PORTABLE MODELS FOR LASER CUTTING

By Thijs Roumen, MSc

a dissertation submitted in partial fulfillment
of the requirements for the degree of:
Ph.D.

Computer science, Human Computer Interaction
June 2022

Hasso Plattner Institute
Faculty of Digital Engineering, University of Potsdam

Published online on the
Publication Server of the University of Potsdam:
https://doi.org/10.25932/publishup-57814
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-578141

To Katrien van Beers whom I dearly
missed in every step along the way

ABSTRACT

Laser cutting is a fast and precise fabrication process. This makes laser cutting a powerful process in custom industrial production. Since the patents on the original technology started to expire, a growing community of tech-enthusiasts embraced the technology and started sharing the models they fabricate online. Surprisingly, the shared models appear to largely be *one-offs* (e.g., they proudly showcase what a single person can make in one afternoon). For laser cutting to become a relevant mainstream phenomenon (as opposed to the current tech enthusiasts and industry users), it is crucial to enable users to reproduce models made by more experienced modelers, and to build on the work of others instead of creating one-offs.

We create a technological basis that allows users to build on the work of others—a progression that is currently held back by the use of exchange formats that disregard mechanical differences between machines and therefore overlook implications with respect to how well parts fit together mechanically (aka engineering fit).

For the field to progress, we need a machine-independent sharing infrastructure.

In this thesis, we outline three approaches that together get us closer to this:

(1) **2D cutting plans that are tolerant to machine variations**. Our initial take is a minimally invasive approach: replacing machine-specific elements in cutting plans with more tolerant elements using mechanical hacks like springs and wedges. The resulting models fabricate on *any consumer laser cutter* and *in a range of materials*.

(2) **sharing models in 3D**. To allow building on the work of others, we build a *3D modeling environment* for laser cutting (kyub). After users design a model, they export their 3D models to 2D cutting plans optimized for the machine and material at hand. We extend this volumetric environment with tools to edit individual plates, allowing users to leverage the efficiency of volumetric editing while having control over the most detailed elements in laser-cutting (plates)

(3) **converting legacy 2D cutting plans to 3D models**. To handle legacy models, we build software to interactively reconstruct 3D models from 2D cutting plans. This allows users to reuse the models in more productive ways. We revisit this by automating the assembly process for a large subset of models.

The above-mentioned software composes a larger system (kyub, 140,000 lines of code). This system integration enables the push towards actual use, which we demonstrate through a range of workshops where users build complex models such as fully functional guitars. By simplifying sharing and re-use and the resulting increase in model complexity, this line of work forms a small step to enable personal fabrication to scale past the maker phenomenon, towards a mainstream phenomenon—the same way that other fields, such as print (postscript) and ultimately computing itself (portable programming languages, etc.) reached mass adoption.

ZUSAMMENFASSUNG

Laserschneiden ist ein schnelles und präzises Fertigungsverfahren. Diese Eigenschaften haben das Laserschneiden zu einem starken Anwärter für die industrielle Produktion gemacht. Seitdem die Patente für die ursprüngliche Technologie begannen abzulaufen, nahm eine wachsende Gemeinschaft von Technikbegeisterten die Technologie an und begann, ihre Modelle online zu teilen. Überraschenderweise scheinen die gemeinsam genutzten Modelle größtenteils *Einzelstücke* zu sein (z.B. zeigten sie stolz, was eine einzelne Person an einem Nachmittag entwickeln kann). Damit das Laserschneiden zu einem relevanten Mainstream-Phänomen wird, ist es entscheidend, dass die Benutzer die Möglichkeit haben Modelle zu reproduzieren, die von erfahrenen Modellierern erstellt wurden, und somit auf der Arbeit anderer aufbauen zu können, anstatt *Einzelstücke* zu erstellen.

Wir schaffen eine technologische Basis, die es Benutzern ermöglicht, auf der Arbeit anderer aufzubauen—eine Entwicklung, die derzeit gehemmt wird durch die Verwendung von Austauschformaten, die mechanische Unterschiede zwischen Maschinen außer Acht lassen und daher Auswirkungen darauf übersehen, wie gut Teile mechanisch zusammenpassen (aka Passung).

Damit sich das Feld sich weiterentwickeln kann, brauchen wir eine maschinenunabhängige Infrastruktur für gemeinsame Nutzung.

In dieser Dissertation präsentieren wir drei Ansätze, die uns zu diesem Ziel näherbringen:

(1) **2D-Schnittpläne, die gegenüber Maschinenvariationen tolerant sind.** Unser erster Ansatz ist ein minimalinvasiver Ansatz: Wir ersetzen maschinenspezifische Elemente in Schnittplänen durch tolerantere Elemente unter Verwendung mechanischer Hacks wie Federn und Keile. Die resultierenden Modelle können auf jedem handelsüblichen Laserschneider und in einer Reihe von Materialien hergestellt werden.

(2) **Teilen von Modellen in 3D.** Um auf der Arbeit anderer aufbauen zu können, erstellen wir eine 3D-Modellierungsumgebung für

das Laserschneiden (kyub). Nachdem die Benutzer ein Modell entworfen haben, exportieren sie ihre 3D-Modelle in 2D-Schnittpläne, die für die jeweilige Maschine und das vorhandene Material optimiert sind. Wir erweitern diese volumetrische Umgebung mit Werkzeugen zum Bearbeiten einzelner Platten, sodass Benutzer die Effizienz der volumetrischen Bearbeitung nutzen und gleichzeitig die detailliertesten Elemente beim Laserschneiden (Platten) steuern können.

(3) **Umwandlung von *legacy* 2D-Schnittplänen in 3D-Modelle.** Um mit *legacy* Modellen umzugehen, entwickeln wir Software, um 3D-Modelle interaktiv aus 2D-Schnittplänen zu rekonstruieren. Dies ermöglicht Benutzern, die Modelle auf produktivere Weise wiederzuverwenden. Wir behandeln dies erneut, indem wir den Rekonstruierungsprozess für eine große Teilmenge von Modellen automatisieren.

Die oben genannte Software ist in ein größeres System integriert (kyub, 140.000 Codezeilen). Diese Systemintegration ermöglicht es, den tatsächlichen Gebrauch voranzutreiben, was wir in einer Reihe von Workshops demonstrieren, in denen Benutzer komplexe Modelle wie voll funktionsfähige Gitarren bauen. Durch die Vereinfachung der gemeinsamen Nutzung und Wiederverwendung und die daraus resultierende Zunahme der Modellkomplexität wird diese Arbeitsrichtung und das daraus resultierende System letztendlich (teilweise) dazu beitragen, dass die persönliche Fertigung über das Maker-Phänomen hinausgeht und sich zu einem Mainstream-Phänomen entwickelt – genauso wie andere Bereiche, z.B. als Druck (Postscript) und schließlich selbst Computer (portable Programmiersprachen usw.), um eine Massenakzeptanz zu erreichen.

ACKNOWLEDGEMENTS

First and foremost, I want to thank Patrick Baudisch for being the most inspiring person I have had a chance to work with. I vividly recall the first semester of my PhD where I came home every night with a head excessively filled with new and amazing ideas, it was that period that convinced me 100% that I wanted to stay in this academic wonderland. The countless (3?) semesters after that, Patrick never stopped to impress, if I can transfer only a fraction of this energy and inspiration to my future students, I would be eternally grateful!

But besides the academic work and remarkable patience (ranging from fideo to starting too far back in writing abstracts), Patrick has also turned into a dear friend over the years. During some of the darker periods within my PhD, Patrick was always there to support, and in an unfortunate turn of events, I got a chance later to reciprocate that support. As a result, we can now make weird puns nobody else seems to be able stomach, thanks for that! Overall, during the rollercoaster with many ups and downs which we call PhD, I have become a much better version of myself, which would never have been possible without Patrick's support.

I want to thank my colleagues at the Human Computer Interaction lab for their unprecedented support, patience, and guidance. I thank Stefanie Mueller for always radiating positive energy and excitement, motivating me early-on to really go for this crazy career. Alexandra Ion, for making me believe in and fight for my ideas and pursuing them even when they go against the current of the lab (some of these will still come to fruition in the next years, I promise!). Pedro Lopes for being a great example and showing that its ok to ignore what the rest of the world (including R2) thinks because just doing good research will get you there no matter what! I thank Lung-Pan Chen for trusting me to run his great TurkDeck project when I was only a rookie PhD student, that must have taken infinite courage! Robert Kovacs for always being there to support anyone who needs help, the more complicated the merrier. And Abdullah Muhammad for being an amazing student, giving me confidence that I may be able to graduate into an advisor as well. And

of course, to my other lab mates Jotaro Shigeyama, Sebastian Marwecki, Shohei Katakura, Conrad Lempert, Martin Taraz, Lukas Rambold, and the many others who came and went for shorter commitments at the lab

Special thanks go to Jack Lindsay, who has made my day countless times as a great friend. I think there has been no day where we did end up laughing out (very loud—sorry Alex) loud in the first 30 minutes of the day. And that would pretty much continue until the end of the day, while miraculously being extremely productive in the process as well. I am excited to be moving close to Boston in the distant future to continue this strike, and I apologize in advance to colleagues at Harvard or Cornell for the resulting noise!

I want to thank HPI and the research school for providing me with dream-conditions to do research, the fact that I have absolutely no idea how to apply for funding proposals and spent more than 80% of my time fully focused on research is unheard of. Special thanks go to the endless efforts of Prof. Robert Hirschfeld and Prof. Andreas Polze for running what later became "our" research school on Service Oriented Systems Engineering. And the valuable personal and academic feedback they have provided over the years.

I thank all my co-authors which would add another page to the acknowledgements if I were to thank them all individually, I therefore refer to the references and highlight two particularly impactful co-authors who for some reason could not get enough and kept working with me through multiple projects: huge thanks Conrad Lempert and Ingo Apel for your immeasurable contributions to my work!

Finally, I want to thank Anna for always believing in the path I was traversing even though none of us ever knew where it was going… It blows my mind that you have always kept seeing the road forwards while I explored every possible side-road before heading there. It has been a long ride, but I am infinitely excited to continue this amazing adventure we call life together! I will probably have to cry, but I cannot wait to cheer on our rooftop terrace on Roosevelt Island while overlooking the beautiful campus of Cornell Tech!

PUBLICATIONS

Some ideas and figures presented in this thesis have appeared in the following publications, specific chapters and publications are listed.

Chapter 3 is composed of these two papers:

Thijs Roumen, Jotaro Shigeyama, Julius Cosmo Romeo Rudolph, Felix Grzelka, and Patrick Baudisch. 2019. SpringFit: Joints and Mounts that Fabricate on Any Laser Cutter. In Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19). ACM, New York, NY, USA, 727–738.
DOI: https://doi.org/10.1145/3332165.3347930

Thijs Roumen, Ingo Apel, Jotaro Shigeyama, Abdullah Muhammad, and Patrick Baudisch. 2020. Kerf-Canceling Mechanisms: Making Laser-Cut Mechanisms Operate across Different Laser Cutters. In Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST'20). ACM, New York, NY, USA, 293–303.
DOI: https://doi.org/10.1145/3379337.3415895

Chapter 4 is composed of these two papers:

Patrick Baudisch, Arthur Silber, Yannis Kommana, Milan Gruner, Ludwig Wall, Kevin Reuss, Lukas Heilman, Robert Kovacs, Daniel Rechlitz, and Thijs Roumen. 2019. Kyub: A 3D Editor for Modeling Sturdy Laser-Cut Objects. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Paper 566, 1–12. DOI: https://doi.org/10.1145/3290605.3300796

Thijs Roumen, Ingo Apel, Thomas Kern, Martin Taraz, Ritesh Sharma, Ole Schlueter, Jeffrey Johnson, Dominik Meier, Conrad Lempert, and Patrick Baudisch. 2022. Structure-Preserving Editing of Plates and Volumes for Laser Cutting. In *submission to UIST'22*

Finally, chapter 5 is composed of these two papers:

Thijs Roumen, Yannis Kommana, Ingo Apel, Conrad Lempert, Markus Brand, Erik Brendel, Laurenz Seidel, Lukas Rambold, Carl Goedecken, Pascal Crenzin, Ben Hurdelhey, Muhammad Abdullah, and

Patrick Baudisch. 2021. Assembler^3: 3D Reconstruction of Laser-Cut Models. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21).* ACM, New York, NY, USA, Article 672, 1–11. DOI: https://doi.org/10.1145/3411764.3445453

Thijs Roumen, Conrad Lempert, Ingo Apel, Erik Brendel, Markus Brand, Laurenz Seidel, Lukas Rambold, and Patrick Baudisch. 2021. AutoAssembler: Automatic Reconstruction of Laser-Cut 3D Models. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (*UIST '21*) ACM, New York, NY, USA, 652–662.
DOI: https://doi.org/10.1145/3472749.3474776

DECLARATION

I hereby confirm that

- this dissertation is the result of my own work, it was prepared without unauthorized help and using only the given literature,
- this dissertation has not been previously submitted, in part or whole, to any university or institution for any degree, diploma, or other qualification
- I am aware of the doctorate regulations of the department for Digital Engineering of the University of Potsdam from November 27, 2019

Ich erkläre hiermit, dass

- ich die vorliegende Dissertationsschrift selbständig und ohne unerlaubte Hilfe angefertigt sowie nur die angegebene Literatur verwendet habe,
- die Dissertation keiner anderen Hochschule in gleicher oder ähnlicher Form vorgelegt wurde,
- mir die Promotionsordnung der Digital Engineering Fakultät der Universität
  Potsdam vom 27. November 2019 bekannt ist.

Thijs Roumen

Potsdam, June 15th, 2022

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

Laser cutters are fast and precise fabrication machines. Maiman built the first research prototype in 1963 using radiation in a ruby [76], and the first CO_2 based functional cutter was built a year later at Bell labs in 1964 [87]. The initial decades of use took place exclusively in industrial and research contexts. As the technology matured and the machines became more accessible, a new class of users started to embrace the power of the technology around the 2000s, mostly for hobbyist use cases [46]. These "tech enthusiasts" have since created millions of models for laser cutting.

In contrast to the industrial users before the 2000s, tech enthusiasts started sharing their models widely on online repositories (e.g., thingiverse [117], GrabCAD [44], myMiniFactory [79]), mostly to showcase what they made and to allow others to reproduce the models. Surprisingly, these models are largely *one-offs* made by single individuals and, as we will explain later, it is hard to reproduce these models for anyone other than the original modeler. For laser cutting to become a mainstream phenomenon and thus become relevant to other users than tech-enthusiasts [13], it is crucial that non-experts can download and reproduce models created by experts, and that those experts build on the work of others to create more advanced and useful models. This progression is currently held back by machine-dependent exchange formats. The central thesis of this work therefore is:

For laser cutting to progress into a mainstream phenomenon, we need a machine-independent (aka portable) sharing infrastructure.

In this chapter we first analyze the underlying problem of the current sharing infrastructure, we then outline three approaches to progress the field that are presented in detail in this thesis and summarize the contributions. Finally, we conclude with an outline of the thesis.

1.1 WHAT IS HOLDING BACK PORTABILITY?

We begin by taking a closer look at the underlying challenge. Models are currently shared in the form of 2D cutting plans; vector drawings the laser traces when cutting. When traversing the paths in the cutting plan the laser removes material, the amount of material removed when cutting is called "kerf" or the width of the cut. As studied in-depth by Uslan [122] kerf depends on the laser used, the material, the thickness of the material, the state of the machine, the air flow within the machine, and various other parameters that may vary within the same day of cutting.

The width of the cut is reasonably harmless when cutting 2D shapes or engraving objects using the laser cutter. However, when creating models that either consist of more than one piece (e.g., consisting of multiple pieces such as 3D constructions or models embedding non-laser cut components), variations in cut width causes severe problems. The canonical way to combine multiple pieces is by embedding joints, mounts, or mechanisms in the cutting plans. As we present below, these rely on tight tolerances and thus fail to operate reliably when the cut width varies.

Figure 1 shows how embedding a plastic arcade button requires the designer to add a mount to the model. The commonly accepted solution is to create a hole that has the same shape as the button but is a *tiny* bit smaller than the button. This type of mount is called a *press fit* mount. When *forcing* the button into such a press fit mount, the tightness of the hole causes the plywood to stretch a tiny bit to accommodate the button. This means that the plywood now acts as a *spring*, and the stretching of that spring holds the button in place.

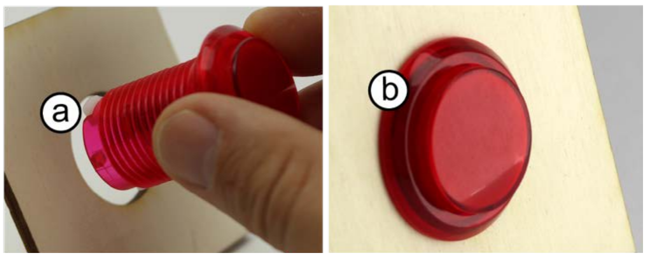


Figure 1: (a) A key design element in laser cutting is the press-fit where a physical object is forced into a slightly smaller opening. (b) The opening now acts as spring, securely holding the object.

Unfortunately, when other users try to reproduce such a model, they switch to *their* laser cutter and *their* material, which introduces variation in the size and stiffness of the mount, this commonly causes the mount to fail. As shown in Figure 2, (a) some user's lasers produce a thinner cut than the original model was designed for. This makes it impossible to assemble the model, as the button does not fit into the mount anymore. (b) other users' lasers produce a wider cut. This causes the button to fall out of the mount. (c) This user tries to reproduce the model from acrylic. Acrylic is more brittle than plywood. The hole size that worked fine with plywood now causes the material to pop when the user forces the button in.

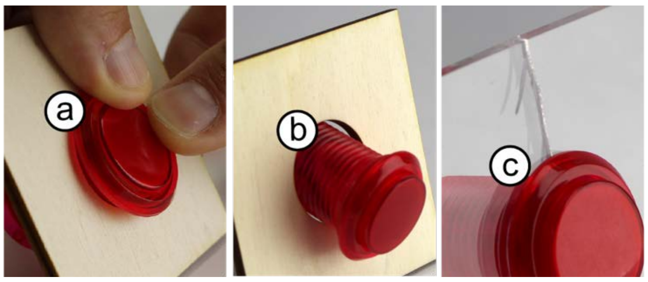


Figure 2: The carefully tuned press fit from Figure 1 fails when fabricated on different laser cutters. (a) On machines with smaller kerf, it cannot be inserted completely. (b) On machines with wider kerf, it is loose and falls out. (c) When cut from brittle material, it breaks the model.

The mount for a button is a somewhat simplified version of connecting laser-cut joints. Joints manifest themselves just like mounts by compressing material, however the same spring tends to be repeated along the surface of the plates resulting in a firm, homogenous fit. As a result, they are even more subject to failure under variations in kerf.

While joints and mounts exert a constant force, a bearing for example should hold its axle in place without applying any force to the axle. In Mechanical Engineering this is referred to as "*loose fit*" [60], achieving this in a laser-cut model requires the size of the opening to be tuned properly. As shown in Figure 3, without proper tuning, a bearing that is too loose introduces slack. This slack tends to cause mechanical issues.

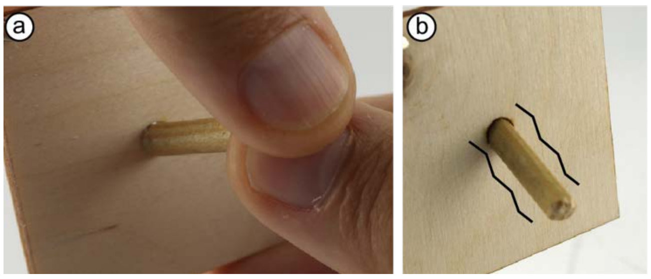


Figure 3: (a) When fabricated on a machine with smaller kerf, this bearing gets too tight. This causes friction or even prevents users from inserting the axle. (b) On machines with wider kerf, bearings are subject to slack, potentially causing adjacent mechanisms to jam.

Unfortunately, tuning tends to get lost when manufacturing a model on a different machine—as is the case when sharing a model, and the resulting models are again subject to slack and/or jamming.

This problem affects a range of mechanisms, including three of the four primary types of mechanisms with moving parts [5] shown in Figure 4. Red highlighting indicates areas where kerf-related problems occur.

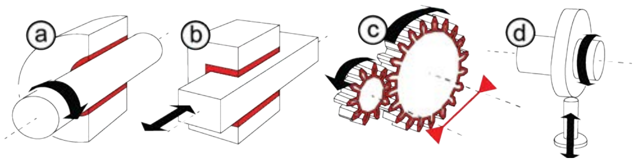


Figure 4: Three out of four elementary classes of mechanisms [5] are subject to kerf-related issues. Susceptible surfaces are marked in red. (a) The revolute pair includes mechanisms that operate like the bearing shown before, (b) a prismatic pair allows a rod with rectangular cross section to slide forth and back, and (c) a pair of gears. (d) Only cam/follower mechanisms remain unaffected, as they are typically spring-loaded.

We conclude that the existing sharing infrastructure built on machine-dependent 2D cutting plans falls short when models are more complex than a single 2D shape. They will fabricate fine once but sharing those cutting plans inevitably leads to dysfunctional models.

1.2 PORTABLE LASER CUTTING

In this thesis, we create a technological basis that allows users to *reproduce* models made by others and *build on the work of others*. We approach this in three steps: (1) converting machine dependent 2D cutting plans to cutting plans that are tolerant to machine variations, (2) sharing cutting plans in 3D instead of 2D, making it easier to build on the work of others and (3) converting legacy 2D cutting plans to 3D. We built software tools to facilitate each of these steps.

1.2.1 2D cutting plans that are tolerant to machine variations

Our initial approach is to patch the existing machine-dependent 2D cutting plans. As identified before, the problems occur when there are elements like joints, mounts, or mechanisms in cutting plans. We propose novel mechanical variations of these elements that support a much broader range of tolerances, by using springs (for joints and mounts) or wedges (for mechanisms). The resulting software tool identifies problematic elements in 2D cutting plans and swaps them out for our elements that are tolerant to machine variations. The tool outputs 2D cutting plans that fabricate on a range of materials and all typical laser cutters, but still in the same format, so they can be shared on the same platforms as before. As shown in Figure 5, the modified models fabricate on *any consumer laser cutter* and *in a range of materials*.

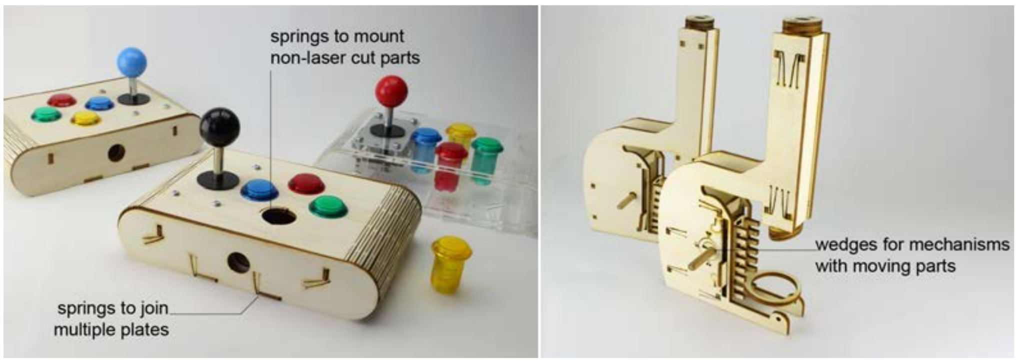


Figure 5: Models made using modified 2D cutting plans fabricate reliably on any typical lasercuter and in a range of materials, modified using our software tools SpringFit [101] and KerfCanceler [97]. The models in the background show the original (machine-dependent) version.

1.2.2 sharing models in 3D

Machine-independent cutting plans allow users to reliably *reproduce* models made by others, but it does not help users *build on the work of others* as we set out to do. To accommodate this, we built a *3D modeling environment* for laser cutting. Representing models in 3D as shown in Figure 6 allows our software to generate machine-optimized joints, mounts, and mechanisms when users export models to 2D cutting plans (to fabricate on their laser cutter). Users then share 3D models and use the 3D environment to modify models made by others. We extend this volumetric environment with tools to edit individual plates, allowing users to leverage the efficiency of volumetric editing while having control over the most detailed elements in laser-cutting (plates).

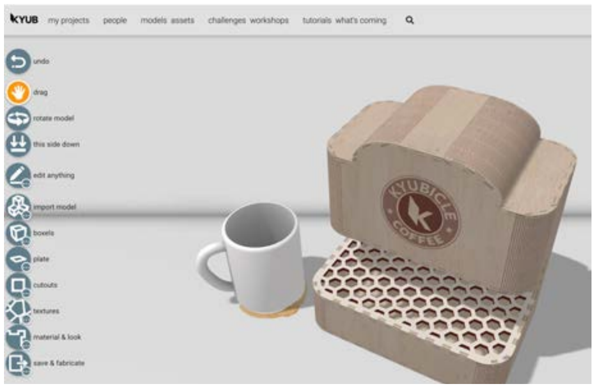


Figure 6: Representing models for laser cutting in 3D, allows users to export the model to 2D cutting plans that are optimized for their machine and material at hand. It furthemore makes it easier to create 3D modifications to the cutting plans when modifying models made by others.

1.2.3 converting legacy 2D cutting plans to 3D models

Despite our attempt to create a new 3D modeling environment, there are still a lot of models represented in 2D formats, and entire engineering/design careers hinge around working with these formats. We suspect 2D cutting plans continue to remain the standard while that is the case. And as a result, the field of laser cutting fails to really move forwards. We therefore revisit this issue by reconstructing the legacy 2D cutting plans into 3D models, which users can modify using our 3D modeling environment.

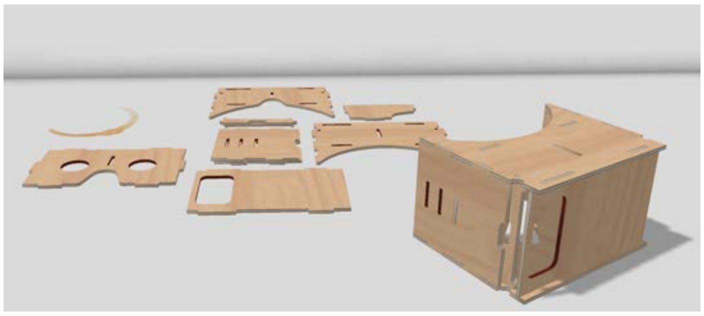


Figure 7: We reconstruct 2D cutting plans into 3D models as shown here at the example of a VR headset represented as 2D plates and the resulting 3D assembled model. We built an automatic pipeline to achieve this, and we provide an interactive fallback for models that turn out to be challenging.

We first extend our 3D modeling environment with the ability for users to interactively reconstruct 2D cutting plans. Our algorithm analyses the geometry of the 2D cutting plan in a 5-step pipeline, upon which users load the cutting plan in the 3D environment and puzzle together the plates. Based on high success rates, we revisit the analysis algorithm by turning it into an automatic pipeline using a heuristics-based beam-search algorithm. The interactive workflow continues to serve as a fallback for the subset of models which cannot be automatically reconstructed.

1.3 CONTRIBUTIONS

3D laser-cut models that work across typical laser cutters. The main contribution of this thesis is that sharing models in 3D allows for machine-independent fabrication, and furthermore allows others to make high-level parametric changes to models. The existing 2D cutting plans did not allow for this as they make implicit reference to specific machines and materials. To accomplish this high-level goal, we make the following contributions:

A pipeline to detect machine-dependent features in cutting plans. We contribute a software-pipeline to detect joints, mounts, material thickness, and structure in 2D cutting plans. We use this to either modify existing problematic features or to present the plates in a 3D editor for users to assemble into a 3D model.

An algorithm to "upgrade" legacy models. We contribute an algorithm to convert 2D cutting plans into 3D models automatically. This allows users to make parametric changes to legacy 2D cutting plans and share models with users that operate different laser cutters.

Novel mechanisms that counter-act kerf. We identify that kerf affects mechanical elements that require precision to operate such as joints, mounts, and mechanisms. We contribute with new variations of these elements based on springs and wedges, which fabricate in a range of materials and on all typical laser cutters.

Software. We contribute a series of software tools and systems to demonstrate the above-mentioned contributions. We integrate the bulk of these tools into a 3D modeling environment for laser-cutting, called kyub, comprised of over 140,000 lines of code to demonstrate practical impact of the work.

1.4 STRUCTURE OF THIS THESIS

We begin this thesis by reviewing the related research on personal fabrication, portability, and the role of laser cutting (chapter 2). Afterwards, we dedicate one chapter to each of the three steps of making models for laser cutting portable: making 2D cutting plans tolerant to machine-variations (chapter 3), representing laser-cut models in 3D instead of 2D (chapter 4), converting legacy 2D cutting plans to 3D models (chapter 5). We conclude this dissertation with a discussion of the benefits and limitations of portable laser cutting and provide our view on the future of personal fabrication and the resulting challenges (chapter 6).

2 RELATED WORK

In this chapter we look at existing software support for laser cutting, interoperability of exchange formats for other fabrication processes, functional (non-geometric) specifications of fabrication models, and we look at the exchange practices of hobbyist fabrication communities. We start out with a short dive into the history of portability in the field of computing.

2.1 SHORT HISTORY OF PORTABLE COMPUTING

Baudisch and Mueller [13], observe an analogy between the history of computing and the state-of-art in fabrication. They use this as a predictor of personal fabrication as the next phase, after the current phase in which tech-enthusiasts reign the field (analogous to say, the homebrew computer club in the late 70s [39]).

If that analogy held up, one would assume a much more advanced state of fabrication, as orders of magnitude more hobbyists are fabricating now, compared to the few who joined computer clubs back then. The observation that users still largely create and share hard-to-reproduce one-offs indicates that we may be on a different, less favorable, track. To understand what lead to this difference, we narrow in on the history of computing *before* the adoption of hobbyists.

2D cutting plans form an abstraction away from G-Code or actual machine code. There has been a long period since the introduction of the first computer, where commands were written in highly machine-specific assembler code and no program would run on other machines than what they were written for. Initially purely academically Böhm [19] described an abstract programming language *L* that could be compiled back to machine code using the language itself. Soon thereafter Hopper coined the term *compiler* and demonstrated the first practical example for the A0 programming language [52].

The "high-level" languages described above, analogous to the 2D cutting plan, form an abstraction over machine code, but do not compile across computer architectures. Strong et al., [113] described this problem of machine-specific languages and how they keep the evolution of computing back. The first cross-platform compiler was written in COBOL and demonstrated in 1961 by compiling a program to run on the UNIVAC II as well was the RCA 501 architectures [69]. In the wake of this, McIlroy introduced the notion of re-usable software components [58] at the first international conference on software engineering. And finally, with the introduction of the Portable C Compiler [111] in 1973 software as complex as operating systems were written in those high-level languages that compile to many different machines.

With this degree of portability and abstraction, the hobbyists who embraced computing back in the late 70s had a technical basis to build on the work of others and run their code across different platforms. A crucial difference to the fabrication tech-enthusiasts we see now. Therefore, the analogy with computing and its related promise of mainstream adoption, does not add up unless we go back and address the problem of portability first.

2.2 SOFTWARE SUPPORT FOR LASER CUTTING

As stated in the introduction, editing 2D cutting plans for laser cutting is hard and requires substantial domain knowledge about the machines and materials at hand [13]. As a result, researchers have explored several tracks to make laser cutting more accessible: reducing the barrier to interface with the materials and machines, embedding the required domain knowledge in design tools and systems, and proposing alternate workflows for laser cutting to bypass such design challenges.

2.2.1 reducing the barriers to interface with materials and machines

Uslan [122] studied mechanical properties of laser cutting and identified that kerf is caused by a combination of many factors such as the air circulation, the material, the material structure, the type of laser, the cleanliness of the lens and more. Yildrim et al., [137] confirm that users, including professionals struggle with the lack of control over these parameters when using fabrication equipment. Controlling either of those dimensions therefore increases the reliability of the laser cutter. *SensiCut* [32] supports users with more accurate data about the specific material inside the machine, they equip laser cutters with a speckle sensor to automatically identify material properties before the cutting process. This additional information allows for better kerf control and prevents users from cutting harmful materials. *Kerfmeter* [63] attaches a motor and camera to the laser cutter to provide an automatic kerf calibration routine before cutting.

Others have focused on building software support for later stages in the laser cutting process, *Fabricaide* [107] lets users preview the nesting of plates, and material consumption while modeling. This feedforward in the fabrication workflow allows users to prevent unanticipated situations. *PacCAM* [103] provides an interface for the nesting itself. In a later stage of the fabrication process where users assemble individual pieces into a 3D model, *Roadkill* [2] reduces effort and increases the speed of the assembly process, and *Daedelus in the dark* [24] makes this process accessible to visually impaired users.

2.2.2 embedding the required domain knowledge in design tools and systems

Researchers have been building design tools and systems to reduce the effort of hard and time-consuming challenges in the modeling process. Elements that require high precision such as joints, mounts, and mechanisms are hardest to design as a user in conventional 2D editors, while software tools excel at providing the required precision. Kim et al., [64] showed that users face challenges when adjusting models for machine-precision, beyond just laser cutting. *Joinery* [140] therefore generates joints in 2D cutting plans automatically. *MetaSVG* [136] encodes data about joints in svg files, allowing the joints to adjust to variations in kerf or material thickness. And *sketch-n-sketch* [49], independent of the specific laser cutting domain, allows users to manipulate SVG files using a parametric design language.

A major milestone was the shift towards 3D modeling for laser cutting, specifically with *FlatFab* [29] focused on intersecting plates, while *FreshPressModeler* [26] generates enclosed structures from 3D models. *CutCAD* [49] lets users still manipulate the model in 2D while previewing the result as a 3D visualization. *CODA* [125] aids with alignment of 3D models in general-purpose editors. In the context of furniture, *DESIA* [127] converts 3D models to interlocking structures and *SketchChair* [104] lets users model using a sketch-based interface. Finally, *Laser Origami* [81] uses 2D cutting plans as input but produces 2.5D models by bending material within the machine, and *LamiFold* [70] achieves that for mechanisms by laminating plates after cutting.

2.2.3 alternative fabrication workflows

A completely different approach to abstract away from 2D cutting plans is the idea of interactive fabrication [129]. In ultimate form this takes away the role of models as an engineered effort and facilitates walk-up use of fabrication equipment. *Constructable* [82] is an early practical example of this where users walk up to a laser cutter and instead of sending a 2D cutting plan, they use laser pointers to directly specify what they want to make on the machine. *RoMa* [92] brings this form of interactive turn taking to 3D printing using AR to preview results, and *FormFab* [83] is the first true interactive fabrication workflow in that it performs real-time deformation while the user uses their hands in 3D as input.

Beyond the specifics of laser cutting, *Carpentry Compiler* [130] is more broadly focused on woodwork, it creates an assembly-level representation of carpentry models and allows users to make parametric changes. While exporting to assembly instructions using a variety of more and less manual fabrication tools. *LaserFactory* [84] aims to extend laser cutters with more broad fabrication workflows consisting of multiple operations like pick-and-place or soldering into a single workflow. Similar to how Katakura et al., [62] let 3D printers extend their capabilities by fabricating a range of new tools.

2.3 INTEROPERABILITY OF EXCHANGE FORMATS

Beyond laser cutting researchers and professional users have struggled to create more interoperable exchange formats. There is a wealth of representations and formats around to capture 3D structures, often targeted at assemblies of parts or 3D printed structures. Attene et al., [8] provide an overview of representations and the trade-offs between them, their high-level categories are representations based on volumes, surfaces, primitives, and procedural generation. This variety of representations forces users to balance upfront how to represent objects for their given use-case [45]. There are interchange formats like STEP [93], which in principle support the full range of representations, but in turn require a definition and maintenance of the content in all different representations and then allow switching between representations.

There are tools to convert between representations, especially converting from shape models as composed in general purpose 3D modeling environments to fabrication-aware representations. These are typically one-way conversions making it hard or impossible to reverse the process. In the context of laser-cutting, the most common conversion tool is *Slicer for Fusion360* [10] (discontinued since 2020). This tool allowed users to convert 3D models to a range of different typical structures of plates approximating the initial shape. *Slices* [28] is a specialized version of this achieving an even stronger relation between the initial volume and resulting plate structure. *Fresh Press Modeler* [26] and the follow-up publication on bevel joints [116] achieves such conversion specifically for volumetric and watertight models. Furthermore, *Platener* [15] and *CoFiFab* [114] convert generic shape models to partially laser-cut and partially 3D printed structures for fast fabrication and iteration.

Outside of the laser-cutting domain, there are various conversion tools [43] however the process tends to be lossy, making features that exist in another mode undiscoverable, and typically the conversion comes at a cost of expressivity, or even require fixing of models that break in the process [91]. For example, Wu et al., [131] recover structure of meshes that may result from poor 3D scans. *InverseCSG* [33] converts primitive models based on triangular boundary representations to a CSG tree, enabling powerful volumetric editing. Other approaches aim to identify higher level structures in the models such as Fish et al., [38] who represent shape families, Tulsiani et al., [119] who machine-learn using primitive volumes in models to identify abstract shapes, and *Grass* [72] detects shape patterns allowing for high-level parametric operations. Finally, *Shape-up* [20] presents a geometry processing framework using projection operators that works reliably across polygonal meshes, volumetric meshes, point clouds and other discrete geometry representations.

2.4 FUNCTIONAL SPECIFICATIONS OF FABRICATION MODELS

Outside of the field of laser cutting, researchers have investigated capturing models using high-level properties besides their geometry.

Researchers express functionality in shape-models by extending them with micro- and meso-level data. *Mesh2Fab* [134] achieves material-specific fabrication by changing the shape to adjust for material stiffness. They use example models made using wood and metal, the software suggests variations based on the material options. In *Deformable Characters* [110] users specify a starting shape and target poses for a character to take, the system computes the material composition and required machine operations to fabricate the result. *OmniAD* [77] captures meso-level data in the form of aerodynamic properties in the models, initially based on a data-driven approach in *Pteromys* [121].

Others have varied structures *within* materials to achieve similar functional properties. Bickel et al., [17] adjust the material structure to support a desired deformation profile. They demonstrate this at the example of a shoe sole, after measuring the deformation of various materials, they represent this in the modeling environment and then use that data to create an abstract representation that varies its material structure based on the desired deformation. And *Metamaterial Mechanisms* [55] use that capability to embed entire mechanisms within the material itself.

Spec2Fab [25] is a reducer-tuner model to capture properties like deformation, optical properties, surface finish, color representation and other higher-level parameters in models for multi-material 3D printing. *Foundry* [123] extends this idea with a hierarchical model of materials to further capture the variety of material properties and their respective outcomes. And *OpenFab* [124] provides a rendering pipeline for multi-material fabrication where they express the material behavior in the form of “shaders” (they call *fablets*) that can be applied to existing models to experiment with different material properties. *Taxon* [71] expands on that idea in more experimental fabrication processes by capturing the model in a formal language and interfacing directly with fabrication machines.

Such functional specifications not only allow for less machine-specific workflows, but also open the space for interesting new applications. Zhang et al., [138] use functional descriptions of mechanisms to embed them in new shapes. Where *Grafter* by Roumen et al., [102] remix mechanisms to be into more advanced mechanical models, allowing some initial building on the work of others. Funkhouser et al., in *Modeling by Example* [41] demonstrate how to turn modeling into a search problem using large amount of data about parts and connectivity. Which later is extended in *Design and Fabrication by Example* [105] by customizing models on a functional level using mechanism templates. Finally, Schulz et al., [106] allow users to evaluate design trade-offs in real-time by parametrizing the search space of 3D model designs using abstracted properties of models.

2.5 SHARING AND REMIXING OF 3D MODELS IN HOBBY COMMUNITIES

Flath et al., [37] studied sharing and remixing behavior on thingiverse and highlight the impact of the thingiverse *customizer* (a tool in the browser to modify other people’s parametric models). They show how an increase in remixing and building on the work of others lead to a massive increase in the number of models being shared on the platform. Alcock et al., [6] agree that the customizer is powerful but add that it lacks expressivity and ease-of-use. *Grafter* [102] is a software tool targeted to facilitate such forms of online remixing in the context of 3D printed mechanisms, and the *PARTS framework* [49] enables users to specify mechanical parametric models, which again fosters remixing and modifying other people's models. Buehler et al., [21] demonstrate how building on the work of others and remixing enabled a growing online community to create a range of assistive technology devices.

ShapeAssembler [61] drives this one step further by developing a domain specific language that describes how geometry is connected and what is structurally sound assembly. They use this language to train a neural network on available shape repositories to then allow users to edit models by synthesizing assemblies as users modify parameters of the "program of the 3D model".

In observing the maker community, Hudson et al., [54] identified the need for better tools for remixing and customizing. And related to that, Stemasov et al., [112] argue for remixing and customization as a sweet spot between modeling and simply downloading models made by others and that this a key enabler for making personal fabrication truly ubiquitous. We have seen this play out in other fields as well such as the open-source software community [68].

3 2D CUTTING PLANS TOLERANT TO MACHINE VARIATIONS

As introduced in the introduction, different laser cutters and materials lead to a variation in the amount of material removed when cutting. Constructions of advanced 3D models using laser cutters are held together by joints and mounts, and some gain mechanical functionality from mechanisms. These components heavily rely on precise tolerances and thus fail under variations in kerf.

In this chapter we introduce a tool that replaces kerf-dependent elements (joints, mounts, and mechanisms) with elements that are much more tolerant to variations in kerf, leveraging mechanical hacks. We first present the mechanical hack for joints and mounts, we then continue with mechanisms, and present an overarching software architecture of the tool.

3.1 MOUNTS AND JOINTS THAT FABRICATE ON ANY TYPICAL LASER CUTTER

We have made the model shown in Figure 8c kerf-independent automatically using our simple web-based software tool we call "SpringFit" [101]. It takes 2D cutting plans as input (e.g., svg), locates mounts and joints and replaces them with spring-based mounts and joints, and produces the same type of 2D cutting plans as output. SpringFit thereby makes models fabricate reliably on a range of materials and any consumer laser cutter.

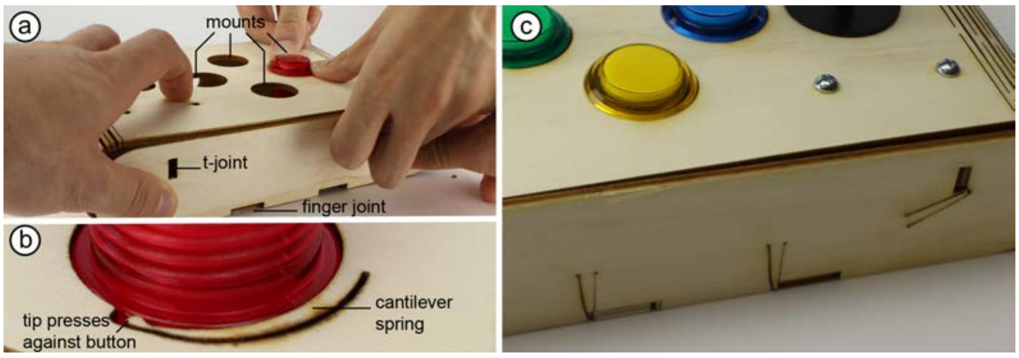


Figure 8: (a) this model fails to assemble when fabricated on a different laser cutter than it was designed for. (b) springFit tackles this by replacing traditional mounts and joints with cantilever-based mounts and joints. (c) the entire model after processed with springFit fabricates reliably on any laser cutter and in a range of materials.

3.1.1 Mounts and Joints based on Cantilever Springs

To address the problem for mounts and joints highlighted in the introduction, we propose replacing press fit-based mounts and joints with a different type of mounts and joints based on *cantilever springs* [14].

The model shown in Figure 8b features a mount based on a cantilever spring (generated by springFit) that holds the button in place. A cantilever spring is a long and thin element that is connected to the laser cut model at one end, while the tip at the free end makes physical contact with the object it is supposed to hold, here the button. This spring is curved to accommodate the shape of the button; cantilever springs generated by our system are otherwise straight.

As shown in Figure 9, the user mounts the button by inserting it into the mount (this works best if done at an angle, so that the button holds the cantilever spring back until fully inserted).

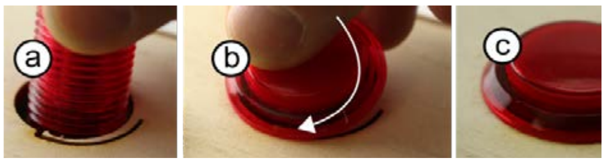


Figure 9: (a) A cantilever spring-based mount. (b) The button is best inserted by sliding it in at an angle and optionally spinning it against the direction of the spring. (c) Done.

Figure 10 shows the resulting benefit of using cantilever-based mounts: these mounts continue to work irrespective of what machine they are fabricated on and what material they are made of. They fabricate reliably on (a) a machine of small kerf, (b) a machine of wide kerf, (c) from different material. (d) They even allow inserting a slightly bigger button.

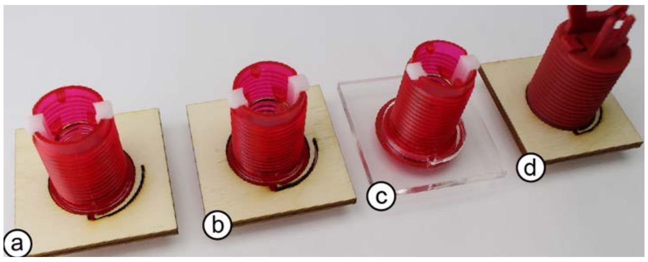


Figure 10: The use of cantilever-based mounts and joints allows one and the same models to fabricate reliably (a) on machines with small kerf, (b) with wide kerf (here simulated by eroding the model by 0.2mm), and (c) different material, and (d) even slightly different sized buttons (this one is 0.3mm bigger in diameter).

3.1.2 Why it works

Figure 11 illustrates why cantilever springs succeeds where press fits fail. (a) The springs formed by a press fit require the surrounding material to *compress*. Such "compression-based" springs are very *stiff*, i.e., even a small compression requires a large force. Implementing a certain desired force thus requires a *very* specific diameter, while minor changes in diameter easily result in a force of zero or a force large enough to break the model.

(b) Cantilever springs, in contrast, act by *bending* material, which makes them much less stiff. Obtaining a certain desired force can thus be achieved with a wider range of diameters. Since the cantilever *tolerates* comparably large changes in diameter, switching to a different fabrication machine or material is less of an issue.

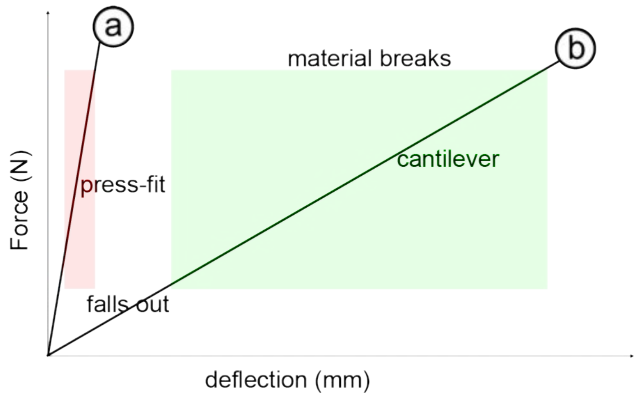


Figure 11: (a) Traditional press fit-based mounts and joints are very stiff, thus only a tiny range of "deflection" allows it to stay in the desired force range. (b) The cantilever solution affords a substantially bigger range of deflection.

Cantilever springs make it easy to tune their stiffness. As illustrated by Figure 12, we can increase (a) a spring's stiffness by a factor of 8 either (b) by doubling its diameter or (c) by cutting its length in half (as both parameters affect stiffness *cubed*).

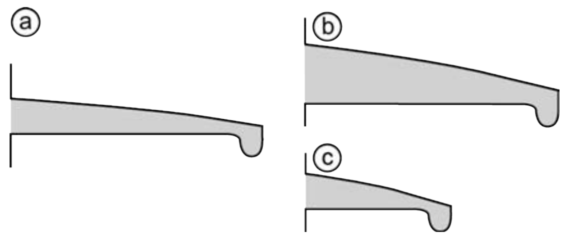


Figure 12: We can increase (a) a spring's stiffness by a factor of 8 either (b) by doubling its diameter or (c) by cutting its length in half.

As shown in Figure 13, the combination of thickness and length allows tuning the spring's desired tolerance. (a) This cantilever spring was designed to allow for typical variation in kerf (e.g., 0.1mm at 10N). (b) This longer (yet thicker) cantilever spring is as stiff as the previous spring but accommodates 7.5x more variation of up to 1.5mm, allowing users to even swap out the button for an (up to 1.5mm) larger button, such as the one from Figure 10d.

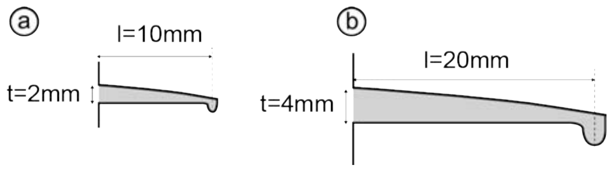


Figure 13: (a) This short and thin cantilever spring and (b) this long and thick cantilever spring are equally stiff. The latter one can deform further though, thus accommodates, for example, larger variations in kerf.

3.1.3 Cantilever-based notch-, finger- and mortise-tenon joints

While so far, we have talked only about *mounts*, we have created cantilever-based equivalents for press-fit *joints* as well. Figure 14 shows cantilever-based (a) finger joints (c) notch joint (aka cross joint) and (e) mortise-tenon joints and how they are assembled. Many models combine multiple joint types, such as the model shown in Figure 8

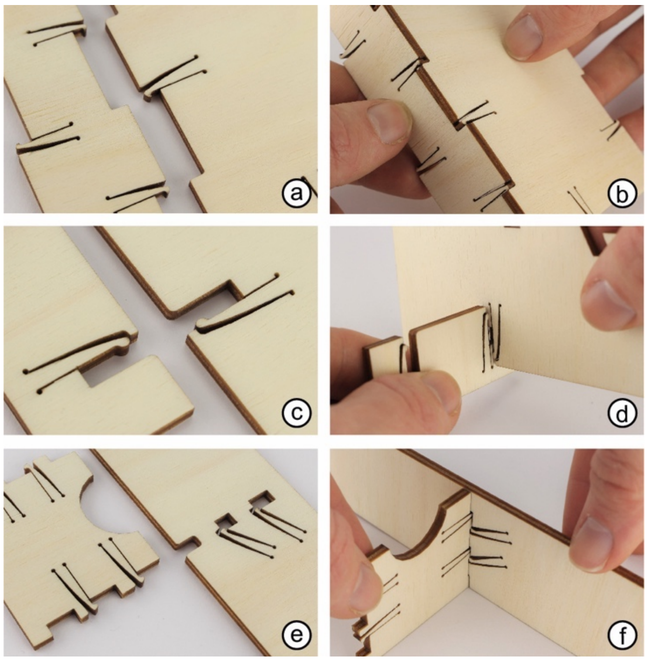


Figure 14: Cantilever spring versions of (a) finger joints and (b) how they assemble. (c,d) notch joints or cross joints and (e,f) mortise-tenon joints.

To simplify assembly, finger-joints generated by springFit provide rounded corners, as shown in Figure 15.

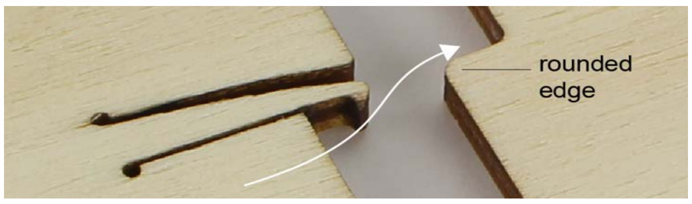


Figure 15: the rounded edge helps to push the spring smoothly when assembling.

3.1.4 Classification and conversion Algorithm

SpringFit proceeds in two automatic steps. First, it analyzes the cutting plan at hand to locate press fit-based mounts and joints, i.e., mounts, finger joints, cross joints, and mortise-tenon joints. Second, it replaces these mounts and joints with cantilever-based counterparts. It thereby computes optimal springs for each individual mount and joint. In a third manual step, a user can come in and override the suggested joints and mounts using a simple browser UI (as shown in Figure 22).

SpringFit's mount and joint classifier

As a first overview, Figure 16 illustrates the criteria springFit's joint classifier uses to locate (a) a circular mount (b) a cross joint, (c) a finger joint, or (d) a mortise-tenon joint in the svg file it is processing.

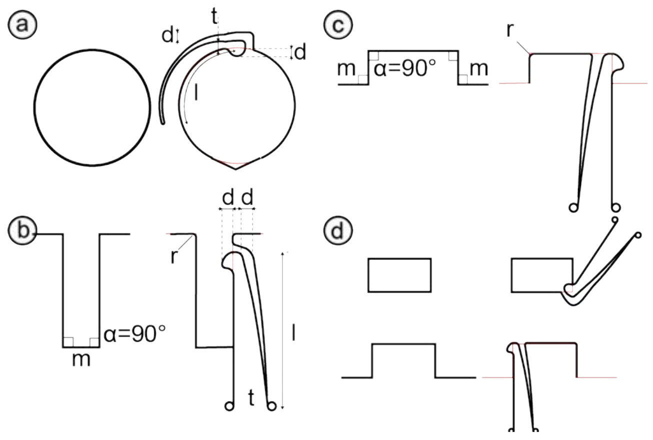


Figure 16: SpringFit's joint classifiers.

As shown in Algorithm 1, springFit uses the joint classifiers of Figure 16 to detect which segments in the svg represent what type of joint. It relies on the heuristic to find material thickness *m*, by plotting all line segments in a histogram ranging from 0 to 30mm with 100 equal bins. The most frequently occurring dimension is considered *m*. After classification, the `SpringOptimizer` (see 3.1.6 Cantilever Spring design) provides springFit with the optimal spring parameters for the given force, tolerance, and spring types. A final pass creates the actual output file in which the identified spring elements are exchanged for actual spring geometry.

Algorithm 1: spring placement

```
Input: labeled SVG file D
        press-fit force F
        tolerance t
Output: SVG file d*
lineLengths ← new array()
for each element ∈ D do
        add element.length to lineLengths
m = max(bucket_sort (lineLengths from 1 to 10mm))
for each class ∈ D do
|       if class is labeled(“press-fit”)
|       for each element ∈ class
|       |       if element= “circle” or “rectangle” then
|       |               delete element
|       |               insert mount-element
|       |       if element = “path” then
|       |       for each segment ∈ path
|       |       |       if segment.lentgh = m and
```

```
                        segment[-2].length = m
                        if segment[-2].angle = segment.angle = 90 or -90 then
                            replace segment with "fingerjoint"
                        if segment[-1].length = segment [-3].length then
                            replace segment with "crossjoint"
spring = SpringOptimizer(springTypes,F,t)
for each element ∈ D do
    if element.type = "fingerjoint" then
        insert spring.finger to element
    if element.type = "crossjoint" then
        insert spring.cross to element
    if element.type = "mount" then
        insert spring.mount to element
    add element to d*
export d*
```

Generating mounts

As illustrated by Figure 17, springFit generates round mounts to make sure that any matching object will be held at three points forming an equilateral triangle. (1) SpringFit finds the inscribed equilateral triangle in the circle, (2) scales it down to fit the minimal required circle (by using the given tolerance requirement), (3) translates that circle until it intersects with the original cutout, and (4) overlaps these two circles so the final cutout will have shape reminiscent of an oval.

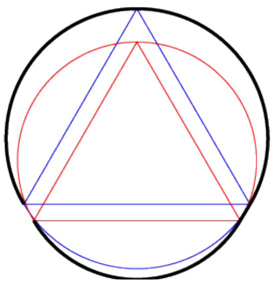


Figure 17: Mounts generated by springFit have the shape of the black shape shown here. It allows holding round physical objects at three points that together form an equilateral triangle. The red circle and the blue circle illustrate this for two specific diameters.

Cross joints

As illustrated by Figure 18, springFit classifies cross-joints by locating pairs of opposing line segments of the same length in the model with a segment of material thickness in between and straight angles.

The feature detector finds a cross joint in the model shown in Figure 18. First, (a) the path is split in segments (the dashes are the start of a new segment). (b) springFit loops through segments until one is found with length *m* (material thickness). (c) it then loops 2 more segments and checks whether element *n-1* and *n-3* have the same length (d) if so, it checks whether the angles are both 90° (or 270°) and uses the direction of the angles to determine how the spring will be inserted.

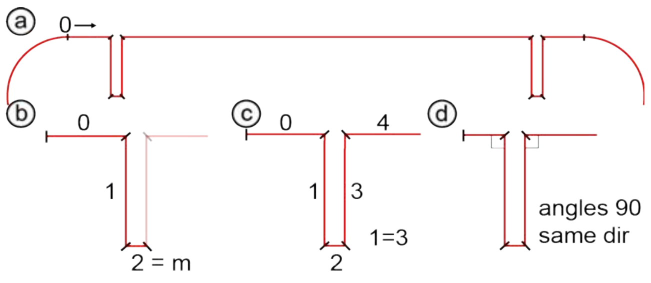


Figure 18: Notch joint classification.

SpringFit modifies this joint by inserting a cantilever spring next to the slit. It rounds the edges to prevent more brittle materials from cracking at sharp corners. The spring for cross joints lines up with the cutout.

Finger joints

SpringFit locates finger joints as illustrated by Figure 19. (a) springFit loops through the segments (b) until a segment with length *m* is found. (c) it considers two segments forwards. springFit checks whether the *n*th segment has also length *m* (d) and confirms by checking the angles. It again uses the direction of the angles to determine how the spring will be inserted.

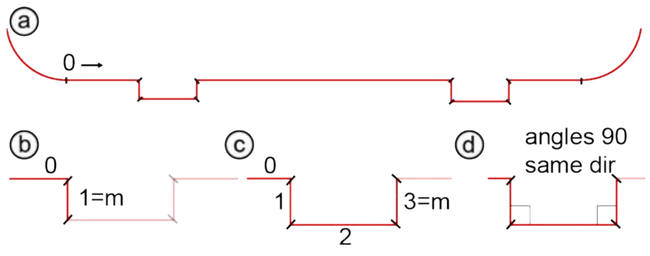


Figure 19: Finger joint classification.

SpringFit’s finger joint classifier shown in Figure 16c, identifies two parallel edges of “material thickness” length and a 90-degree connector between them as well as 90-degree connections on both other sides of the segment. The replacing cantilever spring has the three core properties of length, displacement, and thickness, are generated by springFit’s spring optimizer (more detail in Section 3.1.6 Cantilever Spring design).

Mortise-tenon joints

SpringFit classifies and converts mortise-tenon-joints as a side effect of mount and finger joint classification and conversion. One side of the joint is equivalent to a finger joint. The other side is a rectangular cutout—these are recognized and processed as rectangular mounts.

3.1.5 technical evaluation of conversion

To validate the functionality of springFit, we ran it on 14 models, which we downloaded from Thingiverse. We picked these models as to show the maximum variety of joints and mounts.

We used 7 of these models to test our claim of *material*-independence by converting models and then fabricating them from both plywood and acrylic. We used the other 7 models to test our claims of *machine*-independence by fabricating them on two laser cutters that deferred quite substantially in terms of kerf (0.12 vs. 0.20mm).

Results

Eight of the fourteen models converted in fully automated fashion, Figure 20 shows four of them.

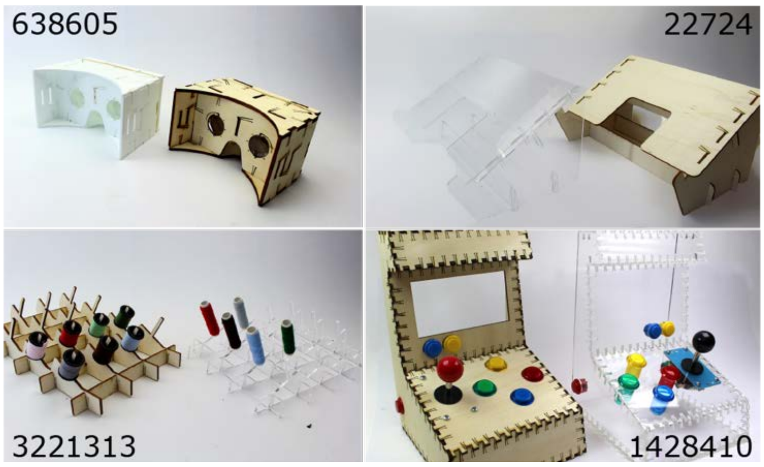


Figure 20: Four of the models we converted and fabricated as part of the first technical evaluation. (thingiverse IDs on the label)

SpringFit completed the conversion also for the other six models; the results, however, required manual fixing. (1) Three models had produced intersecting cantilever springs, which we resolved by manually deleting one or more springs (springFit allows for this, as shown in Figure 22). (2) Two models contained parts that were too small to contain the cantilever springs as shown in Figure 21. (3) One model (a heart shaped box) could not be converted because it contained bent fingers as shown in Figure 21c, springFit was unaware of this joint type.

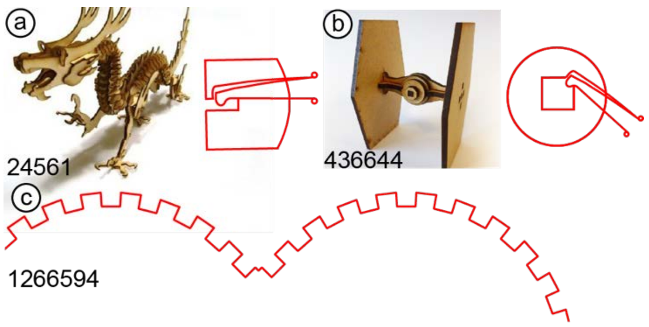


Figure 21: Models springFit could not convert (a,b) two of the models that contained parts too small to hold the required cantilever springs. (c) a model with non-straight finger joints.

All three issues are solvable in the long run. Future versions of springFit should address them by adding a better "routing algorithm" for the cantilever springs, by folding cantilever springs into the available space (similar to how springFit already produces curved springs), and by adding additional joint classifiers.

SpringFit identified all mounts and joints contained with 4% false positives. We inserted on average 105 springs per model (with the arcade of Figure 17c as extreme outlier with 477 springs). On average 4.1 springs were placed at cutouts that are not press-fit (and thus have been removed in the UI). The model with the most redundant springs was a Theremin model which has a lot of holes for bolts that are not press-fit (14 springs (27%) are not needed).

Users may choose to simply leave them (they generally do not really affect the model's functionality) or choose to remove them in the UI. For familiar models, the simple interaction shown in Figure 22 typically takes only a few clicks.

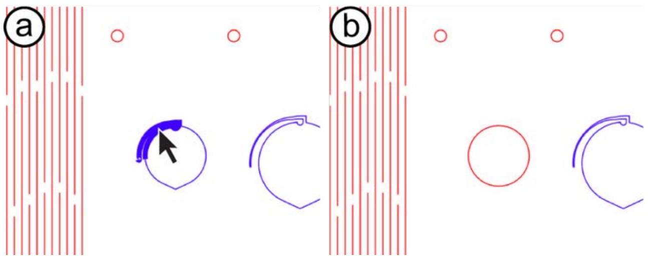


Figure 22: (a) SpringFit falsely classified this cutout as a press-fit and consequently created a cantilever spring for it. Leaving it in does not affect functionality. Alternatively, a mouse click in springFit reverts this mount to (b) the original version.

3.1.6 Cantilever Spring design

At the lowest level, springFit is about cantilever spring design. When converting models, springFit calls the function `generateSpring(force,tolerance)` that generates optimal springs for the mount or joint at hand.

The first objective of cantilever design is to create a spring with a well-defined *holding force*. Holding forces for small buttons and joints may range anywhere from 1 to 5N. We made 5N the default in springFit, but users can override it.

The second parameter springFit optimizes for is the amount of *tolerance* the spring offers, i.e., the size difference between the smallest and the largest object this mount/joint will be able to hold. This parameter defines the deflection that the spring-fit will be able to accept. For example, when the user defines 1mm of tolerance, the system will generate a spring that exerts the desired holding force around ±1mm from the original point of fit.

The `generateSpring` function takes these two input parameters and minimizes the size of the spring. It has to conform 3 additional constraints: (1) the material should not break, (2) the force needs to be consistently applied within the given tolerance and (3) the resulting spring should be able to fabricate (not too small, not too big).

Design parameters and constant stress springs

As described in section 3.1.2 "why it works", the stiffness of a cantilever can be varied by changing its shape. SpringFit specifies the stiffness k in the $F=kd$ relationship of the cantilever by using the shape parameters, l: length, t: thickness and d: deflection.

The smallest possible spring is one that has no redundant material. This happens when the stress of the spring is distributed evenly across the material as shown in Figure 23. To achieve this, springFit uses the *constant stress* cantilever model. Similar to Shin et al. [108].

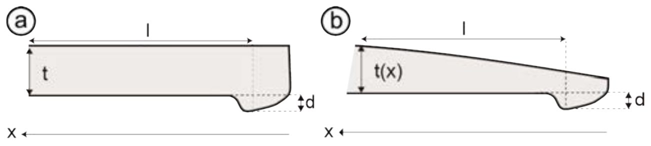


Figure 23: Design for a constant-strength cantilever. (a) Compared to usual 'bar' cantilever, (b) the constant-stress cantilever has constant bending stress along its length induced by input force at the end and thus is more space efficient.

Force/deflection relationship for cantilever springs

Next, springFit needs to know how the shape parameters influence each other and the required spring behavior. This is defined by the force-deflection relationship.

To acquire the force-deflection relationship for a cantilever, we use the Euler-Bernoulli beam theory [1]. The deflection of a cantilever d under the bending moment M is described by the elastic curve equation for the path along x:

$$\boldsymbol{d}''(\boldsymbol{x}) = -\frac{\boldsymbol{M}}{\boldsymbol{EI}} \tag{1}$$

E is the Young's modulus of the material, and I is its moment of inertia of a cross-section which is a shape dependent value which can be calculated as $I=mt^3/12$. The bending moment M is calculated by multiplying the distance from the point of force F to the point of interest (in case of a straight cantilever this will be $M=Fx$).

For curved springs with radius R and angle θ_e, springFit uses Castigliano's method [22] to derive the deflection. With the given elastic energy in the cantilever U, we calculate the deflection at the end.

$$\omega = \frac{\partial U}{\partial F} \tag{2}$$

F is the force applied at the end of the cantilever. To calculate the elastic bending energy stored in the spring, we integrate given the path C and small element ds of the cantilever. We neglected tension/compression energy here.

$$U = \int_C \frac{M^2}{2EI} ds \tag{3}$$

Thus, we can derive *F=kd* relationship resulting from solving equations (1)-(3). Table 1 shows the equations, which shows how much deflection can be caused by force *F*, given the shape of cantilever.

Table 1: The calculations based on the cantilever dynamics model for each cantilever design we used in by springFit. I0 here means moment of inertia of section at x=l.

shape	constant-stress deflection d	thickness t
straight	$\frac{2Fl^3}{3EI_0}$	$\sqrt{\frac{6Fx}{b\sigma_b}}$
curved	$\frac{FR^3}{EI_0}\sqrt{sin\theta_e}\int_0^{\theta_e}\sqrt{sin\,\theta}\,d\theta$	$\sqrt{\frac{6FR}{b\sigma_b}sin\theta}$

Optimization and criteria

SpringFit aims produces the smallest possible springs as this minimizes aesthetic and structural impact in the model. Since there are multiple parameter configurations that lead to the same stiffness (see Figure 12), springFit uses an optimization algorithm to pick the optimal design.

We write these criteria as an objective function L that penalizes for size of the spring (sizeof(π)). For springs to work across materials, we choose the minimum value of Young's modulus E and maximum stress σ_{max} from given set of materials specified by users (default are the typical materials used for laser cutting cardboard, plywood, acrylic), the parameter s is a safety factor to compensate for slight variations within the material (e.g., grains of the wood). We have a lower bound for the force (otherwise it would not be press-fit) Following these criteria, the optimization problem of the cantilever design π with design parameters length, thickness, and deflection π: {l,t,d} can be described as follows:

$$\pi^* = \arg\min_{\pi} L(\pi)$$
$$\text{s.t.}\begin{cases} L(\pi) = \text{sizeof}(\pi) \\ \sigma_{max} < \frac{\sigma_+}{s} \\ F > F_{min} \end{cases} \tag{4}$$

We obtain the actual values of Young's modulus E and the maximum strength σ_{max} of the materials from material testing (see section 3.1.7 "Technical Evaluation"). For solving the nonlinear optimization problem with multiple constraints, we use the COBYLA algorithm from the NLopt C++ library (nlopt.readthedocs.io).

With help of this procedure, springFit determines the optimal length, thickness and deflection of the cantilever that produces required force across different materials or kerfs.

3.1.7 technical evaluation of spring performance

To verify that (1) cantilever springs made from plywood deliver reliable repeatable force and tolerance, (2) to test our claim that a single spring design including its dimensions works in plywood *and* acrylic, and (3) to verify the design parameters of our springs, we measured forces and tolerances of springFit-generated springs using a testing setup.

Test set-up

The set-up we created is shown in Figure 24. It uses a linear actuator to automatically push a spring in increments of 0.1 mm against a force gauge (SAUTER FK-100).

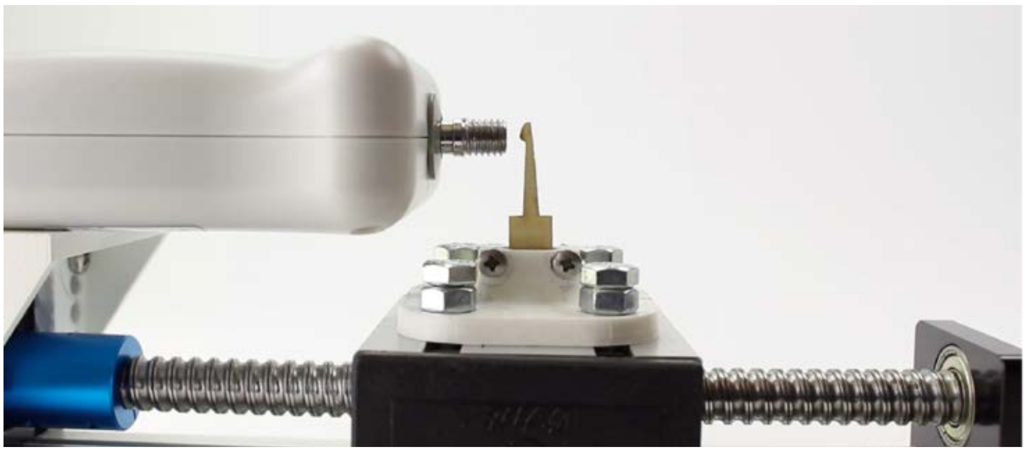


Figure 24: Spring strength example setup. Linear actuator that moves test pieces generated by springFit into a digital force gauge.

Specimen

The springs we tested were at least 3mm thick at their base and were designed to allow for at least 1mm deflection. We repeated each test with 10 springs. We generated springs optimized by springFit with two different forces. We tested straight and curved cantilever springs, from plywood and acrylic (so 2x2x2x10 samples). For plywood test pieces, we laser cut the springs *along* the grain of the material of the outer layers which is the easier side to break with the applied force.

Results

Figure 25 shows the results for the straight cantilever springs. As the diagrams show, the tested springs behaved well, i.e., produced reliable and repeatable force, which proves our optimization criterion (1).

The straight cantilever springs produced a consistent force of around 5[N] largely irrespective of the material (blue line = acrylic vs. orange line = plywood). This means that the tested springs were functional across materials, which also supports our second criterion (2).

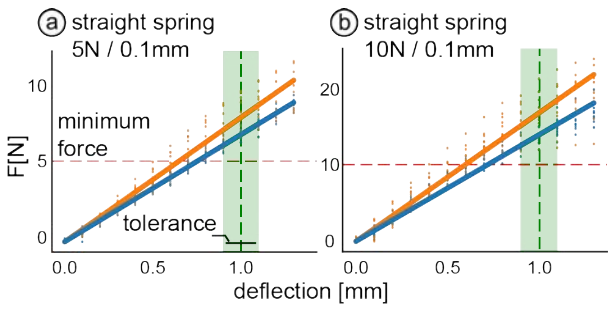


Figure 25: (a) Force-deflection diagram of generated cantilever springs with input force of up to 10N. Red dashed line shows the input minimum force and green band shows the input tolerance=0.1mm. (Blue = acrylic, orange = plywood) (b) Same diagram of bar spring with 10N.

Figure 26 shows the corresponding results for the curved springs, as used in the round mounts shown in earlier figures. As the diagram shows, the curved acrylic springs produced a slightly larger deflection given the same force compared to the straight cantilever springs. Surprisingly, the curved *plywood* springs were roughly half as stiff as their straight counterparts, i.e., they deflected twice as much.

This was caused by the non-elastic behavior of the curved springs breaking the assumption on Castigliano’s method [22]. Since the curved springs have larger deflection compared to bar springs, it will likely fail to produce the linear elasticity assumed in the cantilever model.

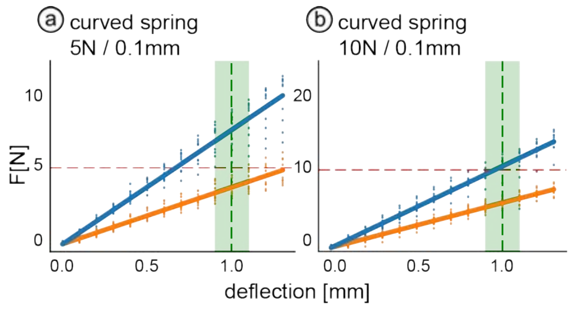


Figure 26: Same diagram for curved cantilever springs.

In summary, all tested springs behaved well. The stiffness of the curved plywood springs was unexpected, but still predictable. This allowed us to embody these findings into springFit by tuning our models there. This now allows springFit to produce springs of desired properties reliably irrespective of materials and shape. More testing would be required to find out whether the springs keep their characteristics under highly frequent or long term (dis)assembly.

3.2 KERF-CANCELING MECHANISMS

While springFit makes models with joints and mounts machine-independent, advanced models for laser cutting such as microscopes, robots, vehicles, etc. contain *mechanisms*. These mechanisms rely on precision and thus suffer from variations in kerf.

We present "kerf-canceling mechanisms" [97]. Kerf-canceling mechanisms replace laser-cut bearings, sliders, gear pairs, etc. Unlike their traditional counterparts, however, they keep working when manufactured on a different laser cutter and/or with a different kerf value. Kerf-canceling mechanisms achieve this by adding an additional wedge element per mechanism (such as the moon-shaped inset in the bearing in the center of Figure 27).

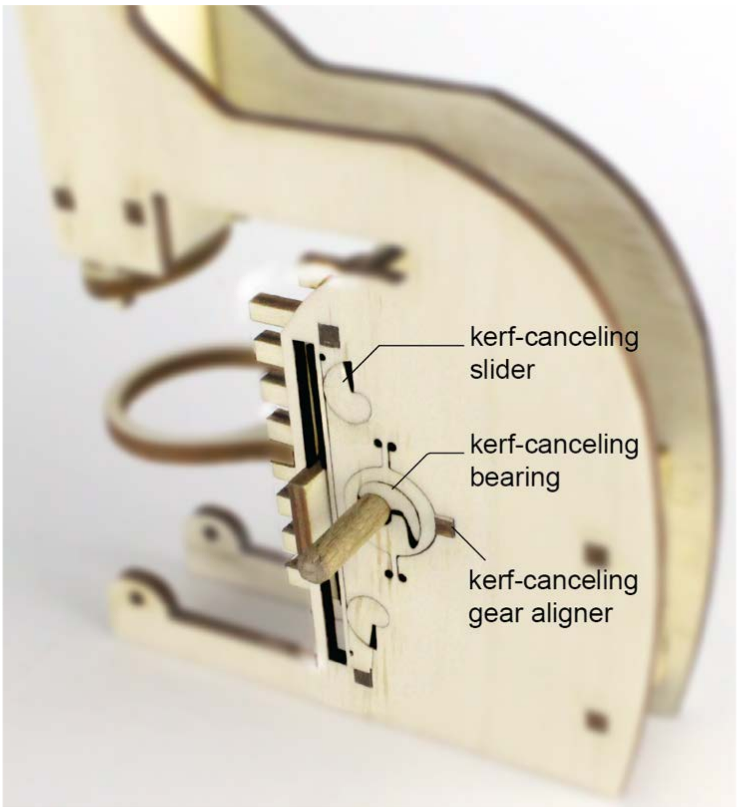


Figure 27: This laser-cut microscope (based on thingiverse id: 31632) contains three types of mechanisms that allow the microscope to adjust focus. By using kerf-canceling mechanisms, the focus adjustment operates reliably, independent of how much material the laser cutter that produced the microscope removes (kerf).

Our software tool *KerfCanceler* locates certain types of mechanisms in SVG files and replaces them with their kerf-canceling counterparts. The resulting models function irrespective of the laser cutter or kerf values they are fabricated on—making these models particularly suitable for sharing.

3.2.1 Kerf-canceling bearings

Kerf-canceling mechanisms, such as kerf-canceling bearings address this issue with the help of one additional component: the crescent-shape inset shown in Figure 28a. The figure shows how the mechanism is assembled by inserting the inset and rotating it clockwise. This jams the inset, locking it in place. At this point, the rotation of the inset has reduced the diameter of the remaining opening. The specific design of the inset causes this opening to always be of the same size, irrespective of the kerf value of the machine it was fabricated on.

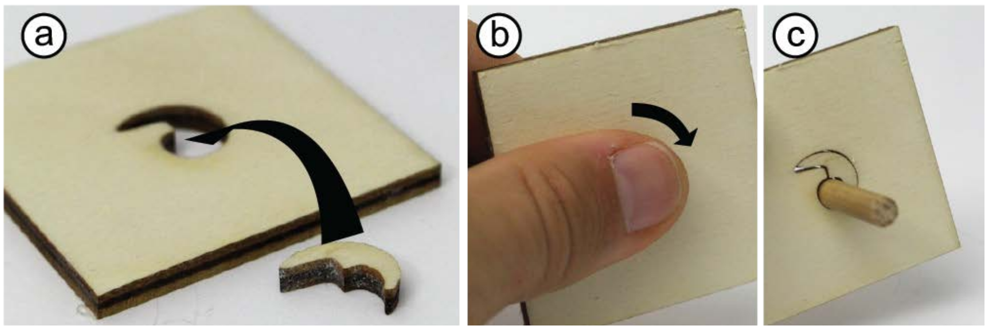


Figure 28: Assembling the kerf-canceling bearing.

As illustrated by Figure 31a, the spiral inset consists of two logical elements, which we call *jammer* and *inverter*.

The jammer is the shape on the outside of the inset. To illustrate how it works, consider a wedge [34]. As illustrated by Figure 29, a wedge-shaped inset in a wedge-shaped cutout jams when slid towards the tapered side of the cutout. If we increased kerf, the inset slides further—but ultimately it will jam just the same. Note that the distance the inset slides is proportional to the kerf of the machine.

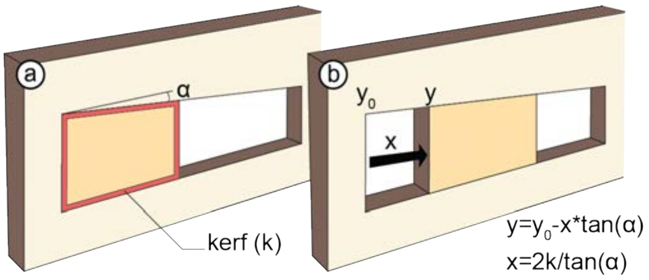


Figure 29: (a) A wedge inset jams by sliding it to the right. A larger kerf value removes the red region, (b) allowing the inset to slide further before it jams.

Applying a polar transformation to the wedge produces the spiral inset we use in kerf-canceling bearings (Figure 30). The spiral version jams when *rotated*. In analogy to the wedge, the inset's final orientation reflects the kerf value.

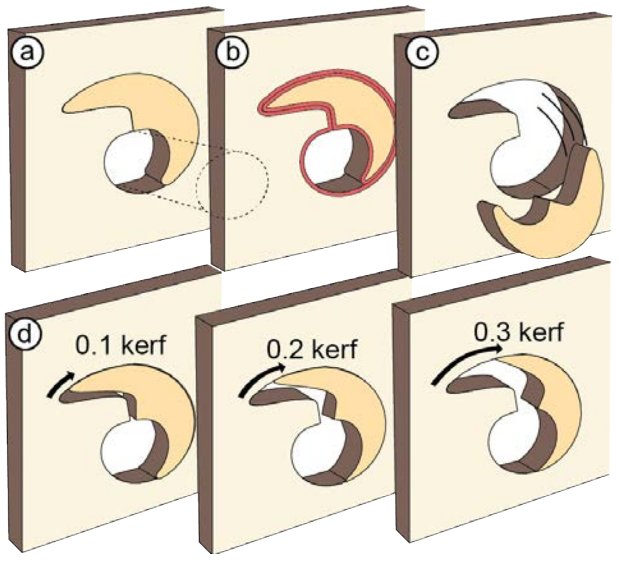


Figure 30: (a) The kerf-canceling bearing. (b) when the model is cut with more kerf, the inset gets smaller while the cutout gets wider. (c) the resulting inset falls out (d), however the self-similar shape of the inset makes that it always jams when rotated in place, even as kerf gets bigger.

The inverter is the shape on the inside of the inset. The key idea behind the inverter is that it bears the same shape as the jammer—but mirrored. Based on the jammer translating size (= kerf) into rotation, the inverter translates rotation back into size. Since its shape is mirrored with respect to the jammer, it does so inversely though, i.e., the further it is rotated, the more it reduces the opening in its center, i.e., the bearing. This allows it to keep the size of the bearing constant. With other words, a larger kerf value makes the opening wider, but also leads to additional rotation of the jammer, which in turn causes the inverter to narrow the opening further, canceling out the effect of kerf.

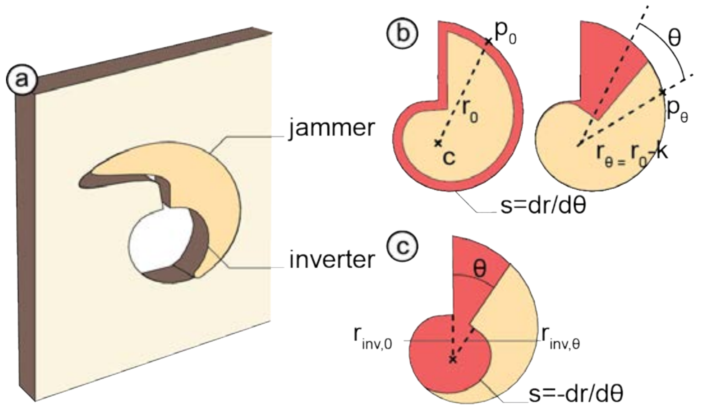


Figure 31: (a) The kerf canceling bearing consists of 2 key elements: (b) the jammer which is characterized by a self-similar nautilus shape that jams in place when rotated and (c) the inverter, which converts the rotation of the jammer back to a bearing, which ultimately holds the axle.

As illustrated by Figure 32, kerf-canceling bearings produce the same fit, irrespective of kerf and thus irrespective of the machine they were fabricated on. Even with a simulated kerf of 0.45mm the bearing continues to produce the desired fit. This exceeds the most extreme typical kerf value in a laser cutting survey by cutlasercut.com [30]. Even when executed on a milling machine with a mill bit of 1.5mm, the axle fits the resulting bearing well.

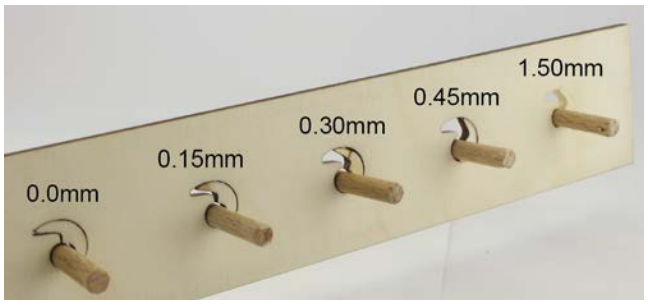


Figure 32: Kerf-canceling bearings fit their axle under variation of a wide range of kerf (by eroding the model). Even when cut on a milling machine with much more kerf.

Technical details

To help readers replicate our designs, we now present the necessary technical details. We begin with the jammer. The slope (s) is constant $s = d_r/d_\theta$, the radius thus decreases proportional to the angle θ from the center of the spiral. A given point p_0 on the contour of the jammer has a radius r_0 and corresponding angle θ_0. Another point on the same contour p_θ rotated by an angle of θ from r_0 is thus $r_\theta = r_0 - s*\theta$. We can rewrite this to calculate the angle θ between two points, given their radii: $\theta = (r_0 - r_\theta) / s$.

The cutout and the jammer have the same slope s. Because of kerf, the radius of the inset is *k* shorter (the red zone in Figure 31b). There is a point on the inset with r_0-k which, before jamming the inset, is aligned with p_0. This point jams in the contour where the radius cutout of the contour is r_0-k. We insert r_0-k as r_θ in the formula derived above, and find that the angle θ is $(r_0 - r_0 - k) / s = -k/s$.

The inverter has the same slope as the jammer, flipped (-s). A point on that spiral can be calculated using: $r_{inv,\theta} = r_{inv,0} + s*\theta$ (Figure 31c). If we substitute θ with -k/s, we get: $r_{inv,\theta} = r_{inv,0} + s*(-k/s)$, this simplifies to $r_{inv,\theta} = r_{inv,0} - k$. Kerf eroded the inset by k, so the radius from the center is k longer for every point, this results in: $r_{inv,\theta} = r_{inv,0} - k + k$ or $r_{inv,\theta} = r_{inv,0}$. We conclude that the kerf added, combined with the jamming of the inset results in a bearing of constant size.

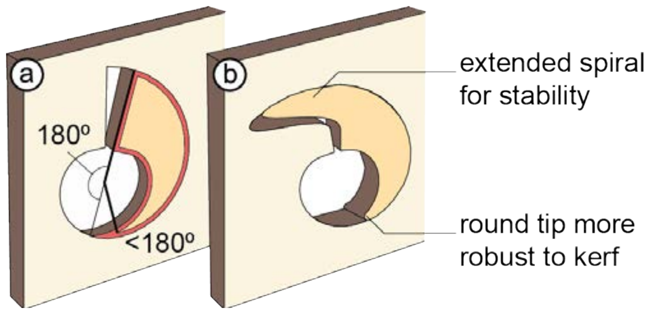


Figure 33: (a) The inset has to span 180 degrees; however, kerf makes it shorter. (b) By extending the spiral and making the tip less sharp, the inset remains stable as kerf increases.

Kerf affects the length of the spiral inset, i.e., if kerf gets wider, the inset gets shorter. To prevent it from spanning less than 180° (Figure 33a), we extend the spiral further than just 180°, by extending it on top (Figure 33b). To make the length of the inset less susceptible to changes in kerf, we round off the bottom tip.

For even better results, we manufacture the inset mirrored. As illustrated by Figure 34, kerf in laser cutting results in a non-straight edge. By mirroring the inset in the cutting plan, it gets cut from the other side, resulting in a part with the slanted edge facing the opposite direction. This allows the slanted edge of the inset to line up with the slanted edge of the rest of the mechanism (Figure 34c). An informal validation shows that flipping the inset increases the friction force by about 60%.

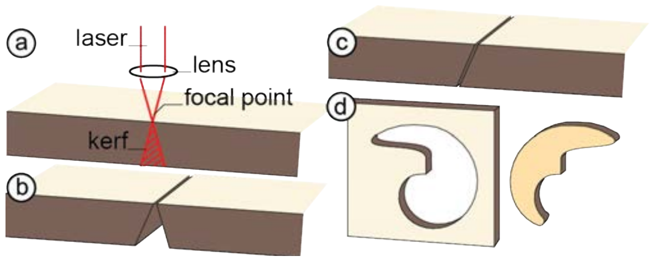


Figure 34: (a) Kerf in a laser cutter is slanted. (b) when cut from the same side, edges poorly align (c) Flipping one side of the plate results in a better fit. (d) Our software tool flips insets by default to support this.

3.2.2 kerf-canceling sliders

We have applied this concept of jammer and inverter to three other types of mechanisms. Sliding mechanisms can be orthogonal or parallel to the surface of the model. In both forms, the kerf canceling variant narrows the slit to counteract kerf. Figure 35 shows kerf-canceling sliding mechanisms. We use the principle of the straight wedge (Figure 29). The V shape between the two prongs of the inset lets it slide down to narrow the slit, a spiral wegde on top locks it in place as shown in Figure 35c.

The parallel slider is narrowed down by pushing a thin spring towards the slit. The self-similar nautilus wedges responsible for this are jammed in the surface and push the spring by 1x kerf from both sides.

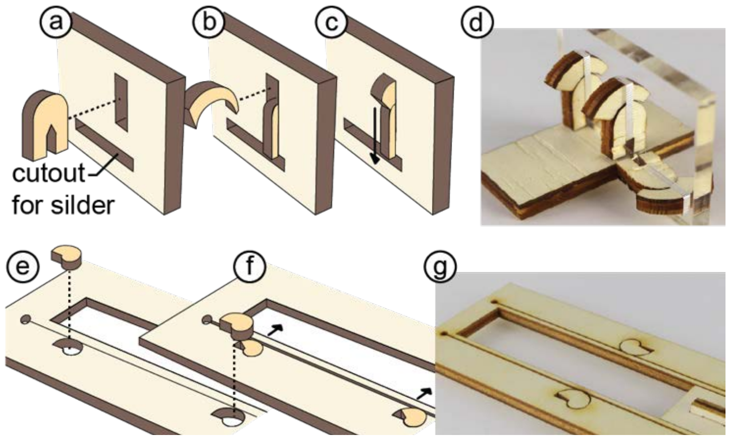


Figure 35: Kerf-canceling sliders (a-d) orthogonal, (e-g) and parallel. (a) The cutout between the prongs lets the shape slide down by 2x kerf. (b) The spiral wedge on top locks it in place. (e) For parallel sliders we insert two simple nautili next to a thin bar (f) the bar gives way as the nautili push by 1x kerf.

3.2.3 kerf-canceling gears

The kerf-problem with gears (and other mechanisms that interlock into each other) is that kerf makes them smaller, resulting in teeth of one gear to be further away from those of another. To cancel out kerf, we push them towards each other. As shown in Figure 36, we cut a slit around the bearing of one gear and add a wedge next to it to push it towards the other gear. The resulting translation makes the gears mesh again. To keep the bearing in the same plate as its surrounding we do not cut it out all the way but keep it connected to the plate with a thin (flexible) extension.

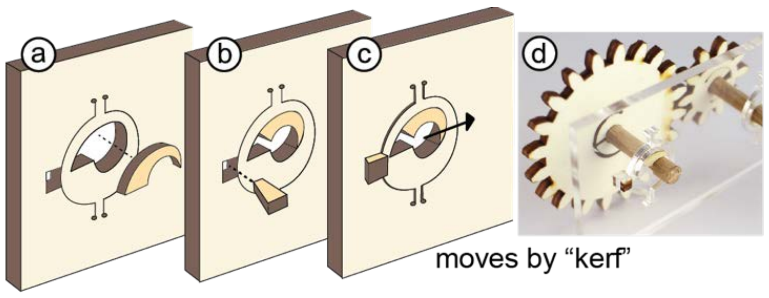


Figure 36: Assembly of the kerf-canceling gear pair. It jams the gears towards each other to compensate for the shorter teeth (a) Insert the bearing wedge, (b) then add a straight wedge next to it, which (c) jams the whole assembly to the right.

Multi-stage gearboxes by combining mechanisms

The kerf-canceling mechanisms described above can be combined to implement more exotic kerf-cancellation techniques. Figure 37 shows a combination of various wedges to form a complex mechanism: a kerf-canceling 3-stage gearbox. Both pairs of gears have to be moved towards each other. A single pair of gears is solved by moving the axles 1x kerf towards one another (Figure 35c). If we naively paired the right and the middle axles, the axle on the left would be 3x kerf away from the middle.

By *nesting* the gear pair on the left together with the middle, they are both moved 1 kerf closer to the gear on the right. Within the nested pair, the left gear is moved 2x kerf closer to the middle gear. The nester corrects kerf equivalent to the angle of the tip: the angle of the left wedge is 2x as narrow as the angle of the middle one making it correct 2x kerf as opposed to the 1x of the nested pair. When compared to the same gearbox with 0.3mm kerf, the normal one jams frequently whereas the kerf-canceled one runs fine.

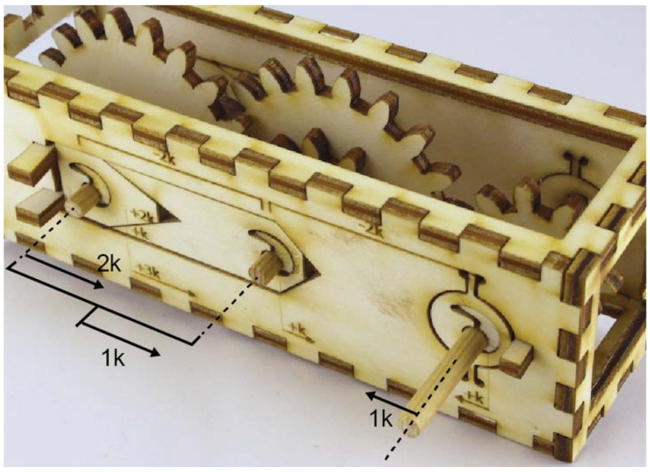


Figure 37: A kerf-canceling multi-stage gearbox.

3.2.4 The software tool: KerfCanceler

Our software tool, *kerfCanceler*, converts traditional mechanisms in 2D cutting plans to kerf-canceling equivalents. The tool takes the commonly shared SVG format as input and produces output in the same format, allowing users to share the result in existing pipelines/repositories. The software is designed to minimize redundant and uninspiring work for the designer of the model. It automatically guesses the locations and types of mechanisms and then allows users to fix if needed.

Walkthrough: converting the microscope of Figure 27

As shown in Figure 38, the conversion starts by loading a 2D cutting plan into the tool, here the microscope from Figure 27. The menu on the left offers 8 tools, three modify revolute pairs (bearings, mounts, and gear pairs), two tools for prismatic pairs (orthogonal and parallel sliders), one utility to set material thickness, a tool to remove suggestions and a tool that calls the algorithm of springFit [101] to make joints kerf tolerant.

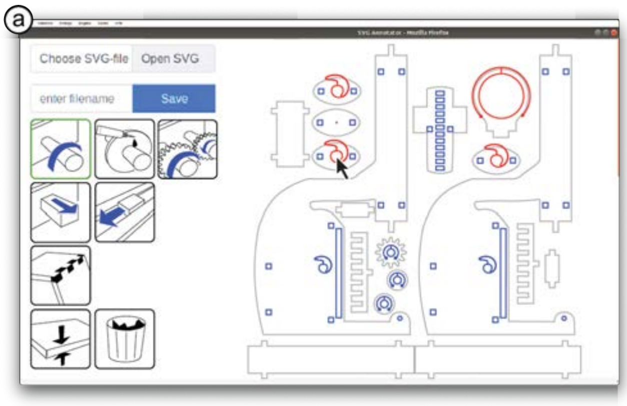


Figure 38: Converting the microscope model of Figure 1.

KerfCanceler classifies polygons when a new cutting plan is loaded (identifying rotary mechanisms with 93% accuracy, see section 3.2.7). It automatically inserts kerf-canceling mechanisms. In this example, kerfCanceler added 9 mechanisms automatically.

Kerf-canceling mechanisms require more space than their traditional counterparts, they can intersect with existing geometry in the cutting plan. KerfCanceler detects such cases and highlights them in red.

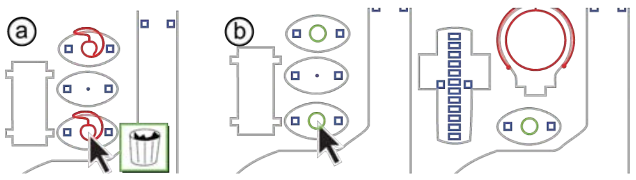


Figure 39: The user removes a kerf-canceling mechanism inserted by kerfCanceler (b) With the "remove mechanism" tool selected; the user clicks on a falsely labeled mechanism. (c) By default, all cutouts with the same diameter now have the mechanism suggestion removed (shown in green briefly to indicate the change).

The microscope has three circles which are glare-holes, but kerfCanceler guessed them to be bearings. The user removes the suggestion as shown in Figure 39b, which reverts them back to the original circular cutout. kerfCanceler recognizes that all three circles are the same size, so the user overrides them in a single click. If the user only wants to modify a single entity, it is possible to turn off "group edit".

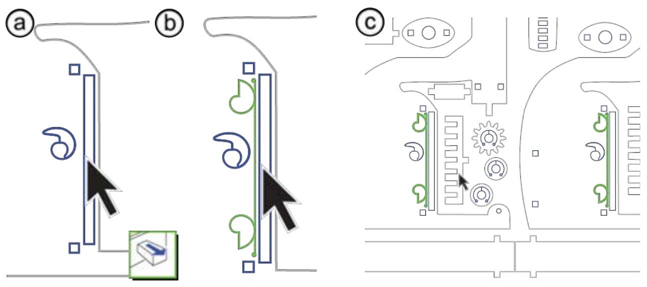


Figure 40: (a) Users add sliding mechanisms manually, using the "slider tool" (b) KerfCanceler creates a kerf-canceling version of that slider (c) both similar cutouts in the model are converted at once.

Sliding mechanisms are rare and hard to identify correctly (any polygon could be a cutout for a sliding mechanism). Based on the principle of *good guesses with little fixing*, KerfCanceler does not automatically place these. As shown in Figure 40, users apply the "slider tool" to manually turn a polygon into a sliding mechanism.

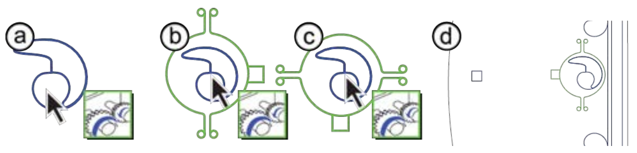


Figure 41: KerfCanceler extends a bearing with the gear tool to compensate for the increased distance between the pinion and the rack as a result of kerf.

The microscope contains a gear (aka pinion), which meshes with the rack. The "gear tool" allows users to align these. It inserts the kerf-canceling mechanism around the already existing bearing (as shown in Figure 41b). Initially, the gear is pushed from the right, by clicking repeatedly, the user rotates this to match the intended orientation. In the first four clicks it rotates by 90-degree steps. After that, granularity goes up.

In a last step, the user calls the springFit [101] algorithm to make joints and mounts kerf invariant. It extends the same data structures as kerfCanceler. We modified the algorithm to not place springs when they overlap with a mechanism (and nullify the fit) as the springFit springs tend to occur in abundance. In some models this requires manual fixing.

This process takes a few minutes, and results in an SVG that is fully kerf independent. The model will reproduce on any machine when the user shares it with others.

Once the model is cut, the user jams the insets in place (before assembling the model) and continues to assemble the model in a regular laser cutting workflow.

3.2.5 Classification and Conversion Algorithm

The algorithm to enable the workflow above proceeds in two automatic steps. First, it pre-processes the cutting plan at hand to identify mechanisms. Second, it replaces these mechanisms with kerf-canceling equivalents.

Pre-processing

KerfCanceler normalizes the SVG by breaking all SVG elements into line segments. This removes ambiguities (e.g., polylines and paths that do the same thing but are defined differently) or document properties like layers that don't influence the laser cutting.

KerfCanceler runs a parts vs cutout detection. It sorts all closed polygons by size. It checks if there is a larger polygon within which the given (smaller) polygon is enclosed and continues to do so until all are checked. It assumes that the outer cuts are outlines of parts and the inner ones are scrap.

As shown in Figure 42, the user's attention is pointed towards the content kerfCanceler assumes to be relevant. The outlines of the parts are greyed out and the cutouts are highlighted (typically the outlines of parts are not mechanisms).

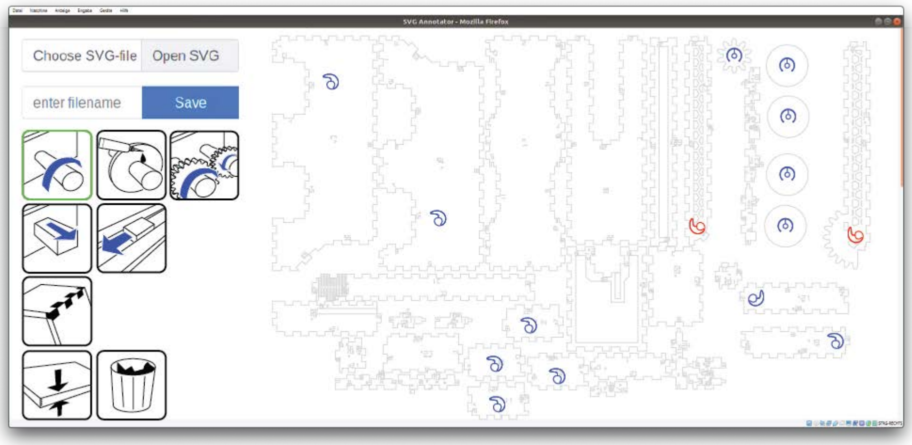


Figure 42: A model presented to the user (a firetruck). All outline geometry is greyed out to put the users' emphasis on the mechanisms guessed by KerfCanceler.

KerfCanceler iterates over the inner geometry to find mechanisms. Revolute pairs (e.g., bearings, gears, wheels, cam/followers) manifest themselves as circles in the model. KerfCanceler groups circles by diameter. As shown in Figure 43, when two similar groups occur, it assumes the group with smaller diameter is press-fit and the other group is loose fit.

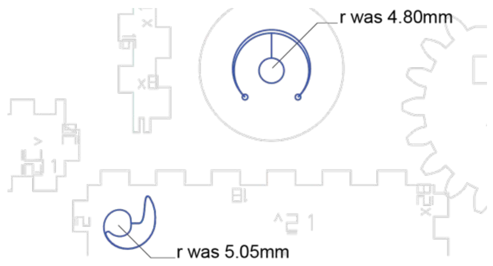


Figure 43: These circle cutouts in the firetruck are of similar size. In the entire fire-truck model, one category turned out to be around 5.05 and one around 4.80mm, KerfCanceler assumes the small opening is press-fit opening and the other one loose fit (it thus placed two different mechanisms).

Replacing mechanisms with kerf-canceling counterparts

KerfCanceler then places kerf-canceling mechanisms. At the positions where it assumed loose or press fit mechanisms, it inserts the correct version. For every circular cutout, it caches three alternatives shown in Figure 44a-c: the original circle, a press-fit mount based on cantilever springs [101] and a kerf-canceling loose-fit bearing. It displays the one it guesses to be the right version. Because these alternatives are generated *before* the user touches them, it allows for interactive response times in the web UI.

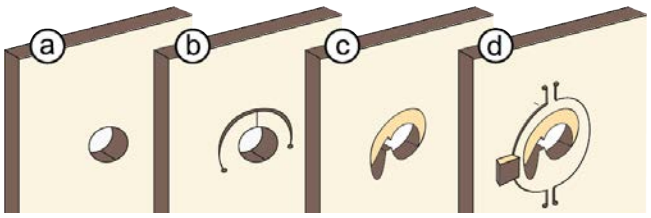


Figure 44: Possible modifications of a circle cutout. (a) The original circle (b) a circle used as a mount (press-fit) (c) the circle used as a kerf-canceling bearing and (d) the same as c but pushed to the right by "kerf" using the wedge on the left, for gears.

KerfCanceler checks for intersections with the model during pre-processing. It uses the shape of Figure 44c overlaid by b. If this intersects with the rest of the SVG model, the mechanism shows up in red, otherwise in blue. It does not use the larger kerf-canceling gear-bearing of Figure 44d as this is a rare case and would produce many false positives. When the user later inserts a gear-pair mechanism, kerfCanceler checks the intersections locally resulting in slightly longer processing time (up to a second).

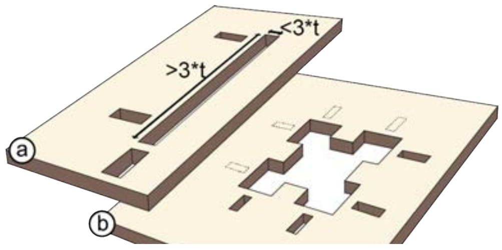


Figure 45: The placement of wedges for a sliding mechanism, (a) half of the edges of the cutout get a kerf adjusting wedge. (b) The same works for non-rectangular cutouts. Multiple placements exist (dotted lines). KerfCanceler, excludes all that cause intersections and picks the best solution.

For guided sliding mechanisms, the wedges do not replace the original polygon, but line up on the sides. As shown in Figure 45, kerfCanceler places two wedges on each edge. It places the wedges as far apart from each other as possible to minimize the risk of jamming the slider. For short edges, it places one wedge in the middle of the edge.

3.2.6 technical evaluation: How well do kerf-canceling mechanisms perform?

We hypothesize that kerf-canceling mechanisms are comparable in performance to the original mechanism, and that with increased kerf, the kerf-canceling mechanisms outperform the original. We evaluate this by measuring the *friction* and the *play* of the mechanism and compare that to plain bearings, while varying kerf.

We measure friction by spinning an axle with 2 flywheels, we start at 1300rpm (=136.14 rad/s) and measure how long it takes until the shaft stops spinning because of angular friction.

We measure the tilt angle of the axle within each of the bearings. We take a photograph with a fixed camera from the side of the bearing, pivot the axle and capture both extremes. The angle between these corresponds to the maximum range of play.

Test setup

We mount the bearing with an 8mm aluminum axle. We attach a 3D printed flywheel with 4x 33g steel balls inside, to both ends of the axle. The shaft is powered using a Bosch drilling machine via a simple clutch. The total inertial moment of the flywheels is 17.4×10^{-5} kgm^2. We use the *Peaktech 2795* contactless rotation sensor to measure the rotation speed. We then calculate the frictional Torque (T) using this basic formula:

$T = I * a$

In which a is the angular acceleration (initial rotation (rad/s)/ time (s)) and I the moment of inertia (kgm^2).

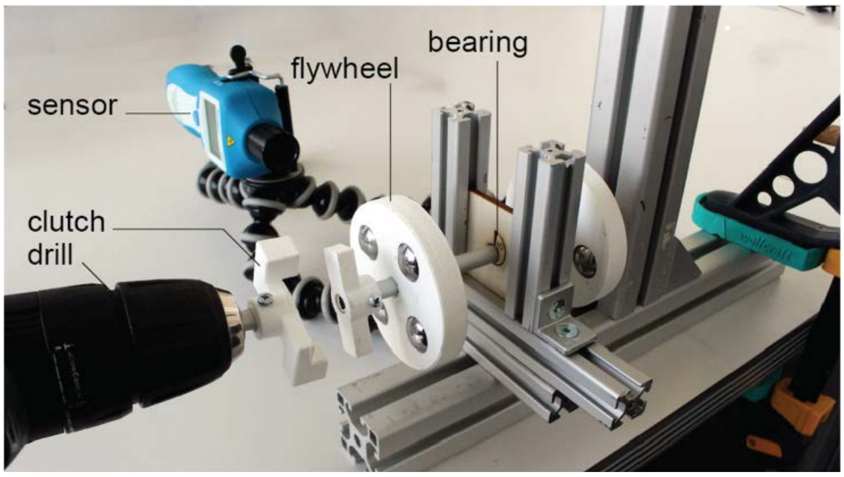


Figure 46: Experimental set-up.

Test pieces

We compare the baseline (a plain bearing) to the kerf-canceling bearing. All pieces were cut out of 4mm plywood, we simulated kerf from 0 to 0.4mm in 0.1mm increments. These kerf values we adjusted for the laser cutter used, so 0mm kerf means the bearing fully touches the axle. We repeated each experiment 3 times and report the average value to compensate for noise.

We used a *Trotec speedy 360 flexx* laser cutter with a kerf of 0.15mm. To reproduce this experiment, we have attached a test piece in the appendix of this paper.

Results

As shown in Figure 47, Kerf-canceling bearings demonstrated constant performance across variations in kerf (between 3.1 and 3.4 mN). Kerf heavily affected the plain bearing's performance. Already at a kerf variation of 0.1mm the friction went up substantially (4.7 mN). And in particular when reducing the kerf further, the bearing essentially got stuck as friction went up by a factor of more than 10. (40.6 mN).

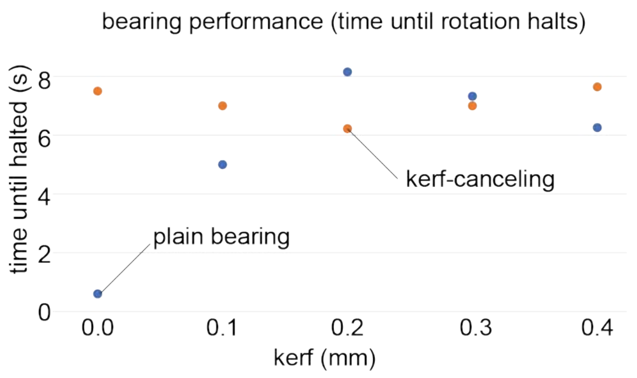


Figure 47: Results of the friction test. Kerf-canceling based bearings perform stable across kerf variations as opposed to plain bearings.

Figure 48 shows the results of the play analysis. For the plain bearing, the play increases roughly linearly with the kerf. The play for the kerf-canceling bearings remained stable.

We found that strong vibrations (e.g., by accidentally misaligning the drill bit) can cause the inset to come out. For mechanisms that are expected to be exposed to such forces, we recommend adding a dot of glue before assembling the mechanism.

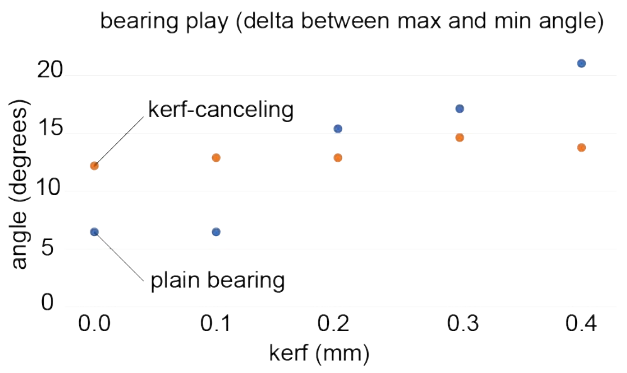


Figure 48: Results of measuring play of the bearings. The kerf-canceling bearing remains relatively stable, while play for the plain bearing almost linearly relates to increasing kerf.

Discussion

Kerf-canceling bearings demonstrate performance independent of the kerf, both when it comes to the play and the friction of the mechanisms. While plain bearings only perform reliably in a narrow range of kerf. We thus conclude that our bearings serve well as kerf-canceling mechanisms.

3.2.7 software evaluation of kerfCanceler

To validate the utility of our software, we ran it on 20 models found online. For each model, we measured what percentage of mechanisms were identified automatically and how many interaction steps were required to modify the mechanisms.

The models in Figure 49 are a subset of the 20 test models which we fabricated to confirm that the generated kerf-canceling mechanisms work.

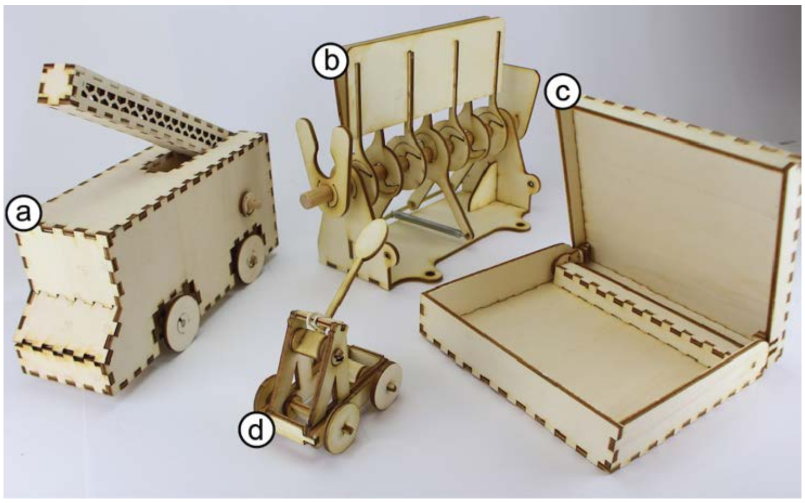


Figure 49: Models of the test set we fabricated on our laser cutter with increased kerf.

KerfCanceler achieved a 93% recognition rate for the rotary mechanisms in the models. It identified false positives in 5 models, which contained engraved (decorative) circles, these were falsely identified as mechanisms.

We used the UI to intervene with 2-21 (9 on average) overrides of the initial guessed mechanisms. Six models worked based on the guessed mechanisms alone. The "group edit" tool reduced the number of edits in most models. Pre-processing of models took on average 5.87ms of time. It took 66s of manual work per model to convert, for a user who knows the model's functionality.

Six models failed to convert. Three of them had too little physical space in the model to insert the kerf-canceling mechanisms. Four models contained lines that were intended to be engraved, which caused intersections. One model showed both problems. These intersections won't break the mechanism but may affect the aesthetics of the model depending on how meaningful the original engravings were. So, in total 17/20 models were converted using our tool with a laser cuttable result.

We conclude that many models online can be converted to become kerf-canceling with only up to three minutes (one minute on average) of user effort.

3.3 SOFTWARE ARCHITECTURE

KerfCanceler and springFit *together* allow making 2D cutting plans more tolerant to machine variations, we therefore designed both tools as separate software modules with a. similar interface, making it easy to integrate both in the same architecture. We implemented this as a lightweight Typescript application that runs in the users' browser. This application leverages the fact that SVG is built on the XML standard to encode annotated information in the cutting plans, and that XML can be directly displayed in the browser.

Figure 50 displays the architecture in a system sequence diagram. Both kerfCanceler and springFit are called under the hood by the central browser application. They are run in a headless fashion and interface with XML in- and output. The geometric optimization scripts to compute the specific dimensions of springs and wedges use the NLopt library in C++ (nlopt.readthedocs.io) for performance reasons, they therefore run on a separate server and interface using JSON strings as exchange format. When users override elements through the UI, the client-side application adds annotations to the XML which are then parsed separately by the individual geometry generators. The server architecture allows each module to be called individually, making it easy to access springFit or kerfCanceler (and/or their respective optimizers) as separate modules to allow them to be easily integrated in other software packages. We demonstrate this with at the example of our simple browser interface.

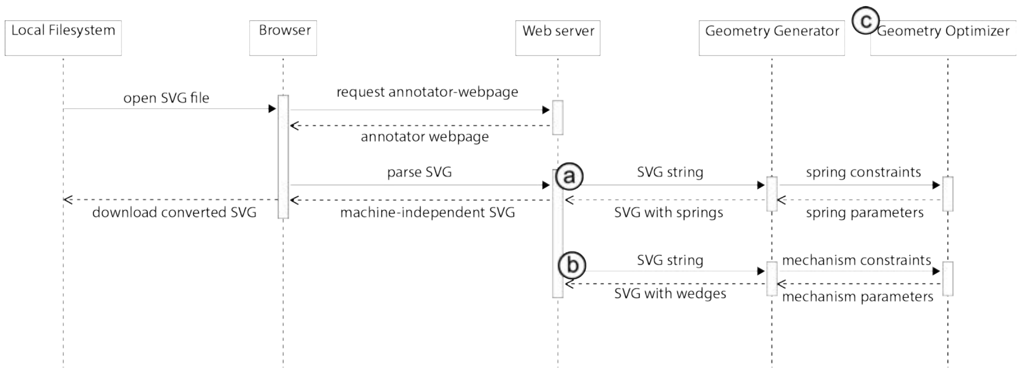


Figure 50: The modular architecture shown in this data flow diagram. (a) springFit and (b) kerfCanceler both are integrated into the system in the form of individual modules. (c) the optimization scripts are implemented in C++ for efficiency and run on their own server.

3.4 CONTRIBUTIONS

With springFit and kerfCanceler, we make three contributions. First, we present and analyze the specific challenges of kerf-dependent models for joints, mounts, and mechanisms. Second, we developed mechanical solutions to these problems by modifying the 2D cutting plans. And third, we present two software plugins that detect joints, mounts, and mechanisms in cutting plans and replaces them with kerf-tolerant version, as well as an overarching browser interface that allows users to modify existing 2D cutting plans to make them fabricate reliably on any consumer laser cutter and in a range of materials.

Our approach is subject to three limitations, kerf-canceling mounts, joints, and mechanisms are less robust than their traditional counterparts, they require additional space in the cutting plan, and they may affect the aesthetics of a model.

Another limitation of this work is that we have only verified this with consumer-grade laser cutters. In theory it should apply to any subtractive fabrication process, but we have not verified it with specialized equipment that uses variations of the laser frequency or modulation that can influence kerf differently.

3.5 CONCLUSIONS

With these two papers, we have presented a mechanical solution to create kerf-tolerant laser cut models with the help of *kerf-canceling* mechanisms and *cantilever-based* joints and mounts. The resulting cutting plans remain valid across machines and kerf values, which, for example, allows users to buy a new laser cutter without invalidating cutting plans created earlier.

Zooming out, kerf-canceling mechanisms address one facet of a larger challenge, i.e., the challenge of portability. Today, the majority of laser-cut models are shared as 2D cutting plans—and these are inherently machine-specific. This is problematic, as this gets in the way of collaboration and sharing, which rely on people's ability to reproduce other users' models, e.g., for the purpose of remixing them.

Our mechanical solution to kerf thus allows users to reproduce models and share in productive ways. However, to address the associated cost of aesthetic and structural integrity and the fact that the models are still shared in a 2D format, we present an alternative solution to the problem in the following chapters.

4 REPRESENTING LASER-CUT MODELS IN 3D

The mechanical solution to create cutting plans that are tolerant to machine variations presented in the previous chapter lets users reproduce models on most laser cutters and in a range of materials. However, in our pursuit of making laser cutting relevant to the "other 99% of people" (other than current tech enthusiasts), in this chapter we show how to make it easier to create laser-cut models, and to make changes to models that are made by others, by modeling in *3D* using a fabrication-aware modeling environment. The development of this environment is a much bigger effort than just the research published in this thesis, and as of now constitutes over 140,000 lines of code. The research in this thesis is what drove the initial development and novel aspects of its data structures and modeling paradigm.

By sharing the models in a 3D representation, we further advance portability, as users export the 3D model to a 2D cutting plan that is optimized for *their laser cutter and material*. And then fabricate that specific cutting plan on their machine, without inserting additional incisions in the models.

In this chapter we present the development of this 3D modeling environment in two phases, first as a primarily volumetric environment which affords efficient structural building with laser cutters. We then revisit this by extending the environment with support for detailed editing by building tools and a subsystem that allows users to manipulate individual plates. These systems combined allow users to make more advanced models than seen before, and furthermore encourage that by letting users build on the work of others in a machine-invariant workflow.

4.1 KYUB: A 3D MODELING ENVIRONMENT FOR LASER CUTTING

Kyub [14] is an interactive editing system for laser cutting called. Kyub allows users to create models efficiently in 3D, which it then unfolds into the 2D plates laser cutters expect. Unlike earlier systems, such as FlatFitFab [29], kyub affords construction based on closed box structures, which allows users to turn very thin material, such as 4mm plywood, into objects capable of withstanding large forces, such as chairs users can actually sit on, as shown in Figure 51.

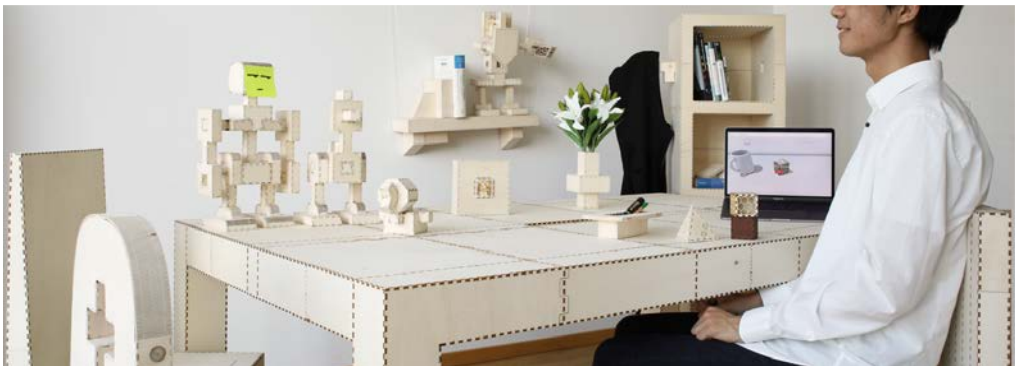


Figure 51: A selection of objects created using kyub, a software system that allows users to design 3D objects for laser cutting. By affording closed box structures, objects made using kyub are very strong. This allows users to make tables, shelves, and chairs that can hold a person. (All shown objects are assembled from 4mm plywood sheets—pressure fit, not glued).

To afford such sturdy construction, every kyub project begins with a simple finger-joint "boxel" (see Figure 52)—a structure we found to be capable of withstanding over 500kg of load. Users then extend their model by attaching additional boxels. Boxels merge automatically, resulting in larger, yet equally strong structures. While the concept of stacking boxels allows kyub to offer the strong affordance and ease of use of a voxel-based editor, boxels are not confined to a grid and readily combine with kyub's various geometry deformation tools.

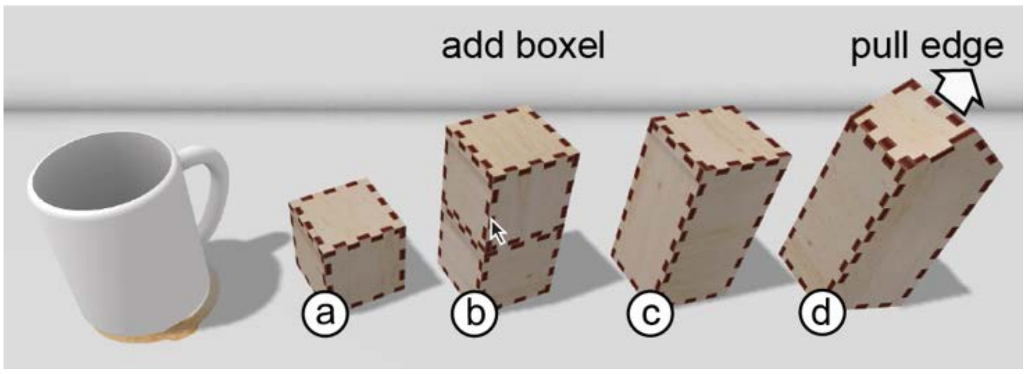


Figure 52: Kyub allows users to create sturdy objects by stacking volumetric elements, which we call boxels. (a) A single boxel can withstand >500kg of load, (b) Added boxels merge automatically, resulting in a larger, yet equally strong structure. (c) While kyub offers the affordance of a voxel-based editor, its objects are not bound to a grid; users can reshape them using a wide range of deformation tools.

4.1.1 3D editing based on boxels

Figure 53 shows a "hello world" example in which we create a simple picture frame, essentially only using kyub's *add boxel tool*. (a) Any editing experience starts with a *boxel* dropping from above. This boxel, like any object in kyub, is represented in a realistic way, i.e., how it will look once laser cut and assembled [1,64]. The boxel bounces off the ground and comes to a rest, demonstrating that this is a physics-based environment. A cup serves as size reference.

(b) The user picks the *add boxel tool* from the menu and clicks into the scene. (c) This produces another boxel. (d) The user selects the *add boxel tool* again, but this time clicks onto a boxel already on stage. The new boxel automatically aligns itself with the clicked one and both merge automatically. This results in a single box the size of two boxels. This merging is an important step in that the resulting geometry not only features a minimal number of plates, but also makes interlocking plates extend across the entire objects; this produces very *sturdy* structures.

By double clicking the *add boxel tool*, the user makes the tool "sticky". (e) Another six clicks cause the *add boxel* tool to create (f) a simple, but stylish picture frame. (g) The user engraves images on each of the six sides of the lone boxel and is now ready to showcase it in the picture frame.

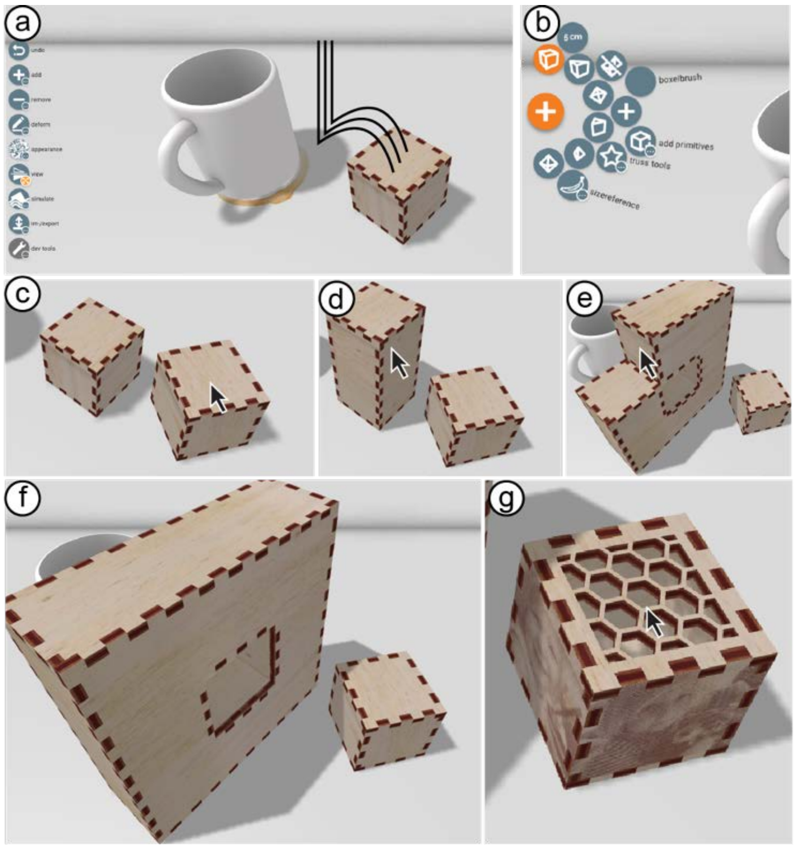


Figure 53: Sole use of the add boxel tool already allows making simple objects, here a picture frame. (a) A boxel falls into the scene. (b) The user selects add boxel and (c) clicks the stage, which produces a second boxel. (d) Holding the add boxel tool, the user clicks a boxel already on stage. This stacks a boxel on top and both merge automatically. (e) Adding another six boxels (f) completes the frame. (g) Engraving six images into the lone boxel prepares it for being displayed in the frame.

The user now exports the box to the laser cutter using kyub's *export* menu. Kyub responds by breaking the 3D model down into individual plates and by correcting for the specifics of the target machine (*kerf* correction [135]). As shown in Figure 54, it adds engraved marks that tell the user which plates to connect when assembling the model. Kyub then lays out plates onto sheets (also known as *nesting*, see also [57, 86]) and writes each sheet into a separate .svg file.

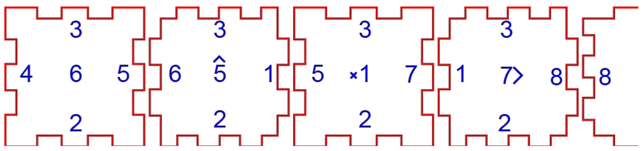


Figure 54: An exported model from Kyub, the red lines are what the laser cutter cuts. Numbers along the edges tell users which plates (numbers in centers) to connect to, when assembling the model. A '^' indicates "this side up"; an 'x' indicates placement at the bottom.

While these look-up numbers provide sufficient data for users to assemble the model, it still presents users with a nasty search problem when there are many plates to be assembled. We have therefore revisited this 3D-2D export pipeline in two distinct approaches: (1) we implemented an optimization algorithm we call FoolProofJoint [89] that makes joints explicitly different from one another so users cannot misassemble them. And (2) we developed a software tool called Roadkill [2] which adjusts the nesting of the plates, so users always assemble adjacent plates with one another. And provide visual assembly instructions on the plate itself, to further simplify the assembly process. While we wrote these papers in the same period as the bulk of this thesis, their contributions are adjacent to the objective of portability, for more details we refer to the respective papers.

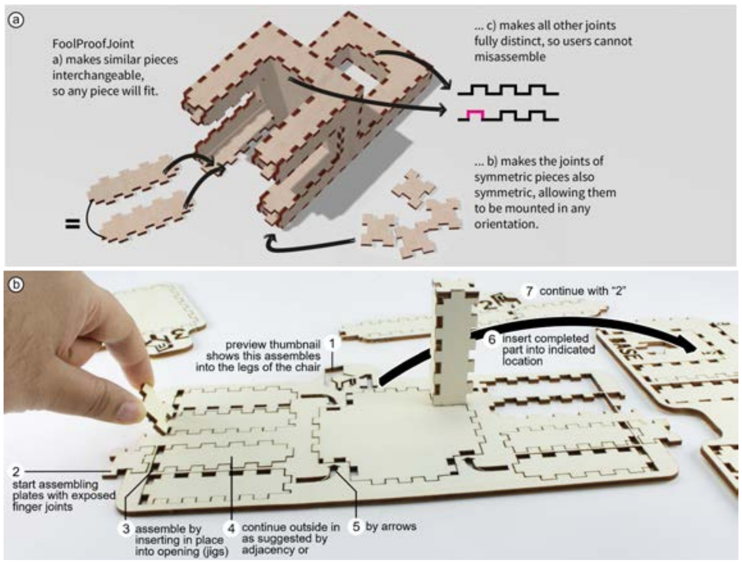

a
FoolProofJoint
a) makes similar pieces interchangeable, so any piece will fit.
... c) makes all other joints fully distinct, so users cannot misassemble
... b) makes the joints of symmetric pieces also symmetric, allowing them to be mounted in any orientation.
b
preview thumbnail shows this assembles into the legs of the chair
1
7 continue with "2"
6 insert completed part into indicated location
2 start assembling plates with exposed finger joints
3 assemble by inserting in place into opening (jigs)
4 continue outside in as suggested by adjacency or
5 by arrows

Figure 55: Follow-up approaches to make it easier for users to assemble models by modifying (a) joints [89] and (b) layout during export [2].

After exporting, the user sends the .svg file(s) to the laser cutter and assembles the fabricated parts. Figure 56 shows the final result, here with manually sanded edges.

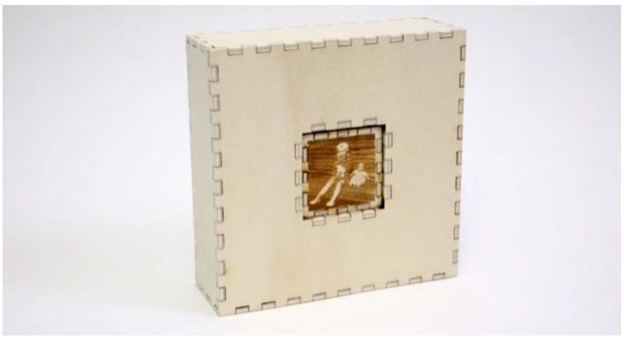


Figure 56: A fabricated, assembled, and sanded picture frame.

Boxels appear to be on a grid—but they are not

The voxel-like behavior of *add boxel* suggests boxels be limited to a grid. This would be a dramatic limitation, as it would eliminate many of the benefits of personal fabrication—which is to create one-off objects that fit a specific use case. Fortunately, kyub boxels *are not* confined to a grid.

As shown in Figure 57, kyub allows users to manipulate boxel-based geometry with a range of deformation tools. For example, (a-b) the user may compress the picture frame using the push-pull tool. Or (c) users may pull out just one of the edges to give the picture frame a slanted top or base.

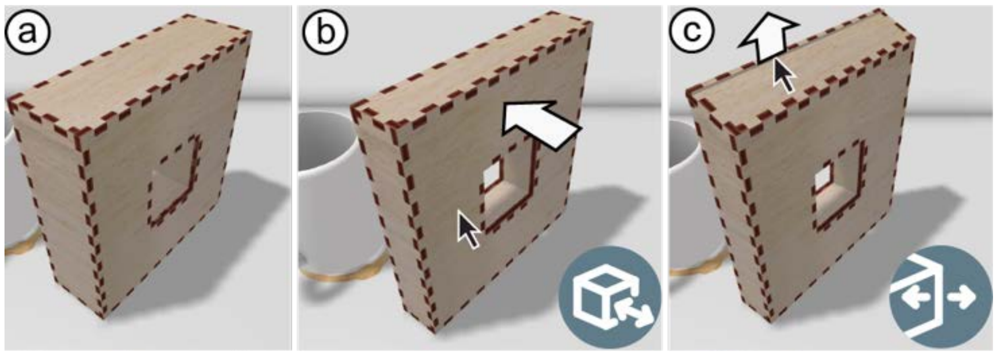


Figure 57: (a) Kyub allows the boxel-based picture frame to be manipulated using (b) push/pull and (b) push/pull edge tools.

Interestingly, *add boxel* continues to be applicable after geometry has been deformed. In Figure 58, we set boxels to half their usual size and then (a-b) apply *add boxel* to produce two prongs that (c) form a stand for our picture frame. *Add boxel* not only remains applicable, but also snaps into an invisible grid, making it easy to align the prongs with the frame.

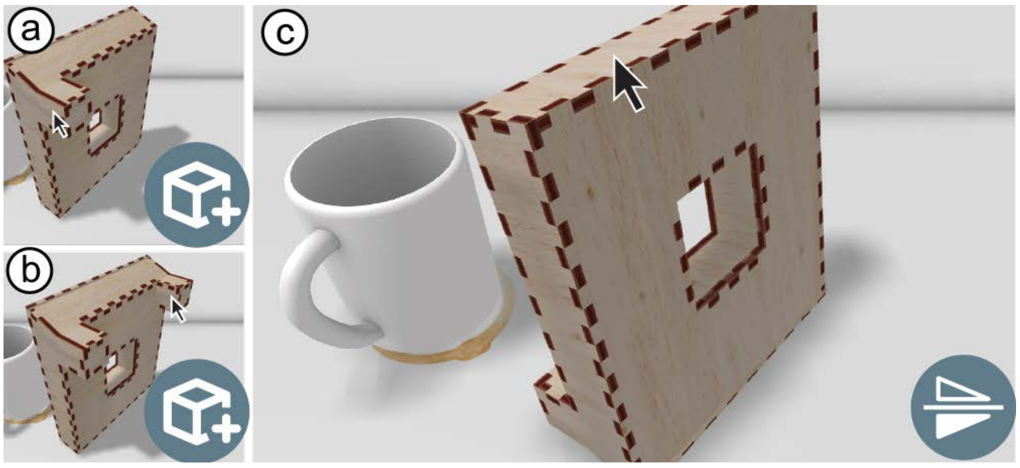


Figure 58: *Add boxel* remains applicable *after* the use of deformation tools.

Kyub's secret to achieving this combination of grid and free deformation is to *not memorize* any grid, but to instead *re-infer* the grid with every tool interaction, i.e., kyub analyzes the current geometry and determines what grid *the user* might be trying to refer to.

Figure 59 further illustrates this. (b) Removing half a boxel on one side of a part and (c) adding it back at the other end looks like it might lead to a grid aligned half-way. (d) However, boxels added now snap into position based on the current shape of the part, not based on its history. We describe the underlying algorithm in detail in section "4.1.3 implementation".

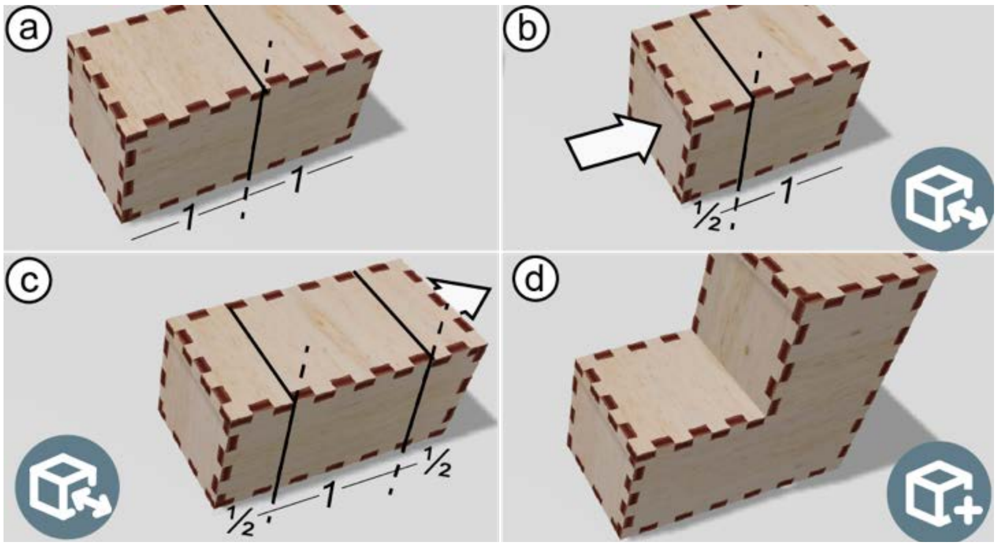


Figure 59: Kyub infers the grid, rather than maintaining it (a) Laying down two boxels, (b) pushing in half a boxel and (c) pulling out half a boxel on the other side results in 2x1 boxel arrangement. (d) Adding a boxel snaps into position based on the current shape of the part, not its history.

Kyub's ability to maintain grid-like behavior allows it to offer good default behavior when placing new geometry. This is key, for example, when running kyub on devices with very coarse input capabilities, such as touch screens (Figure 60).

The fact that kyub determines the currently active "grid" only based on the object's current and thus visible geometry generally relieves kyub of the necessity to display the grid. This allows kyub to adopt a particularly uncluttered and natural look.

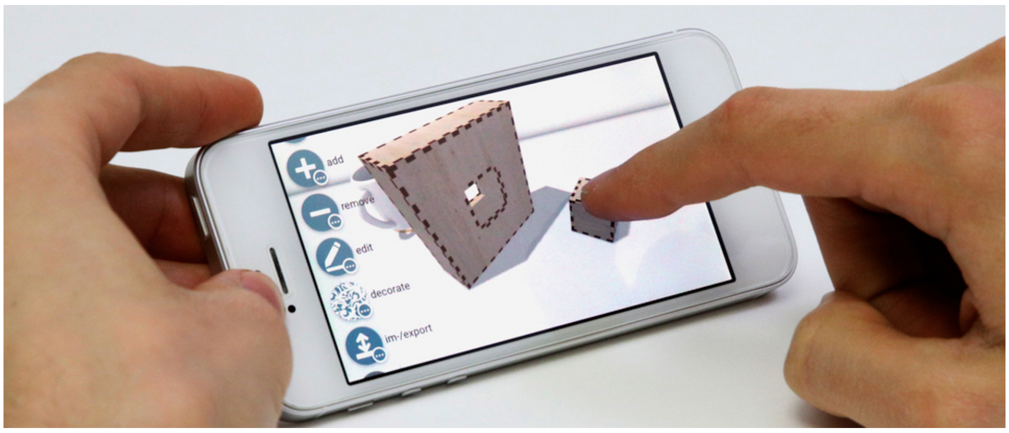


Figure 60: The grid-like behavior of boxels is crucial in allowing kyub users to create precise geometry on low-precision input devices [126] (here an *iPhone SE*).

Making things that fit together

One of the desirable side effects of the boxel-based approach is that everything naturally fits together. Figure 61 illustrates this at the example of a decorative robot figurine, created by one of the participants in our user study (see section 4.1.5 "User Study"). Here the user (a) has modeled the foot of the robot and then cuts two holes into it using the *subtract boxel tool.* (b) The fact that boxel-sized extrusions naturally fit into boxel-sized holes allows the two parts to form a dowel-like connection. Here the user uses the *insert tool* to try this out in the editor. By default, kyub designs all parts with a "fixed fit" (H7/n6) [58] allowing parts to be mounted and removed with a light pressing force. (c) The dowel connections allow the resulting figurine to be posed (d-e) in a variety of ways.

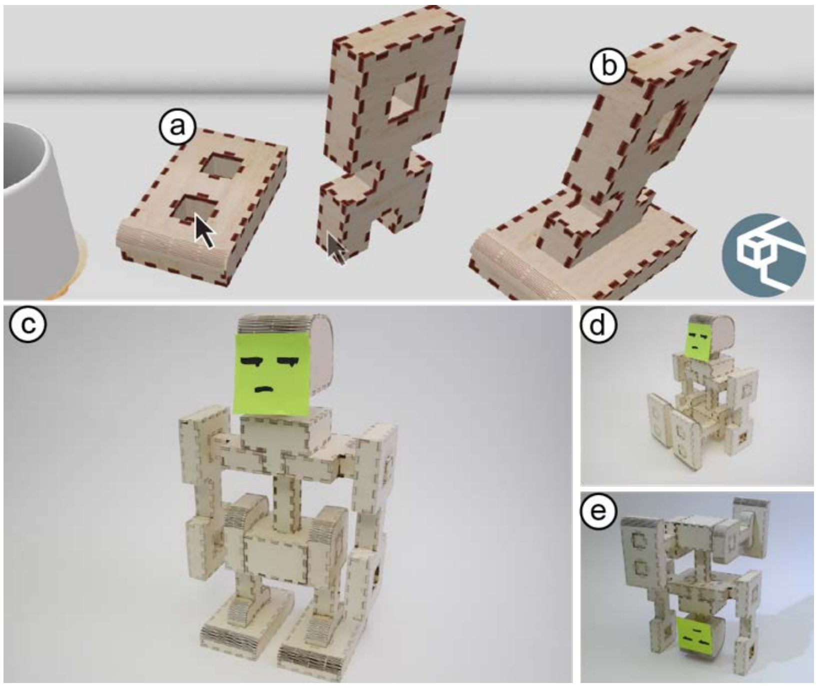


Figure 61: Parts created using kyub naturally fit together. (a) Parts with extruding boxels and parts with boxel-shaped hole naturally match and (b) users can combine them using the insert tool. (c) The resulting dowel joints allow this decorative robot figurine (d, e) to be posed in various ways.

As illustrated by Figure 62, boxel-based dowel connections allow us to create collections of parts that can be combined interchangeably. The result is a custom construction kit that allows making a range of different models, a rudimentary version of something like a LEGO construction kit.

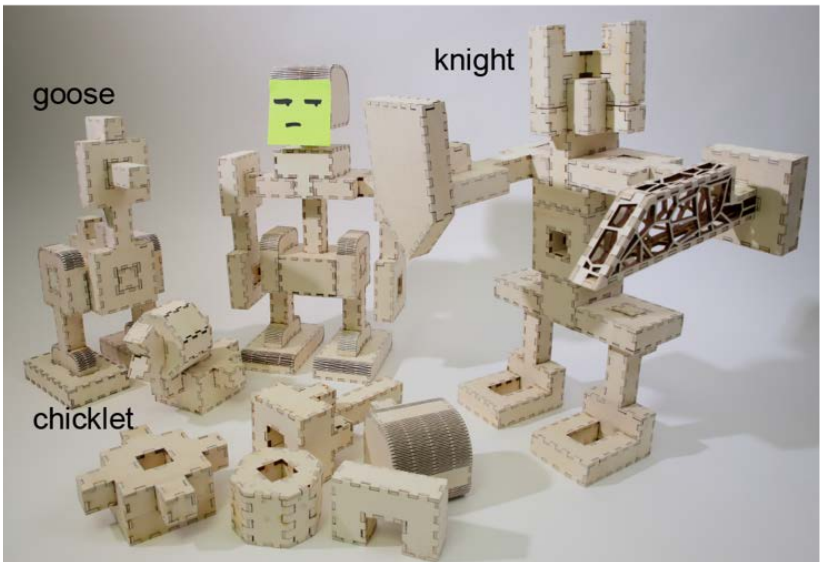


Figure 62: Combining the parts of the robot figurine with a few compatible elements results in a simple construction kit.

While the shown models are clearly designed around the notion of rectilinearity, they also include some non-rectilinear parts. As shown in Figure 63, these were made quickly and efficiently by applying *non-rectilinear* boxels, here specifically *90-degree prisms* and *equilateral prisms.*

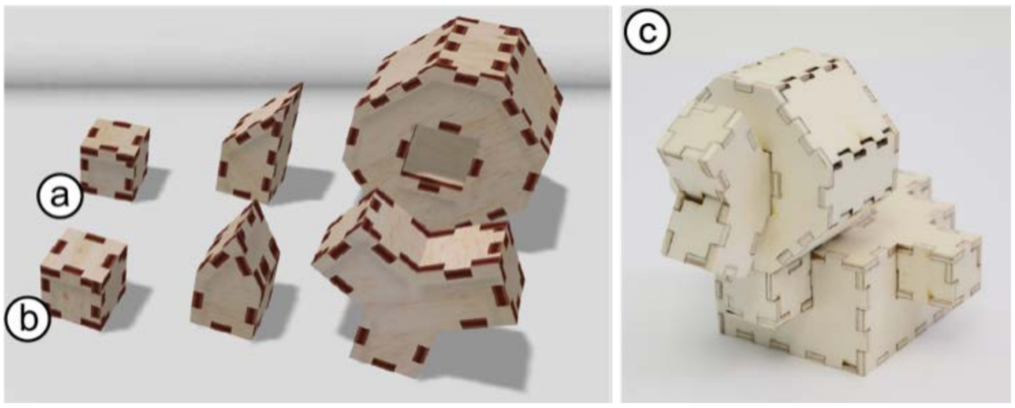


Figure 63: Non-rectilinear boxels add expressiveness to boxel-based construction. (a) Here we use four 90-degree prisms to create a duckling's head and (b) one equilateral prism to make its beak. (c) Resulting duckling.

The concept of non-rectilinear boxels extends past prisms. Figure 64 shows how we recreated the well-known tetrahedron puzzle by complementing (a) a pyramid boxel with (b) two tetrahedron boxels. Cloning the resulting shape completes the puzzle and (c) and with a little bit of trying out it can be assembled into… a tetrahedron.

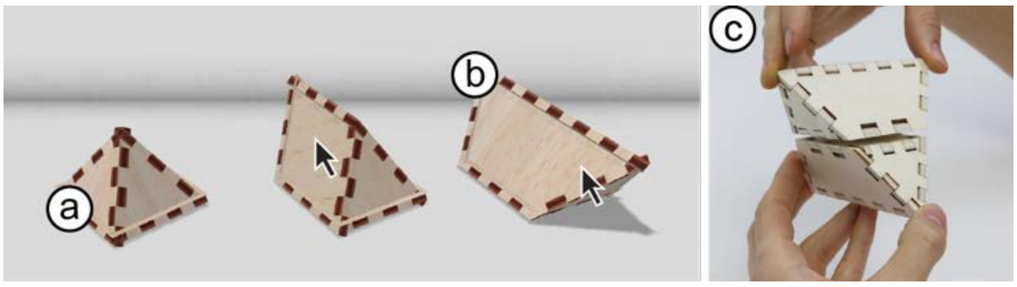


Figure 64: The boxel concept goes beyond rectilinear boxes. The pieces of this tetrahedron puzzle were made by combining a pyramid boxel with two tetrahedron boxels.

4.1.2 designing sturdy structures

As discussed earlier, a key objective behind boxel-based construction is to create *sturdy* structures. The chair shown in Figure 65 is such a structure. This particular model was created using the boxel-based workflow described above, sped up by using a *clone* and an *attach tool*.

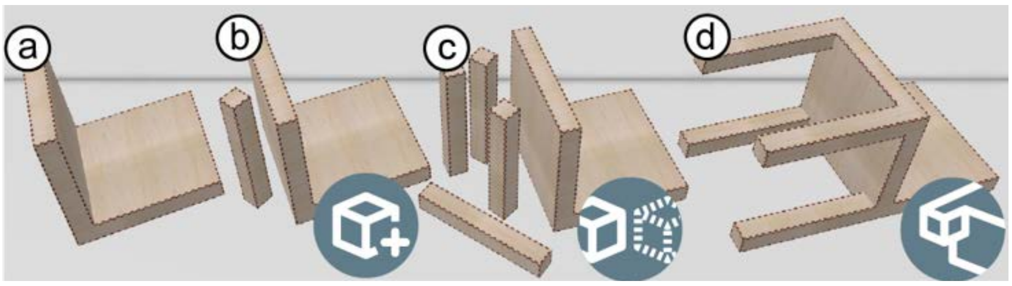


Figure 65: (a, b) Modeling a chair in kyub using *add boxel* (c) the *clone tool* and (d) the *attach tool*.

The common approach to designing furniture is to use thicker material, such as 12mm plywood [104, 120] and as shown in Figure 66, kyub supports arbitrary material thicknesses. We could make the chair from 6mm plywood, for example, and even though plates of half thickness should carry only an eighth of the weight, the result works reliably thanks to the box-based construction.

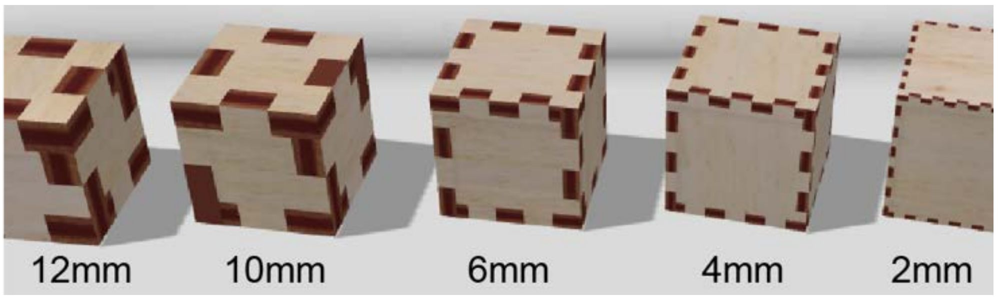


Figure 66: Kyub supports changing material thicknesses.

Figure 67a shows how an experienced user can push the design of our chair design one step further in order to allow manufacturing it from the same 4mm plywood we used in all previous examples. By adding two internal plates that extend through the seat and the backrest using the reinforcement tool, this design prevents the seat from buckling and the backrest from breaking, despite the thin material. The design shown in Figure 67b is even resilient against rocking, as the insides of pairs of legs are combined into contiguous U-shaped plates.

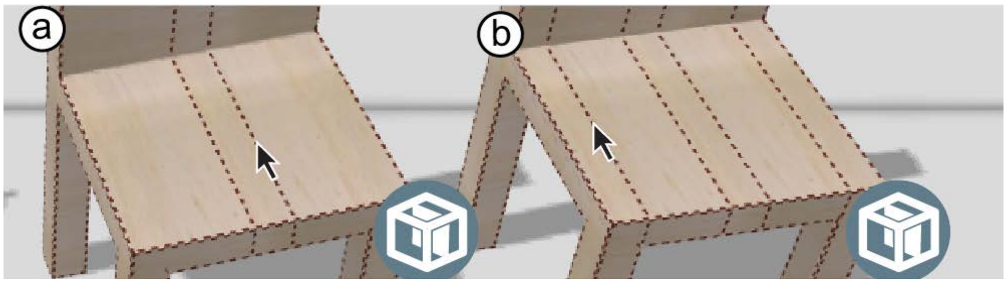


Figure 67: (a) Chair with front-to-back reinforcement, (b) additional reinforcement supporting the legs.

To get the most strength out of internal plates, kyub merges internal plates with other plates whenever possible. As shown in Figure 68a, internal plates are usually centered, but that would prevent the internal plates in Figure 67b from merging with the legs. However, as shown in Figure 68b kyub's internal plates are given some slack, allowing them to snap into the plane of the leg. Kyub heavily relies on its grid inferrer for this functionality (see section "4.1.3 implementation" for details).

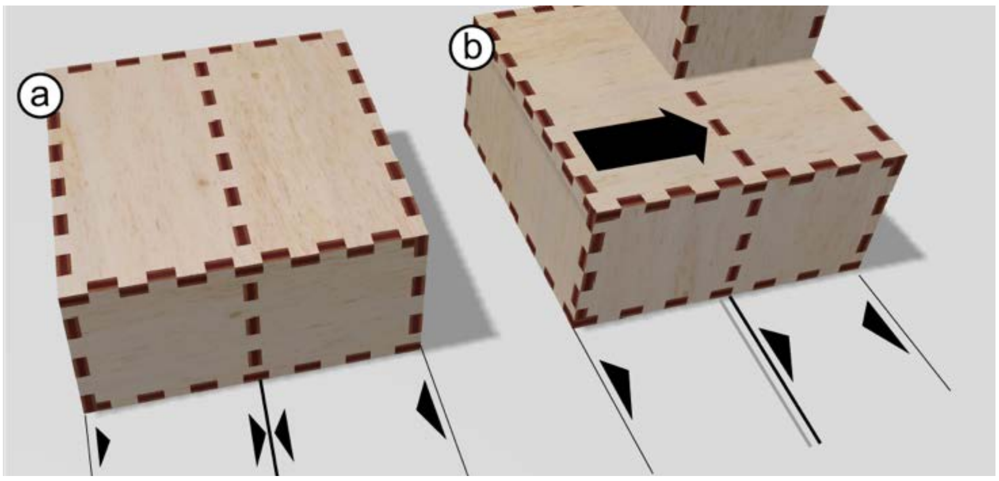


Figure 68: (a) When applying the reinforcement tool to this part, the reinforcement centers itself, (b) However, when adding a boxel, reinforcement automatically shifts by half a plate thickness so as to line up with the left plate of the added boxel, producing a sturdier result.

Finally, Figure 69 shows an expert design that solves the challenge without any internal plates. Instead, a slot cut into seat and backrest using the *subtract boxel* tool reinforces seat and backrest—and creates what we think of as an appealing design element.

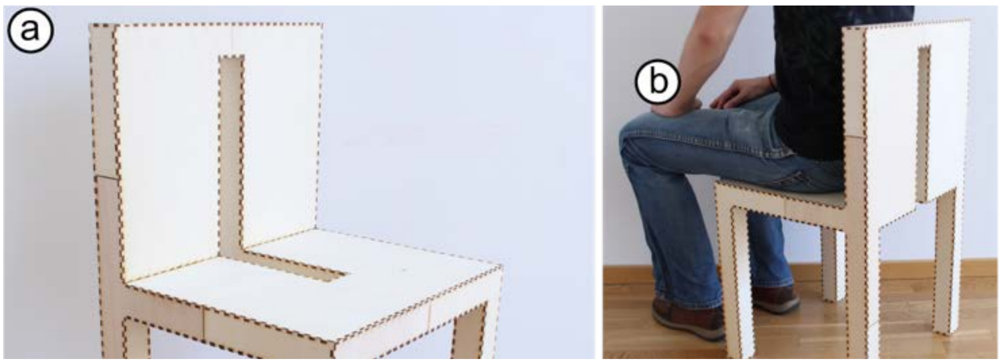


Figure 69: Here a slot cut into the chair reinforces the chair's seating surface and backrest.

In follow-up work beyond the scope of this thesis we have developed an algorithm to detect potential weaknesses in closed-box laser-cut structures. Our extension called *fastForce* [1] places internal plates automatically to reinforce such structures while the user is modeling.

Making large objects using tessellation

The closed-box structures afforded by kyub are generally strong enough to produce sturdy objects even at large scale. While some laser cutters allow cutting very large objects in one go, kyub allows owners of smaller devices to fabricate large objects as well.

Kyub achieves this by composing large objects from smaller plates. Users configure the maximum plate size available to them and when enlarging an object now, kyub breaks down the oversized part into two or more cells as shown in Figure 70a. Figure 70b shows the specific wood joint kyub creates to hold cells together—a supported lengthening joint) Adjacent cells share a single membrane of finger joints; then both cells connect into this membrane using butterfly joint-like tabs.

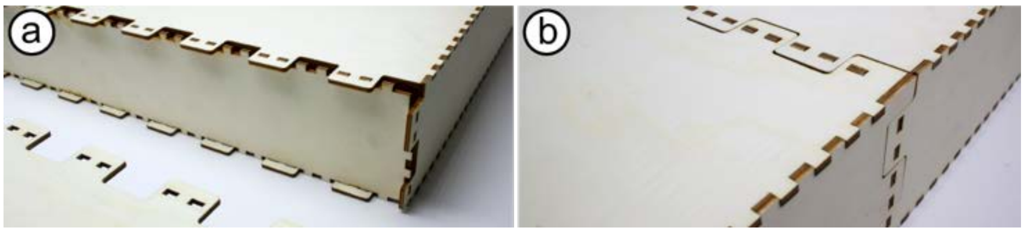


Figure 70: The cell structure created by tessellation. The big finger joints lock the two coplanar plates on the top while supported by a vertical plate.

The joined cells are strong enough to carry human weight even on very large designs, such as the 1.80m dining room table shown in Figure 71, assembled from 40 "A2"-size plates (60 x 40cm; 24" x 16").

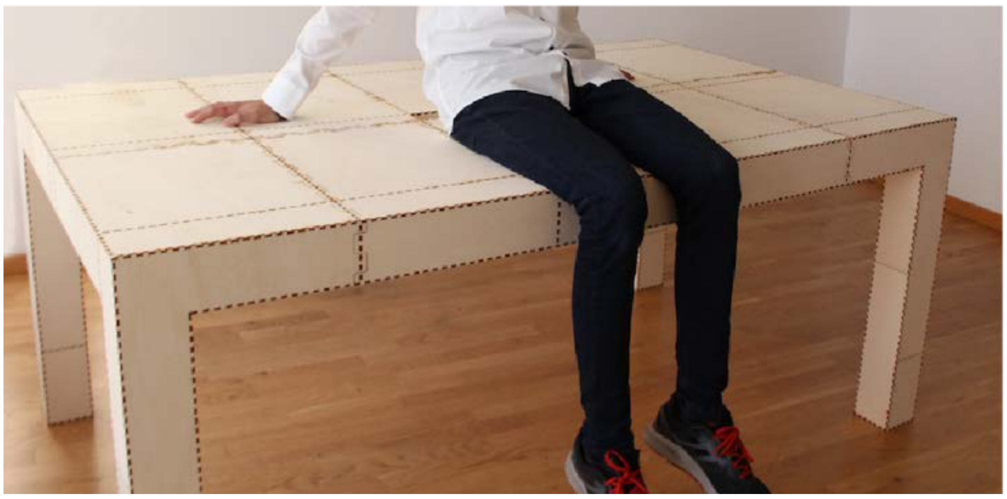


Figure 71: The table from Figure 51 is assembled of separate cells which are capable of holding a human sitting on it.

4.1.3 Implementation

Kyub is implemented as a web-based application using JavaScript and WebGL. It consists of about 70k lines of code. The server runs in Node.js but almost all computation is done on the client, making it easily scalable. The architecture of the client is similar to that of a game engine, because of the different subsystems at play, such as the physics engine (from cannonjs.org) that is continuously running in the background.

Core to the implementation is the *grid inferrer,* shown in pseudo code in Algorithm 2. As introduced before, it provides alignment when adding a boxel to an existing assembly. Figure 72 shows an example, i.e., a user adding a boxel to a somewhat irregular "base" geometry.

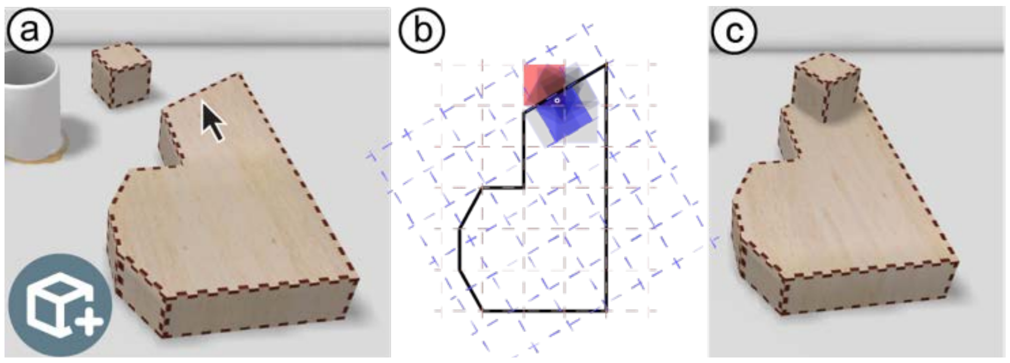


Figure 72: The grid inferrer. (a) a user applies the add boxel tool at the shown location. (b) Kyub takes the projection of the clicked surface and infers all possible grids to which the boxel could be aligned as shown in Algorithm 1. (c) After weighing the different grids, it places the boxel and merges the geometry.

When the user releases the mouse button (Figure 72a), kyub passes the mouse-up location to the grid inferrer. Implementing the algorithm shown below, the grid inferrer traverses all edges in the base geometry and extends each edge into a grid. The blue grid shown in Figure 72b, for example, resulted from the base geometry's top edge. The algorithm now reduces each grid to the one cell that contains the mouse click location. Finally, the algorithm picks the cell that maximizes the number of edges supporting this grid, minimizes the distance between these edges and the mouse click location, and minimizes the resulting number of plates.

Algorithm 2: Grid Inferrer

```
Input:  clicked point $P$
        2D projection of the outline of the base $B$
        2D projections of the outline of the connectors $C$
Output: connector $c^*$, position $p^*$ and rotation $r^*$ of object aligned to (and
placed on) $B$
Parameters: CENTER, IGNORE_PROTRUDING, WEIGHTS
candidates ← new Array()
for each connector ∈ C do
    for each pair of edges ($e_B$, $e_C$) ∈ edges(B) × edges(C)
        alignments ← position and rotation to align start or end of $e_C$
        with either start or end of $e_B$ respectively
        if CENTER then
            add to alignments the position and rotation to align
            center of $e_C$ with the center of $e_B$
        end
        for each alignment ∈ alignments do
            apply alignment to connector
            G ←create grid by repeating the bounding box (AABB) of
            connector
            candidate ← grid cell of G that contains P
            if IGNORE_PROTRUDING then
                add candidate to candidates if fully overlapping with B
            else add candidate to candidates
            end
        end
        candidate.connector ← connector
        candidate.feature_distance ← d(P, $e_B$)
        candidate.mergeable_plates ← the number of coplanar plates after
        placement (see Figure 73)
    end
end
deduplicate candidates, count in candidate.duplicates
normalize candidates' criteria, so each is within [0,1]
best ← select the best candidate considering the weights of the criteria
defined by WEIGHTS
c* ← best.connector
r* ← best.rotation
p* ← best.position
```

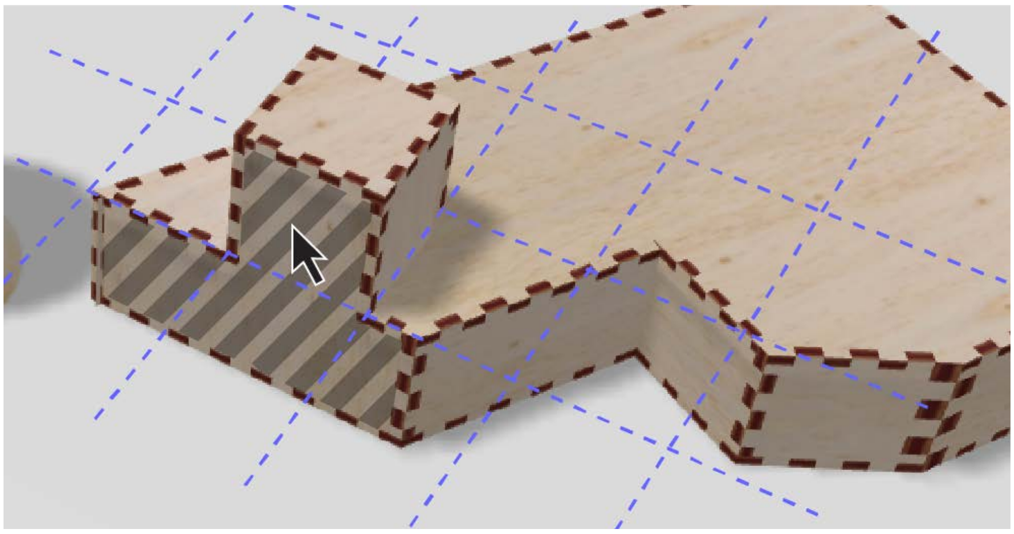


Figure 73: The resulting boxel is merged with the assembly. The dashed surface is an example of a coplanar plate that is unified with the side of the added boxel.

Whenever possible, the grid inferrer tries to place added geometry such that it stays within the circumference of the base geometry. Kyub accomplishes this by calling the grid inferrer with *ignore_protruding* set to *true*. This will produce the desired result in most cases. If the *grid inferrer* returns no result, however, kyub calls the *grid inferrer* again, this time with *ignore_protruding* set to false. This allows the grid inferrer to also explore configurations, where the added geometry protrudes past the edge of the base geometry. This two-pass approach allows kyub to make sure that the *insert tool* works reliably, including such cases as the insertion of the arm of the robot in Figure 61.

4.1.4 technical evaluation

To validate the strength of closed box structures underlying kyub, we conducted a technical evaluation during which we fractured seven types of structures by applying appropriate subsets of up to six different types of forces and measured the force required.

To obtain a conservative lower bound of the sturdiness of the tested objects, we used the weakest and cheapest material we could find, i.e., 4mm 3-layer plywood at a price of roughly 7 Euros/m2. Also, all objects were held together by only press fitting them, i.e., without glue.

Objects tested

Figure 74 shows the objects we tested. The first four objects allowed us to test basic boxel geometry: (a) a 5cm boxel, (b) a 35cm stick made from 7 boxels, (c) an L-shape made from 3 boxels, (d) a 3D L-shape made from 4 boxels.

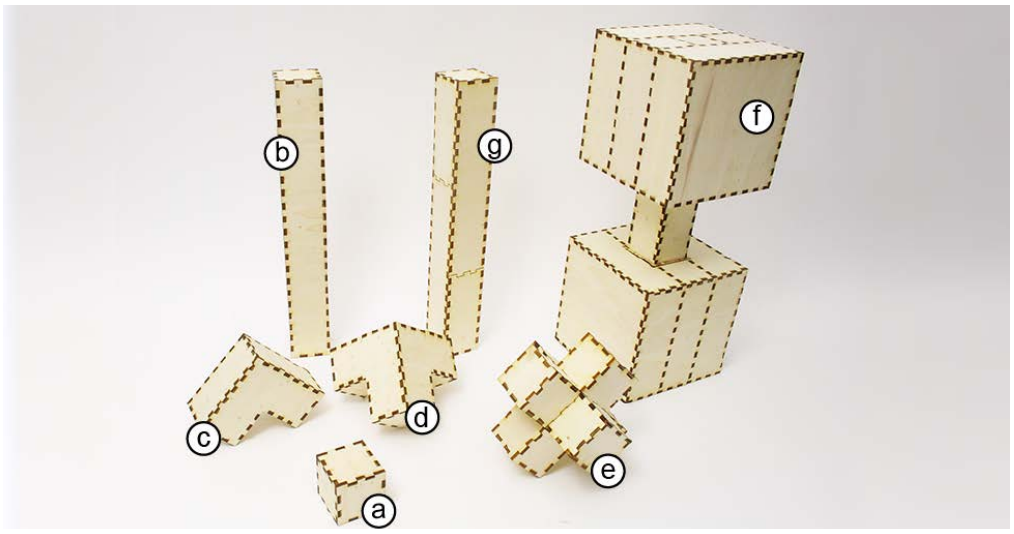


Figure 74: Objects tested

The next two models allowed us to test reinforcement. (e) The star consisted of 7 boxels but was internally reinforced using three pairs of parallel plates. (f) The dumbbell consisted of two 3x3x3 boxels at each end, connected by two boxels in the middle. However, we used reinforcement to extend the center portion into both 3x3x3 boxels.

(g) The final model allowed us to test tessellation. This model was another 7-boxel stick. However, each plate was subdivided into two interlocking plates.

Procedure

To administer the test, we mounted test objects into the custom testing apparatus as shown in Figure 75. The apparatus was essentially a custom vise made from aluminum profiles (from *item Inc.)*. We actuated the device by tightening a nut on a threaded rod until the test object would break. A properly placed force sensor (*forceX 2.30*) measured the applied forces.

We used the apparatus to apply (a) compression along the object's main Cartesian axis, (b) compression against the object tilted by 45 degrees, and (c) compression against the object tilted along two axes. (d) We measured tension by pulling the test object with pairs of straps, and (e) torsion, by holding the test object in place using clamps while twisting the opposite end using a lever. (f) Finally, we measured buckling, by applying a force to test objects at three points.

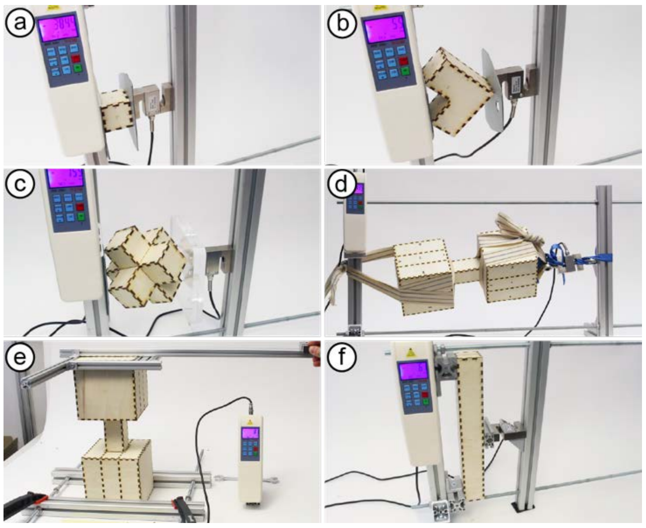


Figure 75: The test apparatus.

Results

Figure 76 shows the main results, i.e., amount of force or torque required to fracture the respective objects. “>500kg” means that the test object was still intact when we exceeded the 500kg value range of our measuring device. “>140kg” indicates that the test object was still intact when our test apparatus started to collapse when applying tension to the dumbbell model.

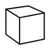
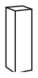
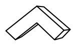
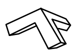
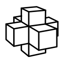
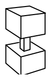
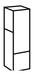
	compression					buckle
	from top	tilted	...2D	tension	torsion	
	>500 kg					
	>500 kg				49Nm	135kg
		293 kg				
			91kg			
	>500kg	>500kg	398kg			
				>140kg	40Nm	230kg
	>500 kg				38Nm	188kg

Figure 76: Forces required to break the respective object.

4.1.5 User study

We evaluate the usability of our system with a user study in which non-engineer participants designed 3D models using kyub, then cut and assembled them. Participants filled in a questionnaire about their experience. We hypothesize that participants find kyub easy to learn and use.

Task

We asked all teams to create a roughly foot-high "persona" figurine for future "design thinking" sessions.

Format

While users spent only 90 minutes with kyub, we conducted our evaluation as part of a two half-day's workshop on laser cutting, a format that gave us time to learn about participants' experience. Participants were in the same space at the same time, as well as several team members. They worked in self-selected teams of two.

During the first day, we gave participants a 2h introduction to the traditional process of laser cutting. During this period participants created their first laser cut designs i.e., figurines to serve as personas for future design thinking activities, which they drew directly in *Adobe Illustrator* or *Inkscape.* Participants added notch joints manually by overlaying rectangles of appropriate width taken from a template onto their designs.

We then showed a demo of kyub and gave participants 90 minutes to design their own models in kyub.

We laser cut the objects offline, and participants reconvened a week later for the second half-day session during which they assembled their models and learned more about personal fabrication. Finally, participants filled in a questionnaire.

Participants

We recruited 18 participants (8 female) with an average age of 35 years. They were part of the design curriculum at an affiliated institution. Few participants (four) had used a laser cutter before. The workshops took place in two rounds, one with 10 participants and one with 8.

Results

All teams succeeded at modeling their personas. Figure 77 shows the resulting designs. Two teams who finished early made additional models, i.e., an advent calendar and the name of their institution in 3D characters.

All participants reported high satisfaction with the models they had made using kyub (6.4/7 on a Likert scale).

Participants rated their enjoyment of using kyub high (5.8/7). Participants report being pleased with the sturdiness of the models (6.6/7). Participants strongly agreed that kyub had helped them create models they could not make before (6.5/7) and strongly indicated that it would be time-consuming to make these models without kyub (6.9/7).

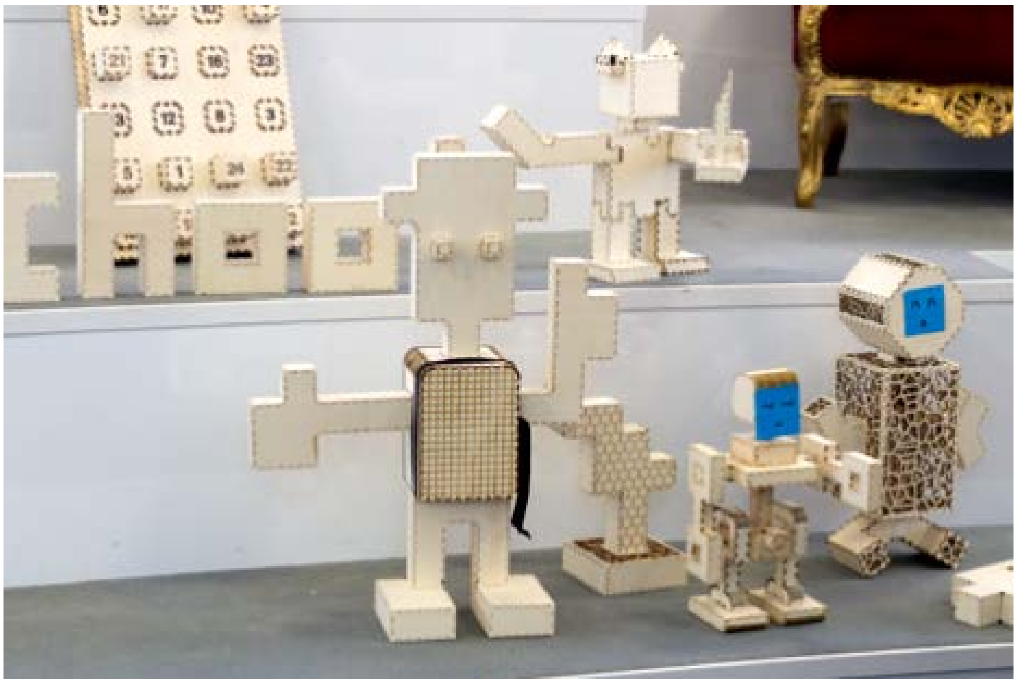


Figure 77: Participant teams created figurines to function as storytelling personas. One team used the remaining time to make an advent calendar one created the name of their institution in 3D characters.

Participants liked the physics simulation in the editor (6.1/7) but said they felt it would be useful to temporarily disable it. P7 explained "gravity helped me model, but once it tipped over it was hard to get it back up".

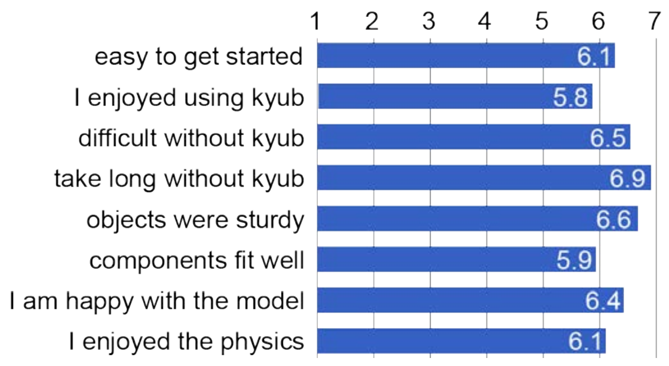


Figure 78: Questionnaire results.

In general, participants found it easy to get started using kyub (6.1/7). Interestingly, some participants credited the physics engine for this. P3: "because everything looks and behaves like the final result, including gravity, it was very easy to understand what was going on". P1: "I just loved how that cube fell in the scene at first, it encouraged me to try things out and model in a playful way with this editor as opposed to Blender which I used before!").

Three participants mentioned appreciating the modularity of kyub. Five participants reported that their favorite feature was the realistic look and feel of the editor. Three mentioned that they liked it best that kyub afforded the creation of sturdy objects.

The overall excitement during the workshop was high and two of the teams approached us later asking for permission to use the software for making prototypes of their regular projects.

4.1.6 Practical use

Kyub is a system that has been live for several years with over 500 beta testers and used for workshops in high schools, Figure 79 shows some of these workshops. To support this, numerous developers have contributed to the system and extended it beyond its mere academic publication. It has served as a platform for various bachelor projects [16,31,35,40] and master theses [66, 109].

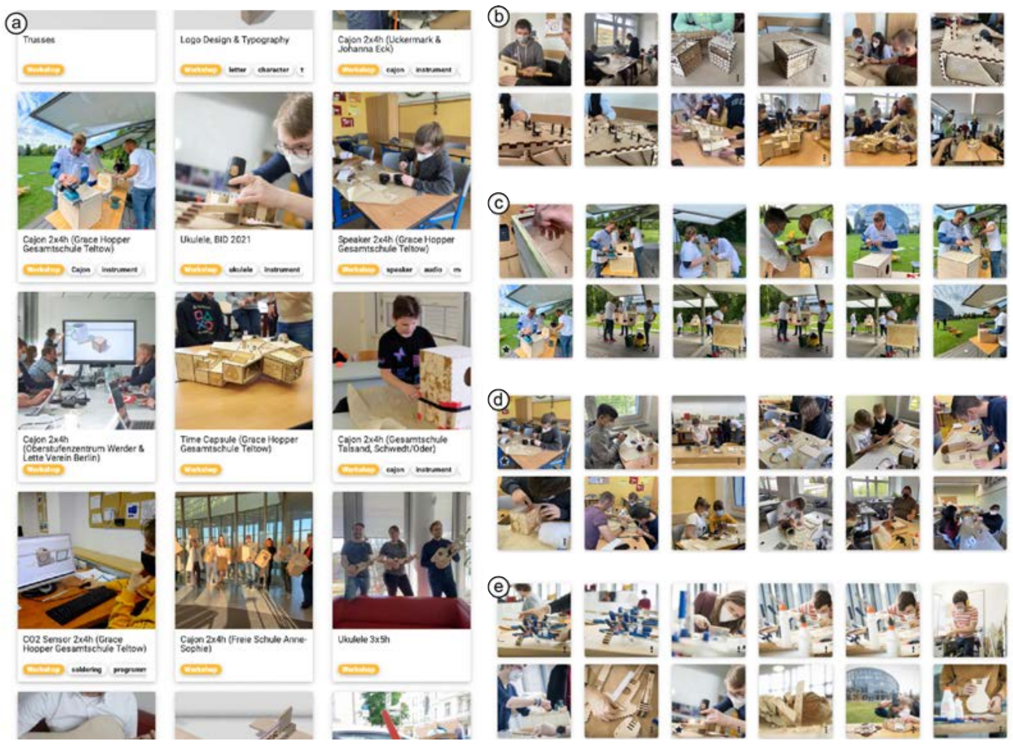

a
Trusses
Logo Design & Typography
Cajon 2x4h (Uckermark & Johanna Eck)
Cajon 2x4h (Grace Hopper Gesamtschule Teltow)
Ukulele, BID 2021
Speaker 2x4h (Grace Hopper Gesamtschule Teltow)
Cajon 2x4h (Oberstufenzentrum Werder & Lette Verein Berlin)
Time Capsule (Grace Hopper Gesamtschule Teltow)
Cajon 2x4h (Gesamtschule Talsand, Schwedt/Oder)
CO2 Sensor 2x4h (Grace Hopper Gesamtschule Teltow)
Cajon 2x4h (Freie Schule Anne-Sophie)
Ukulele 3x5h
b
c
d
e

Figure 79: Kyub in use by pupils around Berlin without modeling or building expertise. (a) An overview of some of the workshops currently offered to high-schools such as (b) building a model of the school of the future, (c) cajons, (d) bluetooth speakers, and (e) ukuleles.

4.2 STRUCTURE-PRESERVING EDITING OF PLATES AND VOLUMES

Kyub as presented thus far, is great at efficient and structural editing, but it lacks support for more complex models that are not based on closed-box structures. We therefore revisited this project by creating a system which gives control over the *detailed* elements of laser cutting, i.e., individual plates and the associated joints, yet at the same time also allows for *efficient* editing by means of volumetric tools while preserving the structure of plates in the model.

As shown in Figure 80, our system consists of four functional groups: (1) We started with a fabrication-aware 3D editor capable of handling volumetric models (*kyub* [14]). This subsystem represents 3D models as a single volume. (2) We added a second subsystem that represents laser-cut models as an arrangement of plates in 3D. This allowed us to add tools that allow manipulating individual plates. (3) We unified these two subsystems by adding a *demotion* mechanism that breaks volumes down into multiple plates, to allow users to apply plate tools to volumes, as well as (4) a *promotion* mechanism, which infers volumetric substructures from sets of plates, to allow users to apply volume-based tools to plate structures.

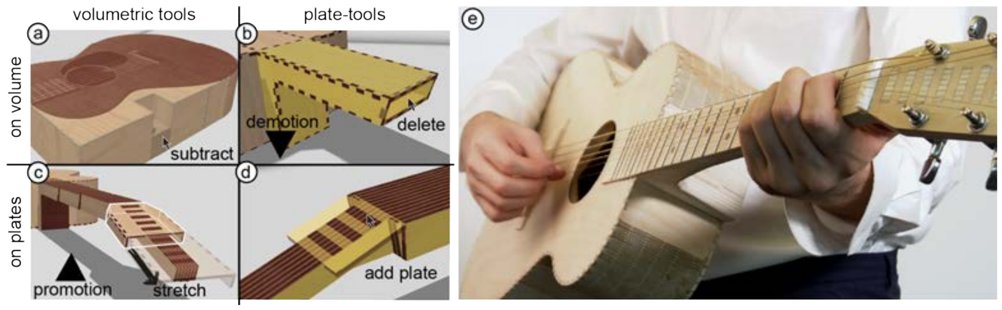


Figure 80: Structure-preserving editing for laser cutting (a) represents laser-cut 3D models as volumes, whenever possible. This allows users to manipulate models efficiently using volume-based tools. (d) It represents laser-cut 3D models as a 3D arrangement of plates, when users want to manipulate models in detail using plate-based tools. (b) The key to making volumetric and plate-based representations work within the same model is that our architecture demotes models represented as volume to plates, when users apply plate-based tools, and it (c) promotes models represented as plates to volumes, when users apply volume tools anywhere. (e) This approach allows users to manipulate 3D models that are complete plate-like elements with volumetric elements, resulting in a level of complexity not possible with previous tools.

As illustrated by Figure 81 our approach allows users to create and manipulate 3D models that are neither all-plate nor all-volume, resulting in a level of complexity not possible with previous tools.

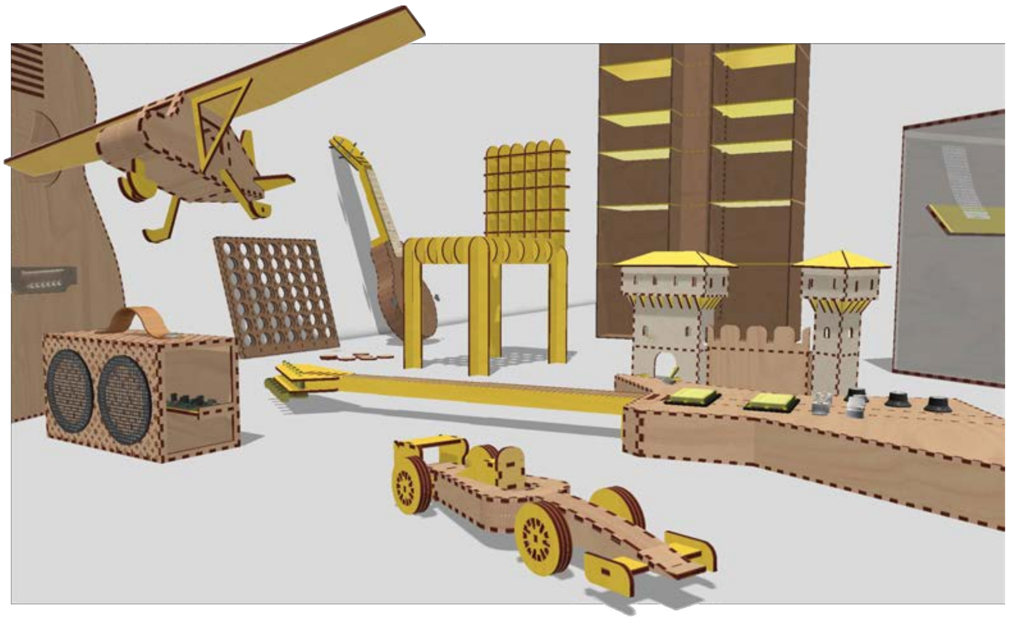


Figure 81: Structure-Preserving Editing allows users to create models that traditionally could only be created and manipulated by hand using "fabrication unaware" modeling. These hybrid models contain plates (highlighted in yellow) and volumetric elements.

4.2.1 the plate-based subsystem

We start by presenting our plate-based subsystem. As illustrated by Figure 82, we designed these tools to be consistent with the volume-based tools provided by the platform we built on: kyub [14].

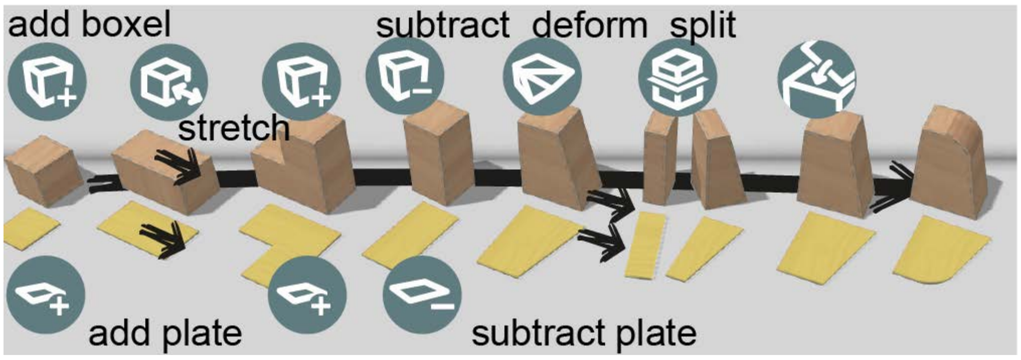


Figure 82: We designed the tools of the plate-based subsystem to be consistent with the volume-based tools provided by the platform we built on kyub [14].

This consistency across subsystems allows for a reduced user interface: as illustrated by Figure 82, it allows us to overload the *edit* functions for plates onto the same functions that manipulate volumes.

In addition to the volume-inspired tools shown above, we added tools that help to arrange plates in 3D. The workflow shown in Figure 83 adds plates at right angles or stacks them onto existing plates. The *move* tool and *rotate* tool allow users to fine-tune the arrangement.

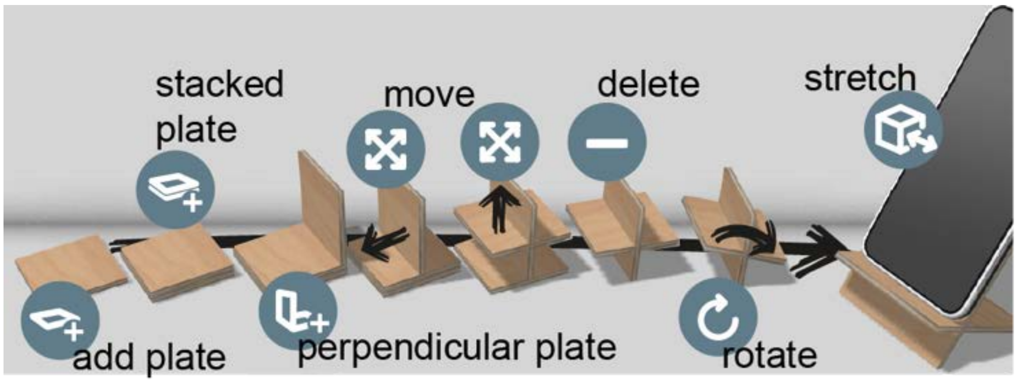


Figure 83: Various add plate tools allow arranging plates in 3D. The move tool allows users to fine-tune their positioning.

The *attach* tool shown in Figure 84 also extends to plates but presents additional options to users on how to arrange plates in 3D after attaching.

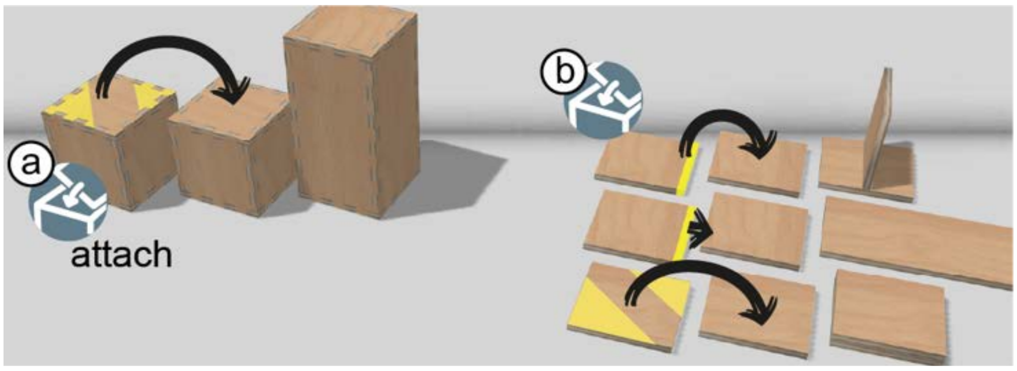


Figure 84: (a) In contrast to the attach tool of the volumetric subsystem, (b) the plate-attach tool provides additional 3D arrangement options.

The plate tools shown above allow constructing a range of basic models, such as the ones shown in Figure 85.

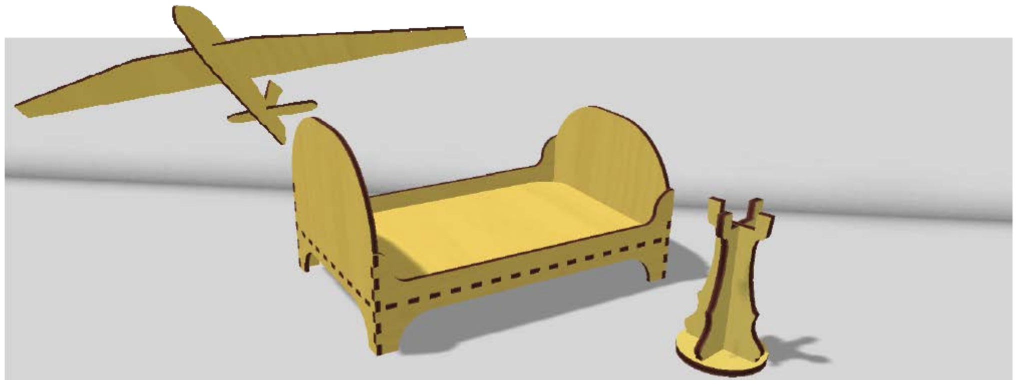


Figure 85: Simple models made using plate tools alone.

The same tools also allow somewhat more complex models, such as the VR headset shown in Figure 86. However, this workflow already hints at the limited efficiency of a purely plate-based workflow.

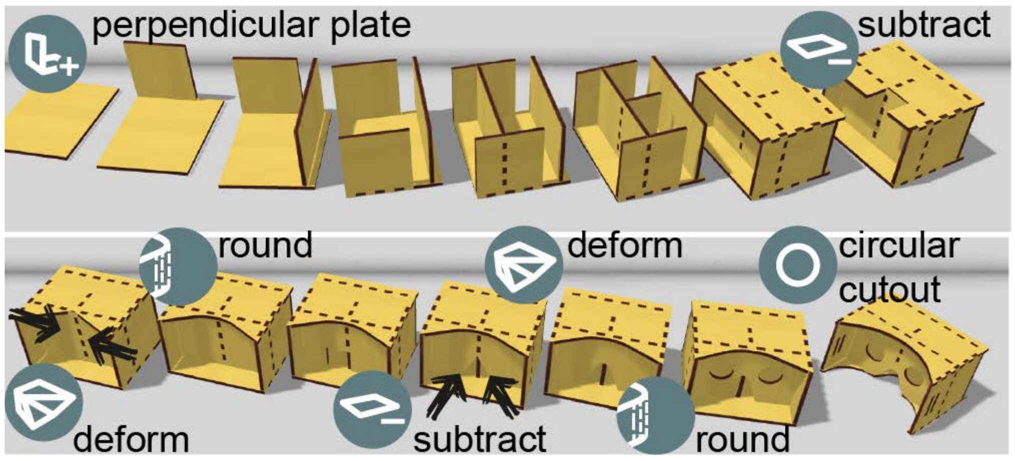


Figure 86: Plate and edit tools allow creating a wide range of models, albeit with limited efficiency (VR headset, id:638605).

4.2.2 promotion

The inefficiency of a purely plate-based workflow becomes obvious when we try to modify the model from Figure 86. As illustrated by Figure 87, making the headset taller now requires users to stretch five plates, move the top plate, doing so in the right order, and getting the resulting alignment right. This is obviously not desirable.

What we want instead is to pull up the top plate and have the rest of the model follow its lead as shown in Figure 87b. We get this type of *volume-based* operation naturally from models that live in the volume-based subsystem. Naturally, we want this type of functionality also for models that originated in the plate-based subsystem.

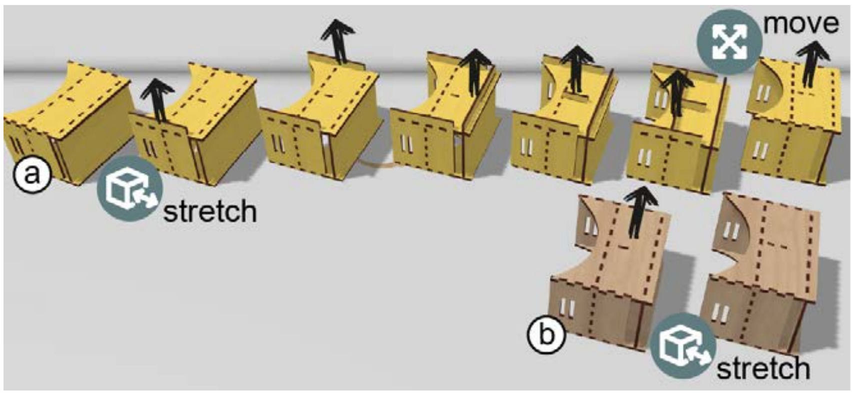


Figure 87: (a) Once demoted to plates, making a VR headset 1cm taller requires six user interactions. (b) Making the same volumetric modification is a single interaction.

We address this by adding what we call the promoter. The promoter is invoked whenever users apply a volume-based tool. The promoter now checks the clicked model: if it is already in volumetric representation, it is done and simply invokes the tool. If the model is in plate-based format, however, the promoter searches the model for volume-like substructures, translates them into a volumetric representation (the promotion), and then applies the tool.

As illustrated by Figure 88, this allows volumetric structures created from plates to be manipulated using volumetric tools, here “stretch”.

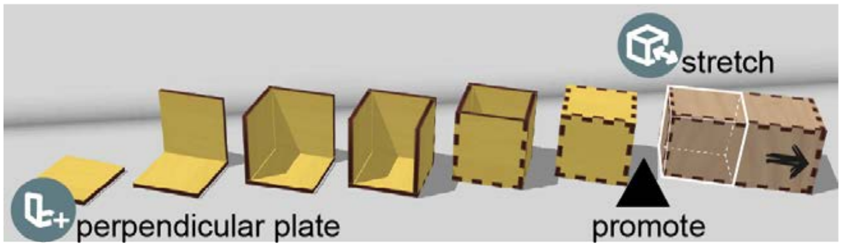


Figure 88: Consecutive add plate tools allow constructing a volume. When applying a volumetric stretch tool, the promoter detects the volume and stretches the plates accordingly.

But the promoter does more. As illustrated by Figure 89a, it identifies volumes also when these are incomplete, and when they are part of slanted models (Figure 89b).

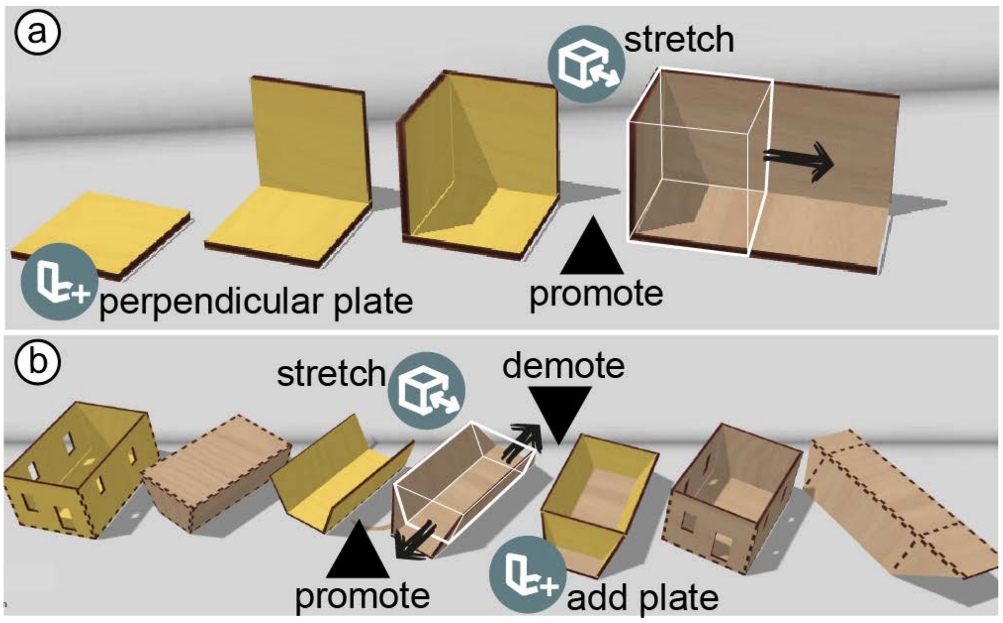


Figure 89: (a) The promoter also identifies incomplete volumes. (b) And works for slanted volumes, here to make a separate rooftop for a dollhouse. To apply the plate tool after, it gets demoted (see next section on demotion).

The key benefit of the promotion mechanism is that it relieves users from the burden to know about how a structure originated, as two structures that look the same can now be treated the same way. Figure 90 shows a three-plate corner created by removing plates from a box, as well as a three-plate corner created by assembling plates. With the help of the promoter, running in the background hidden from the user, either one can be stretched using the stretch tool, producing the same result.

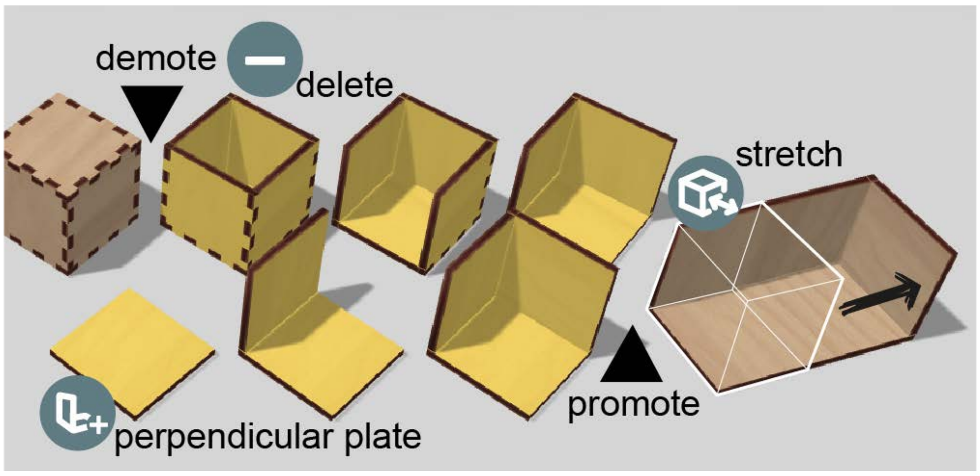


Figure 90: The promoter treats the shown 3-plate assembly the same, irrespective of whether it was created by combining three plates or by removing three plates from a box.

4.2.3 demotion

Going back to the headset, the workflow shown in Figure 86 clearly is not the most efficient way of creating this 3D model. As illustrated by Figure 91, the tools from the volume-based subsystem get users started much faster. However, eventually users need to use plate tools to get the details right, such as the divider between the eyes and the overextended plates.

We enable this scenario with the counterpart to the promoter, the *demoter*. As shown in Figure 91, when users try to apply a plate tool to a model that lives in the volume-based subsystem, the demoter breaks the plates that are touched by the plate into plates, allowing individual plates to be moved or stretched.

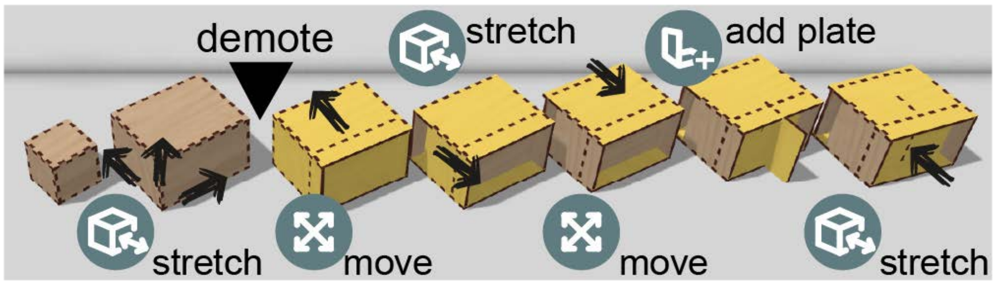


Figure 91: Starting with a volume allows re-creating the VR headset from Figure 86 more efficiently. The part of the model shown in yellow is demoted to plates to allow for the plate tools to apply.

We found this demoter-based workflow, i.e., volume-based tools first, then refinement using plate-based tools to be efficient and the basis for many common models (Figure 92). The promoter, however, is equally crucial for this approach to modeling, as it allows making late modifications, rather than enforcing a strict "waterfall" process.

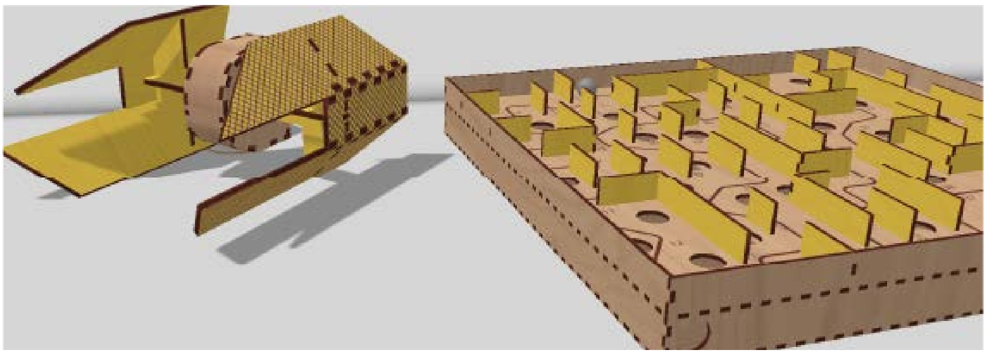


Figure 92: Volume-tools first, then refinement using plate tools is an efficient and thus common workflow.

Figure 92 shows some models that were created using this general "top-down" approach from volume to plates. Most of these models were created by starting with a volumetric element, then adding details using the plate tools, e.g., for structural reasons (e.g., guitar, chair), to mount components inside volumes (e.g., cajon, speaker), or to create small scale structures on a larger model (e.g., race car, airplane). The workflows of more complex models, such as the one shown in Figure 93, may contain multiple invocations of promoter and demoter.

Figure 93 shows the workflow of modeling the guitar of Figure 80 using multiple promotion and demotion invocations. (a) Users start to shape the model with volumetric tools (b) the demoter turns the neck into plates as the user deletes plates and inserts a stack (c) the neck is promoted to a volume when stretching it longer, to then be demoted again as the user modifies detailed plates (d) to make the head, the stretch tool uses the promoter, and to add individual plates the demoter turns it back into plates (e) finally the promoter allows the head plate to be stretched into a volume and (f) the user finishes the model by adding a sound hole, bridge, fretboard and tuners.

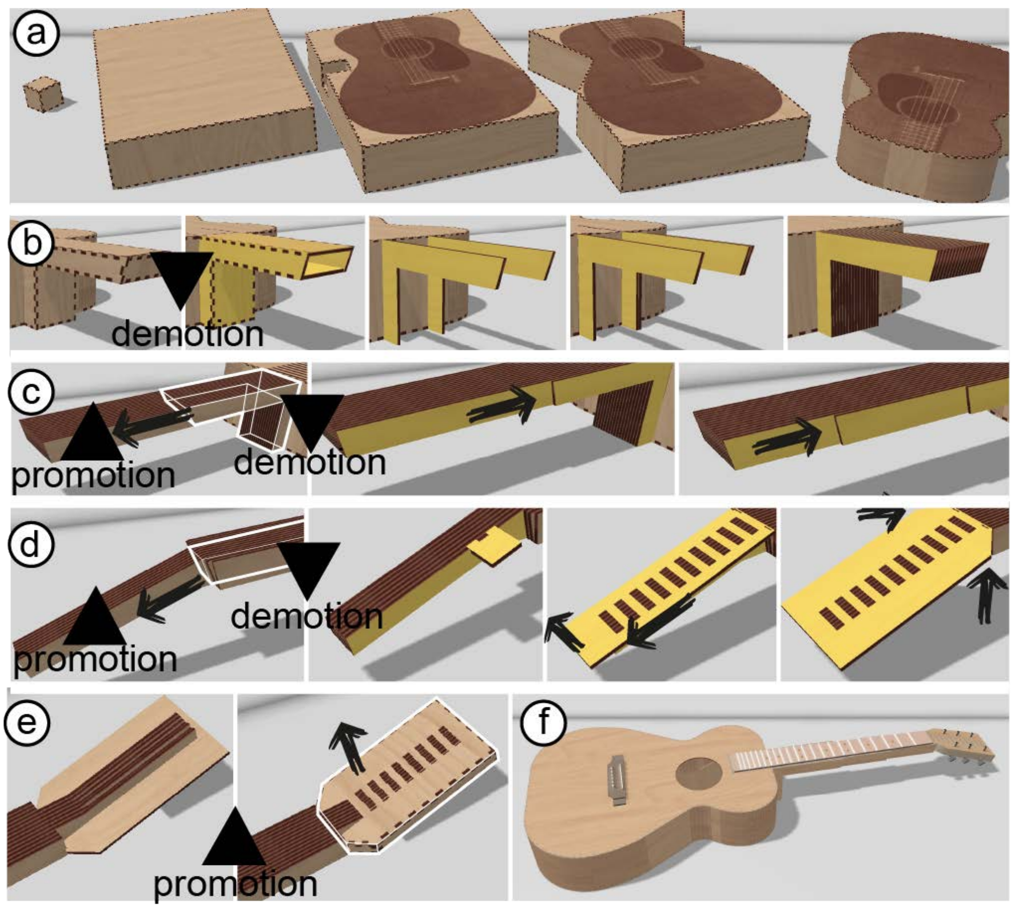


Figure 93: The workflows of more complex models may contain multiple invocations of promoter and demoter.

4.2.4 algorithm and data structures

In this section we present the mechanisms of promotion and demotion. To understand demotion, we take a closer look at the data structure of plates and volumes. As volumes inherently consist of plates, we can break them down relatively easily. To reconstruct a volume, especially when the volume is incomplete, we present the promoter algorithm.

Volume-based vs. plate-based data structures

The promoter and demoter transition the representation of models between volume-based and plate-based data structures.

As shown in Figure 94a, data structures in the volumetric subsystem consist of a single `Mesh` per model, which has its own coordinate system (`Transform`) and operates on a series of linked surfaces (and related edges). Individual plates on the other hand have their own coordinate systems, allowing them to be manipulated without interfering with other plates.

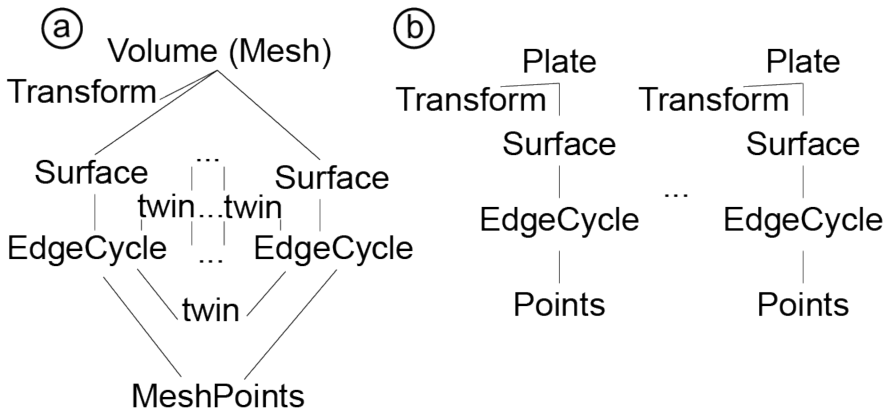


Figure 94: (a) The data structure of a volume vs. (b) data structure if the same model is represented by individual plates.

As shown in Figure 94b, the moment a `Mesh` is "damaged", e.g., by removing a plate, it cannot easily be represented as a `Mesh`. The linked `EdgeCycles` no longer form a fully linked chain, which breaks some of the assumptions the volumetric tools use when operating on `Meshes`. Our system demotes it to a set of plates, as illustrated by going through the `EdgeCycles` and assigning them their own `Transforms`. The cycles remain connected but no longer share `MeshPoints` or a common `Transform`. This gives the plate tools the ability to move them away from one another.

This may seem benign at first, but the demotion means that the volumetric tools no longer apply, as they operate on that single coordinate system and assume full connectedness of the EdgeCycles, turning what could have been a single volume interaction into a long sequence of primitive plate interactions.

This discussion of data structures extends beyond kyub in that fabrication-aware modeling environments for laser cutting would have some representation of plates and how they come together in terms of volumes. While it is possible to maintain both formats in parallel, the volumetric representation remains incomplete upon removal of plates so either the data structure or the resulting volumetric tools are required to handle this.

Promoter Algorithm

At the heart of the presented system lies the promoter. Its purpose is to generate a volumetric description of the model, that tools utilize for volumetric editing operations, such as stretching.
In the example shown in Figure 95, three "plates" are missing to turn the model into a volume. The promoter constructs proxy planes by finding connected edges across two coplanar plates. The L shape on the top of the model, for example, consists of two edges connected at one corner. These edges are coplanar and stretch across two plates. The promoter constructs a proxy plane through these edges and repeats these steps for all connected coplanar edges.

When multiple such connected coplanar edges share a corner, the promoter inserts a proxy edge into the model at the intersection between the proxy planes. When both corners of the edge are shared with other connected edges, the promoter constructs all three planes and inserts a proxy corner at the point where these planes intersect. Finally, it inserts edges between the proxy corner and the edges of the model, resulting in a closed volume.

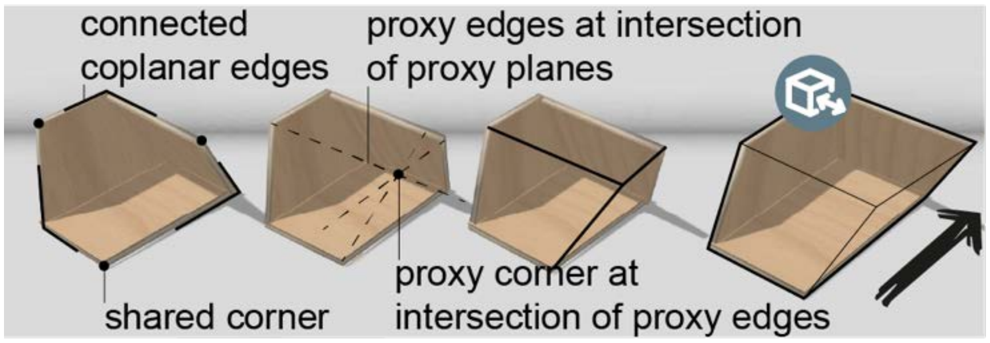


Figure 95: When coplanar edges touch in a corner, they form larger volumes with the adjacent coplanar edges.

When there are no shared corners between sets of connected coplanar edges, there is too little information for the algorithm to locate a proxy corner in 3D. Instead, as shown in Figure 96, the promoter runs the *2D QuickHull* algorithm [11] (which runs in *O*(n log(n))) on the constructed plane and inserts result as edges into the model. In this case forming a basic prism, which can then be used by the volumetric tools. (b) The desk organizer model shows this using a real-world example: After the promoter found the rectilinear volumes, there is a single plate sticking out. Because the convex hull algorithm includes this as a volume as well, it stretches along when users make the model wider.

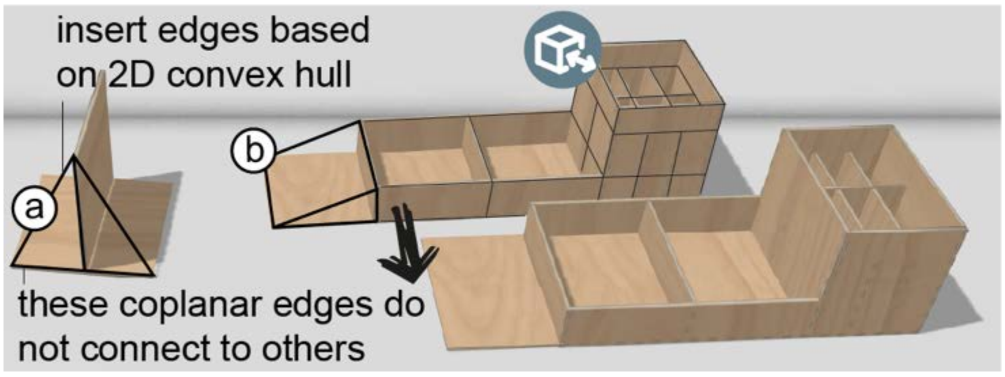


Figure 96: (a) The convex hull of objects where the connected coplanar edges do not share a corner. (b) a practical implication of this case at the example of a desk organizer: because of the proxy prism on the left the base plate stretches with the side plates.

Before the algorithm handles the cases presented thus far, it looks for closed volumes in the overall model. The previous cases therefore typically constitute of the last few plates that were not part of a volume yet. As shown in Figure 97, to detect volumes, the promoter iterates over the edges in the model and groups plates together when an edge connects exactly two adjacent plates. This effectively results in a flood fill for simple, closed volumes, such as the guitar stand of Figure 97.

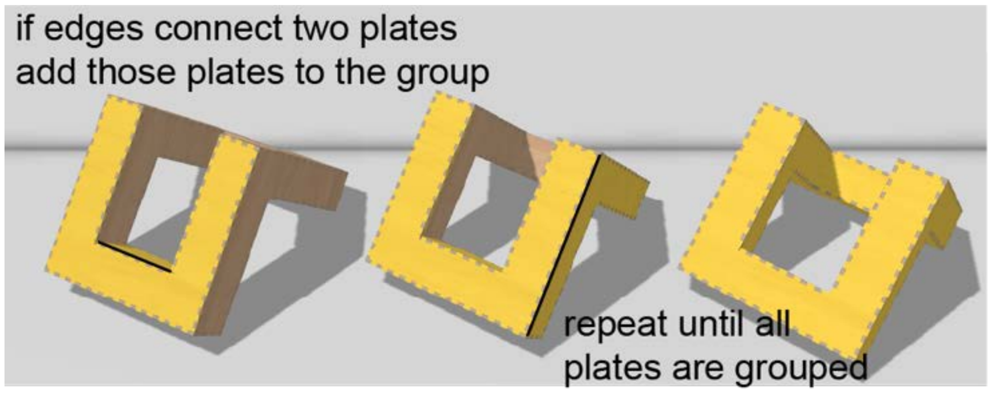


Figure 97: Inferring volumes on this guitar stand, the yellow plates are added to the group.

With full control over plates and volumes, it is possible to construct models which have plates *within* a volume. To respect these, the promoter runs 2D face detection (based on Muller et al. [78]) on the planes before detecting closed volumes. As demonstrated in Figure 98, internal plates within the volume are identified as additional faces, which results in an edge within the top plate that connects to three plates instead of two. This ensures that the internal plate is not simply discarded, but rather causes the volume to be split into two cells when executing the flood fill algorithm, such that volumetric tools behave accordingly. For example, in a stretching operation, the union of the volumes is used, but after stretching, the individual cells restore the internal plate.

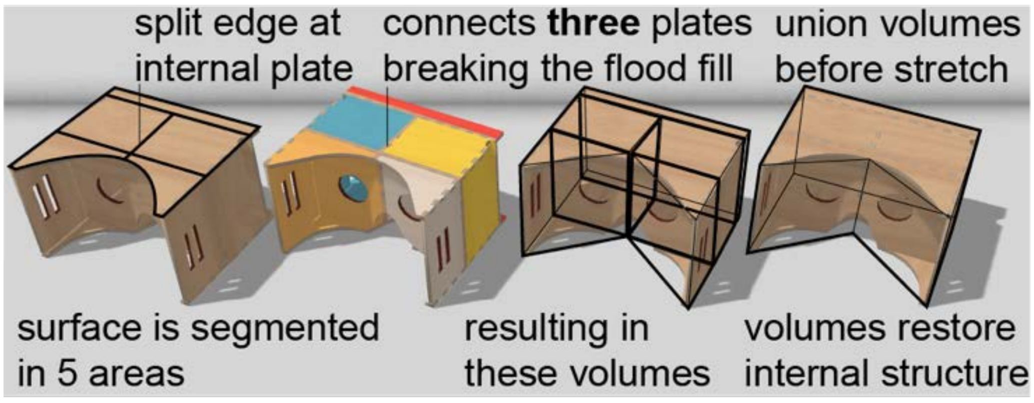


Figure 98: The promoter detects internal structures using face detection 78.

A special case of volumes are stacks of plates. Unlike any of the other plates in models, they are not connected using joints but instead glued on top of each other by users. Because there is no internal structure within a stack (it is all inherently filled with plate), the promoter simply creates a volume composed of the edges of the stack.

Figure 99 shows how the algorithm detected volumetric cells in three example models, and how volumetric stretch operations modify the cells yet keep the overall structure intact.

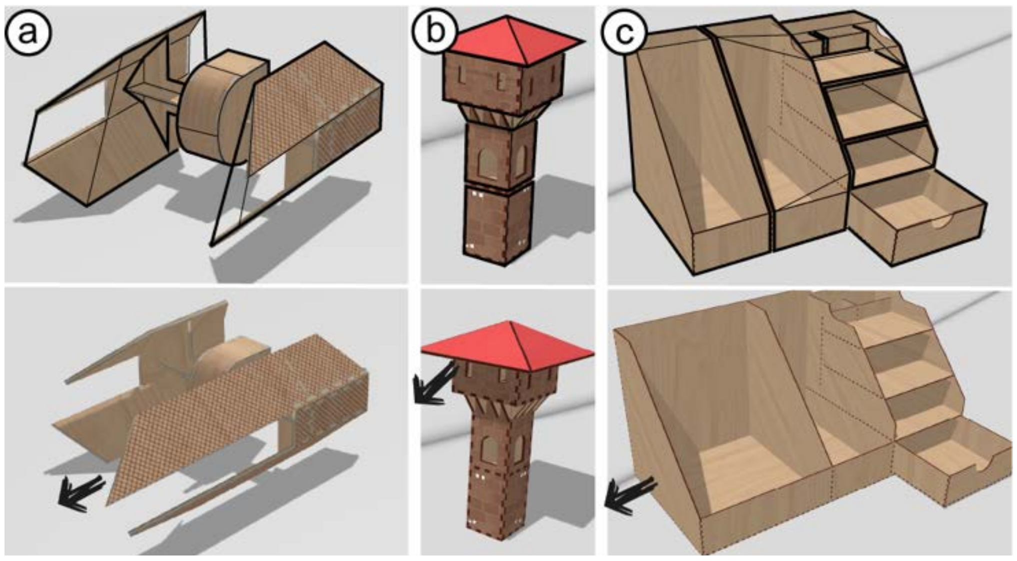


Figure 99: Three example models with their associated volumes as individual cells, the images below show how stretch operations applied to these models stretch these cells while keeping the structure of the model intact.

The explanation of the algorithm so far followed a bottom-up explanation; however, the actual algorithm proceeds in the opposite order, as shown in Algorithm 1. The algorithm recursively inserts proxy planes until all edges of the model are included in a volume. These planes in the next iteration are included as if they were actual plates often resulting in additional or bigger volumes to be found. This approach makes it easy to cache volumes as each tool interaction on the model only requires computing volumes on the newly added plates, extending the previously inferred volume.

Algorithm 3: promoter

Input: List of Edges in the model
Output: Constructed Volume, Cells
Internal data structures: Edges contain a Pointer to their Plate and what Edges on other Plates they connect to, Plates contain a Transform which orients them in 3D space and an EdgeCycle which is a linked list of the related Plates. CoplanarEdges contain pointer to the Edges they belong to.

```
// find all coplanar edges in the graph and store as coplanarEdges
coplanarEdges <- getCoplanarEdges(Edges)
```

```
cycles <- []
for plane in coplanarEdges: // run face detection [78] to split up edges at internal plates, propagate the reference to edges
    cycles.add (faceDetection(plane), plane.Edges)
clusters <- []
for cycle in cycles: // flood fill closed cycles that share 2 plates along an edge
    for edge in cycle:
        if connecting two plates, add as cluster, remove from cycles
// handle non closed volumes and internal plates
while there are still cycles: // internal plates, add to both adjacent clusters
    for edge in cycle:
        if edge connects > two plates, add duplicate of cycle to clusters, remove from cycles
        // check if the cycle connects other cycles and insert proxy edges
        if edge connects to other cycle construct two planes through the points in both cycles and add proxy edge at the intersection, add to clusters, remove cycle from cycles
    // non-closing edges, use convex hull [11] to construct proxy edges into the cycle add to clusters
    clusters.Add(2DQuickHull(cycle)), remove cycle from cycles
cells <- []
for cluster in clusters: // construct cells
    if cluster contains proxy edges, insert corners at intersection between edges or proxy edges, generate proxy planes
    cell <- new Volume from linked list of plates in cluster, unify transforms of plates
    cells.add(cell)
// create the encompassing volume
Volume <-- union all cells
return Volume,cells
```

A limitation of this algorithm are models without clear corners because edges are all curved. Typical examples are skeleton structures with curved "ribs", the algorithm instead considers every point a corner and creates a lot of proxy faces. These produce the right volume, but no currently implemented volumetric tool uses that. More expressive fabrication-aware versions of volumetric operations like *Interactive Images* [141] and symmetry preserving editing [73] support this, but that falls beyond the scope of this paper. As shown in Figure 100b, curved edges perform fine when stretching along the normal of the plane.

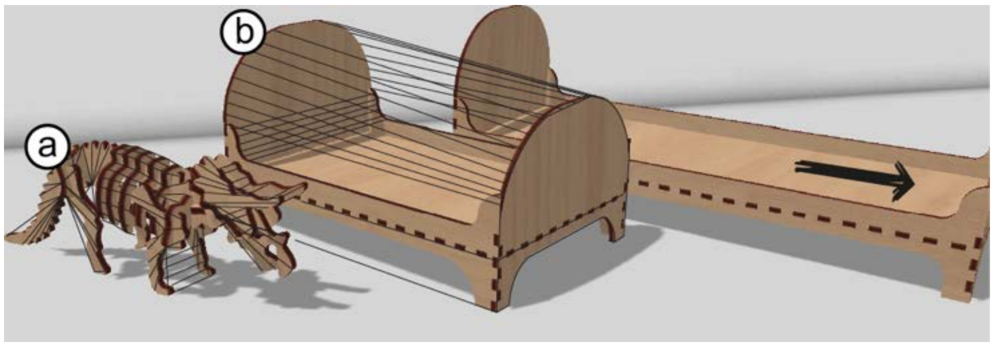


Figure 100: (a) Detected, but less useful volumes. (b) in this case the volume is still useful when stretched along the normal of the plane.

4.2.5 technical evaluation: re-creating 100 models

To evaluate *structure-preserving editing*, we used our system to try and recreate the 100 models from the (assembler[3] benchmark [98], originally from thingiverse [117]). the models from this benchmark were originally created using generic modeling software, thus exhibit a wide variety of construction methods.

We attempted to recreate these models using three systems, i.e., (1) volume-based (original, non-modified kyub [14]), (2) plate-based (FlatFitFab [29]), and (3) volume + plate (the structure-preserving system presented above).

Results

Figure 101 shows the number of models we managed to recreate with each of the three approaches.

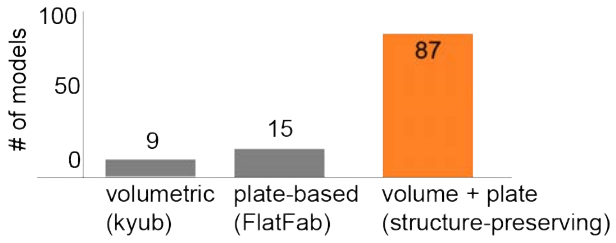


Figure 101: Models recreated using volumetric modeling (kyub), plate-based modeling (FlatFitFab) and our system.

As shown in Figure 102, the models exhibited different modeling workflows: (c) we recreated 53 models with multiple usages of the demoter/promoter, alternating between plate and volumetric workflows, (b) for 12 models we could use a waterfall process where the process is entirely volumetric (if done efficiently) with at the end plate tools demoting the model exactly once, (a) and we made the remaining 35 models using plate tools only like the ones shown in Figure 85.

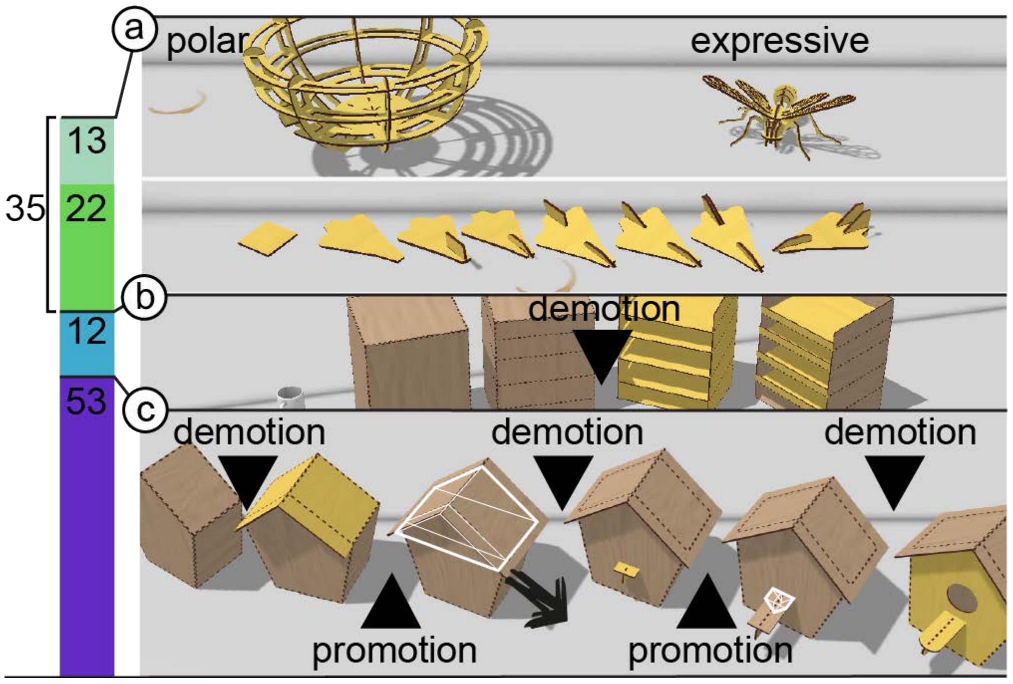


Figure 102: The 100 models of assembler[3] benchmark fall in three categories: (a) 35 models made using individual plate tools (b) 12 models made using a waterfall workflow and (c) 53 models that largely benefit from promotion/demotion in the modeling process.

All 13 models we could not recreate using our tools all fall in that last category; they do not benefit from promotion/demotion but would require a different set of plate tools. Six of them contain plates that are mapped to a polar coordinate system, our tools operate on a cartesian coordinate system, making it hard/impossible to recreate those. The other seven contain highly expressive plates, our system allows for curvature, but such detail is better done achieved using tools optimized for expressiveness (e.g., FlatFitFab [29]).

4.3 SOFTWARE ARCHITECTURE

As outlined in Figure 103, kyub is a typescript application that runs in user's browsers, most of its operations run client-side based on three.js [118]. The notable exceptions are CSG (Constructive Solid Geometry) operations such as creating unions of volumes or splitting geometry, which run on a separate CSG server for performance reasons. These operations take place asynchronously during modeling to not stall the interaction flow of users for this round-trip. The CSG server interfaces with CGAL (written in c++) [23] through an API implemented in node.js [85] and communicates with the front via POST requests, using .off 3D models. 2D user dialogs in kyub take place via angular.js [7] menu items.

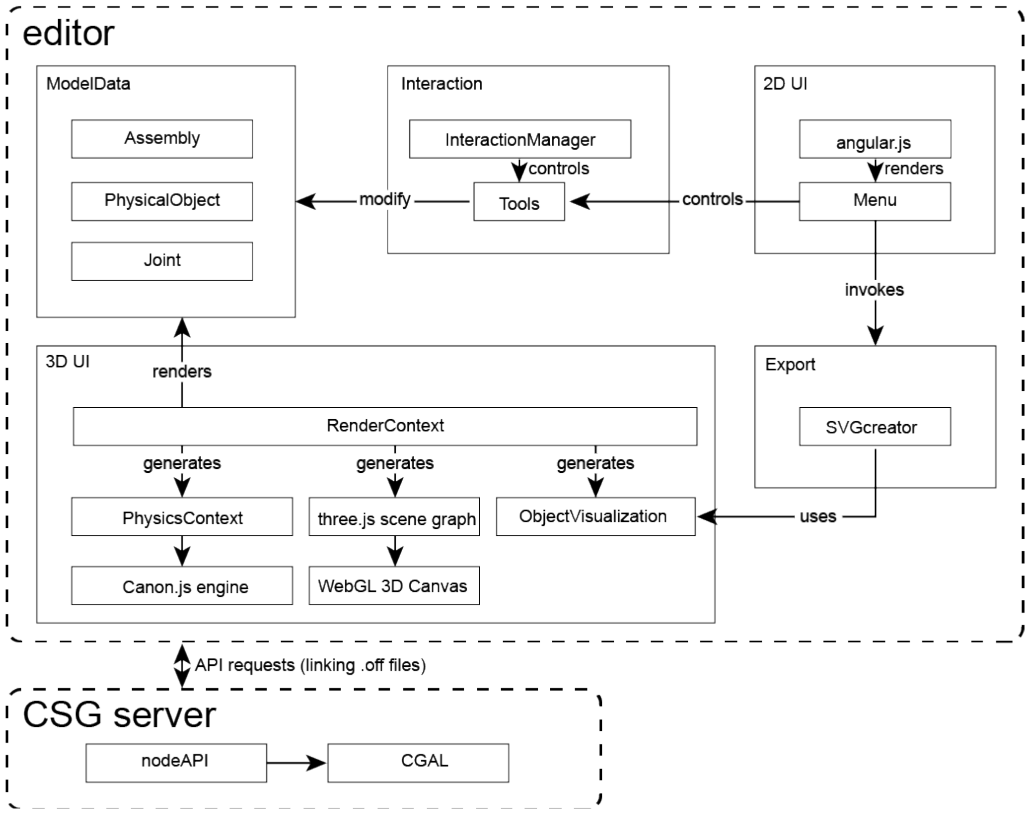


Figure 103: High-level kyub architecture, the editor in front-end and the CGAL server as back-end to handle expensive CSG operations.

Kyub's architecture consists of individual node.js modules allowing for easy extendibility of the functionality of kyub. To give a sense of the breadth of the system, Figure 104 illustrates the scope of kyub modules as of now, structure-preserving editing contributes to highlighted area as well as in the implementation of some individual tools.

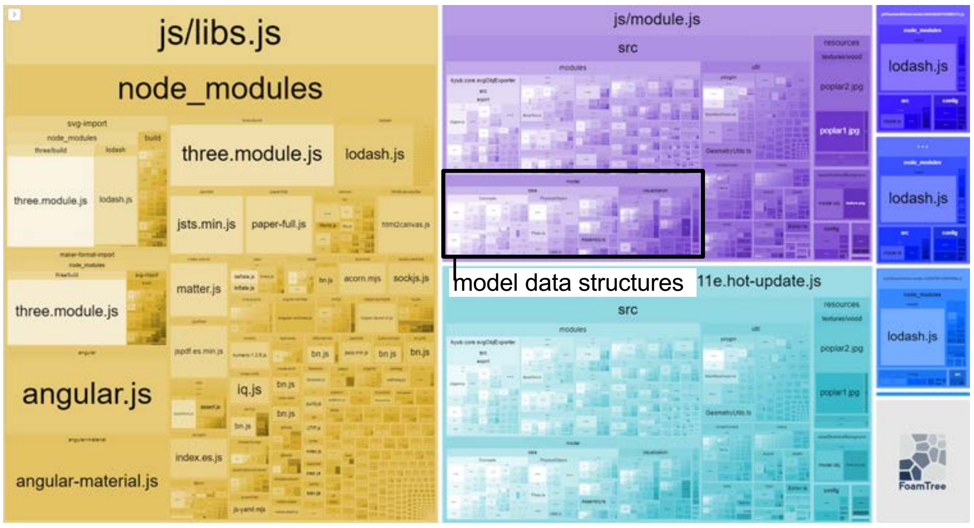
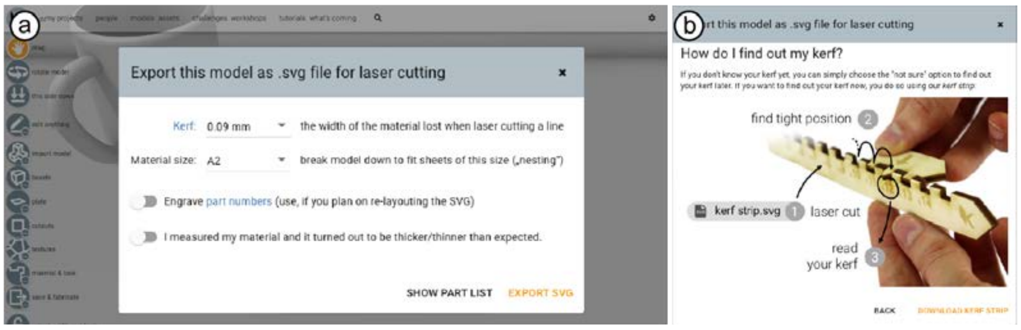


Figure 104: Kyub's modules and high level architecture visualised in a tree map using the NPM package *webpack bundle analyzer* [48]. The purple area contains the editor interface with its underlying data structures. The area with the black outline contains the model data, this is where structure-preserving editing mostly fits into the larger architecture.

The export pipeline enables a high degree of portability. Until exporting, all `ModelData` is represented in a machine-independent form, it consists of `PhysicalObjects`, like plates, which are connected using `Joints`, but it is only when the user exports that these data structures are converted to machine-specific SVG paths. As shown in Figure 105a, in the export dialog users specify the machine's kerf and plate dimensions. In the likely case that users do not know their machine's kerf, kyub exports a gauge as shown in Figure 105b to measure this. The export pipeline uses the SVGnest library [115], building on the algorithm of López-Camacho et al., [75] to lay out the pieces on the plate size, and it adjusts the geometry of all joints based on the kerf value specified by users. This allows users to share 3D models with others as it leaves the machine and material properties generic until users invoke the export dialog, upon which it produces a model optimized for the user's specific machine and material.



Export this model as .svg file for laser cutting

Kerf: 0.09 mm — the width of the material lost when laser cutting a line

Material size: A2 — break model down to fit sheets of this size („nesting")

Engrave part numbers (use, if you plan on re-layouting the SVG)

I measured my material and it turned out to be thicker/thinner than expected.

SHOW PART LIST EXPORT SVG

rt this model as .svg file for laser cutting

How do I find out my kerf?

If you don't know your kerf yet, you can simply choose the "not sure" option to find out your kerf later. If you want to find out your kerf now, you do so using our kerf strip

Figure 105: The export dialog in kyub. (a) during export users specify the machine and material specific dimensions (b) if they do not know their kerf, the software generates a kerf gauge.

4.4 CONTRIBUTIONS

With kyub and its extension of structure-preserving editing, we contribute with a system that lets users create advanced laser-cut models.

Kyub affords construction based on closed box structures, which allows users to make objects capable of withstanding large forces, such as chairs users can sit on. Users construct such models by stacking boxels.

We then integrate volumetric and plate-based modeling paradigms into kyub to allow users to edit laser-cut models in structure-preserving fashion. To accomplish this, we add three elements, i.e., (a) a subsystem for plate-based editing structurally similar to volume-based editing to allow for a tight integration, (b) a demotion mechanism from volumes to plates, and (c) a promotion mechanism from plates to volumes. It is the combination of these four elements that addresses the challenge.

The presented system allows creating models previously only possible with general-purpose 3D or 2D editors, but with the efficiency of fabrication-aware tools, as we demonstrate by recreating models from the *assembler*[3] benchmark [98], as well as complex models, such as acoustic guitars shown in Figure 80e and models shown in Figure 81.

Limitations of our system include that our current set of plate tools does not offer tools for free-form editing (as offered, for example, by *FlatFitFab* [29]) and offers only limited control over alignment, precision, and symmetry. The system is built on the assumption of rigid materials.

4.5 CONCLUSIONS

With kyub and structure-preserving editing we create an environment in which users can create advanced 3D models for laser cutting. It is also a great step forwards in terms of portability: when fabricating such 3D models designed by others, users simply export the model to 2D upon which the kyub exporter generates joints and mounts optimized for the material and machine at hand.

Moreover, the 3D models allow users to make parametric changes to existing models, allowing them to build on the work of others and increase the complexity of models shared online. The guitar shown in Figure 80, in that sense, is not about one guitar, but in the context of structure-preserving editing a starting point that allows users to create a wide-range of *custom* guitars efficiently. This therefore presents a paradigm shift away from the historical machine-specific and hard-to-modify 2D cutting plans and towards 3D formats.

5 CONVERTING LEGACY 2D CUTTING PLANS TO 3D MODELS

The 3D representation and editing methods presented in the previous chapter allow users to create, share, and modify models without having to worry about the underlying machines or materials used to fabricate them. However, large numbers of high-quality models exist in 2D cutting plans and entire sharing communities and industries are centered around these models. As long as that is the case, the chance to have real-world impact with 3D models is inherently limited.

As illustrated by Figure 106, we propose a workflow to overcome these legacy issues by reconstructing 2D cutting plans into 3D models. These 3D models allow users to make 3D parametric changes to their models using software like the 3D modeling environment proposed in the previous chapter. And that environment then in turn exports a 2D cutting plan that is optimized for the laser cutter and material of the user.

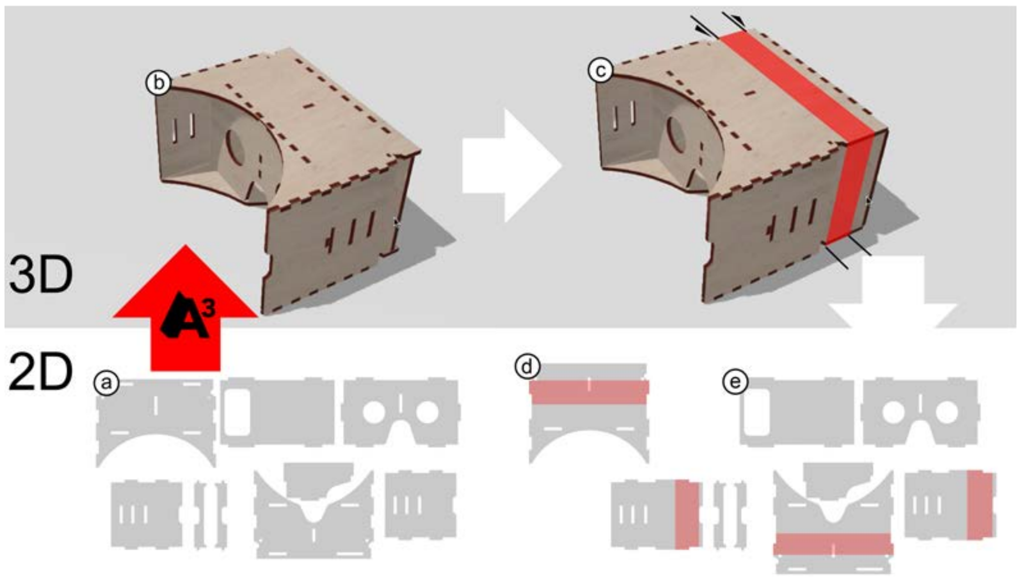


Figure 106: The proposed workflow at the example of modifying the cutting plan of a VR headset: (a) the 2D cutting plan (b) is reconstructed into a 3D model (c) which the user then manipulates in 3D. (d) when done, kyub exports the model to a modified 2D plan.

The resulting workflow allows re-use of models made by others independent of the laser cutter at hand. And as users will be sharing the 3D models instead of 2D cutting plans, it becomes easier to build on the work of others to enable the community to increase the quality and complexity of models shared online. We implemented this workflow in two stages: first we developed a 5-step analysis algorithm followed by an interactive reconstruction process, we captured this in a software tool we call *assembler*^{3} [98].

We revisited assembler^3 by building an automated pipeline using a heuristics-based beam-search algorithm we call *autoAssembler* [100], that allows a large subset of laser cut models to be automatically reconstructed. We integrated both approaches into kyub, in which the automated pipeline is the default process, and the interactive workflow serves as a fallback for users.

5.1 ASSEMBLER^3: INTERACTIVE 3D RECONSTRUCTION

Assembler^3 is an interactive software tool that implements the "mental" workflow shown in Figure 124 as an actual, *software-based* workflow. (a) Assembler^3 allows users to modify *2D* cutting plans by (b) rearranging them into a 3D model, at which (c) users can now apply parametric manipulations using existing 3D editors, before (d) converting back to 2D for cutting. In our study, this workflow allowed participants to apply parametric modifications 10x faster (2:22min on average) than the traditional workflow of rewriting the 2D cutting plan directly. Participants rated the task easy (2/7) and all exported cutting plans assembled into functional models. In a technical evaluation, we furthermore demonstrate the utility of this workflow by reconstructing 100 of 105 models found in online repositories.

5.1.1 assembler^3 workflow

The main step implemented by assembler^3 is the reconstruction of a 3D model from a 2D cutting plan.

While we present our algorithm in full detail in section 5.1.3 "the algorithm of assembler^3", Figure 107 provides a preview. Assembler^3 reconstructs a model, such as (a) this plate of the VR headset, by performing the following steps: (b) plate detection determines what are plates and what is scrap using the nesting order of paths and assigning plate vs scrap in alternating order. (c) Joint detection identifies joint candidates by detecting patterns of left/right turns in the paths. (d) Material thickness detection has every joint candidate vote for a material thickness, the dimension most commonly voted for by the joints determines the material thickness. (e) Joint matching and, for fast retrieval, storage in a hash. (f) Interactive reconstruction in a 3D environment (kyub).

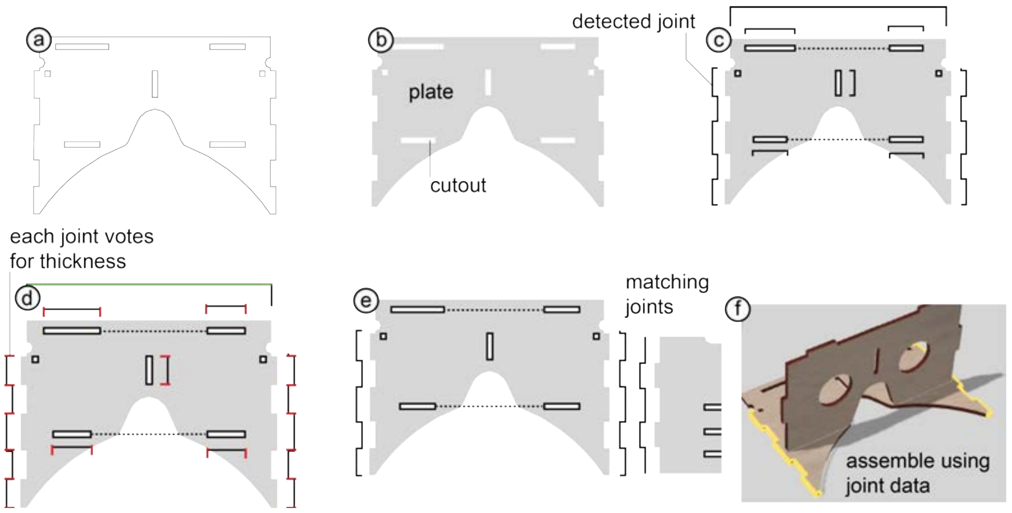


Figure 107: Pipeline of parsing the top plate of the VR headset (in reality it parses the entire SVG).

Figure 108 illustrates the last step, i.e., *interactive reconstruction* in (a) the 3D environment. (b) The main interaction is the user attaching two plates to each other using the *assemble tool*. Here the user has selected a plate, which causes it to highlight in yellow. Assembler[3] responds by highlighting candidate joints that *could* be paired up with the selected plate. This response is instantaneous, as all matches were precomputed and stored in an efficient-to-retrieve format (joint hash).

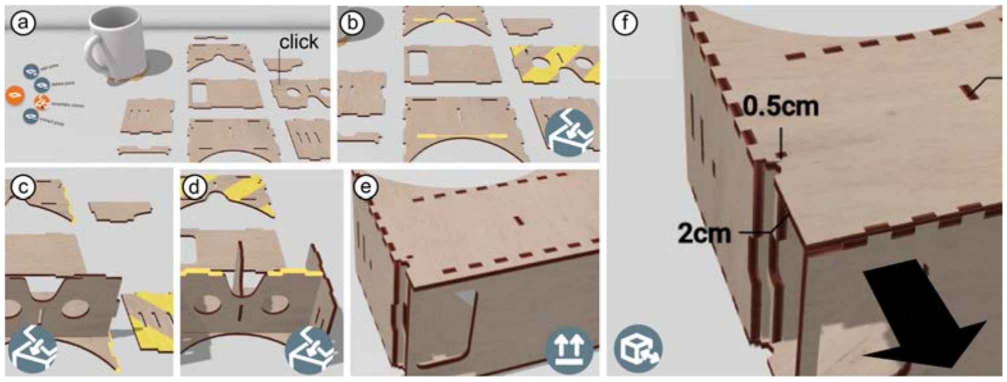


Figure 108: Reconstructing the 3D model of the VR headset. (a) When assembler[3] loads the SVG, all plates are displayed in the 3D modeling environment, here kyub [14] (b) The user clicked the *assemble tool* on the front piece. Assembler[3] responds by highlighting this plate (yellow stripes) and by highlighting joint candidates located on the other plates. (c) Clicking one of the suggested candidates assembles the plate. (d) The user repeats this until the model is assembled. From now on, the user uses standard kyub tools to interact with the model (e) to see the front plate, the user flips the model. (f) Once reconstructed, the user can apply arbitrary parametric changes. Here the user accommodates for far-sightedness by stretching the front plate.

As shown in Figure 108c the user then clicks on a target plate to assemble the selected plate with, assembler^3 responds by attaching the joint candidate to the target plate. The user repeats this step until all plates are assembled into the 3D model.

Sometimes, there are multiple ways how a plate can be attached, including rotations around up to three degrees of freedom. To keep the interaction flowing, assembler^3 picks a default placement. Assembler^3 does so considering the following heuristics: (1) maximizing the probabilities that each of the involved joints actually is a joint, as determined during SVG parsing, (2) picking results that are free of 3D intersection, (3) maximizing the number of joints that will be completed by the attachment operation (such as co-aligned joints elsewhere on the plates).

In our technical evaluation, assembler^3 got orientations right at first attempt for 78.6% of cases.

As illustrated by Figure 109, in those cases where assembler^3 gets the default orientation wrong, successive clicking of the "floating menu" attached to the assembly allows users to cycle through other orientations in order of descending score, until the correct one has been found (on average 1.7 clicks in our evaluation). This score is determined using the same heuristics as for the default orientation and it skips orientations that produce the same outcome because of symmetrical parts.

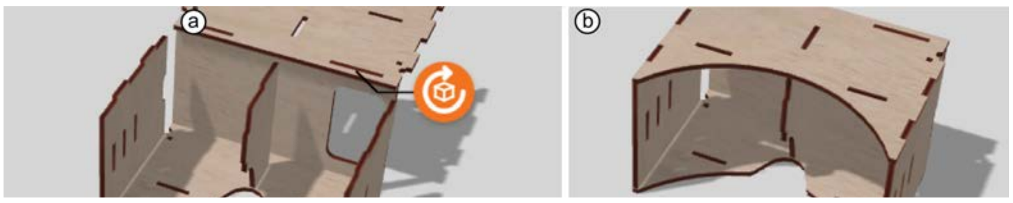


Figure 109: (a) To override a suggestion, the user uses the floating menu item. Assembler^3 presents another orientation of the plate. (b) In this case assembler^3 flips the plate, which leaves the user satisfied with the result.

Whenever combining two plates assembler^3 replaces the joints originally contained in the 2D cutting plan with joints in the format of the surrounding 3D editor, so as to allow joints to accommodate the parametric changes about to happen.

Once the conversion to 3D is complete, assembler^3 allows users to apply any 3D editing tools to the model. In Figure 124e, the user uses this to accommodate the design for the user's far-sightedness.

After reconstructing a model once, other users can leverage that effort by now applying *any number* of additional modifications, such as the changes shown in Figure 110. Users may also store and share the model in 3D format, which lowers the bar for others to build on this model, contribute to it, and help it achieve complexity.

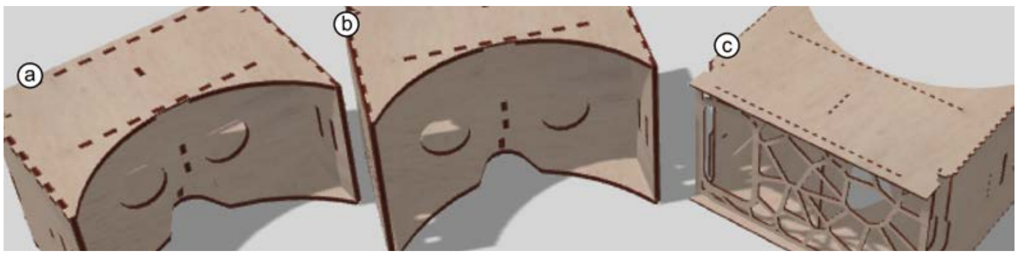


Figure 110: When users have converted the VR headset model to 3D once, the 3D model allows leveraging existing tools (such as kyub [14]) to perform any number of modifications efficiently, such as (a) adjusting the inter-ocular distance, (b) making the headset fit a wider phone by stretching it vertically, (c) or making the headset more comfortable to wear by reducing the weight of the device by decreasing material thickness and removing unnecessary material.

Note how the 3D reconstruction process seamlessly *integrates* into the regular 3D editor environment. Initially, we had expected that 3D reconstruction would have to result in a more wizard/process-funnel like design—a workflow limiting users to back-and-forth navigation along certain process steps. However, once we had fully automated the 2D processing steps, such as material thickness reconstruction, we found the opportunity to implement 3D reconstruction as a *set of tools*, most prominently assemble tool and the floating menu.

The benefit of this integration is that it invites users to tackle plates in any order, fix mistakes recognized late by disassembling selectively using the "extract plate" tool, use the editor environment to perform any other modeling activities along the way, start parametric manipulations before the import is even complete, or even to reconstruct models only partially for the purpose of remixing rather than full reconstruction.

5.1.2 surveying sharing practice

In a short survey, we find that there is surprisingly little remixing/customizing of laser-cut models shared on thingiverse. Flath et al [37] conducted an in-depth survey of thingiverse models in 2017, in which they found 54.7% of all models to be remixed. We repeated their analysis filtered by the search terms "laser cut", "lasercut", and/or "laser-cut", and find a mere 17% of models to be based on work of others (open-source script to reproduce our analysis [94]). We thus conclude that there is a lack of remixing of laser-cut models compared to other practice on the platform. Flath et al furthermore draw attention to the "thingiverse customizer" (a tool to easily make parametric modifications to 3D models) as a catalyst for remixing of 3D models. We believe that assembler^3 could play a similar role in the context of laser-cut models.

We find furthermore anecdotal evidence of a desire to make parametric modifications laser-cut models in the comments to models that are shared. The most popular modifications are changes in material thickness. A strong example is thing 24561, which has these two (out of 16) comments:

"Hi, can you please help me scale this dragon into a 3mm thickness template and share with us …"

"Hi, am new here and I would like to cut this dragon on a 3mm acrylic. Please could you help guide me on how to scale it?? …" (other comments are in the vein of "great model")

Thing *286* mentions the problem in their own description, and a comment to thing 691869 indicates a failed attempt at modifying the thickness. Other thingiverse designers share multiple thickness files to circumvent the problem.

In current-day sharing, assembler^3 could thus already play an important role as a tool to reduce the hurdle of varying material (thickness) of models. However, if the boost of the thingiverse customizer is any indicator, a tool for customizing models can go a long way to structurally change how users make and share models on the platform.

5.1.3 the algorithm of assembler^3

To allow readers to replicate assembler^3, we now provide a detailed description of its algorithm.

Assembler^3 uses a five-step algorithm: (1) normalizing the SVG and detecting plates, (2) detecting joints (3) detecting material thickness and (4) joint matching and hashing the joints for fast retrieval, and (5) rendering the plates in a 3D editor (kyub), to allow users to reconstruct the model. Algorithm 4 provides an overview of the first 4 steps.

Algorithm 4: ParseSVG

```
Input: 2D cutting plan svg,
Output: list of plates, hash of joints jointHash, and material thickness
Internal data structures: lines, closedPaths, votes <-- Lists
lines, closedPaths <-- linearize (svg)
sort closedPaths by area, descending
for each path ∈ closedPaths
    path.children = closedPaths after path that are enclosed in path
end
for each path ∈ closedPaths
    if path.nesting MOD 2 = 0 then
        add path with path.children as cutouts to plates
    end
end
for each plate ∈ plates
    add to plate.joints, jointHash <--  detectJoints (plate.path)
    for each joint ∈ plate.joints
        add joints.assumedThickness to votes
    end
end
thickness = max (frequency(votes,interval 0.1))
updateJointProbabilities (thickness)
return plates, jointHash, thickness
```

Plate detection

Assembler^3 segments the cutting plan into plates and cutouts. It achieves this by determining the nesting order of paths and assigns them alternatingly to plate or cutout. The outer path will produce a plate, if there is another path enclosed within this, it is a cutout etc. To be able to do so, assembler^3 breaks down geometry to line segments (cutting paths in SVG can be polygons, polylines, paths, etc). Assembler^3 then iterates over closed paths to detect if there is any smaller path enclosed within.

In some 2D cutting plans, designers optimize the path the laser cuts by re-using paths between plates. The laser then cuts in one line, two edges of different plates. Naively, assembler^3 would not be able to determine that both are plates. To detect this issue, assembler^3 checks if there are any points where more than two-line segments come together. If this is the case, assembler^3 traverses the path once with clockwise turns and then with counterclockwise turns and assigns both to be plates. Similar to constructing a doubly connected edge list [80].

While pathological cases can be constructed that the algorithm will not properly recognize (specifically cases where waste material, such as the inner cutout of the ocular pieces, would be used as parts) assembler^3 achieved a 100% success rate in recognizing plates in our technical evaluation.

Joint detection

Assembler^3 now detects joints. To illustrate, Figure 111, shows the joint "candidates" assembler^3 finds in the VR headset. Note that this initial set of joint candidates contains several incorrect candidates, here shown in red. This is expected at this stage, as assembler^3 will remove these incorrect candidates after material thickness detection.

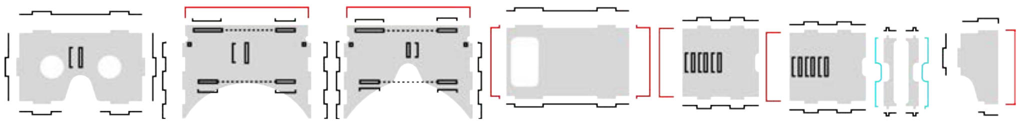


Figure 111: Joint detection on the VR headset. The red lines are false positives, which will get adjusted later, and the blue lines are joints that will receive a low probability because of their odd shape.

Assembler^3 detects line paths in the 2D cutting plans that may or may not be joints. It does so by looking for paths that form certain patterns of left and right turns. As illustrated by Figure 112a, finger joints, for example, follow the pattern left/right/right/left.

Assembler^3 then estimates the probability of a given path actually being a joint. One factor it considers, is how close a joint is to the idealized shape of that joint. As shown in Figure 112 at the example of a finger joint, assembler^3 expects certain characteristic properties. Finger joints, for example, it expects to feature 90-degree angles, top and edge to be parallel, and widths to be larger than heights. However, since the 2D cutting plan might be hand-drawn, assembler^3 will also accept imperfect renditions; it will assign these lower "joint probabilities" though. The feature shown in Figure 112b, for example, follows the "left, right, right, left" pattern of a finger joint, but the width/height ratio is off and the lines are not parallel, assembler^3 is still willing to consider it a finger joint, but with a low probability. (c) Furthermore, assembler^3 increases that probability when it detects repetitions of a pattern.

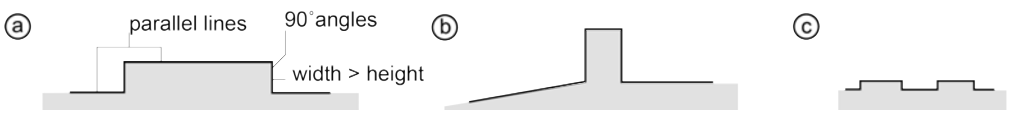


Figure 112: (a) The ideal finger joint has 90-degree angles, a top line parallel to the edge and has a bigger width than height (b) this finger joint has a much lower probability, it could just as well be some aesthetic feature of the model? (c) repetitions of a pattern increase its probability.

While, as mentioned, assembler³'s joint candidate list contains a lot of false *positives* at this stage, the algorithm captures 98% of joints contained in models (see section 5.1.4 'technical evaluation').

Material thickness detection

Assembler³ derives the material thickness from the collected joints. It achieves this by having all joints "vote". In this voting process, each joint votes for a thickness based on its shape, the most frequently mentioned length wins. Figure 113 shows examples of three types of joints and illustrates which line segment is considered as vote for material thickness.

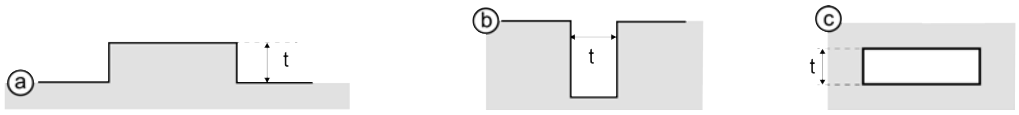


Figure 113: Each joint votes for a material thickness labeled "t" in this figure. (a) finger joint, (b) cross joint, and (c) a mortise-tenon joint.

To allow assembler³ to extract voting information from imprecise (hand-drawn) joints, assembler³ replaces joints with an idealized version of that joint, as shown in Figure 114. The idealized joint then votes with reduced weight.

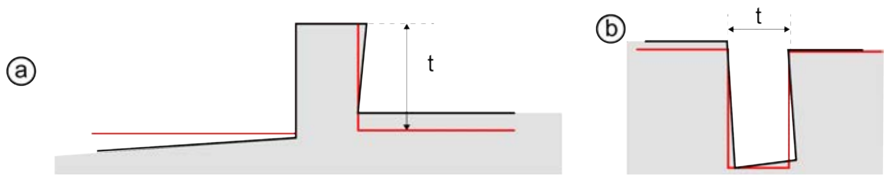


Figure 114: Assembler³ first idealizes non-ideal joints and then haves them vote for the idealized material thickness but with an additional penalty to reduce their impact. The red overlay represents the idealized version of the joints above.

Sorting votes in to buckets (buckets using 0.10mm intervals, picked in reference to the precision of common laser cutters) allows assembler³ to determine the bucket with the most votes quickly, and thus determine material thickness.

Note that the joint candidate list still contains an unknown percentage of false positives—these false positives represent some other design considerations that look at least a bit like joints and are thus allowed to vote. The reason we still have these design features in at this stage is that material thickness detection and joint detection are *mutually* dependent: knowing a joint makes it trivial to tell the material thickness; and knowing the material thickness makes it all but trivial to tell what is a joint. Assembler^3 resolves this mutual dependency by starting with joint detection, but casting a wide net, and delaying the filtering i.e., only now that material thickness is known, assembler^3 uses this information to filter, i.e., it reduces the joint candidate set to those candidates the relevant dimensions of which *are* the material thickness.

The algorithm works despite the mutual dependency because actual joints *all* point to material thickness, while design features tend to point to *random* line segment lengths. This causes joints to outweigh the design features in almost all cases, allowing assembler^3 to achieve a 99% success rate in detecting material thickness. For details, see "technical evaluation".

2D cutting plans tend to contain ornamental features next to the functional joints. Now the joints are known, assembler^3 discriminates between lines that are part of joints and lines that serve ornamental purposes. In SVG files, this is typically denoted using color as the laser cutter uses that information to decide what will be cut and what will be engraved (e.g., burn on the material without cutting for aesthetic purposes). Joints will *have* to be cut, otherwise the model cannot be assembled, so colors without any functional joints are likely ornamental. Assembler^3 uses that information to disambiguate the colors in the cutting plan. Assembler^3 assigns the color(s) used for joints as 'cutting'. It assumes other colors to be engraved and will import these as decorative ornaments on the plates as shown in Figure 115.

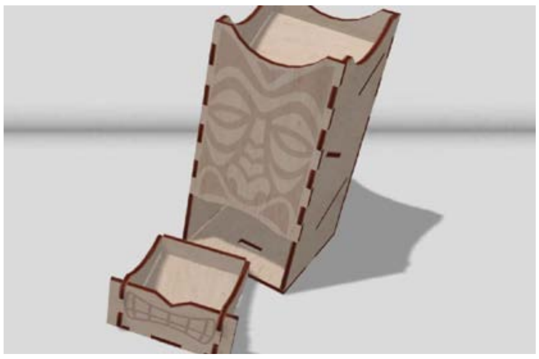


Figure 115: Assembler^3 imports engravings as ornaments on the plates.

Joint matching and storing matches in a hash

Assembler[3] is now just one step away from showing plates and joints to the user who will then try to reconstruct the 3D model. Users will click a plate and assembler[3] will respond by highlighting possible matches. This requires Assembler[3] to know which joints can be matched with which other joints.

In this section we present how assembler[3] precomputes matches and stores them in an efficient data structure, i.e., a hash that allows it to look up matches quickly. This hash is key to allow Assembler[3] to perform reconstruction at interactive rates.

The general objective of joint matching is to identify all other joints that can be fit into the joint at hand: finger joints map to finger joints or t-joints, cross joints map to themselves. As shown in Figure 116, assember[3] determines whether two joints fit by checking what types of joints match (finger, cross, t-joint) and then comparing their respective shapes (signature) to determine which joints interlock.

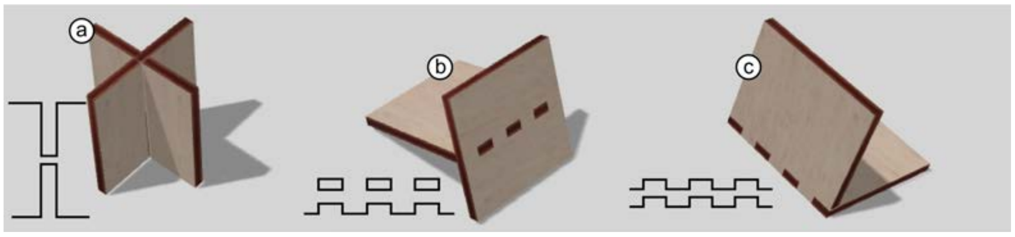


Figure 116: Joint types (a) cross joints, (b) t-joints and (c) finger joints

To make the hash robust to variations in the amount of material the laser removes (aka kerf), for finger joints and t-joints, assembler[3] defines the signature by the sum of a cavity and a protrusion of each joint. As shown in Figure 117, the centers of the features have to align (independent of kerf) otherwise the joints do not fit. This furthermore guarantees that the joint with opposite finger/cavity signature ends up in the same cell of the hash. In the case of cross-joints, the signature is made up from the depth of the joint and the material behind that.

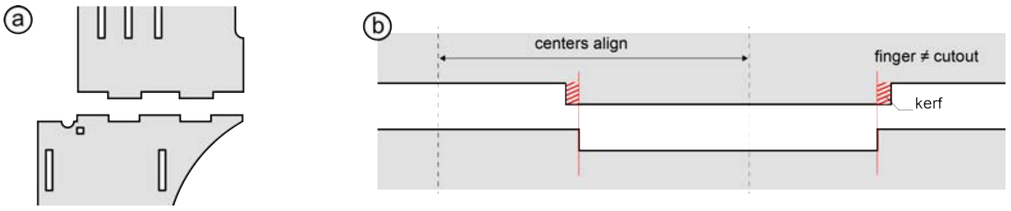


Figure 117: (a) Two plates of the VR headset that fit together. (b) On closer inspection, the fingers of the one joint do not match the cutouts of the other because of the material removed by the laser (aka kerf). However, the centers of joints *have to* align, so assembler[3] hashes the sum of the width of a cutout and a finger as the signature for a finger joint.

For a given joint, assembler^3 looks up that joint in the hash. It will find the joint itself and all other joints that share the same signature (aka collisions), i.e., ones it could possibly match with. If there is only one matching joint, assembler^3 returns it in *O(1)*. If there are more collisions, assembler^3 retrieves an ordered list of joints from the hash. In the case of finger and t-joints, the list is ordered by the number of repetitions of their pattern. A binary search lets assembler^3 get to the right candidate. In the case of cross-joints the list is ordered by joint probability.

Supporting users in assembling the model

Finally, assembler^3 presents the parsed SVG data to the user in the 3D editor. It renders each plate recognized in the *plate detection* step and gives it the thickness determined in the *material thickness detection* step. While not shown to the user yet, each plate knows which joints it contains and each joint knows what other joints it wants to match with.

Assembler^3 lets users assemble the model interactively. It does so by highlighting matching joints on user-selected plates, as presented in section 5.1.1. This is the only step in the algorithm that depends on kyub functionality, up to this point all data structures and implementation apply to any 3D modeling environment for laser-cutting (assuming it has a notion of plates and joints). Kyub as of the moment of publication is the only 3D editor that could handle and make advanced modifications to the models. Particularly with the help of the promotion and demotion mechanisms presented in the previous chapter.

5.1.4 technical evaluation

To validate the technical aspects of our algorithm, we ran assembler^3 on 105 models found online and assessed the results. We drilled down and evaluated the success rate of the steps and performance of our algorithm.

Coverage: assembler^3 allows assembling 95.2% of models

To determine what percentage of 2D cutting plans on the Internet can be reconstructed using assembler^3, we found models online and attempted to reconstruct these. We selected the models by (1) searching *things* for "lasercut" "laser-cut" and "laser cut" on thingiverse and grabCAD, to ensure the models fabricate, we filtered models that have "makes". (2) We then excluded models that were single-part, or that contained features not yet supported by assembler^3: living hinges, moving mechanisms, stacked/glued/bolted plates and joints that connect more than one other joint. This left us with about 40% of models. (3) We randomly selected 105 models of this collection.

Result: assembler[3] managed to reconstruct 100/105 models (95.2% coverage) they are presented in Figure 118. Models varied in thickness from 1mm to 12mm and used a wide variety of construction techniques, e.g., skeletons, grids of cross joints, outside finger joints and more.

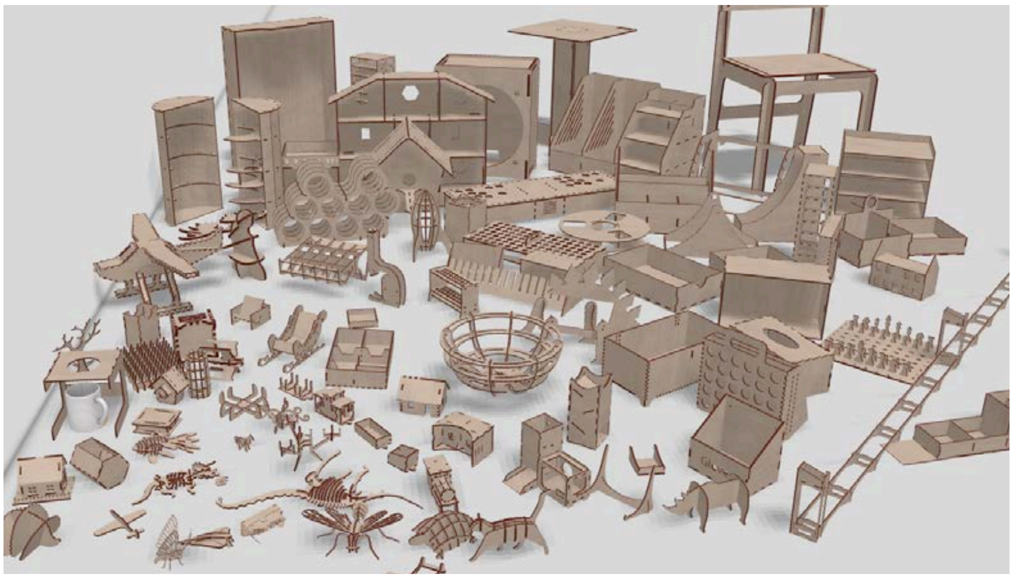


Figure 118: Models used for technical evaluation.

Out of the five models that failed, four failed because their finger joints came in at odd angles. See the model shown in Figure 119b: all vertical plates of this lighthouse are tilted inwards by 10 degrees, assembler[3] could handle that if they did not *also* connect sideways at a 45 degree angle (see Figure 119c). We plan to extend assembler[3] with a more advanced constraint solver to also handle such cases.

Figure 119a shows the last model that could not be assembled, it consisted of cross joints that assembled into t-joints. They come in sideways and then lock in place. Assembler[3] does not check for this combination. Based on these observations, we are planning on extending our joint matching logic in future versions of assembler[3].

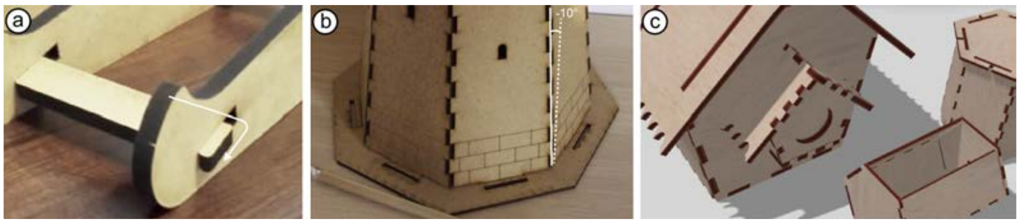


Figure 119: (a) This model has cross joints that assemble into a mortise-tenon joint, assembler[3] fails to pair these up. (b) when all plates are at a non-straight angle with each other, assembler[3] cannot reconstruct the model. (c) luckily, most of the models with plates with non-straight angles still provide sufficient constraints to be reconstructed.

Success rate of the individual steps of our algorithm

To validate the accuracy of each of steps in the algorithm, we compared the outcome of each step to their "ground truth" using 10 selected models shown in Figure 120. We hand-annotated all features in these models and compared it to the data collected in each of the steps of the algorithm of assembler[3].

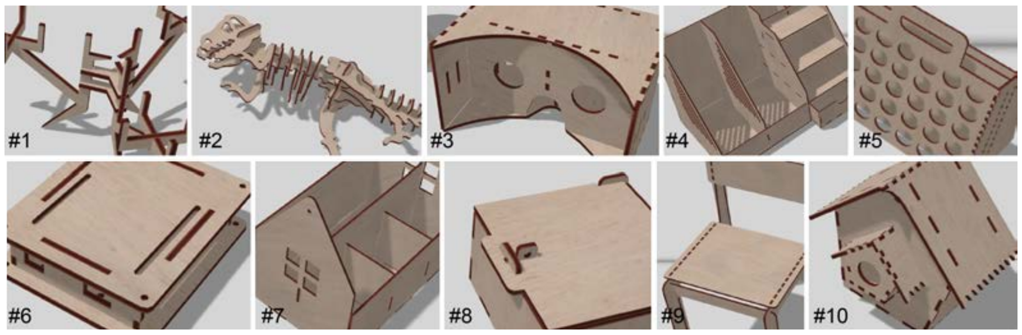


Figure 120: Models used to validate the accuracy of the steps of the algorithm.

Table 2 shows the accuracies the different steps achieve. The overall success rate of 97% of the manual assembly is the key result, this is the percentage of detected features which are required to reconstruct 3D models.

Step	False positives	Success rate
Step 1: plate detection	0	100%
Step 2: joint detection	61	98% (352/356)
Step 3: material detection	-	99%
Overall success rate = product of above		**97%**

Table 2: Success rate of algorithm steps.

Step 1, plate detection worked flawless for the tested models.

Step 2, joint detection achieved 98% of true positives. It still detected 61 false positives, which results in a bigger search space than needed, but models still assemble. The 2% of false negative joints would never show up as suggestions and thus have potential to result in models that cannot be assembled. Both models, #3 and #8, which suffered from this actually *did* assemble nonetheless, because of constraints imposed by other joints.

Step 3, assembler[3] was 98.99% accurate in detecting the material thickness. The two erroneous models were off by 0.1mm, which was caused by rounding errors. The reconstructed models still worked properly (we exported and fabricated both models).

To verify the accuracy of the default placements of plates, we assembled each of the models and counted how often we needed to override initial placements with the floating menu item. Assembler^3 got the first placement right in 78.6% of the clicks. When the initial orientation was wrong, it took on average 1.7 clicks to get to the correct orientation.

Performance of the algorithm

To measure the performance of assembler^3, we used the same 10 models as used to verify the accuracy. We profiled each step in the algorithm to measure the performance of (1) parsing the SVG and generating the hash, (2) suggest matching plates when the user clicks a joint (3) and assembling plates after the user selected a target plate. We ran the performance test on an Intel Core i5-8400 CPU @ 2.80 GHz (6-core) 16GB RAM. We repeated each test 1000 times to account for typical variations in performance.

Model	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**	**#10**
# of plates	8	29	8	16	5	6	8	6	12	13
# of joints	18	55	35	66	10	24	35	20	20	52
parse SVG	12.54	525.96	34.22	65.30	174.77	49.33	27.49	12.83	87.65	77.51
	(12.05)	(101.3)	(72.25)	(13.60)	(223.5)	(11.62)	(47.78)	(6.96)	(187.9)	(9.45)
match	1.72	6.65	5.13	5.59	6.59	2.05	4.60	2.47	11.59	11.41
	(3.00)	(10.54)	(16.66)	(11.90)	(23.98)	(4.42)	(10.07)	(2.39)	(7.78)	(27.37)
assemble	3.12	3.43	4.88	4.21	10.50	3.50	3.99	3.19	9.50	6.09
	(1.69)	(2.79)	(8.26)	(5.91)	(50.88)	(3.36)	(5.44)	(2.97)	(9.42)	(16.22)

Table 3: Average performance of assembler3 when assembling each model 1000 times. Times are in ms (stndev in parentheses)

The steps where the user interacts with the model are all efficient (average of 12ms for every model), the parsing of the SVG initially takes 107ms on average, this only occurs when importing the model initially. Model #2 (dinosaur) took longer (526ms) to parse. The reason is the linearization breaks down curved lines of the dinosaur into a large amount of small line segments.

5.1.5 user study: assembler^3 is 10x faster than the traditional workflow

To verify the claim that assembler^3 is easier and faster than the traditional mental reconstruction workflow, we ran a user study in which participants manually reconstructed the virtual reality headset from Figure 124 and stretched the model to accommodate for far-sightedness.

Task

Participants' task was to modify the VR headset as shown in Figure 107, i.e., stretch the distance between the lenses and the screen in order to accommodate a lens with a bigger focal range. To illustrate the objective, we provided participants with a picture of the *before* and *after* configuration of the headset (Figure 107). Participants were allowed to have a look at these pictures any time during the experiment.

Interface conditions

Participants completed the task in two interface (within subjects) conditions in counter-balanced order.

In the *2D condition*, participants modified the 2D cutting plan using a 2D editing software (gravit.io, runs in a web browser). For training, participants were shown a 4min demo video that demonstrated the relevant editing functionality at the example of extending a box.

In the *assembler^3 condition*, participants modified the 2D cutting plan by converting the headset's 2D cutting plan to 3D using assembler^3, modifying the model in kyub, and exporting it back to a new 2D cutting plan. For training, participants were shown a 1 min demo video showing the process of assembling a box and stretching it using assembler^3, Figure 121 shows a shot of these videos.

In both conditions, participants were allowed to review the demo video until they felt they got the workflow. They were also allowed to revisit the video during the actual task.

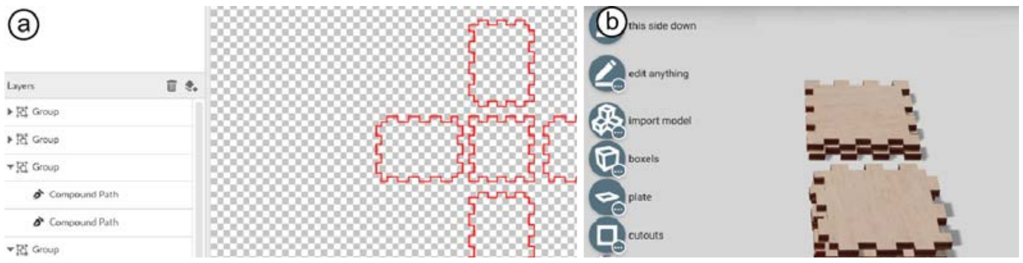


Figure 121: (a) In the baseline condition, users were shown how to modify a boxel in gravit.io, (b) they were shown a video of the same workflow using assembler^3.

Participants

We recruited 13 participants (2f/11m, average age of 21 years) from our institution.

Procedure

We presented both interface conditions to participants in counter-balanced order. In each condition, participants were given up to 30 minutes to complete the task. If they felt they could not complete the task, they were allowed to abort earlier by notifying the experimenter.

Results

Figure 122 summarizes the task times and error rates of the 13 individual participants.

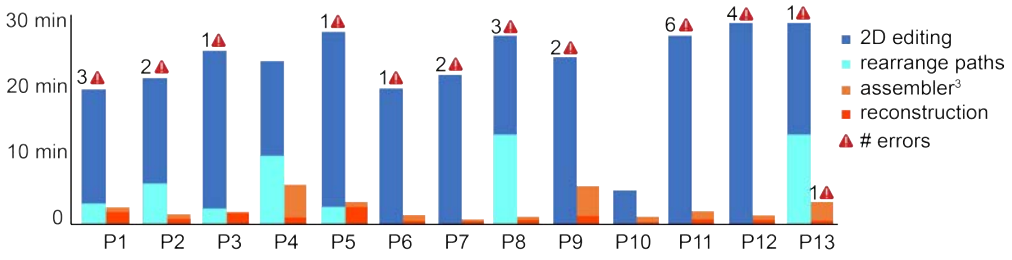


Figure 122: results of the experiment.

Participants performed 10.0x faster when modifying the model using assembler³ (on average 2:22 min vs 24:45 min in 2D condition). This confirms our main hypothesis.

As shown in the diagram, the majority of the time in the assembler³ condition went into moving and scaling the model, 51s on average was used to reconstruct the model. In the 2D condition, users spent on average 4:51 minutes to lay out the model before editing any plate. P5, P12 and P13 did not manage to complete the 2D condition in the given 30 minutes. 11/13 of participants had errors in the model they modified in 2D (on average 2 errors per participant), which would cause the model to not fabricate (typical errors were joints not fitting, cutouts that were moved or stretched t-joints). In the assembler³ condition one participant had an error (the nose piece was flipped, which the participant didn't notice—the model would still assemble and fabricate though).

Participants rated the assembler³ condition as easy (a median of 2/7 on "assembling and modifying the model was easy/hard") while they rated a median of 6/7 for the 2D condition. One participant, P10, considered the 2D workflow easier than the workflow with assembler³, mostly because it took P10 a while to figure out how to use the stretch tool after completing the assembly.

In the 2D condition, 7/13 participants re-layouted paths (see Figure 123) so as to better reflect the 3D nature of the assembly, as shown in Figure 122; they spent on average 7:08 minutes to do so. Those who spent time laying out the model made 1.5 errors on average, vs 2.5 for those who didn't change the layout.

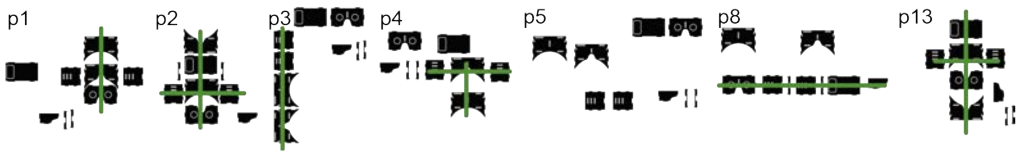


Figure 123: 2D layouts as used by our participants. Green lines indicate plates that are laid out so that their matching joints line up, an indicator that they leverage the space in the 2D editor to help them reconstruct the model in their heads. P5 also spent time laying out paths but grouped by similarity in shape instead of matching joints.

We asked participants about their experience using both tools. P7 commented that "the 3D editor essentially builds the model itself and is easy to change". P8 said "it was a huge relief to do this in 3D" after having struggled for a long time in 2D. P4 mentioned that "the 2D software itself was great but that it's really hard to find out how to connect the plates". On an interesting sidenote, P10, who was extremely fast in 2D mentioned "I prefer to edit in 2D because it helps me learn about the model". This is both a weakness and a strength of assembler^3 as it takes the burden of learning about the model away from the user.

Discussion

We conclude that (1) assembler^3 enabled 11/13 participants to make modifications to the model who were not able to do this without and (2) assembler^3 achieves a speed-up of 10x and (3) the workflow with assembler^3 results in 26x less errors.

5.2 AUTOASSEMBLER: AUTOMATIC 3D RECONSTRUCTION

As shown in the previous section, the pipeline of assembler^3 is performant and effective in reconstructing a broad variety of structures. The weakest link of the algorithm obviously is the semi-automatic assembly step at the end of the process. It comes at a cost of time for the user and requires technical understanding of the underlying models, as users need to puzzle together the individual pieces and thus know how to assemble them. A task which is trivial for original designers of these models, but when customizing a model found online, this can be arbitrary hard.

In many cases, the knowledge the assembler[3] algorithm derives from the 2D cutting plan, proves to be sufficient to reconstruct models automatically. We therefore built an extension of assembler[3], *autoAssembler* [100], since the search space of combining plates is exponential in the number of joints, exhaustive search is impractical for any non-trivial model. AutoAssembler thus pursues only the most "promising" subset of candidates (aka beam search) as illustrated in Figure 124. It considers candidates as promising if they (1) contain no intersecting plates, (2) fit into a small bounding box, (3) use plates whose joints fit together well, (4) the plates do not add many unpaired joints, (5) make use of constraints posed by other plates, and (6) conform to symmetry axes of the plates. The algorithm presents the resulting best candidate assembly to users when they import a model, so they can bypass the manual process and directly modify the 3D assembly if that assembly looks good.

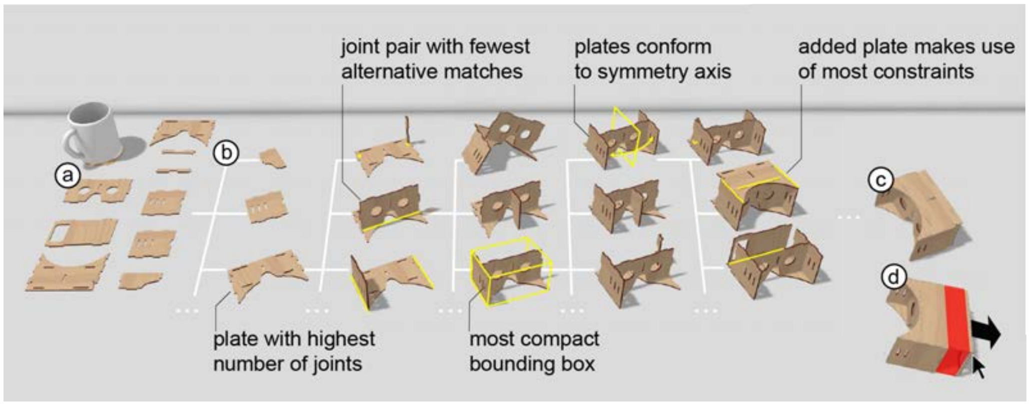

joint pair with fewest alternative matches
plates conform to symmetry axis
added plate makes use of most constraints
a
b
c
d
plate with highest number of joints
most compact bounding box

Figure 124: *AutoAssembler* converts 2D cutting plans to 3D models by (a) importing 2D cutting plans and (b) beam-searching the space of ways to assemble the plates. *AutoAssembler* prefers candidates that (1) have no intersecting plates, (2) fit into a small bounding box, (3) use plates whose joints fit together well, (4) do not add many unpaired joints, (5) make use of constraints posed by other plates, and (6) conform to symmetry axes of the plates. (c) This allows users to load the model into a 3D editor (*kyub* [14]), (d) where they can now apply parametric changes.

5.2.1 the autoAssembler algorithm

When assembling a model, automatically or by hand, one explores a space of possible solutions that is factorial in the number of joints. Even if we reduced the search space by (1) limiting our search to joints that fit, (2) eliminating orientations that lead to intersections between the new plate and what has been assembled already, and (3) adding a method that looks up matching joints for the joint at hand in constant time, the search space remains too large for exhaustive search (see Figure 125). We achieve (3) using the *joint hash* from assembler[3], which achieves this by storing geometric profiles of joints in a hash table—the joint and its counterpart share that profile, when looking up the profile of a joint in the hash table, it returns its counterpart as a collision in the table in constant time.

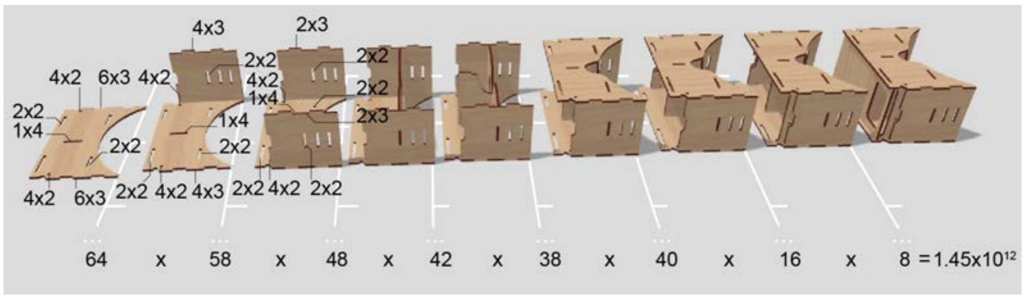


Figure 125: The search space for the simple VR headset consisting of 9 parts and 33 joints after limiting our search to joints that fit and only exploring orientations of plates that do not lead to an immediate collision. (Labels denote the number of joints that fit at a given position x the number of orientations they fit in).

AutoAssembler therefore limits the search to the more promising candidates at each stage (*beam search* [18]). To this end, autoAssembler starts with an empty model and recursively tries to add one plate each time. The time-complexity of beam search is O(*dk*) where depth *d* is the number of plates and *k* is the *beam width* (the number of candidates autoAssembler picks to generate children for, at each stage) multiplied by the maximum fanout at each stage. The fanout in principle is proportional to the depth. Worst case complexity thus is quadratic, however the joint hash table mentioned before reduces the fanout to the joints with the same signature. If all joints have a unique counterpart (ideal case) the complexity is linear, in practice the complexity sits between these bounds. In our technical evaluation, a *beam width* of 4 proved sufficient for achieving the aforementioned success rate (79% + 18% = 97%), allowing for overall very fast execution (median of 0.30s).

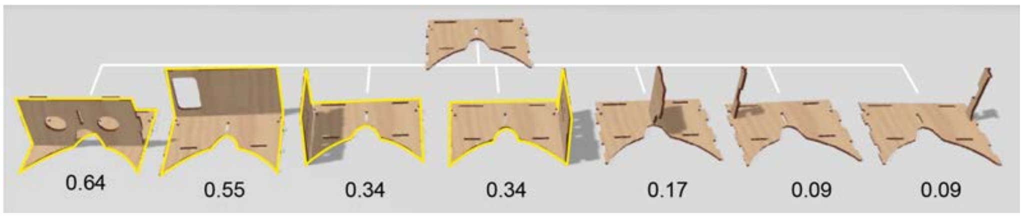


Figure 126: The first stage of search for a VR headset. AutoAssembler picks the initial plate with the most joints, uses the joint-hash to find what plates fit into the open joints, scores the candidates (labeled above) and generates new children from the best candidates.

As shown in Figure 126, performing beam search, autoAssembler selects the four highest scoring candidates to generate the candidates for the next round, detailed in pseudo code in Algorithm 5.

Algorithm 5: Find best candidate

Input: starting plate, detected plates and joints in 2D cutting plan (using the assembler[3] algorithm)
Output: best assembled model

Internal data structures: *candidates* are assemblies of one or more plates, their children are the same assembly with one additional plate. Empirically determined maximum beam width MAX_BEAM_WIDTH=4

```
// These candidates have two plates, (see Figure 126 for examples)
currentCandidates = children of the candidate, which only contains the starting plate
// Based on heuristics in the next section, score each candidate
score(currentCandidates)
while there is at least one currentCandidate and not all plates are used {
            // AutoAssembler groups candidates that use the same plates, but not using the same joint
or
        orientation, see heuristic "minimize candidates that are highly similar"
        groups = group currentCandidates together, which have the same "connection pattern"
        currentCandidates = Highest scoring candidate from each group
        // If there are more states than the maximum beam width, limit them based on score
        currentCandidates = currentCandidates limited to MAX_BEAM_WIDTH
        // Generate children by adding a plate to the current candidates in different orientations
        currentCandidates = children of currentCandidates
        // Based on heuristics in the next section, score each candidate
        score(currentCandidates)
        }
return currentCanddiate with highest score
```

As part of the search process, as illustrated by Figure 127, autoAssembler eliminates duplicate candidates. It achieves this by storing previously visited candidates (*memoization*) in a hash.

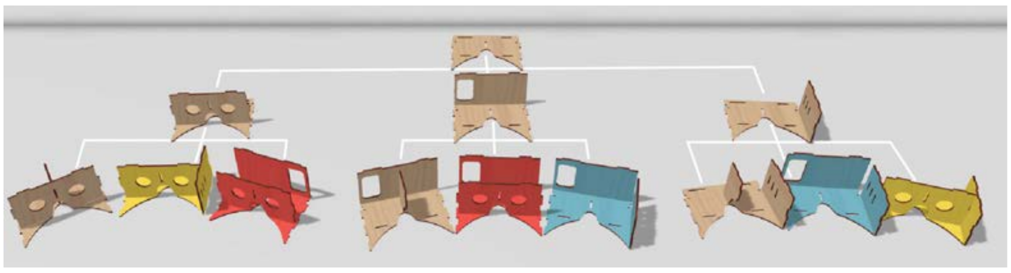


Figure 127: autoAssembler encounters a candidate model more than once (the colored candidates), autoAssembler drops the redundant states (Memoization), by hashing visited candidates. (Here shown with three candidates each for visual clarity).

How autoAssembler picks promising candidates: The heuristic function

The main contribution of autoAssembler is the specific way it selects the candidates it pursues, i.e., how it assesses the potential of each candidate (its *heuristics function* [18]). It computes a weighed sum to prefer candidates (1) that have no intersecting plates, (2) that fit into a small bounding box, (3) that use joints that can be unambiguously matched, (4) that do not add a large number of unmatched joints, (5) that make use of constraints posed by other plates, and (6) that conform to symmetry axes of the plates. We developed these heuristics based on our observation of common patterns in laser-cut models and by manually evaluating candidates in our engineering team. As autoAssembler calculates a score for all candidates at every stage, the implicit objective for these heuristics is that they are efficient to compute.

AutoAssembler aggregates six heuristics as a weighted sum. Two additional heuristics (deduplicating symmetric/similar plates and minimizing highly similar candidates) are procedural in nature as they operate on the *stage* (all "current candidates") rather than scoring individual candidates. We determined the optimal weights of the individual heuristics using hyperparameter optimization (see "technical evaluation" for details):

Table 4: parameter optimization for the heuristic function

parameter	**weight**
compactness of candidates	0.07
intersections between plates	0.63
ambiguity of the joints that are completed	0.88
minimizing the number of unmatched joints	0.95
make use of constraints posed by other plates	0.58
conform to symmetry axes of the plates	0.83
minimize candidates that are highly similar	n/a
deduplicating symmetric plates and similar plates	n/a

1. Give preference to compact candidates: parts that "stick out" of a model break off easily. Since designers generally prefer sturdy designs, laser-cut models tend to be "compact", i.e., fit into a comparably small encompassing volume. AutoAssembler therefore is designed to prefer candidates that fit into smaller bounding boxes. The red plate in Figure 128 for example, could be assembled as shown in (a), but that increases the spanned volume (calculated using the axis-aligned bounding box as this is the cheapest metric to compute). AutoAssembler calculates this "compactness" using the metric: $\frac{(surface\ area)^{1.5}}{volume}$, as proposed by Parker et al. [89]. The compactness score of Figure 128b is much higher (the surface area remains the same, but the denominator is much smaller), so autoAssembler gives this candidate a higher score. Note that the weight of this heuristic is very small (0.07) and thus mostly serves as a tiebreaker for the other heuristics.

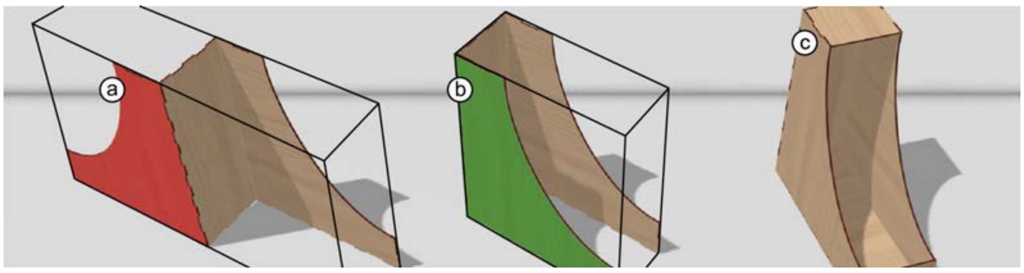


Figure 128: (a) Adding the red plate gives it a much larger bounding box, autoAssembler thus gives (b) this candidate the higher score. (c) The resulting magazine holder.

2. Avoid intersections between plates: laser-cut plates in a model must not intersect. Computing intersections between plates is an expensive operation as it requires comparing every outline feature of the plate to the already existing candidate. To achieve this efficiently, autoAssembler compares the bounding box of the newly added plate to plates already present in the current candidate. In the train wagon of Figure 129, autoAssembler initially prefers to put the wheel mount up because of the compactness metric, however that causes an intersection, which forces autoAssembler to assemble this part in another orientation. Avoiding intersections is not a hard constraint, because the relatively cheap method of computing intersections comes at a cost of accuracy.

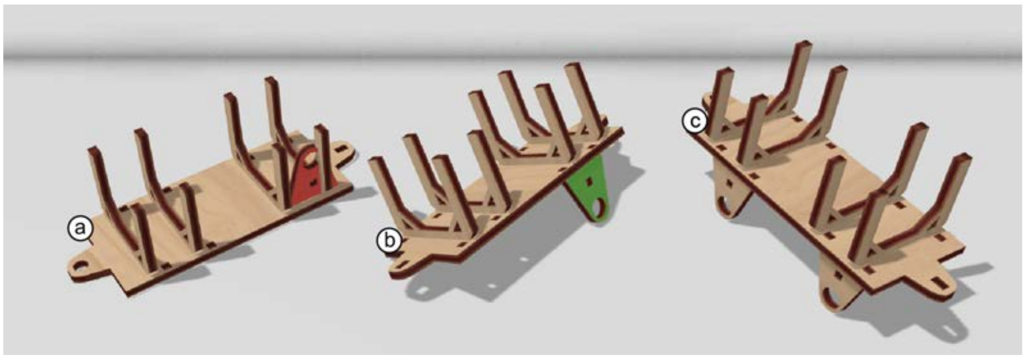


Figure 129: (a) The compactness heuristic suggested mounting this bearing (red) on top of this train wagon, but here it intersects with plates mounted on top of the wagon, causing autoAssembler (b) to flip the bearing plate to its correct position. (c) the same heuristic fixes the other bearings too.

3. Give preference to unambiguous joints: AutoAssembler delays inserting plates that can be mounted in many different ways as long as possible, so as to await additional plates to introduce additional constraints that can help make the decision. AutoAssembler achieves this by giving preference to joints that have few, or ideally only a single matching partner joint. More specifically, autoAssembler assigns a probability to pairs of joints in the joint hash as proposed by assembler[3] and it tries to maximize the ratio between the assigned probability and the sum of the probabilities of all other ways of matching up this joint. When assembling the dice tower shown in Figure 130, for example, starting with the ambiguous single finger joint of the red plate creates many different opportunities for mounting the top plate. (b) AutoAssembler instead first assembles the unambiguous and long finger joints, which then pose constraints on the red plate of Figure 130a, reducing overall ambiguity.

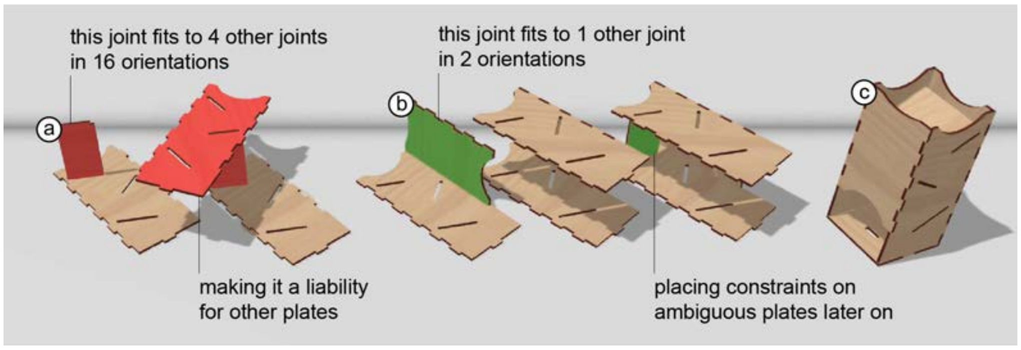

this joint fits to 4 other joints in 16 orientations
a
making it a liability for other plates
this joint fits to 1 other joint in 2 orientations
b
placing constraints on ambiguous plates later on
c

Figure 130: (a) Assembling this ambiguous joint early on forms little or no constraints on other plates, as a result the top plate here can be assembled in many different ways (b) autoAssembler prefers to greedily connect plates with high probabilities. This adds constraints for other plates, (c) to eventually make this dice tower.

4. Minimize the number of unmatched joints: the size of the search space at every candidate correlates with the amount of unmatched (unused) joints. AutoAssembler prioritizes plates that add the fewest incomplete joints. This works because it started out with the plate having the most joints, otherwise autoAssembler would paint itself into the corner (e.g., start with a plate with one joint, then close that joint without opening new ones–done). In the example shown in Figure 131, for example, autoAssembler therefore does not add (a) the side plate that brings in multiple new joints but runs with (b) the middle divider, which only adds one new joint.

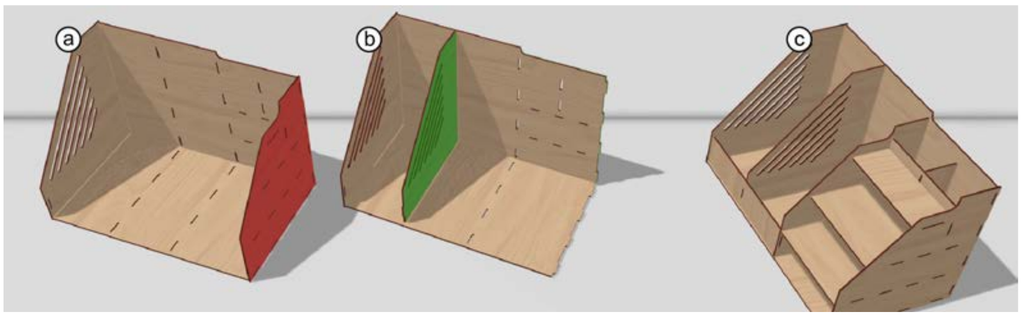


Figure 131: AutoAssembler prioritizes completing joints first as this reduces the search space: (a) inserting the side plate (red) would add four incomplete joints to the model. (b) AutoAssembler therefore rather adds this "divider" plate, which only adds one unmatched joint. (c) leading to this desktop organizer.

5. Make use of constraints posed by other plates: AutoAssembler prioritizes inserting plates whose placement is supported by multiple plates/joints already in the model. It tries to maximize the number of completed joints by adding a plate. This avoids situations as shown in Figure 132, where (a) the nosepiece of this VR headset is under-constrained: it can be assembled in different orientations that all seem equally good according to the other metrics. (b) AutoAssembler thus prioritizes assembling the front plate first, which completes three joints at once and then later (c) adds the nosepiece as the front plate imposes additional constraints on that plate.

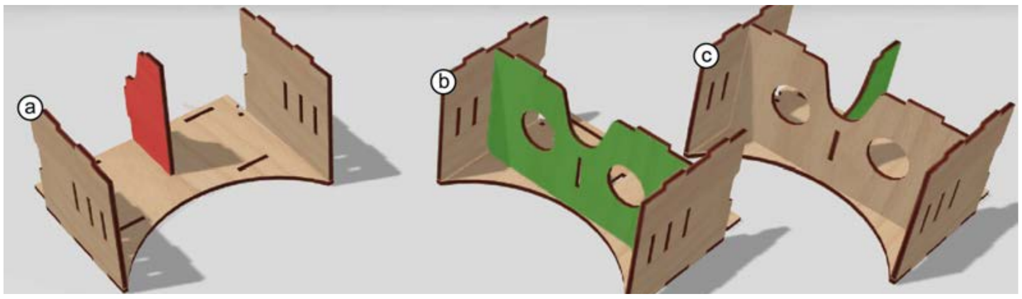


Figure 132: (a) This candidate offers too few constraints to orient the red plate correctly. (b) AutoAssembler therefore prioritizes this plate, which completes three joints. (c) This adds constraints that come in handy when eventually inserting the middle piece.

6. Minimize candidates that are highly similar: as illustrated by Figure 133, if autoAssembler encounters multiple candidates that differ only by one or more plates being flipped, it drops all but the highest scoring one, so as to make room for candidates consisting of a different subset of plates. As shown in Algorithm 5, candidates are filtered by "connection pattern". This is a high-level data structure that describes what plates are connected, using which joints. This prevents AutoAssembler from only looking at similar structures with a flipped plate.

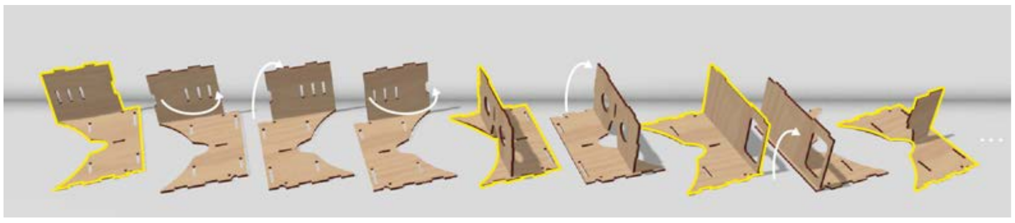


Figure 133: AutoAssembler picks the best four candidates of this stage. To avoid picking the first four candidates which are almost the same, it skips candidates that share the same connection pattern with a candidate that is already picked. Resulting in the four candidates highlighted in yellow.

The algorithm, as described above is functional, but performs poorly on models containing symmetries, such as the models shown in Figure 134. On such models the search space is cluttered with results that are the same, but contain different plate connectivity, resulting in problems similar to the ones in Figure 133. This unnecessarily blows up the search space and deprioritizes asymmetrical plates in the assembly that end up defining the structure. This is problematic, as 3D models designed for laser cutting are commonly symmetrical in nature. Out of the benchmark of assembler[3], for example, 81/100 have reflective symmetries, and 11/100 have rotational symmetries.

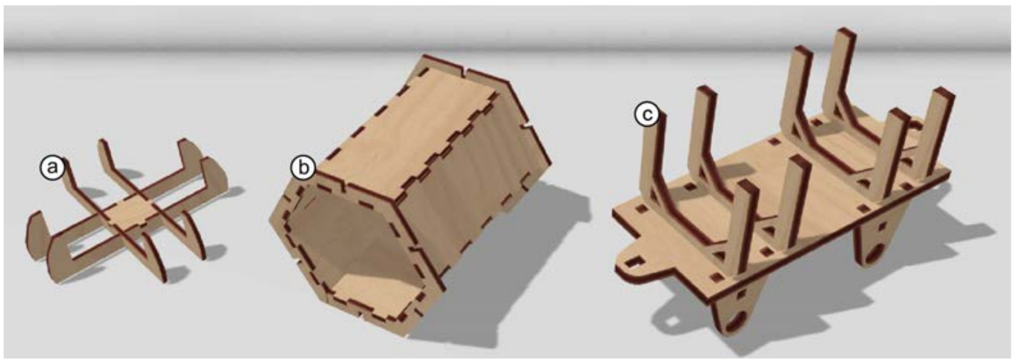


Figure 134: Examples of symmetric laser-cut models from assembler[3] (a) double reflectional symmetry, (b) 6-point rotational symmetry, (c) double reflectional symmetry (and multiple uses of same plate).

We propose two extensions of the algorithm based on symmetries: prioritizing symmetric assembly of plates, and deduplicating plates and orientations when possible.

7. Favor symmetric candidates: AutoAssembler prefers symmetric assemblies over asymmetric ones. As shown in Figure 135b, when autoAssembler adds a plate to a symmetrical plate, it verifies whether there is a similar plate elsewhere in the assembly and if so, it increases the score of a placement that implements symmetry.

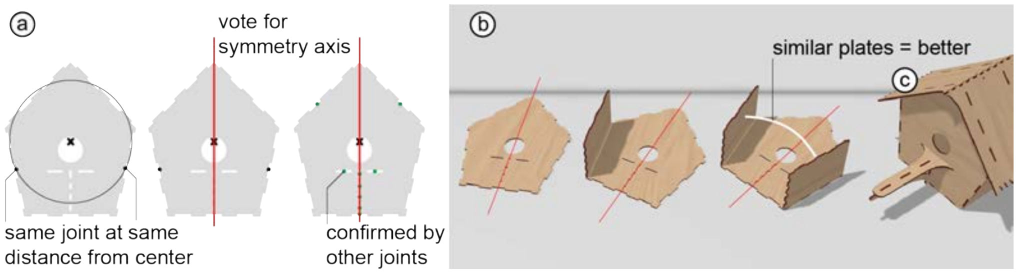


Figure 135: (a) AutoAssembler detects symmetries by having pairs of joints vote for symmetry axes (b) autoAssembler uses the information to prefer similar plates connected to joints on opposite sides of the symmetry axis. (c) resulting in this birdhouse.

As shown in Figure 135a, autoAssembler detects symmetries in three steps: (1) it starts by looking for joints with a similar profile using the joint hash table, at the same distance to the center of the plate, (2) it constructs the symmetry axis this pair of joints conforms to, and (3) then verifies that proposed symmetry axis with the other joints on the plate, similar to Mitra et al. [78].

8. Deduplicating symmetric plates and similar plates: Symmetric plates blow up the search space unnecessarily: When encountering a symmetric plate, such as the one shown in Figure 136, the basic version of autoAssembler considers inserting it in all possible orientations, leading to a much bigger search space with a lot of candidates that turn out to be geometrically identical.

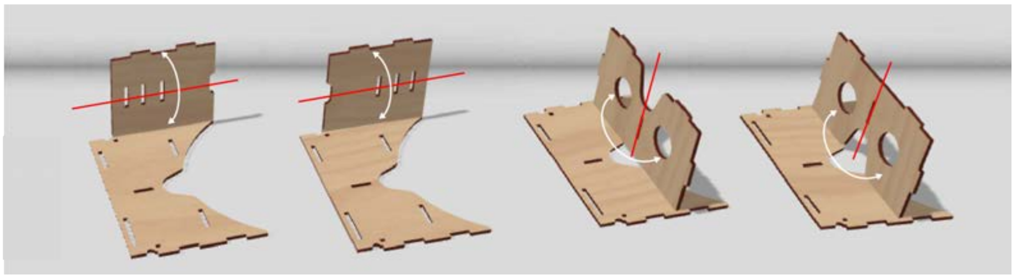


Figure 136: Because of the horizontal symmetry axis, only these two orientations of the side plate produce a unique state. The same for this in-plate symmetry in the vertical orientation.

A similar issue is caused by multiple identical plates, such as the ones shown in Figure 137. Again, the basic version of autoAssembler considers inserting each copy of that plate separately, thereby blowing up the search space and increasing the risk of beam search dropping relevant models, as the algorithm is instead pursuing multiple essentially identical models.

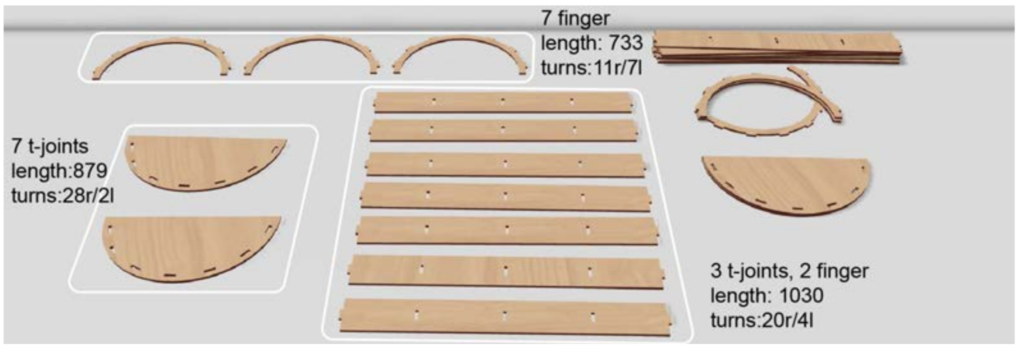


Figure 137: Clustering by similarity (a) autoAssembler characterizes each input plate by three easy-to-compute metrics (b) For this barrel model, autoAssembler considers 3 types of plates with 19 joints as opposed to 12 types of plates with 58 joints.

AutoAssembler determines that two plates are identical by comparing their outlines using efficient-to-compute characteristics: the number and the types of joints, the length of the outline, and the number of left/right turns along each outline.

For hand-drawn models or models subject to rounding errors these metrics may differ by some epsilon. To overcome these imprecisions, autoAssembler uses a density-based clustering algorithm (DBSCAN [36]), which allows autoAssembler to cluster similar plates, without knowing in advance how many clusters to look for.

For the model shown in Figure 137, for example, autoAssembler reduces this model from 12 types of plates featuring 58 joints down to 3 types of plates featuring 19 joints, which heavily reduces the search space.

Favoring symmetry and similarity detection allows autoAssembler to correctly reconstruct the six models shown in Figure 138, thereby increasing autoAssembler's success rate. It also improves the algorithm's performance by a factor of 1.5.

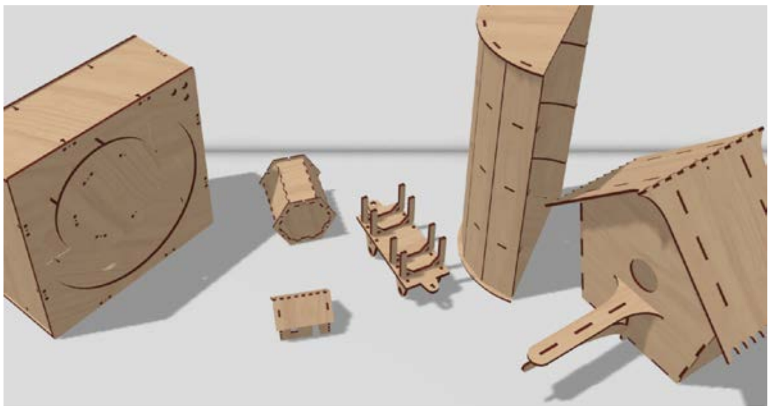


Figure 138: Six models from the test set that assemble correctly in autoAssembler because of the symmetry heuristics.

Manual disambiguation

Some models do not contain sufficient information to automatically complete the model. The triceratops shown in Figure 139, for example, would require *domain knowledge* of the anatomy of dinosaurs to tell how to sort the ribs, or at best a visual reference for what to assemble (as a child may have puzzling the dinosaur together). AutoAssembler does not have this domain knowledge and consequently it precisely fails to assemble models of this type—this is a limitation of the system and the reason we exclude from our analysis these particular type of decorative models, which are based on cross joints alone.

Models that do not solely rely on cross joints, however, tend to have only a small number of such ambiguities and these generally do not derail autoAssembler. The remaining 14 models that autoAssembler did not automatically assemble, had a few plates that were not captured by the general heuristics of the algorithm. We address these by complementing autoAssembler with two manual tools that allow disambiguating these cases:

1. Clicking a plate using the "reorient plate" tool reorients the clicked plate by forcing autoAssembler to re-evaluate its orientation. Users keep clicking until satisfied with the plate's orientation.

2. Clicking a plate using the "swap plate" tool swaps a plate with another selected plate if their joints match. Users click on one of the plates, and then click on the other plate to swap them.

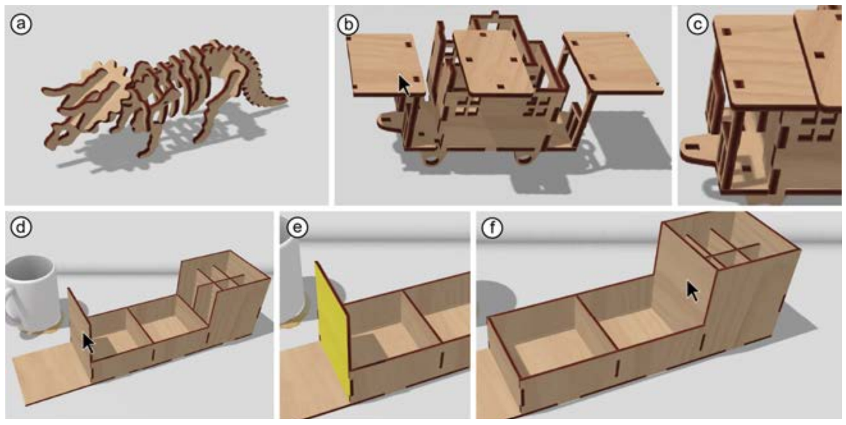


Figure 139: (a) Models such as this dinosaur require domain knowledge, placing them outside the scope of automatic assembly. (b) this train wagon has misassembled plates after automatic assembly, (c) with the reorient tool this is quick to fix. (d) Two plates are swapped in this organizer. (e) The "swap-plate" tool lets users select one of the plates, and (f) by clicking the other one they swap if they share common joints.

In our technical evaluation, fully automated use of autoAssembler assembled 79% of all models correctly. The "re-orient" and "swap" tools allowed fixing an additional 12 of 14 models using 1-4 such clicks, resulting in a 97% success rate.

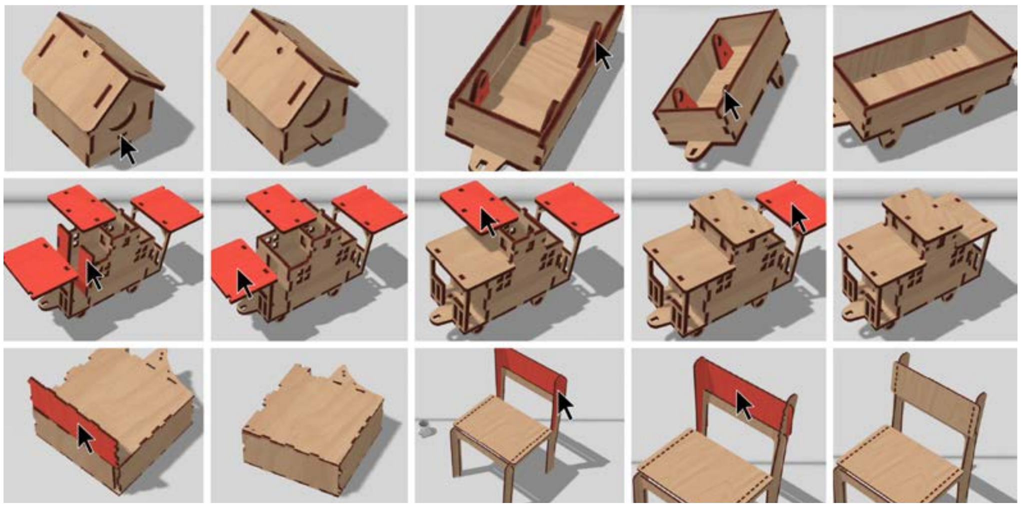


Figure 140: Click sequences of the disambiguate tool. Users click a poorly assembled plate, which autoAssembler then reconsiders. Here are five models that all were fixed by 1-4 manual disambiguation overrides (2.7 on average).

User interface

Figure 141 shows the interface of autoAssembler in kyub. When users import an SVG file, the dialog window shows a live preview of autoAssembler assembling the model. AutoAssembler completes the import after a median of 0.30s, (see section 5.2.3 Technical Evaluation). As shown in Figure 141a on the left, users still have the option to import the individual plates if the result does not look satisfactory. In that case they revert to the assembler[3] tools for the reconstruction process as demonstrated before.

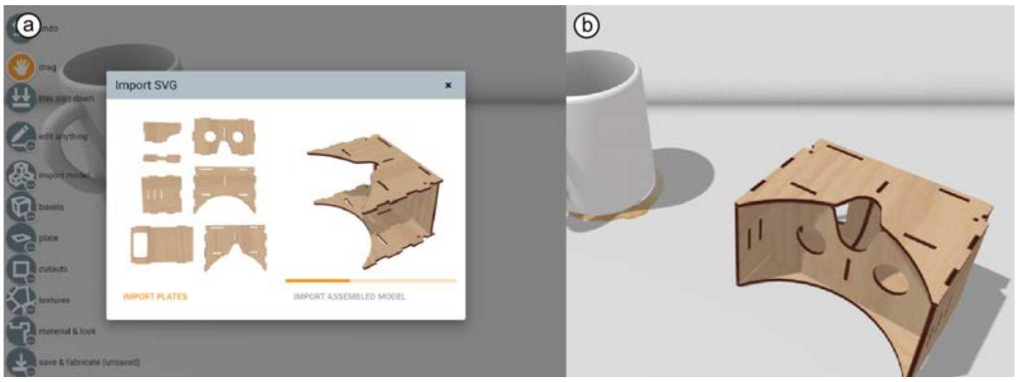


Figure 141: Importing an SVG model automatically into kyub.

5.2.2 Tuning the algorithm

We ran a series of tests to optimize key parameters of the algorithm. (1) Weights for the heuristics (2) determine minimum beam widths, and (3) what plate to start out with.

Test set

We created the test set starting with the test set of the *assembler*[3] project, from which we extracted the 34 "planar intersection" models, as discussed above in the "manual disambiguation" sub-section. When models consisted of multiple assemblies, we split these into separate files as autoAssembler expects one assembly per model. They can be loaded into the same kyub scene though but through 2 import sessions.

We measured the success rate by taking the ground truth models that were manually assembled in assembler[3] as a reference. We automatically verify our test runs by comparing the distance and angle between plates to these ground truth models. We considered a model to be a success only when all the dimensions matched perfectly (e.g., there is no 50% successful assembly).

Procedure

We measured success rate (percentage of models that assembled correctly) and the run time and repeated every measurement 10 times to compensate for performance glitches and any potential delays confounding our measure due to background activities on the machine (MacBook Air 2020 1.2GHz Quad Core Intel Core i7).

Composition of the heuristic function

To determine the right weights for the parameters of the heuristic function, we ran a hyperparameter optimization algorithm using the tree pazen estimator called ATPE, proposed by Wen et al. [128]. We trained the algorithm by feeding it the parameters of random runs on the benchmark, and the corresponding candidates. The candidates are labeled automatically by comparing it to our ground truth distance matrix. After assessing 1000 candidates the algorithm converges, we found the optimal parameters presented in Table 4.

The beam width: from 4 on, autoAssembler achieves maximum success rate

The beam width is the number of candidates autoAssembler selects at every stage after sorting the states. To achieve the optimal success rate and performance trade-off, we ran the benchmark with increasing beam widths until the success rate not increased further. We also did a run with a beam width of 1, which is equivalent to best first search to see if the heuristic function alone (without beam search) would yield sufficiently good results. Results are shown in Figure 142 below.

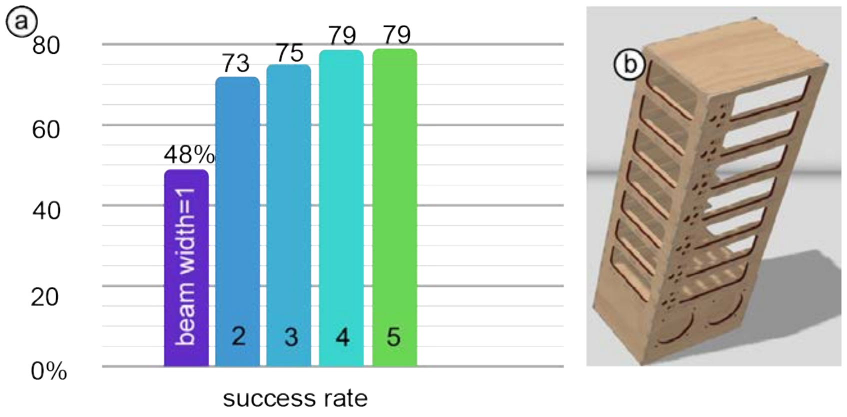


Figure 142: (a) Success rate of autoAssembler on our benchmark while varying the beam width. (b) This model still makes an improvement at a beam width of 8, but the associated performance loss is not worth it.

As shown in Figure 142, after a beam width of 4, the success rate stabilizes. To verify whether if the success rate had reached an upper bound, we ran the remaining 18% of models with a beam width of 10 as well. Apart from the raspberry-pi rack shown in Figure 142b, which after some more testing improved at a beam width of 8, there was no more progress. We also see that with a greedy best first search, we could still achieve a success rate of 48%, which indicates that the heuristic function alone is rather good at picking the right option, but in many cases, we do benefit from searching more alternatives.

The median performance per model in each run was 0.11 (beam width =1), 0.22 (2), 0.22 (3), 0.30 (4), 0.44 (5), increasing the beam width scales the performance roughly linearly. Therefore, doubling the beam width (and thus the run time) from 4 to 8 is not worth it, only to save a single model in our test set.

Start with the plate with most joints

To know what role the starting plate plays in the success rate of the algorithm, we ran the benchmark with different starting plates: (1) the plate with the most joints, as this puts the most constraints on the assembly (2) the biggest plate as this would define most of the shape, and (3) a random plate (the plate that contains the first path in the SVG), as a baseline. Figure 143 shows these strategies at the example of a test tube rack.

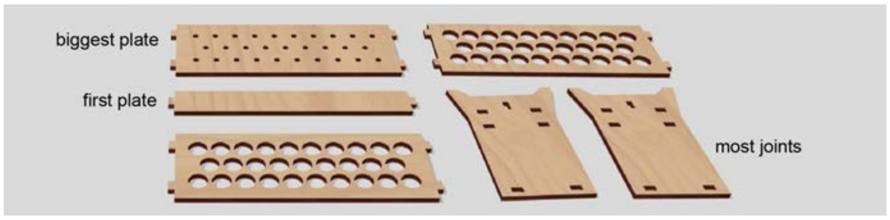


Figure 143: Different starting plate metrics for the test-tube model.

The results show that the best solution is to start with the plate with most joints. To see if there are better options for the models that fail, we ran detailed tests with those models where we started out every plate. Some of the broken models get closer to success by picking a different starting plate, but none were "fixed" by doing so. We thus stick to the plate with most joints as this is cheap to compute.

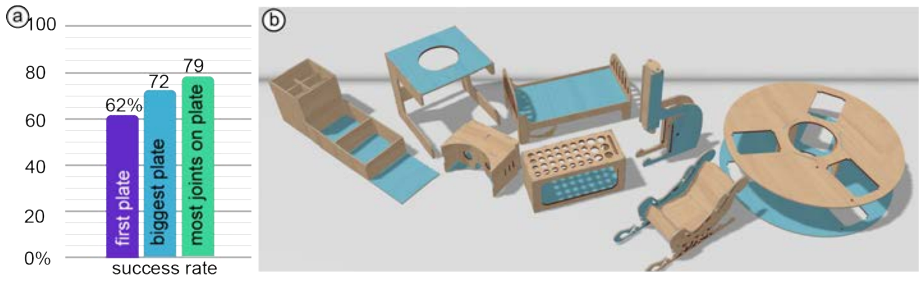


Figure 144: (a) Results of varying the starting plate. (b) some models with their ideal starting plate highlighted.

5.2.3 Technical evaluation: autoAssembler achieves a 97% success rate.

To evaluate the autoAssembler algorithm, we ran it on the benchmark of 66 models encoded as 2D cutting plans. To determine how much the extension of the algorithm contributes to the overall success rate of the algorithm, we ran the tests in four distinct conditions: (1) the basic algorithm, (2) the base with detection of similar plates, (3) the fully extended automatic algorithm with symmetry detection and similarity detection, and (4) with the manual disambiguation tools.

We used the same test set and procedure as presented in the previous section.

Results

As illustrated by Figure 145, the complete algorithm of the eight heuristics and the symmetry/similarity extensions, combined with the manual disambiguation tools resulted in a 97% success rate.

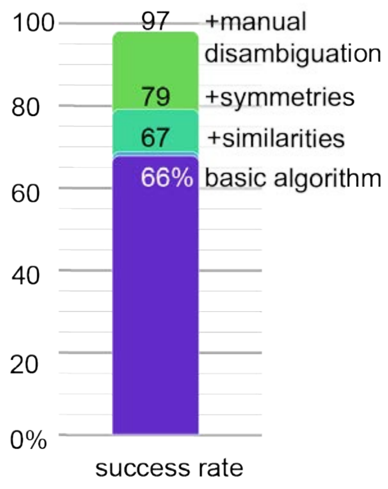


Figure 145: The overall success rate of autoAssembler is 97% based on three extensions of the algorithm: detecting similarities, symmetries and manual disambiguation.

As shown in the diagram, adding the similarity detection alone does not impact the success rate, which is unsurprising as it only reduces the search space (and thus contributes to performance). The model that did get fixed in the process failed before because the search space was overly populated with candidates that were the same.

Symmetry and similarity handling account for 13%: To assess the contribution of the symmetry and similarity detection, we ran the benchmark with each of these steps enabled and disabled. The results shown in Figure 145 show that the symmetries and similarity detection combined increase the success rate from 66 to 79%.

Symmetry and similarity detection reduced the runtime from a median of 0.44s per model to 0.30s per model (a 1.5x performance improvement).

Manual disambiguation accounts for 18%: As shown in Figure 146, 12 models required tweaking using the manual disambiguation tools, although the required user effort was minimal with 1-4 clicks required per model (avg 2.7).

Figure 146b shows the two failed models: a minibar model and a birdhouse the entrance of which was flipped inwards. Both can be fixed using an additional tool to extract 6 plates and manually assemble them.

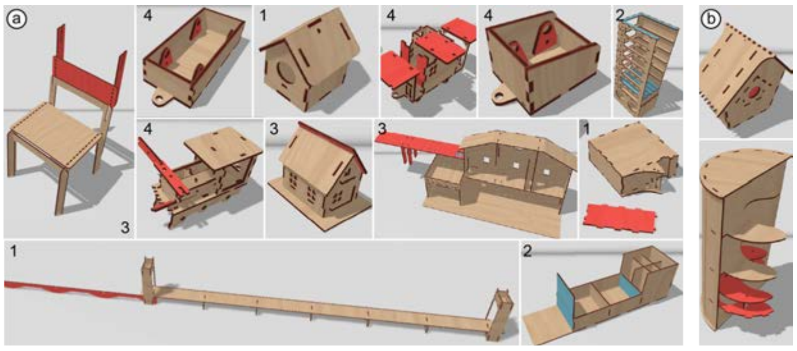


Figure 146: (a) 1-4 clicks using autoAssembler's "re-orient plate" tool fix the mis-oriented red plates in these 12 models. One pair of clicks each using the "swap tool" fixes the swapped blue plates. (b) this birdhouse and minibar would require users to extract 6 plates and re-assemble them.

5.3 A BENCHMARK FOR LASER-CUT MODELS

For the field of laser-cutting to mature, and to make it easier to build on our findings, we release the benchmark of models [95] which we used to evaluate both assembler[3] and autoAssembler. In the Structure-Preserving Editing paper from the previous chapter, we furthermore demonstrated that the benchmark is useful outside of 3D reconstruction.

We sourced the models from public online repositories using random sampling within the subset of laser-cut models consisting of more than one plate and that have been reproduced by at least one other user, they form a representative set of 100 models that capture the variety and complexity of models shared online as of now. Figure 147 presents a high-level overview of the models, as shown they are well distributed across the three main types of laser-cut joints: cross joints, finger joints, and mortise-tenon joints. We did not categorize external types of connections between plates such as adhesives (e.g., glue) because this does not show up in the 2D cutting plans of the models.

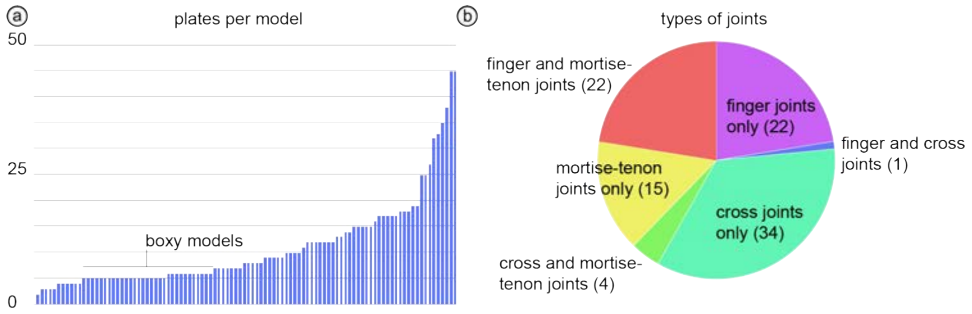


Figure 147: Overview of the models in the benchmark (a) histogram of the number of plates per model and (b) types of joints per model (2 models contained all types of joints).

The histogram of the number of plates is flat for a long time, because many models are rather boxy in nature (4, 5, or 6 plates enclosing a boxy volume). It also clearly indicates that the currently shared models are not that complex yet, arguably caused by the lack of advanced design tools like the ones presented in this thesis.

To allow this benchmark to be reusable for researchers independent of the kyub system we provide for every model a link to the original source model in 2D, an image of the model assembled and rendered in kyub using assembler^3 and an export of the 3D model in generic .obj format so other researchers can use the data in their respective modeling environments. For researchers with access to kyub, we also provide links to the kyub model to allow them to immediately make parametric edits to the models.

This benchmark in combination with the beam search algorithm of autoAssembler furthermore enables the generation of a wealth of assembled states of these models. The fully assembled models form a ground truth of "what is a good assembly", we are currently using this data to traverse yet another route of automatic assembly using machine learning. Without the autoAssembler algorithm and benchmark such approaches were not viable due to lack of data. We see other potential use cases for researchers in NLP to extract a "grammar" of plate construction, a specialized version of shape grammars in computer graphics and architectural design [74], which then allow applications in 3D modeling environments like autocomplete or suggestive interfaces [56], real-time structural feedback (analogous to squiggly underlines in word), or even automatic generation of variations and remixes of models to explore a richer design space [132].

5.4 SOFTWARE INTEGRATION INTO KYUB

As mentioned before, both assembler^3 and autoAssembler are integrated into the import pipeline of kyub. Users either rely on the automation provided by autoAssembler or revert to the interactive tools if the result does not look as expected. The reconstruction logic is spread across two areas of the overall kyub system shown in Figure 103: as illustrated in Figure 148a, the assembler^3 pipeline for processing SVGs is integrated into the infrastructure modules, this provides the distinct benefit that it can be invoked in a headless mode independent of the kyub 3D editor. This allows other tools and systems to use the pipeline as a service which takes SVG as input and returns what we call an `SvgContext` (which can be serialized to JSON) consisting of a set of "knowledge sources" that each capture the data of one of the steps of the assembler³ pipeline. As shown in Figure 148b, the autoAssembler algorithms as well as the tools and logic for the interactive assembly are integrated in the purple (editor-side) modules of the broader system.

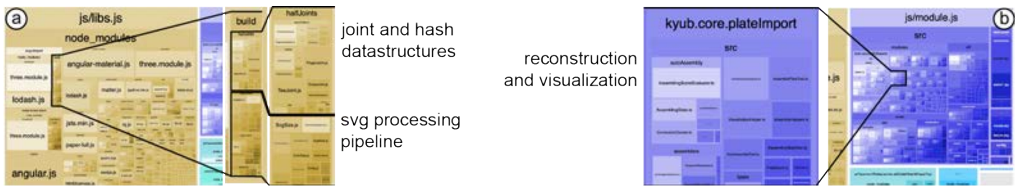


Figure 148: The kyub modules visualized using the *webpack bundle analyzer* [48]. The integration of assembler³ and autoAssembler in the bigger kyub system. (a) On the infrastructure side (yellow) are the modules and data structures to support the 4 analysis steps of the assembler³ algorithm. (b) The editor (purple) is where the tools to invoke reconstruction are as well as the algorithm of autoAssembler.

Figure 149 shows the structure of the `SvgContext` and its underlying "knowledge sources". Each of these modules contains basic information about the SVG they are used to populate the joint and hash data structures used in the editor logic to assemble models. Closer inspection shows that there are two cyclical (dashed) dependencies in this graph. The clearest example of this are the joints and thickness cross-dependency, the algorithm first uses the turns in the paths to derive where joints are, these joints then vote for a material thickness, however when the thickness is known there is extra data to improve the joint detection. We found that 2 cycles of cyclical updating resulted in stable data, as presented before, achieving 99% accuracy on our benchmark.

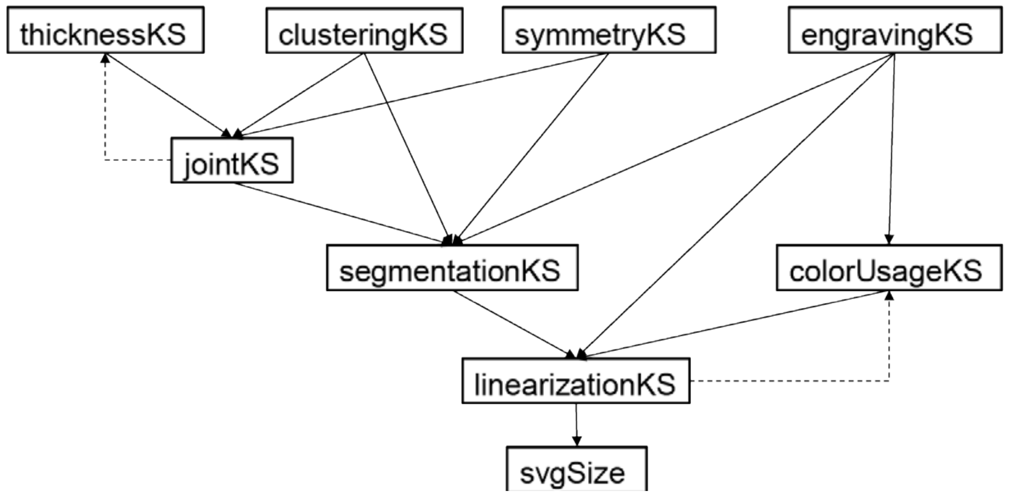


Figure 149: `SvgContext` and its underlying dependency graph of knowledge sources. This determines the structure of the 5-step pipeline of assembler³. There are two circular dependencies balance out after 2 round-trips

Finally, looking back at Figure 103, it becomes evident that all tools in kyub interact on the `ModelData` of the 3D models, the joints and machine-specific properties of the model are regenerated on export. In the current implementation kyub keeps the original imported joints around during the reconstruction effort. However, when users make parametric changes to the models, kyub replaces the original joints with the typical kyub joints (details on the specific algorithm of generating kyub joints can be found in Yannis Kommana's master thesis [66]). In future work we plan to adjust the algorithm so as to respect the specific geometry of the original joints instead of overriding them.

5.5 CONTRIBUTIONS

In this chapter we make four key contributions:

(1) We start with a 5-step pipeline, we call assembler^3, which processes 2D cutting plans to derive high-level data on the structure of the model. We then present tools that allow users to interactively reconstruct a 3D model from this information. Compared the traditional 2D editing workflow this is 10x faster and 26x less error prone.

(2) We release the benchmark of laser-cut models for others to replicate and build upon the work.

(3) Building on the assembler^3 pipeline, we implement AutoAssembler; a 3D reconstruction algorithm that beam searches the exponential space of possible ways of assembling parts. Our main contribution lies in the heuristics that assesses partially assembled models in order to pick the most promising candidates for subsequent exploration; our method prefers candidates that (1) have no intersecting plates, (2) fit into a small bounding box, (3) use plates whose joints fit together well, (4) do not add many unpaired joints, (5) make use of constraints posed by other plates, and (6) conform to symmetry axes of the plates.

(4) We integrate our 3D reconstruction work into the code base of a 3D editor for laser cutting (*kyub*), resulting in an integrated system that allows loading, editing, and writing 2D cutting files.

Our current implementation is limited to three limitations: (1) it handles a basic set of laser-cut elements—we defer living hinges, moving mechanisms, stacked/glued/bolted plates and joints that connect more than one other joint to future versions. (2) In order to work with kyub, our implementation recreates joints from the SVG in which some joint design may get lost. Our automatic workflow does not apply to models where the structure is derived from the *shape* instead of the joints, in particular models only consisting of planar sections held together by cross joints (Figure 139a). As a fallback, users can use the interactive assembly workflow.

5.6 CONCLUSIONS

We present an approach to convert "legacy" 2D cutting plans to 3D models for laser cutting. The resulting models are easy to modify using 3D environments like kyub and allow others to reproduce the physical object using different laser-cutters or materials from the original creator. The proposed automatic workflow succeeds at reconstructing a subset of models if they do not solely rely on cross joints. For the other 36% of models, users fall back to interactive reconstruction. While this involves manual effort, the interactive workflow is still 10x faster and 26x less error prone compared to traditionally editing 2D cutting plans in *2D editors*.

Once a laser cut model has been 3D reconstructed this model will most likely continue its life in this 3D format, making the model easier to process, share, and remix from this point onward. We think that this will help the laser cutting community leave the sharing of 2D cutting plans behind and transition to a 3D format. We anticipate that this will foster collaboration around models und thus ultimately increase the level of model complexity the laser cutting community will be able to achieve.

6 CONCLUSIONS AND OUTLOOK

In this chapter we expand on the contributions of the individual chapters and come back to the overall story of making a small step towards portable models for laser cutting. We discuss the benefits and impact of the work and highlight future opportunities and challenges to transition the field towards mainstream adoption of digital fabrication.

6.1 CONTRIBUTION

In this thesis we presented challenges that prevent designers of laser-cut 3D models from building on the work of others and reproducing models using their materials and machines. We diagnosed that these challenges are caused by using implicitly machine-specific exchange formats.

We presented two ways of approaching this problem: (1) replacing machine-specific elements with more generic counterparts so users no longer need to consider machine and material parameters (within a reasonable range), and (2) representing and sharing the models using a higher level of abstraction to only generate machine-specific formats at fabrication-time, when all machine and material parameters are known.

We contribute with three software systems: (1) a SVG rewriter that makes 2D cutting plans tolerant to variations of machine and material, (2) a 3D modeling environment for laser-cutting that allows users to create, modify, and share models at a higher level of abstraction while providing fabrication-aware tools to support "good laser cutting", and (3) an environment to convert legacy 2D cutting plans to 3D models that can then be manipulated in 3D environments for laser cutting.

To maximize the potential impact of this work, we integrated our tools and algorithms in a platform which currently supports 500+ beta-testers. Moreover, as shown in Figure 79, kyub is currently used in a range of educational contexts where we begin to see how the complexity and usefulness of models is rapidly increasing by facilitating building on the work of others and moving user effort away from dealing with material and machine specifics and towards higher level design. To demonstrate the ability to really design more complex models than the state-of-art in laser cutting, we conducted a workshop in one of our classes where students designed and built fully functional ukuleles as shown in Figure 150.

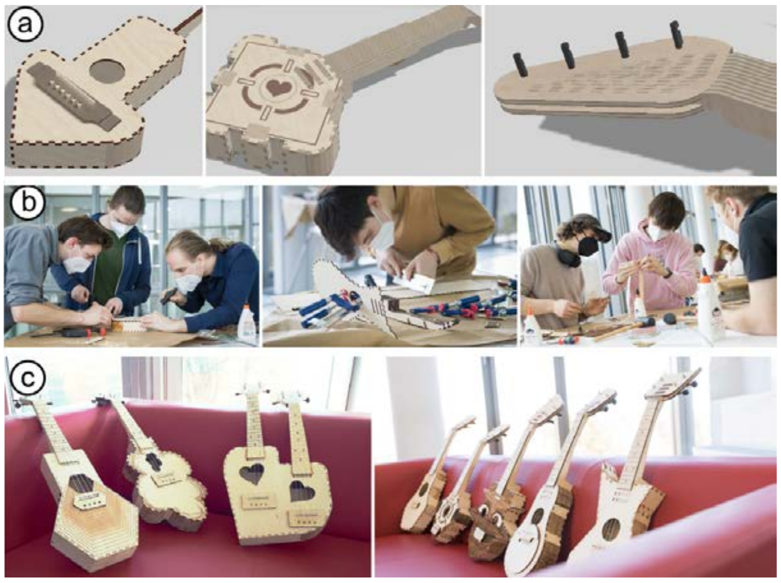


Figure 150: (a) 8 teams of students designed and (b) assembled (c) their instruments.

6.1.1 limitation: loss of knowledge of materials and machines

Abstracting away from the material and machine at hand will allow more (lay) users to engage with laser cutters, who will produce more advanced and useful models. It does come at a cost as well, similar to how only few software engineers today write assembler code or truly understand the interaction with bits and transistors within a computing system, this level of abstraction means a loss of knowledge of the actual machines and materials at hand. We see this as an inevitable trade-off with an increase of complexity of models (nobody should have to reimplement OpenCV when building a simple image manipulating pipeline). We do not discount the value of that knowledge and acknowledge that there is certainly a space for applications on the other side of the trade-off as well: expert tools to design and invent new types of joints/mechanisms or fabrication routines.

We furthermore believe that using higher level modeling systems can be used as a vehicle to bring back the material and machine knowledge when used in carefully constructed workshops. We demonstrate this potential by bringing our software to schools, teachers have shown to be very receptive of inclusion of fabrication classes where pupils not only design but also fabricate and assemble their models and thus learn various hardware skills in the process.

6.2 FUTURE OUTLOOK: UPCOMING OPPORTUNITIES AND CHALLENGES

The work in this thesis forms a small step towards the more ambitious goal of allowing users to build on the work of others and increase the complexity and relevance of models that are shared and fabricated using laser cutters. In this section we extrapolate the directions we set out to study and present related opportunities and challenges for future work.

6.2.1 trade-offs between machine and material properties

In this thesis we identified joints, mounts, and mechanisms as being hard to design and highly machine/material specific, this assumption is a direct by-product of the level of complexity of models that are currently designed and made. Advanced models are more than just a structure of jointed plates, assemblies have other mechanical properties like weight, structural integrity, compliance, electrical or heat conductivity, and many more. Analogous to the discussion of functional properties in 3D models in the related work (chapter 2.4) it would be interesting to abstract away some of these properties too. Imagine a user who designed and shared a chair model, another user with a different material would be able to reproduce a model that holds the same weight and feels equally compliant when sitting down, these may be more important factors that reproducing the exact same 3D structure of plates.

Looking back at the development of compilers, each attempt to virtualize a material or machine property adds a specific rule in the compiling function from a high-level description to machine code. It is tempting to formalize such rules, forming an automatic process. Databases of material properties exist for advanced software packages. The next step would be to identify relevant substructures that perform specific mechanical roles in models and formulate compiling functions with material/machine properties as variable. Resulting functions will conflict (e.g., no metal chair exists which emulates *all properties* of a wooden chair), so a user interface could enable exploring trade-offs.

Pushing this notion further would allow for abstraction of the *fabrication process*. In this thesis we looked at laser cutting only, but a representation of the model independent of whether it would be laser cut, die stamped, milled, 3D printed, or injection molded could be used as input into a "machine compiler" that generates joints and connectors or geometry to support the desired fabrication process. This could enable interesting applications like upgrading a prototype to a model for a more advanced manufacturing process when scaling up to fabricate a small series.

6.2.2 extracting reusable content

The work in this thesis lays a foundation that allows users to build on the work of others by making parametric edits to models and allowing users to increase the complexity of models by adding new plates and volumes. At the current state this is mostly customizing and remixing, similar to what we have presented in the context of 3D printing with *Grafter* [102]. In looking back to the development of reuse in the history of computing, a crucial step was to determine the right level of abstraction for reusable components [58]. Rather than reusing entire models and mechanisms, future work could investigate what the right granularity of reusable components for laser cut/fabrication models is. Based on our findings we assume this to be roughly at the level of joints, mounts, and mechanisms. However, interesting hierarchies and combinations of these will likely provide more useful than single cutouts.

6.2.3 designing products as opposed to processes

A key insight we derived during this research is that the current fabrication workflows describe *processes*, which makes sense because the users (either industrial or tech enthusiast) are interested in *how to use the machine at hand* to fabricate a desired product. For example: "how do I make a guitar *using a laser cutter*". Those are interesting puzzles for engineers, but when we think of laser cutting (and personal fabrication) to transition towards a mainstream phenomenon, these objectives change. The more relevant question for those users is "how do I get a custom guitar": describing a *product*. Like a search query on Amazon, which is where they get their products now.

Designing products instead of processes however has some major implications in how to build future systems for digital fabrication. Consumer products are almost never made from a single material or using one fabrication technique alone. This stands in stark contrast to the way fabrication tools are designed today. It will thus require developing systems that handle multiple fabrication machines under the hood as well as generating joints between different materials and processes. This is domain expertise typically in the hands of Industrial Designers or Engineers. This will further require expressing fabrication and assembly workflows in a more formal language so as to have full control over the machines, *Taxon* [71] is a promising initial step in that direction.

Beyond the technical underbelly of how to interface with machines and materials and adjusting the models, it also requires rethinking the user facing components and the ecosystem of design. End-users as stated before are interested in the product and how to customize it, this requires a different interface from a user who designs the model from scratch. We expect to see an end-user facing system integrated into services like Amazon with carefully designed degrees of freedom. The designer-facing side would allow designing the entire model in its full complexity while exposing specific degrees of freedom to end-users.

6.2.4 manifestation of fabrication in everyday life

The work in this thesis makes a small contribution to advance the field towards mass adoption of fabrication, but it will take more than formats and interfaces for this to happen. Bringing fabrication to educational contexts will be an important step forwards, and so is transitioning to interfaces that allow customizing products. It will further require advances in fabrication hardware and our understanding of what usage context will be most viable. We ran an early exploration into one such context for 3D printing focused on *mobile fabrication* [99], but much more can and will have to be done to find out if and how fabrication can become a relevant mainstream technology.

7 REFERENCES

1. Muhammad Abdullah, Martin Taraz, Yannis Kommana, Shohei Katakura, Robert Kovacs, Jotaro Shigeyama, Thijs Roumen, and Patrick Baudisch. 2021. FastForce: Real-Time Reinforcement of Laser-Cut Structures. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21).* ACM, New York, NY, USA, Article 673, 1–12. DOI: https://doi.org/10.1145/3411764.3445466
2. Muhammad Abdullah, Romeo Sommerfeld, Laurenz Seidel, Jonas Noack, Ran Zhang, Thijs Roumen, and Patrick Baudisch. 2021. Roadkill: Nesting Laser-Cut Objects for Fast Assembly. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21).* ACM, New York, NY, USA, 972–984. DOI: https://doi.org/10.1145/3472749.3474799
3. Sameer Agarwal, Yasutaka Furukawa, Noah Snavely, Ian Simon, Brian Curless, Steven M. Seitz, and Richard Szeliski. 2011. Building Rome in a day. *Commun. ACM* 54, 10 (October 2011), 105–112. DOI: https://doi.org/10.1145/2001269.2001293
4. Anand Agarawala and Ravin Balakrishnan. 2006. Keepin' it real: pushing the desktop metaphor with physics, piles and the pen. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06)* ACM, New York, NY, USA, 1283-1292. DOI: http://dx.doi.org/10.1145/1124772.1124965
5. Aftab Ahmad, Kjell Andersson, and Ulf Sellgren. "An optimization approach toward a robust design of six degrees of freedom haptic devices." *Journal of Mechanical Design* 137, no. 4 (2015): 042301.
6. Celena Alcock, Nathaniel Hudson, and Parmit K. Chilana. 2016. Barriers to Using, Customizing, and Printing 3D Designs on Thingiverse. *In Proceedings of the 19th International Conference on Supporting Group Work (GROUP '16).* Association for Computing Machinery, New York, NY, USA, 195–199. DOI: https://doi.org/10.1145/2957276.2957301
7. Angular.js, last accessed March 2022, http://www.angularjs.org

8. Marco Attene, Marco Livesu, Sylvain Lefebvre, Thomas Funkhouser, Szymon Rusinkiewicz, Stefano Ellero, Jonàs Martínez, and Amit Haim Bermano. "Design, representations, and processing for additive manufacturing." In *Synthesis Lectures on Visual Computing: Computer Graphics, Animation, Computational Photography, and Imaging* 10, no. 2 (2018): 1-146.
9. Autodesk Fusion 360, laser accessed Jan 2022, https://www.autodesk.com/products/fusion-360
10. AutoDesk, Slicer for Fusion 360. last accessed March 2022, https://knowledge.autodesk.com/support/fusion-360/downloads/caas/downloads/content/slicer-for-fusion-360.html
11. Bradford C. Barber, David P. Dobkin, and Hannu Huhdanpaa. "The quickhull algorithm for convex hulls." *ACM Transactions on Mathematical Software (TOMS)* 22, no. 4 (1996): 469-483.
12. Oliver A. Bauchau, and James I. Craig. "Euler-Bernoulli beam theory." In *Structural analysis*, pp. 173-221. Springer, Dordrecht, 2009. DOI: https://doi.org/10.1007/978-90-481-2516-6_5
13. Patrick Baudisch and Stefanie Mueller. "Personal fabrication." Foundations and Trends® in Human–Computer Interaction 10, no. 3–4 (2017): 165-293.
14. Patrick Baudisch, Arthur Silber, Yannis Kommana, Milan Gruner, Ludwig Wall, Kevin Reuss, Lukas Heilman, Robert Kovacs, Daniel Rechlitz, and Thijs Roumen. 2019. Kyub: A 3D Editor for Modeling Sturdy Laser-Cut Objects. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. ACM, New York, NY, USA, Paper 566, 1–12. DOI: https://doi.org/10.1145/3290605.3300796
15. Dustin Beyer, Serafima Gurevich, Stefanie Mueller, Hsiang-Ting Chen, and Patrick Baudisch. 2015. Platener: Low-Fidelity Fabrication of 3D Objects by Substituting 3D Print with Laser-Cut Plates. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (CHI '15). Association for Computing Machinery, New York, NY, USA, 1799–1806. DOI: https://doi.org/10.1145/2702123.2702225
16. Joana Bergsiek, Jannis Bolik, Tim Garrels, Paul Methfessel, Antonius Naumann, Martin Taraz, Robin Wersich. 2019. Enhancing the Kyub Softwaresystem to Push the Boundaries of Physical Prototyping. In *bachelor thesis at HPI, internal publication.*

17. Bernd Bickel, Moritz Bächer, Miguel A. Otaduy, Hyunho Richard Lee, Hanspeter Pfister, Markus Gross, and Wojciech Matusik. 2010. Design and fabrication of materials with desired deformation behavior. In *ACM SIGGRAPH 2010* (SIGGRAPH '10). Association for Computing Machinery, New York, NY, USA, Article 63, 1–10. https://doi.org/10.1145/1833349.1778800
18. Roberto Bisiani, 1987. Beam search. In Shapiro, S., ed., *Encyclopedia of Artificial Intelligence*. John Wiley and Sons. 56–58.
19. Corrando Böhm. *Calculatrices digitales du déchiffrage de formules logico-mathématiques par la machine meême dans la conception du programme*. PhD thesis, ETH, Zuïrich, 1954. Thesis written under supervision of E. Stiefel and P. Bernays and defended in 1951. Published in Ann. Math. PuraAppl. 37 (1954), 5-47. DOI: doi.org/10.3929/ethz-a-000090226.
20. Sofien Bouaziz, Mario Deuss, Yuliy Schwartzburg, Thibaut Weise, and Mark Pauly. "Shape-up: Shaping discrete geometry with projections." In Computer Graphics Forum, vol. 31, no. 5, pp. 1657-1667. Oxford, UK: Blackwell Publishing Ltd, 2012.
21. Erin Buehler, Stacy Branham, Abdullah Ali, Jeremy J. Chang, Megan Kelly Hofmann, Amy Hurst, and Shaun K. Kane. 2015. Sharing is Caring: Assistive Technology Designs on Thingiverse. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (CHI '15). Association for Computing Machinery, New York, NY, USA, 525–534. https://doi.org/10.1145/2702123.2702525
22. Alberto Castigliano. 1879. Théorie de l'équilibre des systèmes élastiques et ses applications. Vol. 1. AF Negro, 1879.
23. Computational Geometry Algorithms Library (CGAL), last accessed March 2022, http://www.cgal.org
24. Ruei-Che Chang, Chih-An Tsao, Fang-Ying Liao, Seraphina Yong, Tom Yeh, and Bing-Yu Chen. 2021. Daedalus in the Dark: Designing for Non-Visual Accessible Construction of Laser-Cut Architecture. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (UIST '21). Association for Computing Machinery, New York, NY, USA, 344– 358. DOI: https://doi.org/10.1145/3472749.3474754
25. Desai Chen, David I. W. Levin, Piotr Didyk, Pitchaya Sitthi-Amorn, and Wojciech Matusik. 2013. Spec2Fab: a reducer-tuner model for translating specifications to 3D prints. In *ACM Trans. Graph.* 32, 4, Article 135 (July 2013), 10 pages https://doi.org/10.1145/2461912.2461994

26. Lujie Chen, and Lawrence Sass. "Fresh press modeler: A generative system for physically based low fidelity prototyping." In *Computers & Graphics 54* (2016): 157-165.
27. Tao Chen, Zhe Zhu, Ariel Shamir, Shi-Min Hu, and Daniel Cohen-Or. 2013. 3-Sweep: extracting editable objects from a single photo. *ACM Trans. Graph.* 32, 6, Article 195 (November 2013), 10 pages. DOI: https://doi.org/10.1145/2508363.2508378
28. James McCrae, Karan Singh, and Niloy J. Mitra. 2011. Slices: a shape-proxy based on planar sections. *ACM Trans. Graph. 30, 6, Article 168 (December 2011),* 12 pages. DOI: https://doi.org/10.1145/2070781.2024202
29. James McCrae, Nobuyuki Umetani, and Karan Singh. 2014. FlatFitFab: interactive modeling with planar sections. In *Proceedings of the 27th annual ACM symposium on User interface software and technology (UIST '14).* ACM, New York, NY, USA, 13-22. DOI: https://doi.org/10.1145/2642918.2647388
30. CutLaserCut kerf manual http://www.cutlasercut.com/resources/tips-and-advice/what-is-laser-kerf. Accessed January 2020.
31. Benjamin Daniel, Jeffrey Johnson, Ole Schluter. 2021. Plate-Based Modelling for Laser Cutting in Kyub. In *bachelor thesis at HPI, internal publication.*
32. Mustafa Doga Dogan, Steven Vidal Acevedo Colon, Varnika Sinha, Kaan Aksit, and Stefanie Mueller. 2021. SensiCut: Material-Aware Laser Cutting Using Speckle Sensing and Deep Learning. In The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21), October 10–14, 2021, Virtual Event, USA. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3472749.3474733
33. Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. 2018. InverseCSG: automatic conversion of 3D models to CSG trees. *ACM Trans. Graph.* 37, 6, Article 213 (November 2018), 16 pages. DOI: https://doi.org/10.1145/3272127.3275006
34. Heinrich Dubbel, and B. J. Davies. *Dubbel-Handbook of mechanical engineering*. Springer Science & Business Media, 2013.
35. Richard Ebeling, Leonard Geier, Ben Hurdelhey, Dominik Meier, Marcel Schmidberger. 2018. Enhancing the Kyub Software System to Facilitate Laser Cutting of Complex Objects. In *bachelor thesis at HPI, internal publication.*

36. Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. "Density-based spatial clustering of applications with noise." In *Int. Conf. Knowledge Discovery and Data Mining*, vol. 240, p. 6. 1996.
37. Christoph M. Flath, Sascha Friesike, Marco Wirth, & Frederic Thiesse, Copy, transform, combine: exploring the remix as a form of innovation. *Journal of Information Technology*, 1-20. DOI: https://doi.org/10.1057/s41265-017-0043-9
38. Noa Fish, Melinos Averkiou, Oliver van Kaick, Olga Sorkine-Hornung, Daniel Cohen-Or, and Niloy J. Mitra. 2014. Meta-representation of shape families. *ACM Trans. Graph.* 33, 4, Article 34 (July 2014), 11 pages. DOI: https://doi.org/10.1145/2601097.2601185
39. Paul Freiberger, Michael Swaine. 1984. Fire in the Valley: The Making of the Personal Computer. McGraw-Hill, Inc., NY. ISBN: 9780071358927.
40. Lukas Fritzsche, Lukas Heilmann, Bastian Kӧnig, Jonas Noack, Milan Proell. 2017. Extending the Kyub Interactive Editor for Laser Cutting Towards the Fabrication of Large-Scale Objects. In *bachelor thesis at HPI, internal publication*
41. Thomas Funkhouser, Michael Kazhdan, Philip Shilane, Patrick Min, William Kiefer, Ayellet Tal, Szymon Rusinkiewicz, and David Dobkin. 2004. Modeling by example. ACM Trans. Graph. 23, 3 (August 2004), 652–663. https://doi.org/10.1145/1015706.1015775
42. Tinsley A. Galyean and John F. Hughes. 1991. Sculpting: an interactive volumetric modeling technique. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques (SIGGRAPH '91).* ACM, New York, NY, USA, 267-274. DOI: http://dx.doi.org/10.1145/122718.122747
43. Jian Gao, Detao Zheng, and Nabil Gindy. "Mathematical representation of feature conversion for CAD/CAM system integration." *Robotics and Computer-Integrated Manufacturing*20, no. 5 (2004): 457-467.
44. GrabCAD, last accessed May 2022, http://www.grabcad.com
45. Salvatore Gerbino. 2003. "Tools for the interoperability among CAD systems." In *Proc. XIII ADM-XV INGEGRAF Int. Conf. Tools and Methods Evolution in Engineering Design*. 2003.
46. Neil Gershenfeld. 2005. Fab: The Coming Revolution on Your Desktop--from Personal Computers to Personal Fabrication. Basic Books publisher, ISBN: 9780465027453

47. Saul Greenberg and Bill Buxton. 2008. Usability evaluation considered harmful (some of the time). In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08).* Association for Computing Machinery, New York, NY, USA, 111–120. DOI: https://doi.org/10.1145/1357054.1357074
48. Yuriy Grunin and Vesa Laakso (maintainers) Webpack bundle analyzer. In *npm packages*. Last accessed May 2022, https://www.npmjs.com/package/webpack-bundle-analyzer
49. Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (UIST '19). Association for Computing Machinery, New York, NY, USA, 281–292. https://doi.org/10.1145/3332165.3347925
50. Florian Heller, Jan Thar, Dennis Lewandowski, Mirko Hartmann, Pierre Schoonbrood, Sophy Stönner, Simon Voelker, and Jan Borchers. 2018. CutCAD - An Open-source Tool to Design 3D Objects in 2D. In *Proceedings of the 2018 Designing Interactive Systems Conference (DIS '18).* ACM, New York, NY, USA, 1135-1139. DOI: https://doi.org/10.1145/3196709.3196800
51. Megan Hofmann, Gabriella Hann, Scott E. Hudson, and Jennifer Mankoff. 2018. Greater than the Sum of its PARTs: Expressing and Reusing Design Intent in 3D Models. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18).* ACM, New York, NY, USA, Paper 301, 12 pages. DOI: https://doi.org/10.1145/3173574.3173875
52. Grace Murray Hopper. 1952. The education of a computer. In Proceedings of the 1952 ACM national meeting (Pittsburgh) (ACM '52). Association for Computing Machinery, New York, NY, USA, 243–249. https://doi.org/10.1145/609784.609818
53. Qi-Xing Huang, Simon Flöry, Natasha Gelfand, Michael Hofer, and Helmut Pottmann. 2006. Reassembling fractured objects by geometric matching. *ACM Trans. Graph.* 25, 3 (July 2006), 569-578. DOI: https://doi.org/10.1145/1141911.1141925
54. Nathaniel Hudson, Celena Alcock, and Parmit K. Chilana. 2016. Understanding Newcomers to 3D Printing: Motivations, Workflows, and Barriers of Casual Makers. *In Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16).* Association for Computing Machinery, New York, NY, USA, 384–396. DOI: https://doi.org/10.1145/2858036.2858266

55. Alexandra Ion, Johannes Frohnhofen, Ludwig Wall, Robert Kovacs, Mirela Alistar, Jack Lindsay, Pedro Lopes, Hsiang-Ting Chen, and Patrick Baudisch. 2016. Metamaterial Mechanisms. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (UIST '16). Association for Computing Machinery, New York, NY, USA, 529–539. https://doi.org/10.1145/2984511.2984540
56. Takeo Igarashi and John F. Hughes. 2001. A suggestive interface for 3D drawing. In *Proceedings of the 14th annual ACM symposium on User interface software and technology* (UIST '01). Association for Computing Machinery, New York, NY, USA, 173–181. https://doi.org/10.1145/502348.502379
57. Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. 1999. Teddy: a sketching interface for 3D freeform design. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques (SIGGRAPH '99).* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 409-416. DOI: http://dx.doi.org/10.1145/311535.311602
58. Malcolm Douglas McIlroy, J. Buxton, Peter Naur, and Brian Randell. "Mass-produced software components." In *Proceedings of the 1st international conference on software engineering, Garmisch Pattenkirchen, Germany*, pp. 88-98. 1968.
59. ISO 286-1:2010 - Geometrical product specifications (GPS) - ISO code system for tolerances on linear sizes - Part 1: Basis of tolerances, deviations and fits. *2010: International Organization for Standardization (ISO).*
60. ISO, DIN. "286-1: ISO-System für Grenzmaße und Passungen." *Grundlagen für Toleranzen, Abmaße und Passungen* (1990).
61. Kenny R. Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. 2020. ShapeAssembly: learning to generate programs for 3D shape structure synthesis. *ACM Trans. Graph. 39, 6, Article 234* (December 2020), 20 pages. DOI: https://doi.org/10.1145/3414685.3417812
62. Shohei Katakura, Yuto Kuroki, and Keita Watanabe. 2019. A 3D Printer Head as a Robotic Manipulator. In Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19). Association for Computing Machinery, New York, NY, USA, 535–548. https://doi.org/10.1145/3332165.3347885
63. Shohei Katakura, Martin Taraz, Paul Methfessel, Abdullah Muhammad, Conrad Lempert, and Patrick Baudisch. 2022. Kerfmeter: Automatic Kerf Calibration for Laser Cutting. *In submission to UIST'22.*

64. Jeeeun Kim, Anhong Guo, Tom Yeh, Scott E. Hudson, and Jennifer Mankoff. 2017. Understanding Uncertainty in Measurement and Accommodating its Impact in 3D Modeling and Printing. In *Proceedings of the 2017 Conference on Designing Interactive Systems* (DIS '17). ACM, New York, NY, USA, 1067-1078. DOI: https://doi.org/10.1145/3064663.3064690
65. Ralph Kimball, and B. Verplank E. Harslem. "Designing the Star user interface." *Byte* 7 (1982): 242-282.
66. Yannis Kommana. 2019. An Implementation of a 3D Modeling Ssystem for Sturdy Laser-cut Objects. In *master thesis at HPI, internal publication.*
67. Robert Kovacs, Anna Seufert, Ludwig Wall, Hsiang-Ting Chen, Florian Meinel, Willi Müller, Sijing You, Maximilian Brehm, Jonathan Striebel, Yannis Kommana, Alexander Popiak, Thomas Bläsius, and Patrick Baudisch. 2017. TrussFab: Fabricating Sturdy Large-Scale Structures on Desktop 3D Printers. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17).* ACM, New York, NY, USA, 2606-2616. DOI: https://doi.org/10.1145/3025453.3026016
68. Karim Lakhani, and Eric Von Hippel. "How open source software works:"free" user-to-user assistance." In *Produktentwicklung mit virtuellen Communities*, pp. 303-339. Gabler Verlag, 2004.
69. Harold "Bud" Lawon and Howard Bromberg. "The World's First COBOL Compilers". Computer History Museum Lecutre Series, Stanford University, June 12 1997. https://web.archive.org/web/20111013021915/http://www.computerhistory.org/events/lectures/cobol_06121997/index.shtml
70. Danny Leen, Nadya Peek, and Raf Ramakers. "LamiFold: Fabricating Objects with Integrated Mechanisms Using a Laser cutter Lamination Workflow." 2020. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (UIST'20), pp. 304-316. 2020.
71. Jasper Tran O'Leary, Chandrakana Nandi, Khang Lee, and Nadya Peek. 2021. Taxon: A Language for Formal Reasoning with Digital Fabrication Machines. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (UIST '21). Association for Computing Machinery, New York, NY, USA, 691–709. https://doi.org/10.1145/3472749.3474779

72. Jun Li, Kai Xu, Siddhartha Chaudhuri, Ersin Yumer, Hao Zhang, and Leonidas Guibas. "Grass: Generative recursive autoencoders for shape structures." *ACM Transactions on Graphics (TOG)* 36, no. 4 (2017): 1-14.
73. Jinjie Lin, Daniel Cohen-Or, Hao Zhang, Cheng Liang, Andrei Sharf, Oliver Deussen, and Baoquan Chen. 2011. Structure-preserving retargeting of irregular 3D architecture. *ACM Trans. Graph. 30, 6 (December 2011)*, 1–10. DOI: https://doi.org/10.1145/2070781.2024217
74. Tianqiang Liu, Siddhartha Chaudhuri, Vladimir G. Kim, Qixing Huang, Niloy J. Mitra, and Thomas Funkhouser. 2014. Creating consistent scene graphs using a probabilistic grammar. ACM Trans. Graph. 33, 6, Article 211 (November 2014), 12 pages. https://doi.org/10.1145/2661229.2661243
75. Eunice López-Camacho, Gabriela Ochoa, Hugo Terashima-Marín, and Edmund K. Burke. "An effective heuristic for the two-dimensional irregular bin packing problem." Annals of Operations Research 206, no. 1 (2013): 241-264.
76. Theodore H Maiman. "Stimulated optical radiation in ruby." (1960): 493-494.
77. Tobias Martin, Nobuyuki Umetani, and Bernd Bickel. 2015. OmniAD: data-driven omni-directional aerodynamics. ACM Trans. Graph. 34, 4, Article 113 (August 2015), 12 pages. https://doi.org/10.1145/2766919
78. Niloy J. Mitra, Leonidas J. Guibas, and Mark Pauly. 2006. Partial and approximate symmetry detection for 3D geometry. *ACM Trans. Graph. 25, 3 (July 2006)*, 560–568. DOI: https://doi.org/10.1145/1141911.1141924
79. MyMiniFactory, last accessed May 2022, https://www.myminifactory.com
80. David E. Muller, and Franco P. Preparata. "Finding the intersection of two convex polyhedra." *Theoretical Computer Science* 7, no. 2 (1978): 217-236.
81. Stefanie Mueller, Bastian Kruck, and Patrick Baudisch. 2013. LaserOrigami: laser-cutting 3D objects. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '13). Association for Computing Machinery, New York, NY, USA, 2585–2592. https://doi.org/10.1145/2470654.2481358

82. Stefanie Mueller, Pedro Lopes, and Patrick Baudisch. 2012. Interactive construction: interactive fabrication of functional mechanical devices. In *proceedings of the 25th annual ACM symposium on User interface software and technology* (UIST'12). Association for Computing Machinery, New York, NY, USA, 599–606. https://doi.org/10.1145/2380116.2380191
83. Stefanie Mueller, Anna Seufert, Huaishu Peng, Robert Kovacs, Kevin Reuss, François Guimbretière, and Patrick Baudisch. 2019. FormFab: Continuous Interactive Fabrication. In *Proceedings of the Thirteenth International Conference on Tangible, Embedded, and Embodied Interaction* (TEI '19). Association for Computing Machinery, New York, NY, USA, 315–323. https://doi.org/10.1145/3294109.3295620
84. Martin Nisser, Christina Chen Liao, Yuchen Chai, Aradhana Adhikari, Steve Hodges, and Stefanie Mueller. 2021. LaserFactory: A Laser Cutter-based Electromechanical Assembly and Fabrication Platform to Make Functional Devices & Robots. In Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21). Association for Computing Machinery, New York, NY, USA, Article 663, 1–15. https://doi.org/10.1145/3411764.3445692
85. Node.js, last accessed March 2022, http://www.nodejs.org
86. Thomas Oster, Rene Bohne, and Jan Borchers. "Visicut: An application genre for lasercutting in personal fabrication." *RWTH Aachen University* (2011).
87. Kumar C.N. Patel, "Continuous-wave laser action on vibrational-rotational transitions of C O 2." Physical review 136, no. 5A (1964): A1187.
88. Georgios Papaioannou, Tobias Schreck, Anthousis Andreadis, Pavlos Mavridis, Robert Gregor, Ivan Sipiran, and Konstantinos Vardis. 2017. From Reassembly to Object Completion: A Complete Systems Pipeline. J. In *Comput. Cult. Herit.* 10, 2, Article 8 (March 2017), 22 pages. DOI: https://doi.org/10.1145/3009905
89. Keunwoo Park, Conrad Lempert, Muhammad Abdullah, Shohei Katakura, Jotaro Shigeyama, Thijs Roumen, and Patrick Baudisch. 2022. FoolProofJoint: Reducing Assembly Errors of Laser Cut 3D Models by Means of Custom Joint Patterns. In *CHI Conference on Human Factors in Computing Systems (CHI '22).* ACM, New York, NY, USA, Article 271, 1–12. DOI: https://doi.org/10.1145/3491102.3501919
90. Kevin J Parker, Saara Marjatta Sofia Totterman, and Jose Tamez Pe. "System and method for 4d reconstruction and visualization." *U.S. Patent 6,169,817*, issued January 2, 2001.

91. Pieter Pauwels, Davy Van Deursen, Jos De Roo, Tim Van Ackere, Ronald De Meyer, Rik Van de Walle, and Jan Van Campenhout. "Three-dimensional information exchange over the semantic web for the domain of architecture, engineering, and construction." *Ai Edam* 25, no. 4 (2011): 317-332.
92. Huaishu Peng, Jimmy Briggs, Cheng-Yao Wang, Kevin Guo, Joseph Kider, Stefanie Mueller, Patrick Baudisch, and François Guimbretière. 2018. RoMA: Interactive Fabrication with Augmented Reality and a Robotic 3D Printer. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18). Association for Computing Machinery, New York, NY, USA, Paper 579, 1–12. https://doi.org/10.1145/3173574.3174153
93. Michael J. Pratt, "Introduction to ISO 10303—the STEP standard for product data exchange." *Journal of Computing and Information Science in Engineering* 1, no. 1 (2001): 102-103.
94. Thijs Roumen, Thingiverse Analysis Script, last accessed August 2020 https://github.com/ThijsRoumen/thingiverse-browser.
95. Thijs Roumen, Benchmark of assembler^3 models, to be released http://www.thijsroumen.eu/data/benchmark-lasercut-models.zip
96. Thijs Roumen, Ingo Apel, Thomas Kern, Martin Taraz, Ritesh Sharma, Ole Schlueter, Jeffrey Johnson, Dominik Meier, Conrad Lempert, and Patrick Baudisch. 2022. Structure-Preserving Editing of Plates and Volumes for Laser Cutting. In *submission to UIST'22.*
97. Thijs Roumen, Ingo Apel, Jotaro Shigeyama, Abdullah Muhammad, and Patrick Baudisch. 2020. Kerf-Canceling Mechanisms: Making Laser-Cut Mechanisms Operate across Different Laser Cutters. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST'20).* ACM, New York, NY, USA, 293–303. DOI: https://doi.org/10.1145/3379337.3415895
98. Thijs Roumen, Yannis Kommana, Ingo Apel, Conrad Lempert, Markus Brand, Erik Brendel, Laurenz Seidel, Lukas Rambold, Carl Goedecken, Pascal Crenzin, Ben Hurdelhey, Muhammad Abdullah, and Patrick Baudisch. 2021. Assembler^3: 3D Reconstruction of Laser-Cut Models. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21).* ACM, New York, NY, USA, Article 672, 1–11. DOI: https://doi.org/10.1145/3411764.3445453

99. Thijs Roumen, Bastian Kruck, Tobias Dürschmid, Tobias Nack, and Patrick Baudisch. 2016. Mobile Fabrication. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (UIST '16). Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2984511.2984586

100. Thijs Roumen, Conrad Lempert, Ingo Apel, Erik Brendel, Markus Brand, Laurenz Seidel, Lukas Rambold, and Patrick Baudisch. 2021. AutoAssembler: Automatic Reconstruction of Laser-Cut 3D Models. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (*UIST '21*) ACM, New York, NY, USA, 652–662. DOI: https://doi.org/10.1145/3472749.3474776

101. Thijs Roumen, Jotaro Shigeyama, Julius Cosmo Romeo Rudolph, Felix Grzelka, and Patrick Baudisch. 2019. SpringFit: Joints and Mounts that Fabricate on Any Laser Cutter. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. ACM, New York, NY, USA, 727–738. DOI: https://doi.org/10.1145/3332165.3347930

102. Thijs Roumen, Willi Müller, and Patrick Baudisch. 2018. Grafter: Remixing 3D-Printed Machines. 2018. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Paper 63, 12 pages. DOI: https://doi.org/10.1145/3173574.3173637

103. Daniel Saakes, Thomas Cambazard, Jun Mitani, and Takeo Igarashi. 2013. PacCAM: material capture and interactive 2D packing for efficient material usage on CNC cutting machines. In *Proceedings of the 26th annual ACM symposium on User interface software and technology* (UIST '13). Association for Computing Machinery, New York, NY, USA, 441–446. DOI: https://doi.org/10.1145/2501988.2501990

104. Greg Saul, Manfred Lau, Jun Mitani, and Takeo Igarashi. 2010. SketchChair: an all-in-one chair design system for end users. In *Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction (TEI '11)*. ACM, New York, NY, USA, 73-80. DOI: http://dx.doi.org/10.1145/1935701.1935717

105. Adriana Schulz, Ariel Shamir, David I. W. Levin, Pitchaya Sitthi-amorn, and Wojciech Matusik. 2014. Design and fabrication by example. ACM Trans. Graph. 33, 4, Article 62 (July 2014), 11 pages. https://doi.org/10.1145/2601097.2601127

106. Adriana Schulz, Jie Xu, Bo Zhu, Changxi Zheng, Eitan Grinspun, and Wojciech Matusik. 2017. Interactive design space exploration

and optimization for CAD models. ACM Trans. Graph. 36, 4, Article 157 (August 2017), 14 pages. https://doi.org/10.1145/3072959.3073688

107. Ticha Sethapakdi, Daniel Anderson, Adrian Reginald Chua Sy, and Stefanie Mueller. 2021. Fabricaide: Fabrication-Aware Design for 2D Cutting Machines. In *CHI Conference on Human Factors in Computing Systems* (CHI '21), May 8–13, 2021, Yokohama, Japan. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3411764.3445345

108. Sung Ho Shin. 2004 Analytic integration of tolerances in designing precision interfaces for modular robotics. *PhD diss., UTexas*. http://hdl.handle.net/2152/2195

109. Arthur Silber. 2019. Integrating 2D Elements into 3D Modelling for Laser Cutters. In *master thesis at HPI, internal publication.*

110. Mélina Skouras, Bernhard Thomaszewski, Stelian Coros, Bernd Bickel, and Markus Gross. 2013. Computational design of actuated deformable characters. In *ACM Trans. Graph. 32, 4, Article 8*2 (July 2013), 10 pages. https://doi.org/10.1145/2461912.2461979

111. Alan Snyder. A portable compiler for the language C. MIT CAMBRIDGE PROJECT MAC, 1975.

112. Evgeny Stemasov, Enriko Rukzio, and Jan Gugenheimer, "The Road to Ubiquitous Personal Fabrication: Modeling-Free Instead of Increasingly Simple," in *IEEE Pervasive Computing, vol. 20, no. 1, pp. 19-27, 1 Jan.-March 2021*, DOI: https://10.1109/MPRV.2020.3029650

113. J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel. 1958. The problem of programming communication with changing machines: a proposed solution. In *Communications of ACM 1, 8 (Aug. 1958)*, 12–18. https://doi.org/10.1145/368892.368915

114. Peng Song, Bailin Deng, Ziqi Wang, Zhichao Dong, Wei Li, Chi-Wing Fu, and Ligang Liu. 2016. CofiFab: coarse-to-fine fabrication of large 3D objects. *ACM Trans. Graph. 35, 4, Article 45 (July 2016),* 11 pages. DOI: https://doi.org/10.1145/2897824.2925876

115. SVGNest, last accessed in May 2022, https://svgnest.com

116. Zhilong Su, Lujie Chen, Xiaoyuan He, Fujun Yang, and Lawrence Sass. 2018. "Planar structures with automatically generated bevel joints." *Computers & Graphics* 72 (2018): 98-105.

117. Thingiverse, last accessed April 2022, http://www.thingiverse.com

118. Three.js, last accessed April 2022, http://www.threejs.org

119. Shubham Tulsiani, Hao Su, Leonidas J. Guibas, Alexei A. Efros, and Jitendra Malik. "Learning shape abstractions by assembling

volumetric primitives." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2635-2643. 2017.

120. Nobuyuki Umetani, Takeo Igarashi, and Niloy J. Mitra. 2012. Guided exploration of physically valid shapes for furniture design. *ACM Trans. Graph.31, 4, Article 86 (July 2012),* 11 pages. DOI: https://doi.org/10.1145/2185520.2185582

121. Nobuyuki Umetani, Yuki Koyama, Ryan Schmidt, and Takeo Igarashi. 2014. Pteromys: interactive design and optimization of free-formed free-flight model airplanes. ACM Trans. Graph. 33, 4, Article 65 (July 2014), 10 pages. https://doi.org/10.1145/2601097.2601129

122. Uslan, I. CO2 laser cutting: kerf width variation during cutting. in *Proceedings of the institution of mechanical engineers, Part B: Journal of engineering manufacture* 219, no. 8 (2005): 571-577. DOI: https://doi.org/10.1243%2F095440505X32508

123. Kiril Vidimče, Alexandre Kaspar, Ye Wang, and Wojciech Matusik. 2016. Foundry: Hierarchical Material Design for Multi-Material Fabrication. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16).* ACM, New York, NY, USA, 563-574. DOI: https://doi.org/10.1145/2984511.2984516

124. Kiril Vidimče, Szu-Po Wang, Jonathan Ragan-Kelley, and Wojciech Matusik. 2013. OpenFab: a programmable pipeline for multi-material fabrication. In *ACM Trans. Graph*. 32, 4, Article 136 (July 2013), 12 pages. https://doi.org/10.1145/2461912.2461993

125. Tom Veuskens Florian Heller, and Raf Ramakers. "CODA: A Design Assistant to Facilitate Specifying Constraints and Parametric Behavior in CAD Models." *In Proceedings of the 47th Graphics Interface Conference on Proceedings of Graphics Interface 2021 (GI'21)*

126. Daniel Vogel, and Patrick Baudisch. 2007. Shift: a technique for operating pen-based interfaces using touch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07).* ACM, New York, NY, USA, 657-666. DOI: https://doi.org/10.1145/1240624.1240727

127. Ziqi Wang, Peng Song, and Mark Pauly. "DESIA: A general framework for designing interlocking assemblies." In *ACM Transactions on Graphics (TOG)* 37, no. 6 (2018): 1-14.

128. Long Wen, Xingchen Ye, and Liang Gao. "A new automatic machine learning based hyperparameter optimization for workpiece quality prediction." *Measurement and Control* 53, no. 7-8 (2020): 1088-1098. https://doi.org/10.1177/0020294020932347

129. Karl D.D. Willis, Cheng Xu, Kuan-Ju Wu, Golan Levin, and Mark D. Gross. 2010. Interactive fabrication: new interfaces for

digital fabrication. In Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction (TEI '11). Association for Computing Machinery, New York, NY, USA, 69–72. https://doi.org/10.1145/1935701.1935716

130. Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. 2019. Carpentry compiler. ACM Trans. Graph. 38, 6, Article 195 (December 2019), 14 pages. https://doi.org/10.1145/3355089.3356518

131. Jianhua Wu, and Leif Kobbelt. "Structure Recovery via Hybrid Variational Surface Approximation." In *Comput. Graph. Forum*, vol. 24, no. 3, pp. 277-284. 2005.

132. Kai Xu, Hao Zhang, Daniel Cohen-Or, and Baoquan Chen. 2012. Fit and diverse: set evolution for inspiring 3D shape galleries. ACM Trans. Graph. 31, 4, Article 57 (July 2012), 10 pages. https://doi.org/10.1145/2185520.2185553

133. Mingliang Xu, Mingyuan Li, Weiwei Xu, Zhigang Deng, Yin Yang, and Kun Zhou. 2016. Interactive mechanism modeling from multi-view images. *ACM Trans. Graph*. 35, 6, Article 236 (November 2016), 13 pages. DOI: https://doi.org/10.1145/2980179.2982425

134. Yong-Liang Yang, Jun Wang, and Niloy J. Mitra. "Mesh2Fab: Reforming Shapes for Material-specific Fabrication." *arXiv preprint arXiv:1411.3632* (2014).

135. B. S. Yilbas, "Effect of process parameters on the kerf width during the laser cutting process." *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture* 215, no. 10 (2001): 1357-1365.

136. Nur Yildirim, Matthew Franklin, Daniel Zeng, John Zimmerman, and James McCann. metaSVG: A Portable Exchange Format for Adaptable Laser Cutting Plans. In *Graphics Interface* 2022.

137. Nur Yildirim, James McCann, and John Zimmerman. 2020. Digital Fabrication Tools at Work: Probing Professionals' Current Needs and Desired Futures. Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376621

138. Ran Zhang, Thomas Auzinger, Duygu Ceylan, Wilmot Li, and Bernd Bickel. 2017. Functionality-aware retargeting of mechanisms to 3D shapes. ACM Trans. Graph. 36, 4, Article 81 (August 2017), 13 pages. https://doi.org/10.1145/3072959.3073710

139. Robert C. Zeleznik, Kenneth P. Herndon, and John F. Hughes. 2007. SKETCH: an interface for sketching 3D scenes. *In ACM*

SIGGRAPH 2007 courses(SIGGRAPH '07). ACM, New York, NY, USA, Article 19. DOI: https://doi.org/10.1145/1281500.1281530

140. Clement Zheng, Ellen Yi-Luen Do, and Jim Budd. 2017. Joinery: Parametric Joint Generation for Laser Cut Assemblies. In *Proceedings of the 2017 ACM SIGCHI Conference on Creativity and Cognition* (C&C '17). Association for Computing Machinery, New York, NY, USA, 63–74. DOI: https://doi.org/10.1145/3059454.3059459

141. Youyi Zheng, Xiang Chen, Ming-Ming Cheng, Kun Zhou, Shi-Min Hu, and Niloy J. Mitra. 2012. Interactive images: cuboid proxies for smart image manipulation. *ACM Trans. Graph. 31, 4, Article 99 (July 2012)*, 11 pages. DOI: https://doi.org/10.1145/2185520.2185595