and Applications, 2007. PG'07. 15th Pacific Conference on, pages 343–350. IEEE, 2007.

Gabe Johnson, Mark Gross, Ellen Yi-Luen Do, and Jason Hong. Sketch it, make it: sketching precise drawings for laser cutting. In CHI'12 Extended Abstracts on Human Factors in Computing Systems, pages 1079–1082. ACM, 2012.

Jack Fastag. egami: Virtual paperfolding and diagramming software. In Origami4: Fourth International Meeting of Origami Science, Mathematics, and Education, pages 273–283, 2009.

Asem Kasem and Tetsuo Ida. Computational origami environment on the web. Frontiers of computer science in China, 2(1):39–54, 2008.

Ludmila Zamiatina. Computer simulations of origami. Journal/Anthology, 3, 1994.

Wendy Ju, Leonardo Bonanni, Richard Fletcher, Rebecca Hurwitz, Tilke Judd, Rehmi Post, Matthew Reynolds, and Jennifer Yoon. Origami desk: integrating technological innovation and human-centric design. In Proceedings of the 4th conference on Designing interactive systems: processes, practices, methods, and techniques, pages 399–405. ACM, 2002.

Susan L Hendrix, Michael Eisenberg, et al. Computer-assisted pop-up design for children: computationally enriched paper engineering. Advanced Technology for Learning, 3(2): 119–127, 2006.

Steven J Bell. Design thinking. American Libraries, pages 44–49, 2008.

Robert Cecil Martin. Agile software development: principles, patterns, and practices. Prentice Hall PTR, 2003.

Donald A Norman. The design of everyday things: Revised and expanded edition. Basic books, 2013.

Donald A Norman and Jakob Nielsen. Gestural interfaces: a step backward in usability. interactions, 17(5):46–49, 2010.

Stine Liv Johansen. The ipad as a tool for play: Methodological considerations. nordmedia 2013: Defending Democracy, 2013.

Duncan Birmingham. Pop-up!: A Manual of Paper Mechanisms. Tarquin, 1997.

Barbara Valenta. Pop-o-mania: how to create your own pop-ups. 1997.

David Wood. Scaffolding, contingent tutoring, and computer-supported learning. International Journal of Artificial Intelligence in Education, 12(3):280–293, 2001.

Marissa Allen. Foldlings: Visualization tools for interactive pop-up card design. Master's thesis, Dartmouth College, 2015.

Masahiro Chatani. Pop-up greeting cards: a creative personal touch for every occasion ; origamic architecture. Ondoriha, Tokyo, 1986. ISBN 978-0-87040-733-8.

Edwin Catmull and Raphael Rom. A class of local interpolating splines. Computer aided geometric design, 74:317–326, 1974.

Matthias Veit and Stephan Herrmann. Model-view-controller and object teams: A perfect match of paradigms. In Proceedings of the 2nd international conference on Aspect-oriented software development, pages 140–149. ACM, 2003.

Apple. Gesture Recognizers — iOS Developer Library, August 2015. URL `https://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/GestureRecognizer_basics/GestureRecognizer_basics.html`.

Anpathintersection, August 2015. URL `https://github.com/unixpickle/PathIntersection/`.

George M Phillips. A de casteljau algorithm for generalized bernstein polynomials. BIT Numerical Mathematics, 37(1):232–236, 1997.

G Hart et al. Modular kirigami. Proceedings of Bridges Donostia, 2007.

Melina Blees, Peter Rose, Arthur Barnard, Samantha Roberts, and Paul L McEuen. Graphene kirigami. In APS Meeting Abstracts, volume 1, page 30011, 2014.

Holly A Taylor and Allyson Hutton. Think3d!: Training spatial thinking fundamental to stem education. Cognition and Instruction, 31(4):434–455, 2013.

Alton T. Olson. Mathematics through paper folding. National Council of Teachers of Mathematics, Reston, Va, 2004. ISBN 978-0-87353-076-7.