Assembler^3: 3D Reconstruction of Laser-Cut Models

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

Hasso Plattner institute
University of Potsdam, Germany
Firstname.lastname@hpi.de

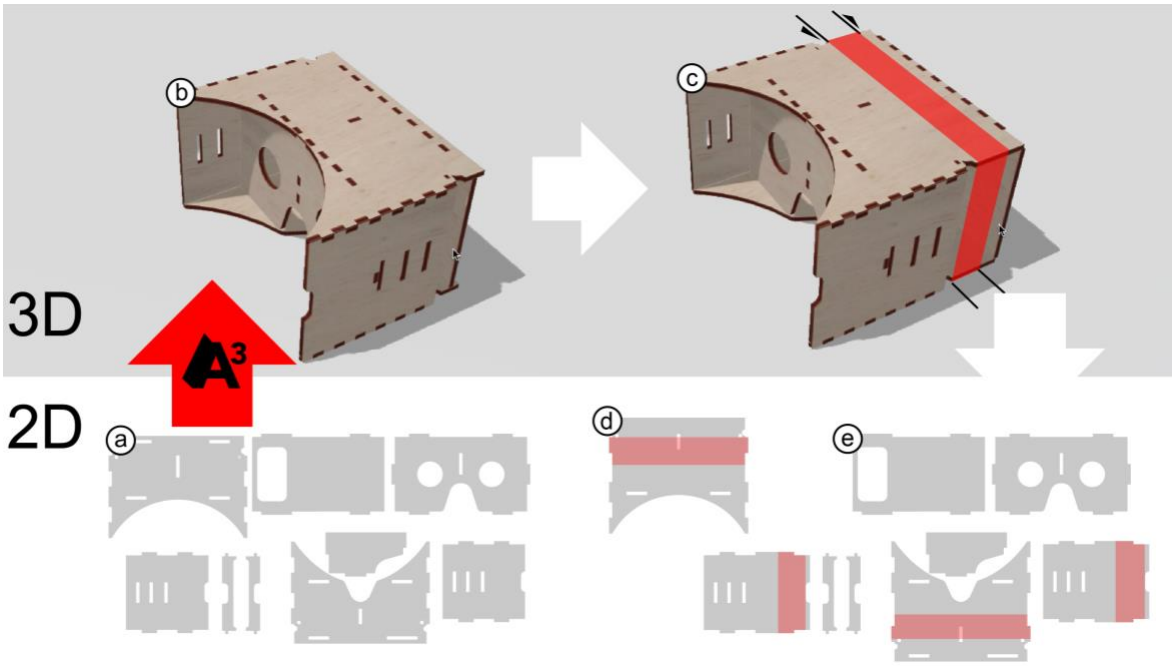

Figure 1: Assembler^3 allows users to modify 2D cutting plans by (b) rearranging them into a 3D model, where (c) users can now apply parametric manipulations using existing 3D editor kyub [3], before (d) converting back to 2D for cutting. In our study, this workflow allowed participants to apply parametric modifications 10x faster, more easily (6/7 vs. 2/7), and 26x more reliably, than when employing the traditional workflow of rewriting the 2D cutting plan directly.

1 ABSTRACT

We present Assembler^3 a software tool that allows users to perform *3D* parametric manipulations on *2D* laser cutting plans. Assembler^3 achieves this by semi-automatically converting 2D laser cutting plans to 3D, where users modify their models

using available 3D tools (kyub), before converting them back to 2D. In our user study, this workflow allowed users to modify models 10x faster than using the traditional approach of editing 2D cutting plans directly. Assembler[3] converts models to 3D in 5 steps: (1) plate detection, (2) joint detection, (3) material thickness detection, (4) joint matching based on hashed joint "signatures", and (5) interactive reconstruction. In our technical evaluation, Assembler[3] was able to reconstruct 100 of 105 models. Once 3D-reconstructed, we expect users to store and share their models *in 3D*, which can simplify collaboration and thereby empower the laser cutting community to create models of higher complexity.

CCS CONCEPTS • Human-centered computing~Human computer interaction (HCI)**;**

Additional Keywords and Phrases: laser cutting, personal fabrication, reuse, remixing

ACM Reference Format:

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 CHI Conference on Human Factors in Computing Systems (CHI '21), May 8–13, 2021, Yokohama, Japan. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3411764.3445453

2 INTRODUCTION

While laser cutters are capable of creating complex 3D objects [2], illustrated by the professionally designed model shown in Figure 2a (ugearsmodels.com). Laser-cut models created *by the community*, in contrast, do not reflect this potential: the most often fabricated model on the sharing site thingiverse.com, for example, consists of only 5 parts (Figure 2b).

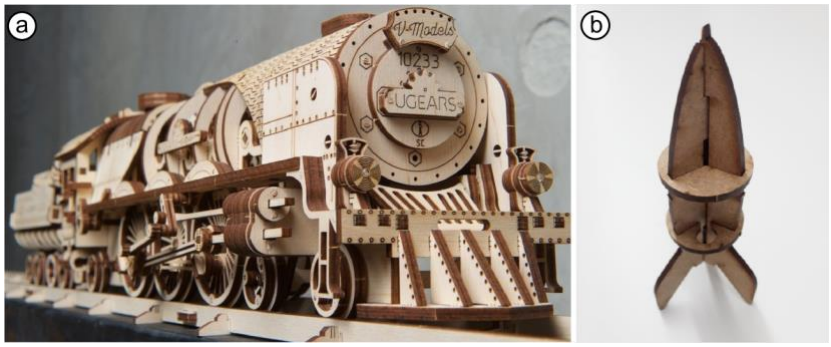

Figure 2: (a) This 3D models designed by a closed design teams (here ugearsmodels.com) consists of 538 parts. (b) The most often fabricated model shared on thingiverse.com (IDs 2440907), in contrast, is of much lower complexity.

We feel that this lack of complexity is not coincidental. Communities *are* capable of producing complexity (e.g. the open source code community [14] [23]), but this requires community members to be able to have efficient access to the shared project. In laser cutting, however, 95% of laser cut 3D models are still shared in the form of *2D* cutting plans—a format that makes 3D modifications all but impossible and thus makes it difficult to build on the work of others.

To illustrate this point, we use Figure 1 for visual reference. (a) A user wanting to accommodate the 2D cutting plan of this virtual reality headset for far sightedness might start by (d) extending the top plate of the model, but would then also have to realize that (e) three additional plates require modifications as well.

This sounds straightforward, but in order to figure out *which* plates to modify, the user would most likely (b) *mentally* reconstruct the 3D model, (c) *mentally* scale the model in 3D, and (e) *mentally* apply those changes to the 2D cutting plan. Only now could the user realize that the side plates of this particular model have to be scaled, but for example *not* the middle piece. In our user study, it took participants on average 24:45min to complete this particular task, they rated it as "hard" (6/7), and only 2/13 of the resulting cutting plans assembled into functional models.

In this paper, we present *Assembler³*. *Assembler³* is an interactive software tool that implements the "mental" workflow shown in Figure 1 as an actual*, software-based* workflow. (a) Assembler³ 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.

3 ASSEMBLER³ WORKFLOW

The main step implemented by Assembler³ is the reconstruction of a 3D model from a 2D cutting plan.

While we present our algorithm in full detail in the section 'The Algorithm of Assembler³', Figure 3 provides a preview. Assembler³ 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).

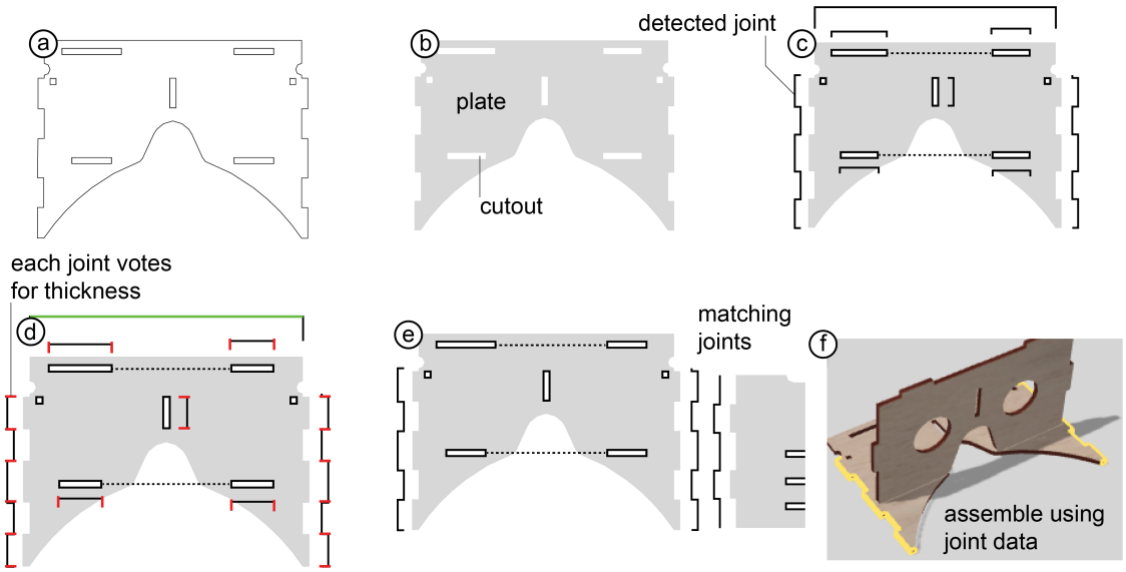

Figure 3: Pipeline of parsing the top plate of the VR headset (in reality it parses the entire SVG).

While we explain each of these steps in detail in the 'Algorithm' section, Figure 4 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).

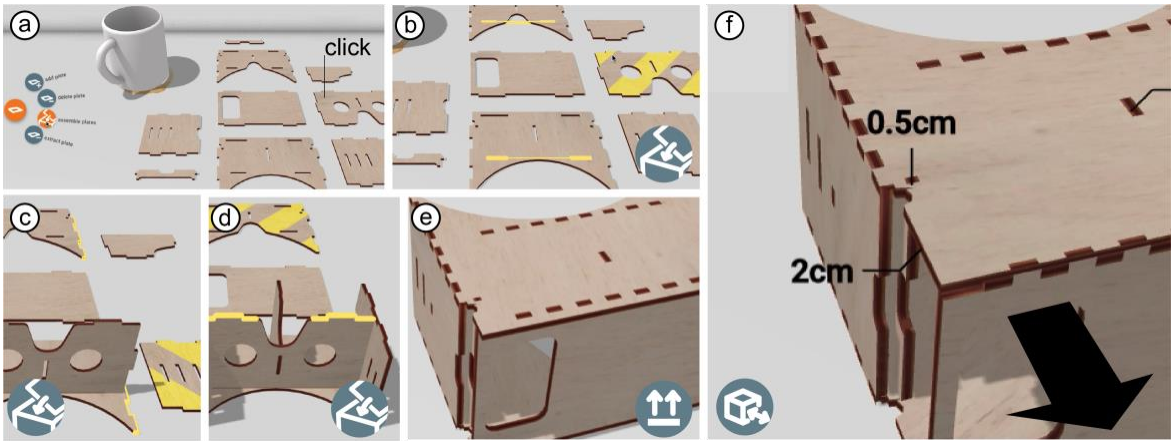

Figure 4: Reconstructing the 3D model of the VR headset. (a) When Assembler[3] loads the SVG, all plates are displayed in the 3D modeling environment, here kyub [3]. (b) The user clicked the *assemble tool* on the front piece. Assembler[3] responds by highlighting this plate (yellow stripes) and by highlighting 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 4c 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 5, 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 5: (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 1e, 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 6. 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 6: When users have converted the VR headset model to 3D *once*, the 3D model allows leveraging existing tools (such as kyub [3]) 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.

4 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 [10] 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 [21]). 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, Assembler³ 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 CONTRIBUTION, BENEFITS AND LIMITATIONS

In this paper, we make four contributions:

(1) We propose an algorithm, we call Assembler³, that allows users to make parametric modifications to 2D cutting plans by allowing them to convert their cutting plans to 3D models. (2) To show its relevance and to leverage existing support to modify and export models, we demonstrate this algorithm by implementing it as an extension of existing modeling software, kyub. (3) We compare our approach to the 2D editing workflow and find that the 3D workflow is 10x faster and 26x less error prone. And finally, (4) we reconstruct 100 models found online to 3D models that can be modified and find that Assembler³ achieves a coverage 95.2% of supported models.

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. (3) Currently, models in which all angles between plates are non-straight (3.7% in our evaluation) cannot be reconstructed with Assembler³.

6 RELATED WORK

Our work builds on research into reconstruction of 3D models, puzzle solvers, sharing and remixing of models for 3D printing, and collaborating on laser-cut models.

6.1 Reconstruction of 3D models and non-parametric to parametric

In the field of computer vision, one approach to reconstructing 3D models is to reconstruct based on one or more 2D images: *3-Sweep* [5] reconstructs a 3D model from a single photograph, and Xu et al. [24] reconstruct 3D mechanisms from multi-view images and even reconstruct the movement of the mechanisms from a corresponding video. Agrawal et al. [1] reconstruct 3D models of historical monuments by aggregating massive amounts of pictures found online.

In a very different context, archaeologists have a tradition of leveraging computing methods to reconstruct models of excavated artefacts both to find out what it could have looked like (aka "object completion") [17] and as a way to figure out how to physically re-assemble the model [13].

Once a 3D model has been created, a next *InverseCSG* [8] lets users turn a non-parametric 3D model (e.g. STL) into parametric models, allowing users to modify the models using advanced 3D operations. It reconstructs what geometric primitives and CSG operations were required to get to an input 3D model. Assembler³ solves this bottom up, the primitives are given (i.e. the plates) and Assembler³ reconstructs how they come together.

6.2 Solving combinatoric issues, puzzle solvers

When arranging 2D laser-cut plates in 3D Assembler³ requires users to solve a combinatoric problem. Combinatoric problem solving has a long history in the algorithmic solving of *2D* puzzles. Demaine & Demaine [7] show that jigsaw puzzles are in principle NP-hard. Nevertheless, Sholomon et al. [22] show that large jigsaws can be solved using a genetic algorithm, given good "compatibility" metrics describing how well pieces fit together.

Chen et al. [4] solve puzzles based on a complete joint map between any kind of "objects". We feel their approach to 3D reconstruction might also be worth investigating in the context of laser cut models.

6.3 Sharing and remixing benefits from models being in a parametric format

Online sharing repositories, such as thingiverse, GrabCAD, 3DContentCentral etc. contain millions of models. Flath et al. [10] studied sharing behavior on thingiverse in detail and particularly target the importance of the *customizer* (a tool to modify other people's parametric models) to foster the complexity of models as well as the concept of remixing across users. Grafter [18] is a software tool targeted to facilitate such forms of online remixing in the context of mechanisms, and the *PARTS framework* [12] enables users to specify mechanical parametric models, which again fosters remixing and modifying other people's models.

6.4 Sharing and remixing in laser cutting

We see one of the key contributions of Assembler^3 in its ability to move the current 2D sharing format to 3D, which we feel will empower collaboration. Earlier work in laser cutting has dealt to address other hurdles inherent to the current 2D sharing formats. SpringFit [19] made the case that 2D cutting plans are not inherently reproducible, as they encode the kerf (material removed) of the laser cutter. SpringFit and KerfCanceler [20] both tackle the problem of how to enable generic 2D cutting plans to lead to well-defined physical output; their approach is to modify mechanisms and joints so as to become invariant to kerf.

A major milestone towards collaborating in 3D laser cutting was the introduction of 3D exchange formats. This took place as a side-effect of the introduction of 3D modeling environments for laser cutting, specifically FlatFab [6]. CutCAD [11] follows a similar approach but uses a combination of 2D input and 3D visualization. Kyub [3] further increases the level of abstraction by representing laser cut models as volumetric models. We thus decided to integrate into one of these software systems (kyub) so as to build on the state of the art.

7 THE ALGORITHM OF ASSEMBLER^3

In order 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 1 provides an overview of the first 4 steps.

Algorithm 1: ParseSVG

```
Input: 2D cutting plan svg,
Output: list of plates plates, hash of joints jointHash, and material thickness 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
```

7.1 Plate detection

Assembler³ starts by segmenting the 2D cutting plan (commonly in SVG format) into plates and cutouts. Assembler³ 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, Assembler³ breaks down all geometry to line segments (any cutting path in an SVG can be a polygon, polyline, path, and more). Assembler³ 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, Assembler³ would not be able to determine that both of them are plates. To detect this issue, Assembler³ checks if there are any points where more than two line segments come together. If this is the case, Assembler³ 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 [16].

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) Assembler³ achieved a 100% success rate in recognizing plates in our technical evaluation.

7.2 Joint detection

Assembler³ now detects joints. To illustrate, Figure 7, shows the joint "candidates" Assembler³ 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 Assembler³ will remove these incorrect candidates after material thickness detection.

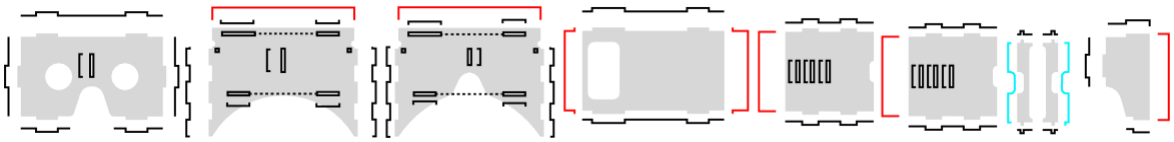

Figure 7: Joint detection on the VR headset. The red lines are false positives, which will get adjusted later, and the blue lines are joints that will receive a low probability because of their odd shape.

Assembler³ 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 8a, finger joints, for example, follow the pattern left/right/right/left.

Assembler³ 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 8 at the example of a finger joint, Assembler³ 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, Assembler³ will also accept imperfect renditions; it will assign these lower "joint probabilities" though. The feature shown in Figure 8b, 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, Assembler³ is still willing to consider it a finger joint, but with a low probability. (c) Furthermore, Assembler³ increases that probability when it detects repetitions of a pattern.

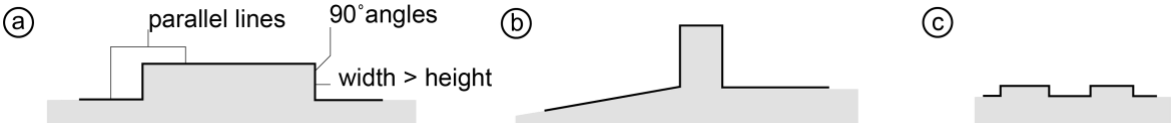

Figure 8: (a) The ideal finger joint has 90-degree angles, a top line parallel to the edge and has a bigger width than height (b) this finger joint has a much lower probability, it could just as well be some aesthetic feature of the model? (c) repetitions of a pattern increase its probability.

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 technical evaluation).

7.3 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 9 shows examples of three types of joints and illustrates which line segment is considered as vote for material thickness.

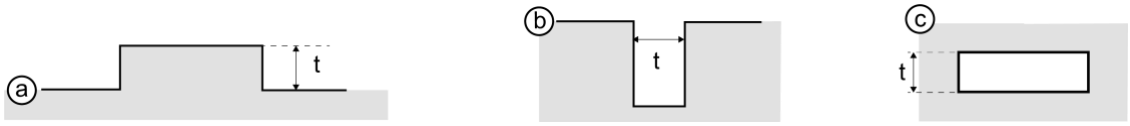

Figure 9: Each joint votes for a material thickness labeled "t" in this figure. (a) finger joint, (b) cross joint, and (c) a t-joint.

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 10. The idealized joint then votes with reduced weight.

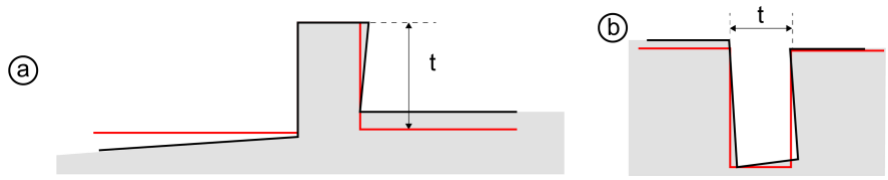

Figure 10: Assembler³ first idealizes non-ideal joints and then haves them vote for the idealized material thickness but with an additional penalty to reduce their impact. The red overlay represents the idealized version of the joints above.

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. Assembler³ 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, Assembler³ 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 Assembler³ 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, Assembler³ 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. Assembler³ uses that information to disambiguate the colors in the cutting plan. Assembler³ 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 11.

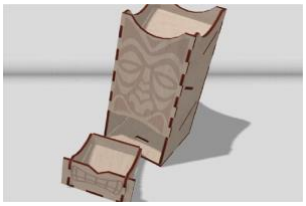

Figure 11: Assembler^3 imports engravings as ornaments on the plates.

7.4 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 12, Assember^3 determines whether two joints fit by checking that joint types match (finger, cross, t-joint) and then comparing their respective shapes (signature) to determine which joints interlock.

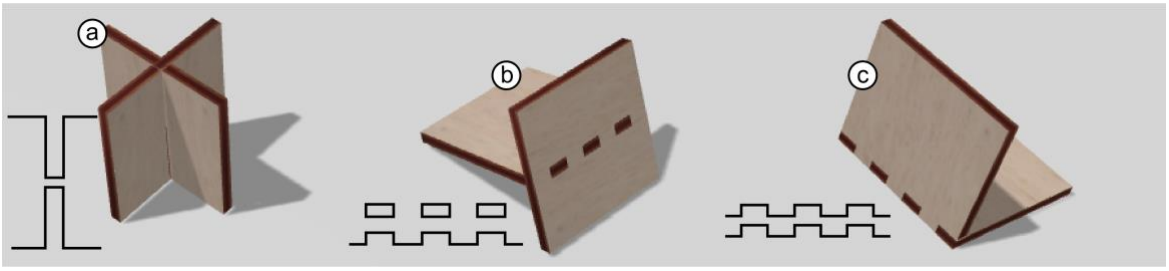

Figure 12: Joint types (a) cross joints, (b) t-joints and (c) finger joints

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 13, 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.

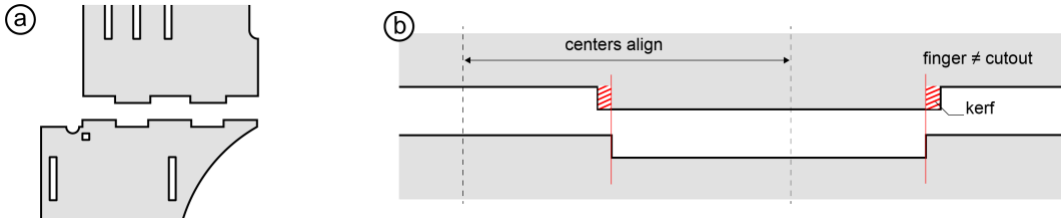

Figure 13: (a) Two plates of the VR headset that assemble together. (b) On closer inspection, the fingers of the one joint do not match the cutouts of the other because of the material removed by the laser(aka kerf). However, the centers of joints *have to* align, so Assembler^3 hashes the sum of the width of a cutout and a finger as the signature for a finger joint.

For a given joint, Assembler^3 now uses the hash to look up that joint. 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 their joint probability.

7.5 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 now supports users in assembling the model interactively. It does so by highlighting which joints match to joints on user-selected plates, as already presented in section 'Assembler^3 workflow'. 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 found on thingiverse.

8 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.

8.1 Coverage: Assembler^3 allows assembling 95.2% of supported 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 14. 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 14: 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 15b: 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 15c). We plan to extend Assembler^3 with a more advanced constraint solver to also handle such cases.

Figure 15a 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.

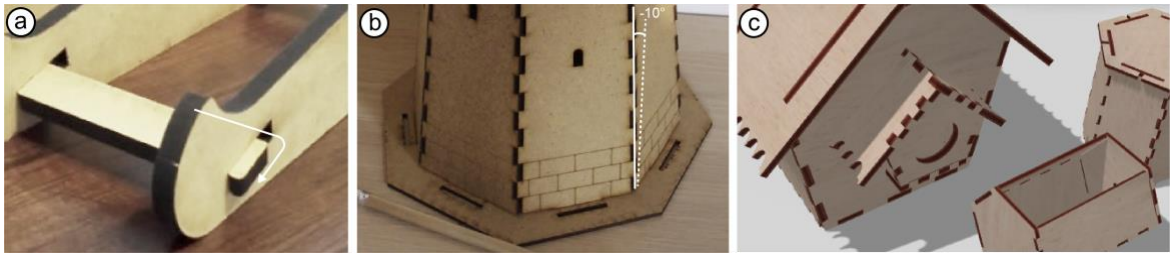

Figure 15: (a) This model has cross joints that assemble into a t-joint, Assembler^3 fails to pair these up. (b) when all plates are at a non-straight angle with each other, Assembler^3 cannot reconstruct the model. (c) luckily, most of the models with plates with non-straight angles still provide sufficient constraints to be reconstructed.

8.2 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 16. 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.

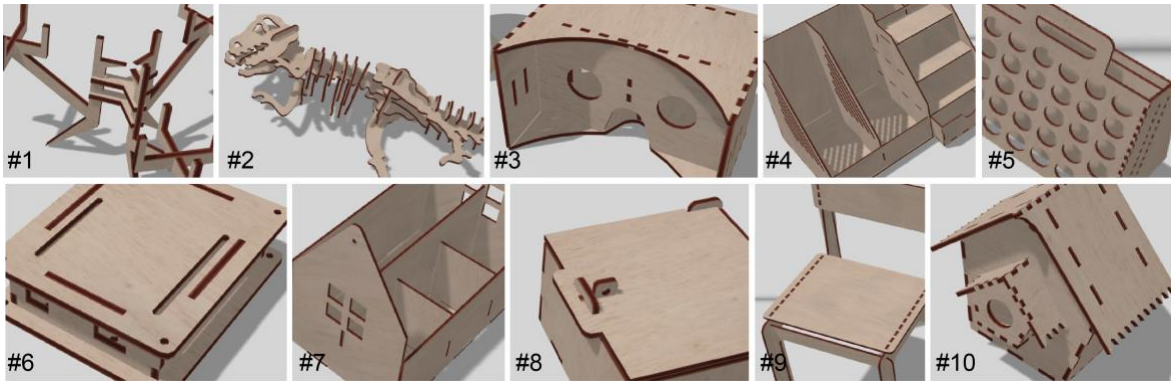

Figure 16: Models used to validate the accuracy of the steps of the algorithm.

Table 1 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.

Table 1: Success rate of algorithm steps.

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**%

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.

8.3 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.

Table 2: Average performance of Assembler^3 when assembling each model 1000 times. Times are in ms (stndev in parentheses)

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 (12.05)	525.96 (101.34)	34.22 (72.25)	65.30 (13.60)	174.77 (223.5)	49.33 (11.62)	27.49 (47.78)	12.83 (6.96)	87.65 (187.92)	77.51 (9.45)
match	1.72 (3.00)	6.65 (10.54)	5.13 (16.66)	5.59 (11.90)	6.59 (23.98)	2.05 (4.42)	4.60 (10.07)	2.47 (2.39)	11.59 (7.78)	11.41 (27.37)
assemble	3.12 (1.69)	3.43 (2.79)	4.88 (8.26)	4.21 (5.91)	10.50 (50.88)	3.50 (3.36)	3.99 (5.44)	3.19 (2.97)	9.50 (9.42)	6.09 (16.22)

The steps where the user interacts with the model are all efficient (average of 12 ms 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.

9 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 1 and stretched the model to accommodate for far-sightedness.

9.1 Task

Participants' task was to modify the VR headset as shown in Figure 1, 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 1). Participants were allowed to have a look at these pictures any time during the experiment.

9.2 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 17 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.

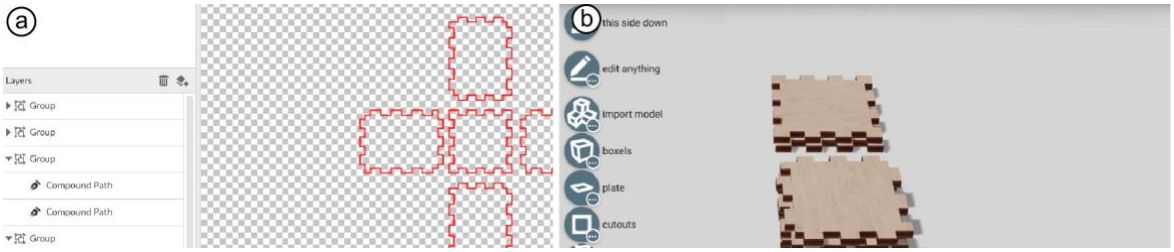

Figure 17: (a) In the baseline condition, users were shown how to modify a boxel in gravit.io, (b) they were shown a video of the same workflow using Assembler[3]

9.3 Participants

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

9.4 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.

9.5 Results

Figure 18 summarizes the task times and error rates of the 13 individual participants.

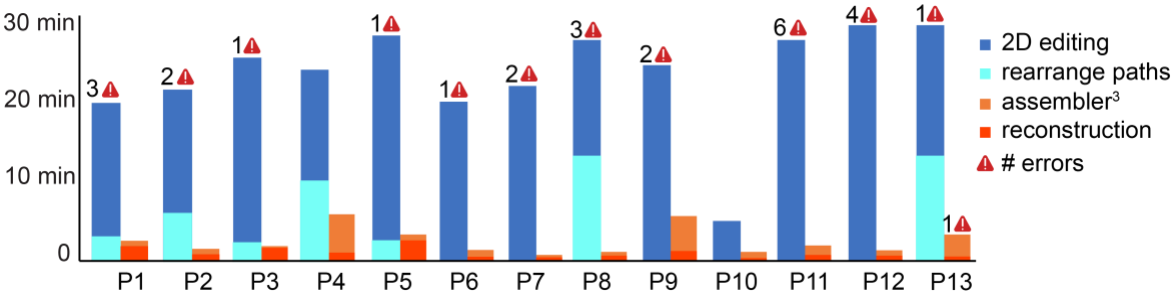

Figure 18: results of the experiment.

Participants performed 10.0x faster when modifying the model using Assembler[3] (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 Assembler[3] 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 Assembler[3] 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 Assembler[3] 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 19) so as to better reflect the 3D nature of the assembly, as shown in Figure 18; 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.

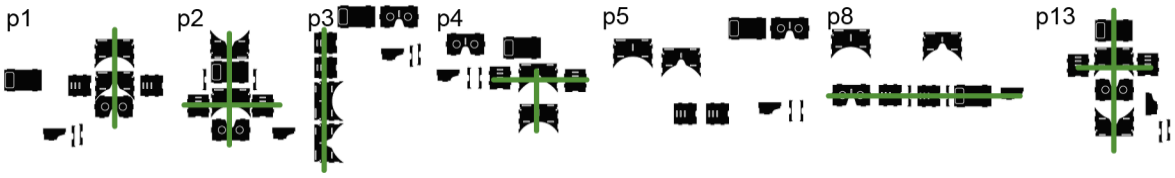

Figure 19: 2D layouts as used by our participants. Green lines indicate plates that are laid out so that their matching joints line up, an indicator that they leverage the space in the 2D editor to help them reconstruct the model in their heads. P5 also spent time laying out paths but grouped by similarity 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 side-note, 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 Assembler³ as it takes the burden of learning about the model away from the user.

9.6 Discussion

We conclude that (1) Assembler³ enabled 11/13 participants to make modifications to the model who were not able to do this without and (2) Assembler³ achieves a speed-up of 10x and (3) the workflow with Assembler³ results in 26x less errors.

10 CONCLUSION

In this paper, we presented *Assembler³* a software tool which allows users to perform parametric manipulations on 2D cutting plans of 3D models in 3D. Assember³ achieves this by allowing users to convert 2D laser cutting plans to *3D* models, modifying their models using available 3D tools (kyub), before converting them back to 2D. This makes users 10x faster and results in 26x less errors than the traditional approach of editing 2D cutting plans directly.

Once a laser cut model has been 3D reconstructed using *Assembler³* 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 to transition to a more sharing-friendly and collaboration-friendly format—which must be 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.

REFERENCES

[1] 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

[2] Patrick Baudisch and Stefanie Mueller. 2017. Personal Fabrication, Foundations and Trends in Human–Computer Interaction: Vol. 10: No. 3–4, pp 165-293. DOI: http://dx.doi.org/10.1561/1100000055

[3] 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 *2019 CHI Conference on Human Factors in Computing Systems Proceedings (CHI 2019)*, May 4–9, 2019, Glasgow, Scotland, UK. ACM, New York, NY, USA. DOI: https://doi.org/10.1145/3290605.3300796

[4] Yuxin Chen, , Leonidas J. Guibas, and Qi-Xing Huang. "Near-optimal joint object matching via convex relaxation." *arXiv preprint arXiv:1402.1473* (2014).

[5] 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

[6] 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

[7] Erik D. Demaine, and Martin L. Demaine. "Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity." *Graphs and Combinatorics* 23, no. 1 (2007): 195-208.

[8] 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

[9] 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.

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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.

[15] 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

[16] David E. Muller, and Franco P. Preparata. "Finding the intersection of two convex polyhedra." *Theoretical Computer Science* 7, no. 2 (1978): 217-236.

[17] 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

[18] Thijs Roumen, Willi Müller, and Patrick Baudisch. 2018. Grafter: Remixing 3D-Printed Machines. 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

[19] Thijs Roumen, Jotaro Shigeyama, Julius Cosmo Romeo Rudolph, Felix Grzelka, and Patrick Baudisch. SpringFit Joints and Mounts that Fabricate on Any Laser Cutter. In *Proceedings of the 32th annual ACM symposium on User interface software and technology* (UIST '19). ACM, New York, NY, USA, DOI: https://doi.org/10.1145/3332165.3347930

[20] Thijs Roumen, Ingo Apel, Jotaro Shigeyama, Abdullah Muhammad, and Patrick Baudisch. “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, DOI: https://doi.org/10.1145/3379337.3415895

[21] Thijs Roumen, Thingiverse Analysis Script, last accessed August 2020 https://github.com/ThijsRoumen/thingiverse-browser

[22] Dror Sholomon, Omid David, and Nathan S. Netanyahu. "A genetic algorithm-based solver for very large jigsaw puzzles." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1767-1774. 2013.

[23] Molly McLure Wasko, and Samer Faraj. "Why should I share? Examining social capital and knowledge contribution in electronic networks of practice." *MIS quarterly* (2005): 35-57. DOI: https://doi.org/10.2307/25148667

[24] 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