PLATENER: GENERATING 2D LASER CUTTABLE CONSTRUCTION PLANS FROM 3D MODELS

3D MODELLE AUS DEM LASERCUTTER – AUTOMATISCHES ERSTELLEN VON 2D SCHNEIDEPLÄNEN AUS 3D VORLAGEN

DANIEL-AMADEUS J. GLÖCKNER, SVEN MISCHKEWITZ, DIMITRI SCHMIDT, KLARA SEITZ, LUKAS WAGNER

A thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science in IT Systems Engineering

Supervised by Stefanie Müller and Prof. Dr. Patrick Baudisch

Human-Computer Interaction Group
Hasso Plattner Institute
University of Potsdam

July 25, 2016

Daniel-Amadeus J. Glöckner, Sven Mischkewitz, Dimitri Schmidt, Klara Seitz, Lukas Wagner :
Platener
July 25, 2016

Supervised by Stefanie Müller and Prof. Dr. Patrick Baudisch

ABSTRACT

We present platener, a web service that converts 3D models originally designed for 3D printing to a representation that can be fabricated using a laser cutter.
Designing three-dimensional objects for laser cutting is difficult today, because it requires users to break down their idea into a set of two-dimensional parts and to add appropriate joints between these. Both aspects require substantial engineering knowledge. Our service eliminates the need for engineering knowledge by offering an alternative workflow. In this workflow, users model their 3D idea using a 3D editor of their choice; users then obtain the 2D representation required for the laser cutter by converting their 3D model with platener. As a side effect, this workflow allows converting models from 3D printing model repositories; this is beneficial, as those repositories already hold many more models as there currently are models for laser cutting.

platener creates two-dimensional cutting plans as follows. First, it identifies flat surfaces in the 3D model. Second, it locates laser cuttable elements by finding plates of appropriate thicknesses as pairs of parallel surfaces with appropriate distance. Third, the software generates joints to connect the plates.
The software system is designed primarily with the objective of converting functional objects, such as mechanical tools and low-fidelity prototypes. Unlike existing conversion systems, such as *123DMake* and *Brickify* it focuses on maintaining the functional aspect and stability of the input model while still enabling the benefits of laser cutting, such as the ability to use a wider choice of materials.

ZUSAMMENFASSUNG

Wir präsentieren platener, einen web service, der 3D Modelle, die ursprünglich für den 3D Druck gedacht waren, in eine Representation umwandelt, die mit einem Lasercutter hergestellt werden kann.
Das Designen von drei-dimensionalen Objekten für den Lasercutter ist heutzutage noch schwer, denn der Nutzer muss seine Idee in zwei-dimensionale Teile aufspalten und zwischen ihnen Verbindungen hinzufügen. Beide Aspekte erfordern erhebliche technische Vorkenntnisse. Unser Webservice eliminiert die Notwendigkeit von solchem technischen Vorwissen durch die Bereitstellung eines alternativen Vorgangs. In diesem Vorgang modelliert der Nutzer seine Idee in 3D mit einem 3D Editor seiner Wahl; Anschließend erhalten Nutzer die 2D Representation, die für den Lasercutter benötigt wird, dank der Konvertierung des 3D Modells von platener. Ein Seiteneffekt ist, dass dieser Vorgang die Konvertierung von Modellen aus bereits existierenden 3D Druck Modell-Sammlungen ermöglicht; Dies ist sehr nützlich, da solche Sammlungen bereits mehr Modelle beinhalten, als aktuell für den Lasercutter verfügbar sind.

platener erstellt zwei-dimensionale Schnittpläne wie folgt: Zunächst werden ebene Flächen im 3D Modell erkannt. Als zweites werden daraus lasercutbare Elemente extrahiert durch das Finden von Platten mit passenden Dicken. Platten bestehen aus einem Flächenpaar, die in einem angemessenen Abstand zueinander parallel stehen. Letztlich erzeugt die Software Verbindungen, um die Platten zu verknüpfen.
Das Softwaresystem wurde hauptsächlich zum Zweck der Konvertierung von funktionalen Objekten, wie mechanischen Werkzeugen und Prototypen erstellt. Anders als bei bisherigen Konvertierungssystemen, wie *123DMake* und *Brickify*, fokussiert sich unser System darauf die Funktionalität und Stabilität des Modells zu erhalten; Währenddessen können trotzdem die Vorteile von Lasercutting genutzt werden, wie die Möglichkeit aus einer breiten Palette an vielfältigen Materialien zu wählen.

CONTENTS

LIST OF FIGURES

1

INTRODUCTION

1.1 FABRICATION MACHINES ENABLE ANYONE TO BRING THEIR IDEAS TO LIFE

People always had lots of creative ideas, regardless their background or skill set. Though back in time it was difficult realizing these due to the lack of specific knowledge or tools. For example in the field of engineering, when one has an idea how airplanes fly better (Figure 1a). There are ideas concerning creative processes, like designing a fashion line for shoes which offer correct fit to the feet of its wearer (Figure 1b). Even in health care broad ideas evolve. In 2013 Jake Evill presented *Cortex*, a customizable plaster that is "fully ventilated, super light, shower friendly, hygienic, recyclable and stylish"[1]. Cortex is shown in Figure 1c.

Before, it was hard to shape these ideas into a physical reality. But the way ideas are turned into reality is changing. Today machines exist that, connected to a computer, will take over the fabrication of objects. Users of fabrication machines can focus on ideas instead of requiring and acquiring additional skills to actually fabricate.

Such a fabrication machine is a 3D printer. 3D printing is an emerging technology spreading across the consumer market. According to *Wohlers Associates*, in the year 2015 the 3D printing market was worth 6.5 billion USD. Their prognosis estimates a roughly 300% growth in the next five years [6]. With 3D printers literally any user can produce any free-form object with a button press. A common technique in 3D printing is fused deposition modeling (FDM). With the FDM technique thermoplastic material is heated and then pressed through a nozzle, mounted on a print head. Material is extruded layer by layer to create 3D objects [19].

Though 3D printing brings a substantial progress to the field of fabrication, it shows two flaws: 3D printing is slow, and it is limited in its materials. Even small models need a couple of hours to be printed. The miniature bird house (3.5 x 3.5 x 5.5cm) in Figure 2 printed for three hours[2]. In its original dimensions (20 x 20 x 30cm) it requires

1 http://www.evilldesign.com/cortex

2 Printed with the *Ultimaker 2+*, https://ultimaker.com/en/products/ultimaker-2-plus

about two days of printing time[3]. The FDM technique requires material to be extruded through a nozzle. Thus the material is melted and loses its original structure and characteristics, like aesthetics or haptics.

Another fabrication machine is the laser cutter, shown in Figure 3a. A laser traces the outlines of a planar cutting plan, producing plates which are cut out from flat materials (Figure 3b). Such materials are wood or acrylic. High-performance devices can cut textile, metal, and stone. The cutting process requires very short time, compared to the printing time of a 3D printer. Figure 3c and Figure 3d show how the cut-out plates are assembled to a 3D object. The entire fabrication is completed in several minutes.

With a laser cutter low-fidelity functional objects are fabricated. Low-fidelity means producing functional objects that fulfill their purpose, but lack the amount of detail a full-fledged product would have. If there is a minor relevance to structural details of the designed object, the laser cutter is the perfect choice. Using a laser cutter for low-fidelity objects will increase the overall efficiency of the fabrication process. A user can iterate on objects several times a day to achieve better results. Such a functional object could be a quadcopter produced from wood, as shown in Figure 4.

With a laser cutter users can design objects with decorative aspects as well (Figure 5). The fabricated object benefits from the original characteristics of the material.

When creating 3D objects with a laser cutter, users face two challenges: They need a vast amount of spatial orientation and craft men's knowledge about the used materials. The final object has to be assembled from the cut-out plates. The user has to know where the plates will resemble on the object in 3D space. Then, the user has to attach the plates to each other. In the case of wooden plates this would require knowledge about carpentry. In Figure 3d we combined the plates of a bird house with finger joints. When creating cutting plans manually for larger objects, detailed work is required. All connections have to be positioned perfectly. Otherwise the plates stuck or fall apart during assembly.

In this thesis, we present a solution which enables every user to produce three dimensional objects with a laser cutter. We produce cutting plans from 3D models automatically. We improve the creation process of cutting plans, so that users do not require any additional skills. Similar to 3D printing we begin the fabrication process with a 3D model. Our software system *platener* converts the 3D model to a 2D cutting plan. *platener* analyzes the geometries and approximates

3 Estimated with the *Cura* 2 3D printer software, https://ultimaker.com/en/products/cura-software

the model with plates. The system is aware of common materials used with a laser cutter. The software generates connections between the plates. All connections are calibrated, meaning the plates hold together without using any glue or other means of attachment. With our software users fabricate 3D models easily while benefiting from the working speed of a laser cutter and free choice of materials. Users do not require additional skills.

We present our system architecture in Chapter 4 ARCHITECTURE and give an overview of the most important data structures in Chapter 5 PROCESSING PIPELINE. From Chapter 6 APPROXIMATION to Chapter 10 CURVES we explain the conversion process of 3D models to 2D cutting plans.

1.2 RELATED WORK

platener Beyer et al. [7] did the groundwork for this thesis. Their work introduced the concept of converting 3D models to laser cuttable SVG files as a low-fidelity fabrication technique. Our work improves the introduced methods, while we focused on building a user-centered conversion service.

BRICKIFY Silber et al. [24] present a low-fidelity fabrication approach using 3D printers and *LEGO*^{TM}[4] bricks. Similar to *platener* they use a 3D editor to convert input models. The conversions result in hybrid models consisting of printed material and bricks. They improve fabrication time noticeably. They built a user-oriented web service using WebGL. The implementation presented in this thesis is based on the implementation of brickify. brickify is grounded on previous work by Mueller et al. [16].

LASERORIGAMI Mueller et al. [15] present a low-fidelity prototyping system for creating 3D models with laser cutters. Though they have the same goal as *platener*, they use a different approach. LaserOrigami cuts and folds the resulting 3D object from a single piece of acrylic material. Mueller et al. provide an editor for cutting plans which incorporates the additional laser cutter instructions for folding. The objects produced by *platener* need a manual assembly step. In return *platener* allows fabricating objects from a 3D template.

SKETCHCHAIR With the software SketchChair novice users can design and fabricate their own chair using a laser cutter. Similar to *platener* Saul et al. [22] provide a system to produce cutting plans for

4 http://www.lego.com

functional objects from 3D models. These 3D models are generated from two dimensional sketches which are drafted in an editor. In contrast to SketchChair, *platener* attempts to convert any given 3D model into a laser cuttable template.

123DMAKE Autodesk[5] ships a 3D editor which converts 3D models into slices[6]. These slices can be cut with a laser cutter and stacked onto each other to approximate the original model. This method solely provides an approximation of the external shape of the object. Though *platener* provides this technique as well, we focus on maintaining the functional aspect of the input model.

5 http://www.autodesk.de/
6 http://www.123dapp.com/make

(a) A self-fabricated model airplane.

Source: http://thingiverse-production-new.s3.amazonaws.com/renders/05/d7/e9/e9/88/Take_Off_03_preview_featured.jpg (visited on July 25, 2016)

(b) A shoe with correct fit to its wearer.

Source: © Olivier van Herpt, http://oliviervanherpt.com/img/3d-printed-shoes-foot.jpg (visited on July 25, 2016)

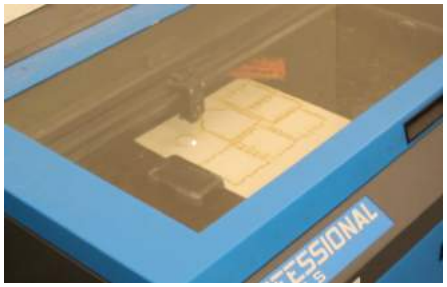
(c) *Cortex*, a fully ventilated plaster.

Source: © Jake Evill, http://www.evilldesign.com/image/41499/2000x0-5/p18qh3vpntsq91sg21a261dr114dc6.jpg (visited on July 25, 2016)

Figure 1: With fabrication machines, broad ideas come to life.

Figure 2: A 3D printed bird house.

(a) A laser cutter.



(b) The laser cuts plates from wood.

(c) Plates are assembled into a 3D object.

(d) The wooden bird house is connected with finger joints.

Figure 3: Fabrication steps for building a bird house with a laser cutter.

Figure 4: A laser cutted quadcopter.

Source: http://wedreamabout.com/wp-content/uploads/2016/01/drone-plywood-mixedbg-2.jpg (visited on July 25, 2016)

Figure 5: Wooden designer glasses built with a laser cutter.

Source: https://i.ytimg.com/vi/q0dDhesoC08/maxresdefault.jpg (visited on July 25, 2016)

2

USERINTERACTION

As our software is a converter, we minimized the required user interaction.

2.1 DRAG N DROP AS MAIN INTERACTION

Because the main function of our web service is converting custom objects, uploading a model is made easy by providing a drop area. This spread out over the whole web site. Alternatively we provide a upload button.

2.2 THE LANDING PAGE COMBINES CONVERSION AREA WITH REPOSITORY

The landing page is horizontally divided into two parts. At the top of the page is the conversion and preview area. Underneath this we show a repository of known 3D models (see Figure 6).

2.2.1 *The Conversion and Preview Area Provides the Main Functionality*

The conversion and preview area shows the currently selected or uploaded model. It also provides some conversion settings at the bottom left and a visualization selection at the bottom right.

2.2.1.1 *Conversion Settings for Custom Material Selection*

To change which materials should be used for the conversion, the user can select multiple plate thicknesses in the top selection drop down (see Figure 7a). Additionally the drop down below provides the choice of the base material. Acrylic, wood, cardboard and paper are available yet (see Figure 7b). The *upload .stl* button adds a discoverable way of uploading an own model for conversion, unlike drag an drop as upload possibility.

At the bottom we placed the *download .svg* button. If the user clicks this, the download is not started directly. It opens the download dialog (see Figure 7c).

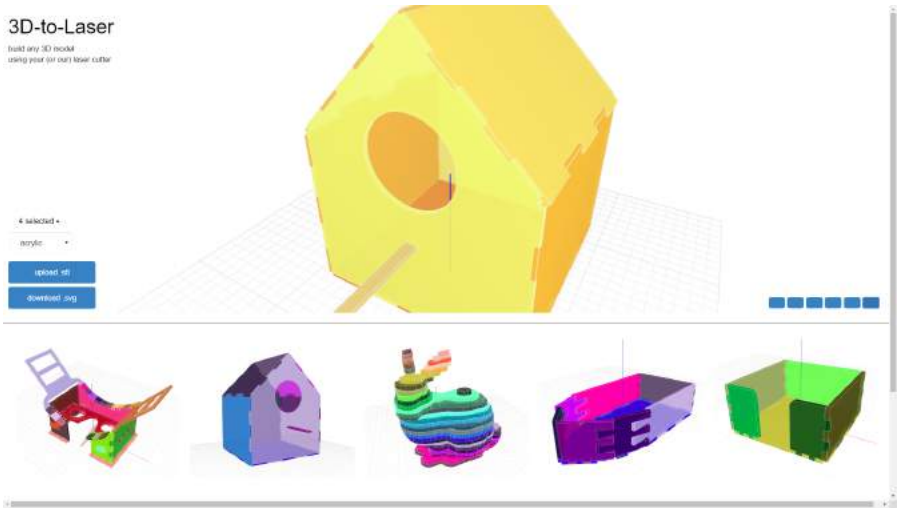
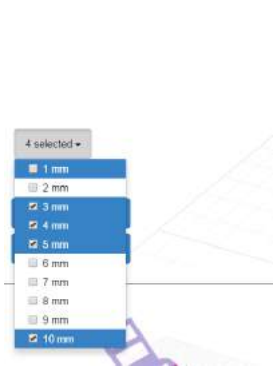
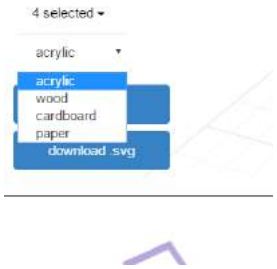


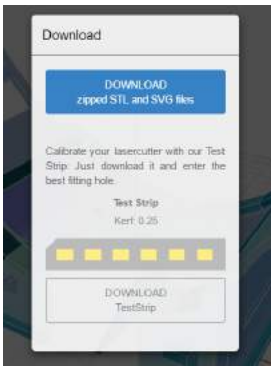
Figure 6: The landing page with the conversion and the repository area.



(a) The thickness selection provides several options to choose from and combine.



(b) Currently this four materials are supported by our software.



(c) The download dialog also provides the possibility to calibrate the SVG file to the used laser cutter.

Figure 7

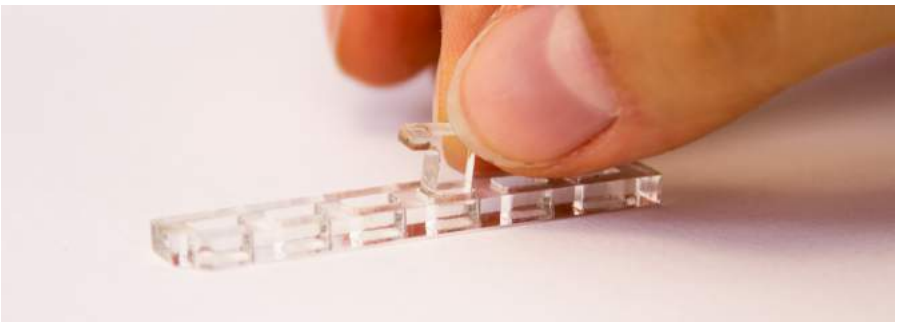


Figure 8: The test strip with the spike.

The download dialog contains the *DOWNLOAD zipped SVG files* button. This directly starts the download of the zipped result of the conversion, including one SVG file for each used plate thickness. Underneath this the dialog shows a short explanation of the test strip (see 2.2.1.2 The Test Strip Enables Easy Kerf Error Reduction) and an interactive test strip. It makes it possible to select the best fitting hole by clicking it. To create the corresponding strip the dialog provides also a download button for this.

2.2.1.2 *The Test Strip Enables Easy Kerf Error Reduction*

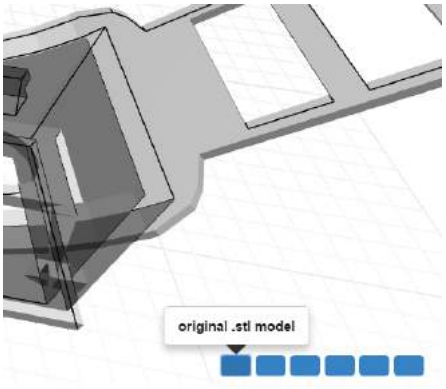
The test strip is a small cutting plan with multiple rectangular holes and a small spike (see Figure 8). The holes all have slightly different widths. The User can try in which hole the spike fits best. If he selects the corresponding hole in the user interface, our software calculates the kerf of the laser cutter. This makes it easy to determine the kerf without special equipment. We use the result for the calibration (see 5.2 Pipeline Steps);

2.2.1.3 *The Visualizations Helps to Understand the Conversion*

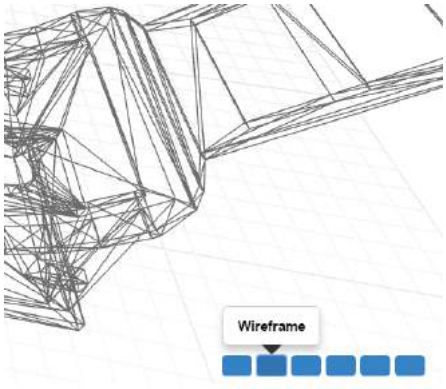
The visualization selection contains six buttons where each of this activate another visualization. The selection includes in this order: the original .stl model, a wireframe view, a visualization of the coplanar faces, a view of the generated plates, a visualization of the plates with highlighted bend plates and one of the final model including the generated joints (see Figure 9).

2.2.2 *The Repository Guarantees Fast Access to Lasercut-able Models*

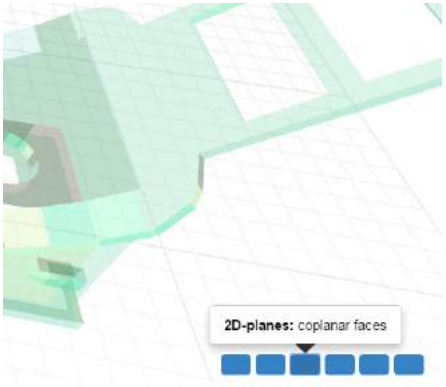
The bottom part of the web page is a gallery of a repository of converted 3D models. If the mouse hovers over one of the shown models,



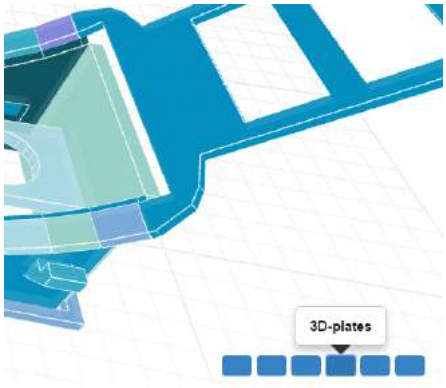
(a) The model visualization shows the original mesh.



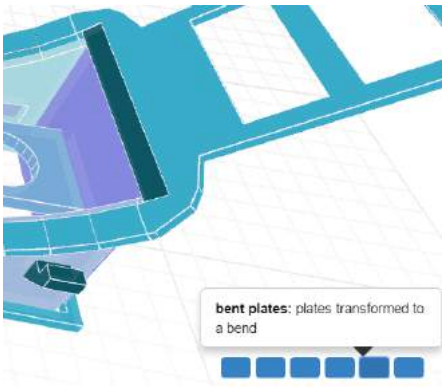
(b) The wireframe visualization shows the triangles of the original mesh.



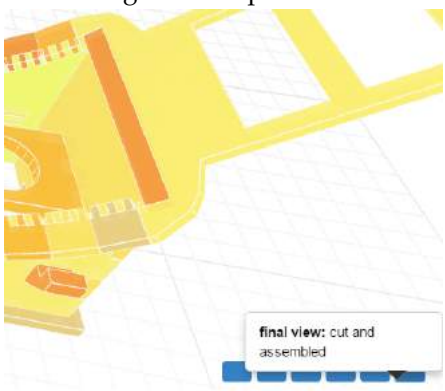
(c) The coplanar visualization shows which faces create a flat surface.



(d) The plate visualization shows the generated plates.



(e) The bend plate visualization uses for plates of the same bent plate the same color.



(f) The final visualization shows all plates with the generated joints.

Figure 9

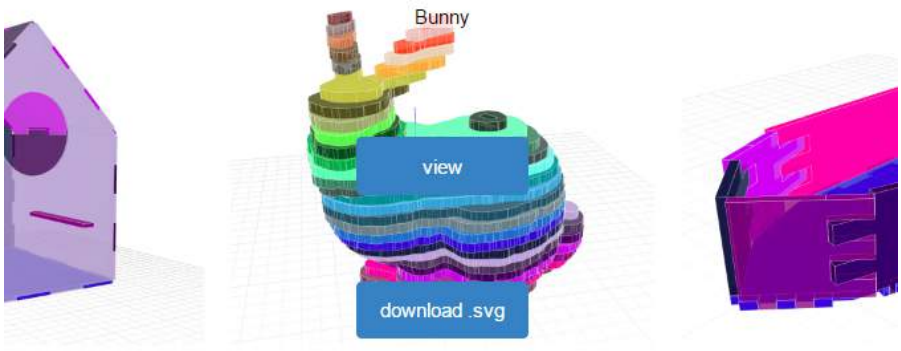


Figure 10: The hover menu in the gallery.

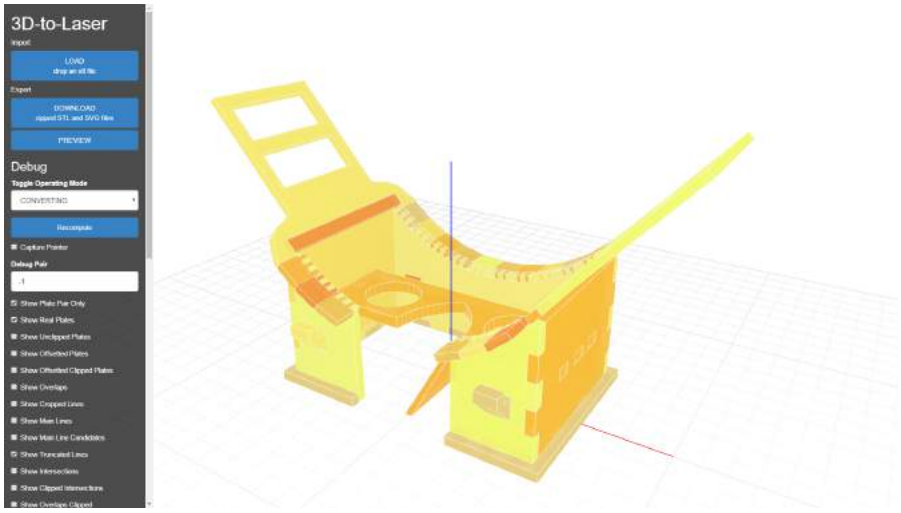


Figure 11: The debug view for easier debugging of the pipeline and single steps.

the gallery shows a overlay menu over this. This overlay includes the name of the model, a *view* and a *download .svg* button (see Figure 10).

The *view* button loads the selected model in the conversion area to show more details of the conversion and fine tuning it. The *download .svg* button starts the download of the converted model directly.

2.3 DEBUG VIEW FOR MORE PROFOUND CONVERSION ANALYSES

For detailed analyses of the conversion of an 3D modeland single conversion steps the debug view can be used. It can be found at `localhost:3000/#/debug` (see Figure 11).

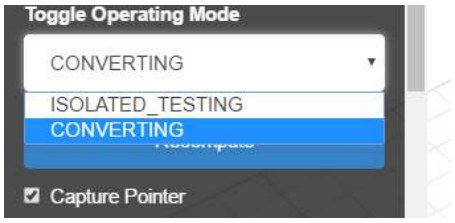


Figure 12: There are two different operation modes.

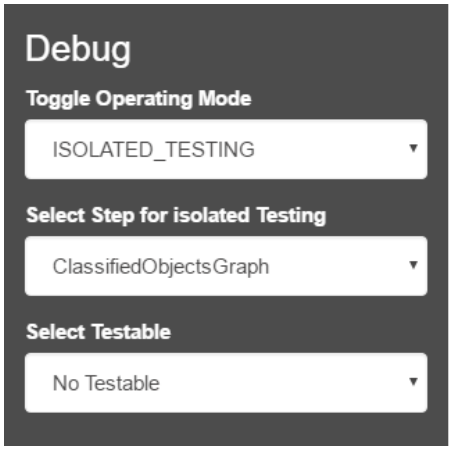


Figure 13: There are two different operation modes.

2.3.1 *Operating Mode for Testing the Pipeline or One Step*

We implemented two operation modes (see Figure 12).

2.3.1.1 *Isolated_testing for Single Step Debugging*

While *Isolated_testing* is selected there are only two additional options (see Figure 13). The first is the step selection. With this the step that should be tested is selectable. The second is the *testable* selection. This shows, depending on the selected step, test sets that could be used to test the step. Currently there are only for the steps *GeometryClassification* and *ClassifiedObjectsGraph* test sets available.

2.3.1.2 *Converting Enables Execution of Hole Pipeline*

If *converting* is selected the hole pipeline is executed. A 3D modelcan be uploaded and is processed until the SVG file is generated.

The *Pipeline Methods* that are selectable are *Stack Plates*, *Find Plates* and *Classifiers*.

For *Stack Plate* and *Classifiers* there are just the following additional options:

A radio button menu to change the rendering of the original model to hidden, solid, wireframe or just the vertices. A drop down offering multiple visualizer depending on the chosen operating mode and representing the corresponding pipeline steps. The test strip user interface to set the kerf for calibrating including the download button for the test strip.

In addition to these, while using the pipeline method *Find Plates*, a field for setting the welding distance and several check box options are shown.

The following of those options effect only the *PlateGraphVisualizer*: *Show Plate Pair Only*, *Show Real Plates*, *Show Unclipped Plates*, *Show Offset Plates*, *Show Offset Clipped Plates*, *Show Overlaps*, *Show Cropped Lines*, *Show Main Lines*, *Show Main Line Candidates*, *Show Truncated Lines*, *Show Intersections*, *Show Clipped Intersections* and *Show Clipped Intersections*. The other effect only the *FingerJointVisualizer*.

In both of these two visualizers the *Capture Point* option can be used to click on a plate and get plate name and connection id's printed into the debugging console.

The option *Platefinding Algorithm* offers the possibility to use only inherent plates for creating the plate graph and the following steps or using them with the extruded plates together.

2.4 *platener* AS A COMMAND LINE INTERFACE FOR ADVANCED USERS

We have seen in the preceding sections, how *platener* is used in a web browser. Though converting a 3D model can be realized in a mere drag-and-drop action using a web browser, we are limited to a single conversion at a time. The web solution focuses on the results of the conversion. Developers require a tool, that reports about the conversion status and internal successful or failed processes. Our software *platener* provides a command line interface (CLI), which is used in a terminal window. The CLI exposes commands for converting 3D models, processing multiple models in sequence and logging progress reports. We explain the usage of the CLI in Section 2.4.1 Usage Instructions of *platener*'s CLI and we demonstrate the logging capabilities in Section 2.4.2 Tracking Down Errors with Conversion Reports.

2.4.1 *Usage Instructions of platener's CLI*

A CLI is self-documenting. Listing 1 shows the available commands.

```
node platener-cli.js -h

Specify an output directory.

  Usage: platener-cli [options] <output-dir>

  Options:

    -h, --help                output usage information
    -V, --version             output the version number
    -p --convertPath <input>  Convert multiple stl-Models to plates.
                              Give path to an folder with stl-Files.
    -f --convertFile <input>  Convert an stl-Model to plates.
                              Give path to an stl-File.
    -v --verbose              Enable Verbose Logging
    -s --subReports           Log Reports for each Fabrication Method
    --maxFilesize <input>     Limit filesize (MB),
                              so large files are skipped.

Full Help:

-f --convertFile    Generate all conversions for each
                    fabrication mode for the given stl-Model.
                    Each solution is stored in a seperate zip-File.
                    The zip-Files are written to the output directory.
```

Listing 1: The help of *platener*'s CLI.

The command line interface mirrors all the computational behavior of *platener* as a web application. With the CLI, we can convert a single 3D model by loading data from the file system. The conversion result is again a ZIP file. It is written to a given target directory. We can recursively read input directories to convert each STL file for the laser cutter. Both commands are given in Listing 2.

```
# convert a single file
node platener-cli.js -f ./stl/cubeFilled5cm.stl ./conversions

# convert an input directory
node platener-cli.js -f ./stl ./conversions
```

Listing 2: Converting an STL file with *platener*'s CLI.

We can use the *–maxFilesize* filter, to limit the input files by their size in megabytes.

```
info: Sample models loaded
info: setup convertify on server-side
debug: Dispatch init
info: init platener-pipeline
info: init scorer
info: init solution-selection
debug: Deactivate method CLASSIFIER for this session.
info: Loading file ./stl/cubeFilled5cm.stl
info: File ./stl/cubeFilled5cm.stl loaded.
debug: Dispatcher: Adding Node
debug: Evaluation did start.

debug: Solution -- STACKED
debug:   hasFailed: false
debug:   error:
debug:   score: 0.3396232718951086
debug: Solution -- PLATE
debug:   hasFailed: false
debug:   error:
debug:   score: 0.5138983423697507
debug: Evaluation did finish.
info:
info: ConversionReport
info: File: ./stl/cubeFilled5cm.stl
info:
info: Best Conversion
info: Method: PLATE
info: Score: 0.5138983423697507
debug: DONE
```

Figure 14: A *Report*, showing a summary of the conversion.

2.4.2 *Tracking Down Errors with Conversion Reports*

Using the web application we either receive a successfully converted SVG file or the conversion fails. As the web interface is a user-centered design we do not show extensive error messages or logs of similar kind. For a developer it is crucial know where and under what circumstances the software failed. With reports, we document the conversion process. A report provides a short summary of the conversion and gathers all error messages along the way, so a developer can track down the issue. Figure 14 shows a summary report. A report itemizing the results of our three conversion approaches can be seen in Figure 15. In Figure 16 an error message is depicted. When multiple conversions were enqueued by *platener*, we receive each conversion report when the process finishes, as shown in Figure 17.

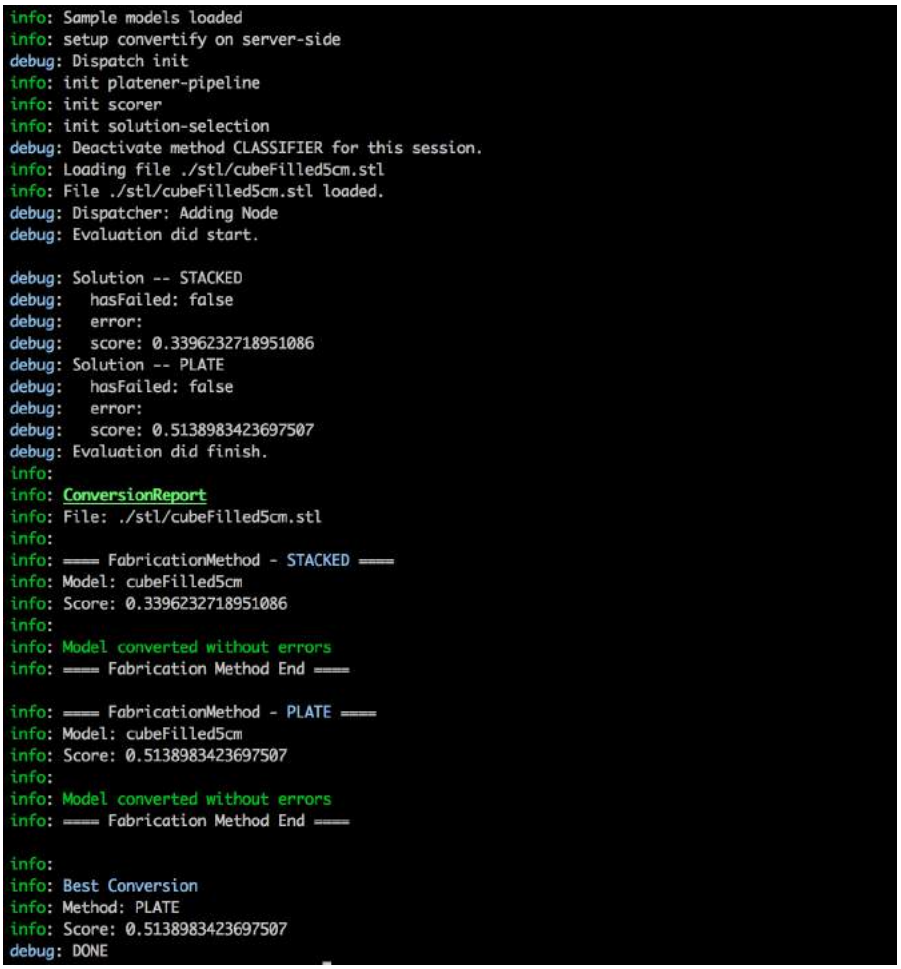


Figure 15: A *Report*, showing a separate summary for each fabrication method of the conversion.

Figure 16: When a conversion fails the *Report* shows the error message.

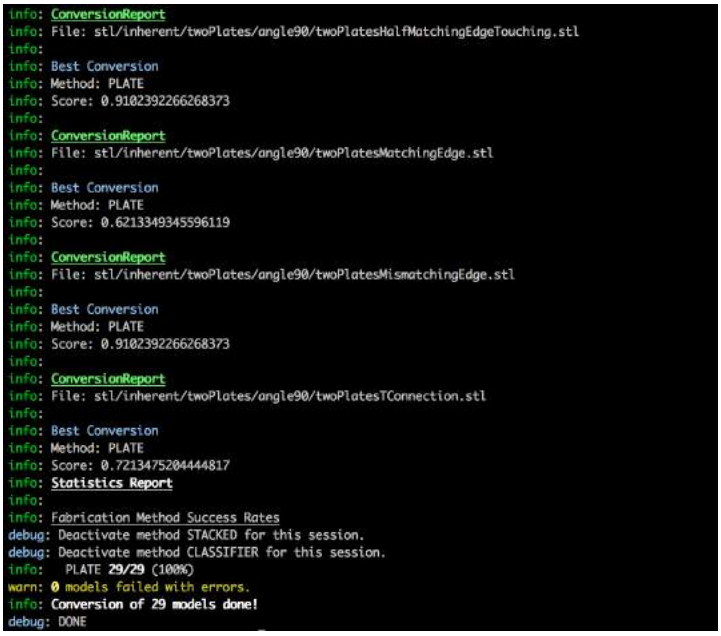


Figure 17: A sequence of conversions is summarized when the computation completed.

3

IMPLEMENTATION GOALS AND TOOLCHAIN

3.1 THE SOFTWARE IS A WEB SERVICE

We aim to enable any user to produce 3D objects with a laser cutter. Our application *platener* is a web service. Therefore it can be used without installation on desktop and mobile devices. We minimize the system requirements of our software by providing a cross-platform web application. *platener* is based on HTML 5, CSS, *CoffeeScript*[1] and WebGL. The web server runs in *Node.js*. The application is written in such a way that it can be executed without a browser in *Node.js* only. This enables power users to run *platener* as a command line interface. Third party applications can utilize *platener* by integrating the *Node.js* package.

3.2 LIBRARIES IMPROVE THE DEVELOPMENT EXPERIENCE

JavaScript and *Node.js* are backed by a huge community. The *node package manager* (npm) ecosystem[2] hosts about 250.000 code packages. By reusing such packages we can significantly speed up our development process. In the following, we list the most important tools and libraries used by *platener*.

- *CoffeeScript* is an indent-based language which transpiles to *JavaScript*. It provides useful syntactic sugar for function and class definitions.
- *React*[3] is a *JavaScript* library for building modular user interfaces.
- *Redux*[4] is a predictable state container which manages the application state in a data-driven way.
- *stylus*[5] is a powerful CSS pre-processor. Instead of writing pure CSS we can benefit from *stylus'* scoping, inheritance and mixin features. *platener* is based on the *Bootstrap 3*[6] CSS framework.

1 http://coffeescript.org/
2 https://www.npmjs.com/
3 https://facebook.github.io/react/
4 http://redux.js.org/
5 http://stylus-lang.com/
6 http://getbootstrap.com/

- *three.js*[7] provides high-level abstractions of the WebGL API.
- *webpack*[8] is a module bundler. It prepares all source code and assets into an deployable package.
- *Grunt*[9] is a *JavaScript* task runner with which we automate our setup and build process.

3.3 DEVELOPMENT WITH A BUILD SYSTEM

CoffeeScript requires transpilation to *JavaScript* and generation of source maps. *stylus* affords pre-processing to CSS. All dependencies need to be included into the web site because the browser is not authorized to access the local filesystem. These processes have to be done each time a line of code changes or a new package is installed. We utilize *Grunt* and *webpack* to create a build system. A build system is a tool, which automates the process of compiling a computer program [25]. Figure 18 gives an overview of our toolchain. *Grunt* runs the *webpack* tasks for the web application and the CLI application in either development or production mode.

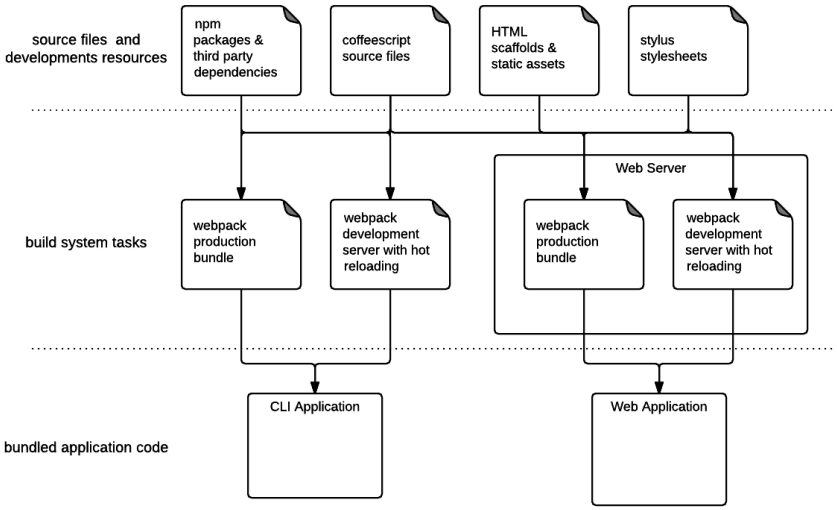


Figure 18: Build system of *platener* with development and production bundles.

In development mode the code automatically rebuilds when files are changed by a developer. Once the new code is built, the sources

7 http://threejs.org/
8 https://webpack.github.io/
9 http://gruntjs.com/

are loaded into the browser or served as the CLI. Loading sources from the server can take several seconds. We reduce the waiting time by performing hot reloads. A hot reload patches the existing and already loaded code base with a change set. This greatly improves the development speed.

4

ARCHITECTURE

In this chapter we outline the architecture and composition of the application *platener* and the framework *convertify*. Our software *platener* is implemented in our web framework *convertify*. The framework allows building interactive WebGL applications in a modular manner. Features are decoupled into plugins. Each application implements its user interface independently from *convertify*.

We introduce the concepts of graphics programming in web environments in Section 4.1 Computer Graphics in Web Environments. The design of *convertify* is explained in Section 4.2 *convertify*. Finally, we give details of the structure of *platener* in Section 4.3 *platener* Is Implemented in *convertify*.

4.1 COMPUTER GRAPHICS IN WEB ENVIRONMENTS

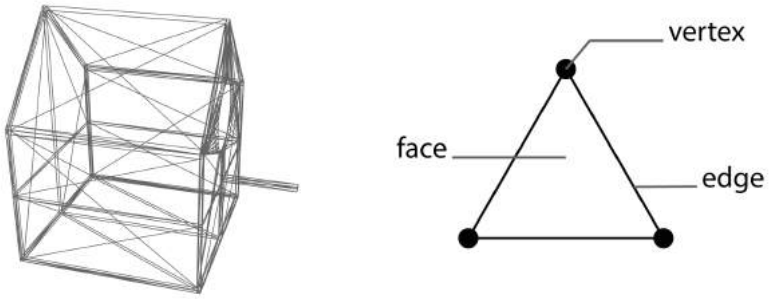
In this section we give a brief overview of graphics programming fundamentals in general and in web environments. Knowledge about fundamentals helps to understand the presented work of this thesis. These fundamentals are concepts of 3D model data representation in Section 4.1.1 3d Model Representation and render loops and scene graphs in Section 4.1.2 Render Loop and Scene Graphs.

4.1.1 *3d Model Representation*

This section explains the mesh data structure and the STL file format that is used to represent 3D models on disk.

The model geometry is represented by a set of points in 3D space. Such a point is called vertex. A single vertex is described by three floating point numbers for the x-, y- and z-coordinates. The vertices of the model are connected into triangles. Each triangle consists of exactly three vertices, where different triangles can share up to two vertices. These connected triangles form a mesh. Figure 19a shows a mesh. A triangle in the mesh is called face. The line between two connected vertices is called edge [8, p. 3]. Figure 19b illustrates the terminology.

platener supports all geometry which is arranged in two-manifold meshes. Two-manifoldness is a constraint on the mesh, which re-



(a) A triangle mesh of a bird house.

(b) A single face with vertices and edges.

Figure 19: Terminology of a Mesh

quires each egde to exactly touch two neighboring faces. Figure 20 shows a non-manifold part of a mesh. Each triangular face of a two-manifold mesh has to have three adjacent faces. This constraint enforces the mesh to be a fully connected graph without holes [7, p. 28]. In order to make sophisticated assumptions when designing the conversion algorithms, the software requires two-manifold meshes.

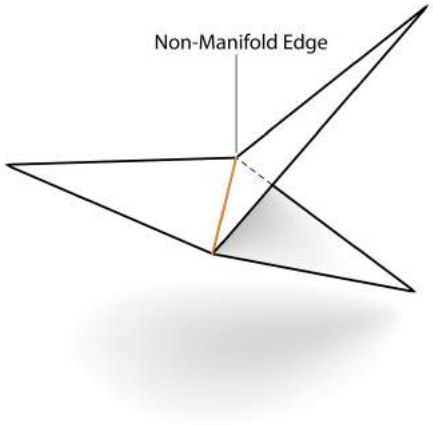


Figure 20: A non-manifold cut-out of a mesh.

We load 3D models in the Standard Tessellation Language (STL) format. STL is a common file format for 3D model representation in the context of 3D printing. The reason we focus on the STL format is the possibility to make 3D printer formats available for the laser cutter. An STL file consists of a list of faces associated with their face normal. Each face is given as a list of vertices. We cannot say in which direction the face points judging only from the vertices. The

face normal determines the orientation of the face. Listing 3 shows an STL file in ASCII encoding. An STL file can also be stored in a binary format. The binary format is commonly used because it requires less disk-space than the ASCII encoding [11, p. 8].

```
solid name
 facet normal n1 n2 n3
  outer loop
  vertex p1x p1y p1z
  vertex p2x p2y p2z
  vertex p3x p3y p3z
  endloop
 endfacet
endsolid name
```

Listing 3: General format of a STL-file in ASCII encoding.

The vertices are stored as floating point numbers. We stated above that two different faces can share the same vertex. In the STL format the data of shared vertices is duplicated. Due to rounding errors the same vertex can be represented by slightly different floating point numbers. Without a look-up table, in which vertices are referred to by a unique index, vertices cannot be distinguished fail-safe. We have to assume that two points with equal coordinates represent an identical vertex. Equality is thereby determined by a threshold. We use the library *meshlib*[1] to create an indexed face-vertex mesh from the possibly ambiguous STL file. The face-vertex mesh is converted to a *three.js* geometry by *meshlib*. With a *three.js* geometry we can render a 3D model in *convertify*.

To support a variety of 3D printer optimized models, we use the STL file format. We import the files with the *meshlib* library. The imported face-vertex mesh structure is then converted for usage with *three.js*.

4.1.2 *Render Loop and Scene Graphs*

To understand how *convertify* and *platener* work, we have to familiarize ourselves with the concepts of rendering and scene graphs first.

A typical pattern in graphics software is the render loop. Rendering is the process of turning the 3D model representation into an array of pixels which can be displayed on the screen [8, p. 2]. A 3D model

1 https://github.com/brickify/meshlib

is rendered in a continuous loop to produce an interactive experience rather than a single static image. The render loop pattern consists of three steps: processing input, updating the 3D model representation and rendering of the scenen [17]. Figure 21 shows an exemplary flow.

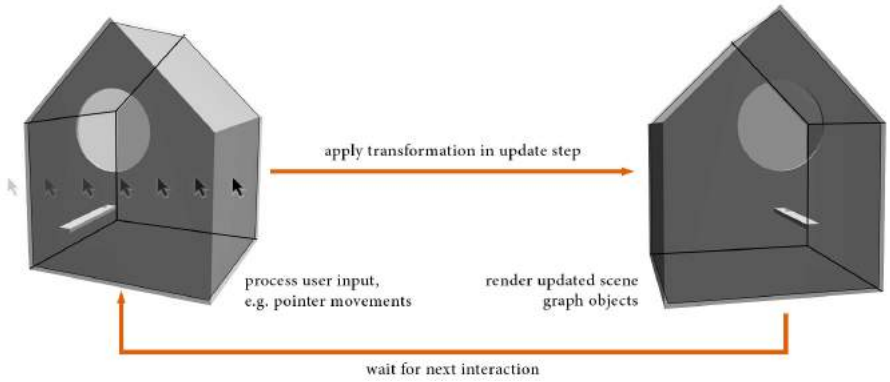
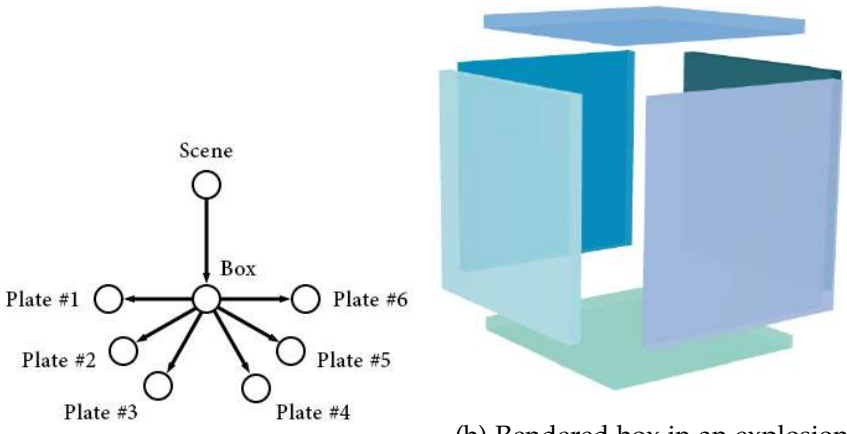


Figure 21: Pointer input is processed by the render loop.

3D model representations can be part of a scene. A scene is the visual space, in which the rendered 3D model can be seen. We use representations of models which are organized in a hierarchical tree data structure. We refer to this hierarchical structure as scene graph [9]. A simple scene graph is depicted in Figure 25. The graph contains a box as single root node. The sides of the box are children of the root node. With this abstraction transformations are applied to parts of the model only, without necessarily touching each face or vertex. Every model that is recognized by *convertify* is part of a scene graph. Finally, the vertices of a model in the scene graph are rendered to the screen.



(a) Graph notation of the composed box.

(b) Rendered box in an explosion view, showing decomposition into plates.

Figure 22: A scene graph showing a box, composed of plates.

The framework *convertify* uses common computer graphics patterns like render loops and scene graphs. With the help of WebGL and *three.js* we are able to bring these concepts into a web environment.

4.2 *convertify*

In this section we present *convertify*, a web framework for working with 3D models. It consists of three code packages: a frontend package, a backend package and a common package. Figure 23 shows its package diagram. The common package provides utilities that can be used by frontend and backend applications. Such utilities are math helpers or the plugin manager.

Apart from that, there are two packages which integrate with either frontend or backend applications that use *convertify*. The frontend package contains components for rendering and scene management. These components implement a render loop and scene graphs which we discussed in Section 4.1 Computer Graphics in Web Environments. The backend package provides similar components as the frontend package, but they work without a document object model (DOM). The DOM is the main data structure for visual elements in a web browser. When applications require the DOM in the *JavaScript* engine, then these applications can only be used in a browser. Our backend package can be run in *Node.js* and does not require a browser environment.

convertify expects the plugins package and the client package to exist, but both packages are not part of the *convertify* framework. These

packages are implemented by an application like *platener* that uses *convertify*. The plugins package provides an exchangeable set of features. Such features are for example analysis algorithms or visualizations of input 3D models. A plugin interacts with the scene and its 3D models via lifecycle events. In *platener* the *PlatenerPipeline* plugin provides all computation logic to transform a face-vertex mesh of a 3D model into laser cuttable SVG file. We elaborate on plugins in Section 4.2.2 *convertify* Provides a Plugin System. The client package gives the look and feel of the application. It contains frontend components which produce HTML elements and it wires up the user interface with the computation logic.

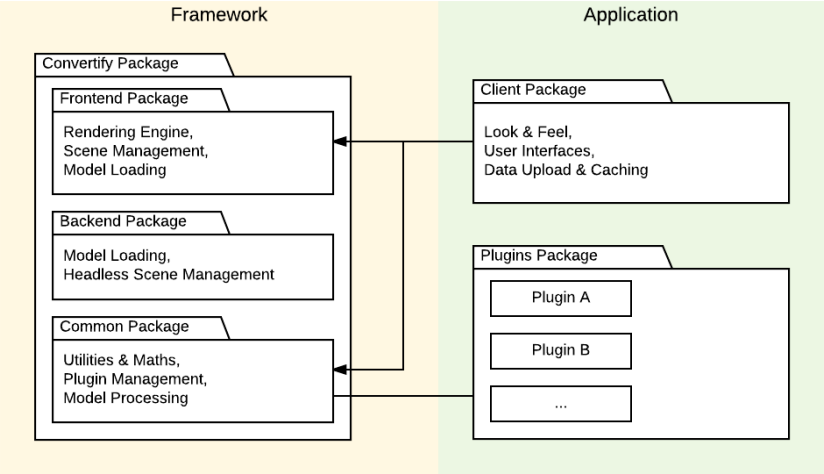


Figure 23: Packages of *convertify*

We will explain core features of *convertify* in Section 4.2.1 Introduction to the Core System of *convertify*. Then we will dive into its plugin system in Section 4.2.2 *convertify* Provides a Plugin System. *convertify* is based on previous work by Silber et al. [24]. We will give a short comparison of their work with *convertify* in Section 4.2.4 *convertify* Is Built on brickify. We elaborate on the client package when we talk about our application *platener* in Section 4.3 *platener* Is Implemented in *convertify*.

4.2.1 *Introduction to the Core System of convertify*

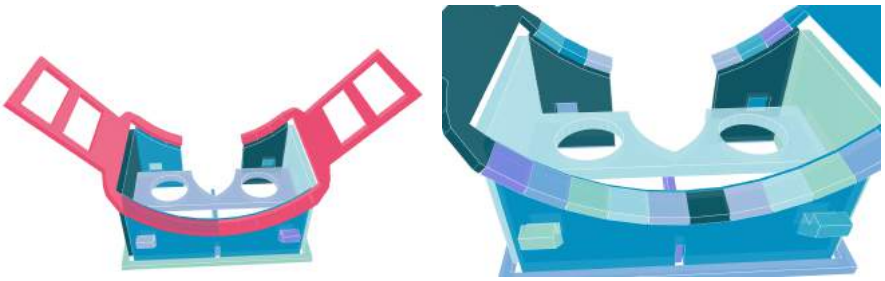
In this section we present the core system of *convertify*. *convertify* helps to build WebGL applications by providing abstractions to the rendering engine and by composing features into plugins. We will focus on

important design decisions, which are scene management, rendering and plugins.

4.2.1.1 *3D models Are Managed in a Flat Scene Graph*

We use a flat scene graph to manage loaded STL files. A flat scene graph has only one level of hierarchy. Such a graph is sufficient because the STL files do not represent nested models.

We implement the flat scene graph using *Nodes*. *Nodes* are abstract objects which represent an entity in our scene graph. Each *Node* references an input model. The input model is a face-vertex mesh that is either used for rendering or used by algorithms for further computation. When the model is used for rendering it is displayed in the scene. Important parts of the model can be highlighted. Figure 24a shows virtual reality glasses where the bendable area is highlighted in red. When the 3D model is used for computation, geometries can be analyzed and manipulated. Figure 24b illustrates computed plates of the virtual reality glasses.



(a) The bend area is highlighted in red.

(b) Plates are computed by an algorithm.

Figure 24: The input model are virtual reality glasses.

All *Nodes* are part of a *Scene*. Figure 25 shows how these classes relate. A *Scene* holds references of *Nodes*, so we can access the input models for later usage. *Nodes* are added to or removed from the *Scene* via the *SceneManager*.

4.2.1.2 *3D objects Are Rendered with three.js*

In the prior sections we explained that the input data for *convertify* is loaded from STL files. These 3D model representations are stored in face-vertex meshes and attached to *Nodes*. Though *Nodes* represent entities in the scene of *convertify*, only *three.js* objects can actually be

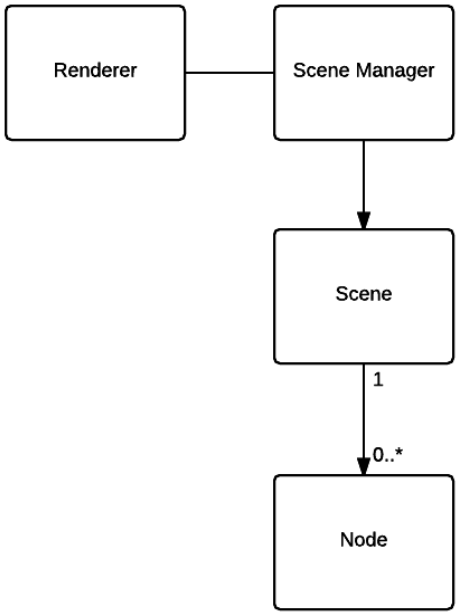


Figure 25: Relation of scene graph components

rendered. Such *three.js* objects are instances of *THREE.Object3D*[2]. This is a generic object which can be rendered into a WebGL scene. The scene graph of *three.js* is not flat. A *THREE.Object3D* can have a hierarchy of objects of any size. Figure 26 shows the *THREE.Object3D* in the context of *Nodes* and the *Renderer*.

We use a *Renderer* instance to bring *three.js* entities to the screen. The *Renderer* sets up a WebGL context and initializes *three.js*. In *convertify* we use plugins in which we can provide additional features. Plugins are described in detail in Section 4.2.1.3 *convertify* Emits System Events. We associate each plugin with a *THREE.Object3D* and vice-versa. Thus, we enable plugins to enhance the scene with new objects. The *Renderer* then traverses the hierarchy of each associated *THREE.Object3D* and renders it.

The *Node* contains the input model. A plugin accesses *Nodes* via system events. We explain system events in Section 4.2.1.3 *convertify* Emits System Events. The plugin uses the *Node* to append the input model data to its associated *THREE.Object3D*.

4.2.1.3 *convertify Emits System Events*

The framework allows building interactive and modular WebGL applications. To be capable of interacting with the user, *convertify* handles touch and pointer events. As an application is composed of mul-

2 http://threejs.org/docs/#Reference/Core/Object3D

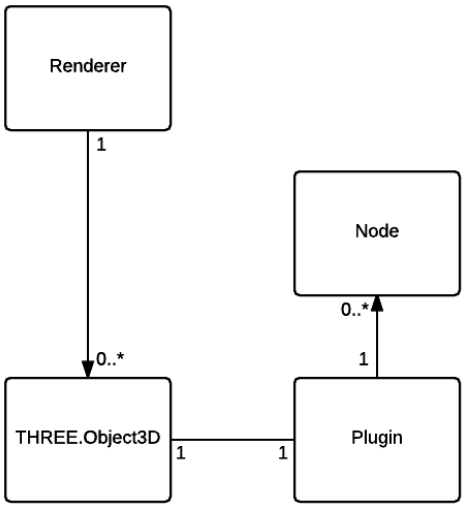


Figure 26: *Nodes* and indirectly associated *THREE.Object3D*

tiple independent plugins, these plugins are integrated with the core of *convertify*'s system. To realize interactions and modularity, *convertify* emits system events during the loading and render process.

A system event notifies subscribers about changes of the state of *convertify* during its lifecycle. Such changes can be touch or pointer interactions with rendered models in the scene (*onPointerEvent*) or change events to the scene graph (*onNodeAdd*, *onNodeRemove*). Figure 27 shows the full lifecycle and event loop of *convertify*. Events concerning the scene management will have the *Node* as payload, so plugins can access the *Node*.

4.2.1.4 *convertify Integrates Additional Features Via Plugins*

convertify loads additional features into its system via plugins. Plugins encourage to decompose feature sets into independent components. A plugin is a separate code package that contains visualizations for the WebGL scene or algorithmic computation units. Therefore, it provides a set of methods which can be called upon emitted system events. We refer to these callbacks as *PluginHooks* or simply hooks.

4.2.2 *convertify Provides a Plugin System*

In this section we present a plugin system that reacts to *convertify*'s system events. Additionally, we introduce a method which organizes communication between plugins.

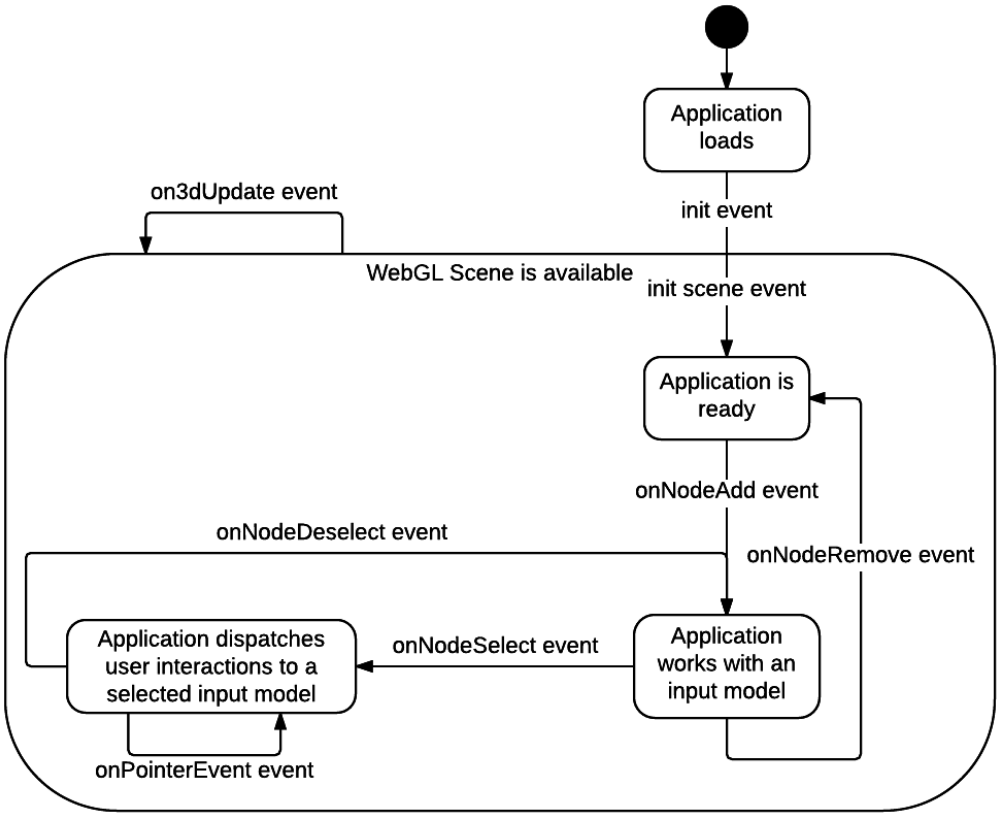


Figure 27: The complete lifecycle of *convertify*

A plugin bundles an exchangeable set of features which will interact with the WebGL scene or will react on changes to the scene. Such features are touch interactions with the rendered model or computations on the model data, triggered when a *Node* is added to the scene. Any full fledged application implements a set of plugins to provide the scene functionality. For example, a concrete conversion strategy provided by *platener* is implemented by the *PlatenerPipeline* plugin. Figure 28 shows our seven plugins and how they relate to each other. We give an overview for each plugin in Section 4.3.1 platener Uses Plugins to Implement Its Features.

4.2.2.1 *Plugins Interact with the Internal System via Lifecycle Events*

convertify emits system events from internal components. Figure 27 depicts the system's lifecycle and the emitted events. The subscribers to system events are plugins. Each event can be handled by a *PluginHook* if a plugin implements it. To implement a *PluginHook* each plugin registers a callback for that event. These callbacks get called when the event is dispatched by the system. Thus, plugins can react to each event and apply their own functionality to the scene or even

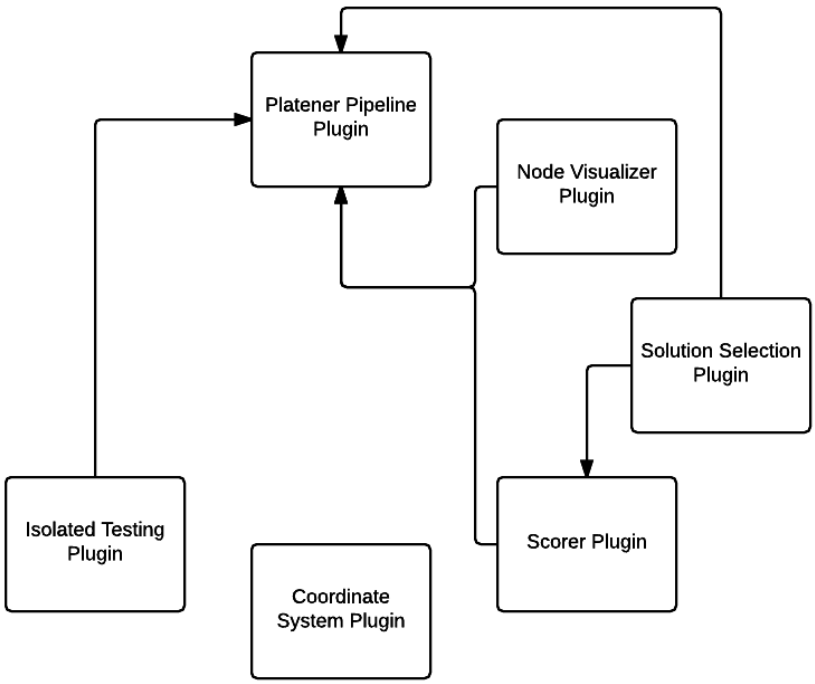


Figure 28: Platener's plugins and their dependencies.

the model geometry. With this we emphasize compact computation units in plugins which can still interact freely with the system.

For better illustration of the functionality of plugins, we have a look at the *PlatenerPipeline* plugin. Figure 29 shows how this plugin integrates with *convertify* events. The *PlatenerPipeline* plugin is a plugin implemented for *platener*. The plugin reads the model data from a *Node* after a 3D model was loaded into the scene. Then it processes the data to create a laser cutter conversion. Once the model is deleted from the scene by a user, the plugin releases the processed data. This plugin does not react to further user interactions, like clicking or rotating the model.

4.2.2.2 *Organizing Plugin Communication with a Dispatcher*

As *convertify* manages multiple plugins, which either represent computation logic or render components, we have to know exactly when each of these plugins will interact with the system. As Figure 28 shows, the dependencies between plugins are hard to oversee. We propose a *Dispatcher* component, behaving similar to the *mediator* pattern. Figure 30 shows how the dependencies between plugins are managed with a *Dispatcher*.

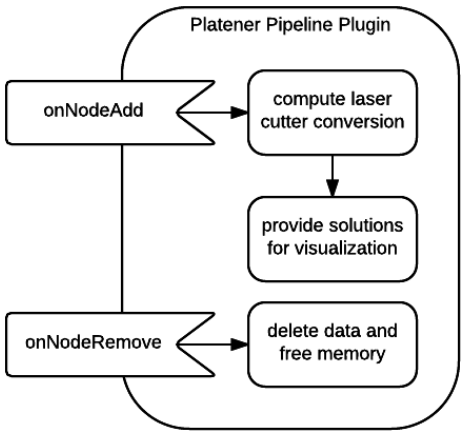


Figure 29: Workflow of the *PlatenerPipeline* plugin.

According to Gamma et al. [10], the *mediator* pattern is a behavioral software design pattern. The mediator encapsulates interconnections of components. It acts as a communication hub and coordinates its clients. The client components are loosely coupled as the clients communicate with the mediator instead of communicating with each other directly. With a mediator we can model many-to-many relationships [10, p. 273].

The order of *PluginHook* invocations has to be determined explicitly, because for different events, different plugins have to run first. For example, *platener* wants to process the model data before it is rendered when a user adds a node to the scene. But it has to destroy the visualization before releasing all the computed data. It is not possible to solve the problem by dispatching all events to all plugins in the same order.

That is why the *Dispatcher* implements every *PluginHook*. All system events are emitted to the *Dispatcher* first, before the *Dispatcher* will reemit them to the plugins. This can be seen in Figure 31. The *Dispatcher* is a mediator whereas the plugins are the mediator's clients. With this component in the middle we can control the order in which the events will be received by the plugins.

Plugins are modular feature sets for our applications. Sometimes it is advisable to separate functionality into multiple plugins, even if they share the same data. In *platener* the *PlatenerPipeline* plugin holds all computed conversions. This plugin does not produce any visual output. This is necessary as we want to use this plugin in our CLI tool as well. The *NodeVisualizer* plugin has to read the computed data from the *PlatenerPipeline* plugin in order to displays it.

To model many-to-many relationships between plugins we enhance the *Dispatcher* with *Protocols*. Figure 32 shows how the mediator con-

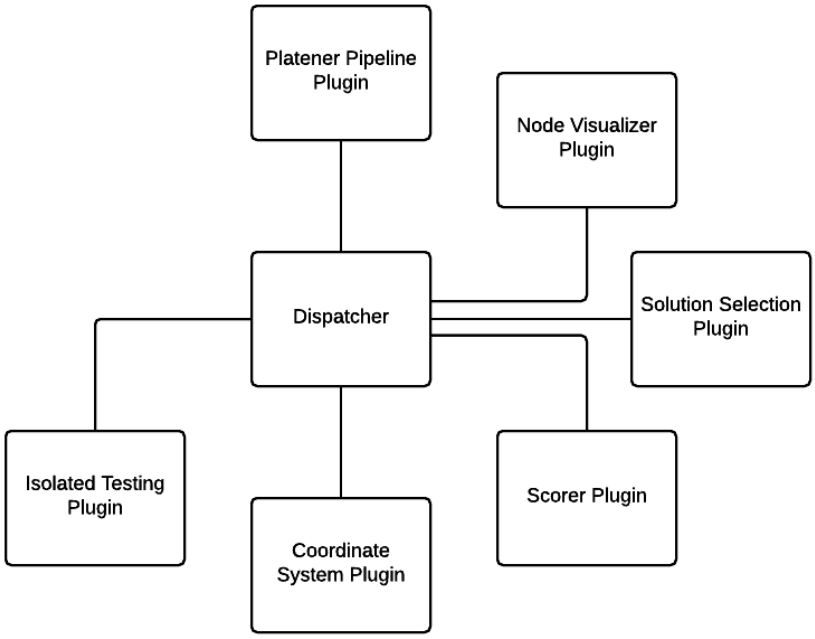


Figure 30: Platener's plugins, managed by a *Dispatcher*.

nects two plugins. A *Protocol* is a mixin implemented by the *Dispatcher*. A mixin adds functionality to an object by object composition [18, p. 81]. A plugin accesses functionality of another plugin by sending messages to the *Dispatcher*. Therefore, the *Protocols* of the *Dispatcher* use the delegation pattern. The delegation pattern achieves the results of multiple inheritance by object composition. It introduces delegating objects and delegates. A delegating object calls external functionality on a delegate object which is available through a predefined interface [23]. The *Dispatcher* is the delegating object and the plugin is the delegate. The *Protocol* ensures that the *Dispatcher* delegates an action to another plugin.

When applications grow it is hard to observe all messaging between components at once. With the *Dispatcher*, we wire up all communication with plugins in one place. Due to explicit ordering of callback executions we have fine-grained control for each plugin. *Protocols* define clean interfaces for plugin communication.

4.2.2.3 *A* Bundle *Is the Entry Point for Applications Using convertify*

Setting up a framework typically requires configuring an entry point. A *Bundle* is the entry point for applications that use *convertify*, e.g *platener* uses the *Bundle* to initialize the web application with *convertify*'s WebGL scene. The *Bundle* is a controller for *convertify*. It loads models into the system and exposes controls for the scene. With this

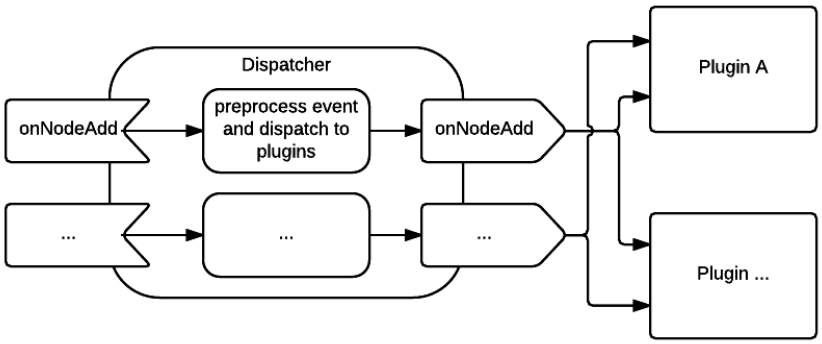


Figure 31: The *Dispatcher* remits system events in an explicit order.

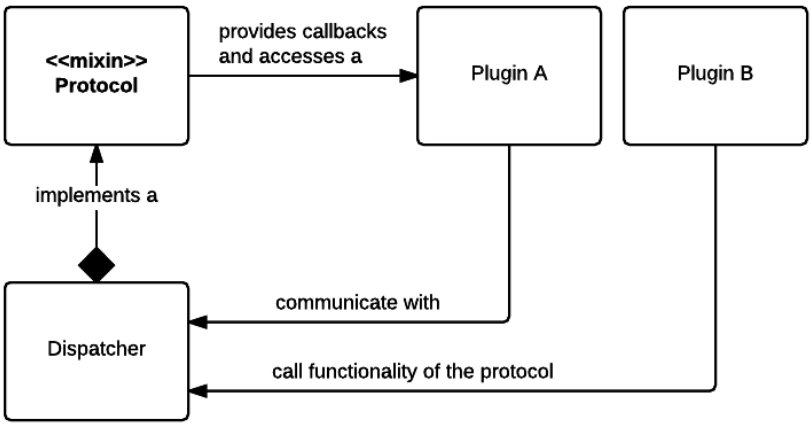


Figure 32: Plugin B can use features of Plugin A by calling functionality of the *Protocol* implemented by the *Dispatcher*.

we can animate the scene programmatically or sync scenes of multiple *convertify* instances. The framework requires a *Bundle* for both, frontend and backend package, because data is accessed differently in browser environments and *Node.js*.

In the preceding section, we introduced a *Dispatcher*, which organizes the communication between plugins and *convertify*. As *convertify* is a framework, multiple applications can be realized with it. Now, every application wants to organize its plugins and event handlers differently. That is why each application implements its own *Dispatcher* instance. The *Bundle* can be configured with such a *Dispatcher*. Figure 33 shows the *Bundle* in the context of an application.

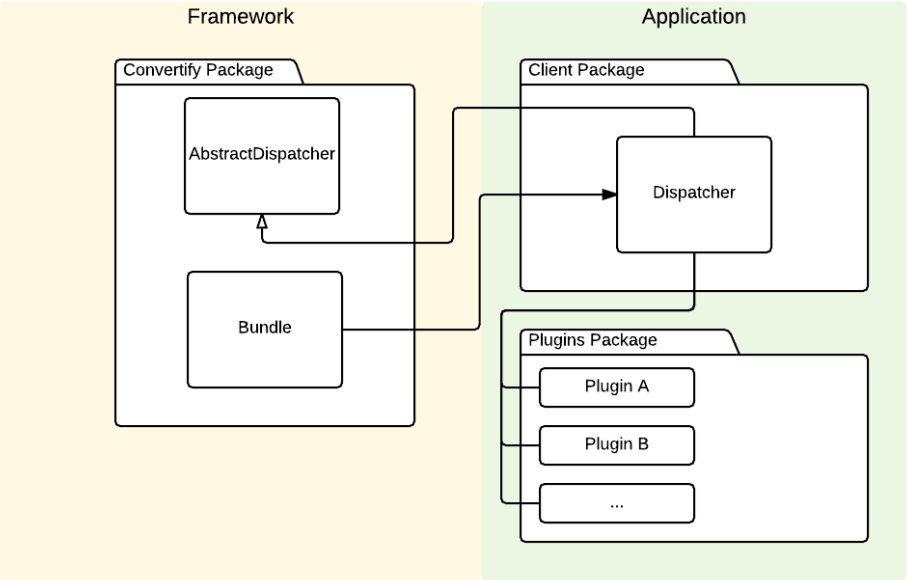


Figure 33: A *Bundle* is configured with a *Dispatcher*.

4.2.3 *convertify Is a Cross-platform Isomorphic Framework*

convertify is a cross-platform framework. *JavaScript* code does not run seamlessly in multiple browsers out of the box. The code has to be aware of their different implementations and APIs. To enable cross-platform *JavaScript* support we use polyfills and libraries that are cross-platform as well.

Legacy *JavaScript* engines do not support data structures like *Map*, *Set* or *Promise*. We provide these features with the *es6-shim*[3] polyfill. A polyfill is "a piece of code or plugin that provides technology that [we] expect the browser to support natively"[4].

Using libraries that have cross-platform support themselves enables us to maintain *convertify* as a cross-platform framework. Such libraries are for example *lodash*[5] or *three.js*. *lodash* provides functional utilities which greatly improve the way we work with our data structures. *three.js* is the 3D library which we use for WebGL rendering. The WebGL API is not consistent for every browser [6].

convertify is an isomorphic framework. We speak of isomorphic *JavaScript* code when the code can be seamlessly executed in browser environments and non-browser environments[7]. *convertify* can use its

3 https://github.com/paulmillr/es6-shim
4 https://remysharp.com/2010/10/08/what-is-a-polyfill
5 http://lodash.com
6 https://docs.unity3d.com/Manual/webgl-browsercompatibility.html
7 http://nerds.airbnb.com/isomorphic-javascript-future-webapps/

full feature set in *Node.js*. This enables us to write applications that can be integrated into other environments or workflows. In the case of our application *platener* we can batch process 3D-models from a command line interface. The CLI is described in Section 4.3.5 The Backend Package Provides a Command Line Interface for Batch Processing.

4.2.4 *convertify Is Built on brickify*

The web application brickify was introduced in Section 1.2 Related Work. brickify provides several features that we reuse for *convertify*. We integrate the rendering engine with its scene management and event system. We greatly improve the event system by using a *Dispatcher*. Also, brickify provides the basic functionality of our plugin system. Though we enhance plugin communication via *Protocols*. In general brickify combines user interface elements with its logic components. With *convertify* we have a modular framework that is completely decoupled from any user interface code.

4.2.5 *A Variety of Possible Applications Can Be Built with the convertify Architecture*

Based on brickify's groundwork and our improvements to the system, we provide a set of common functionality which is useful for any web based 3D application. We focus on 3D-Models as primary data representation. We enable users to perform customization to these models in real-time. The computed results are visualized for the user. We support cross-platform browsers using WebGL technology. We allow mouse and touch interactions with the scene. This enables users to interact with *convertify* from desktop or mobile systems. As *convertify* is fully isomorphic its applications can be integrated into other software by packaging the code as an independent *Node.js* module.

4.3 *platener* IS IMPLEMENTED IN *convertify*

platener is a web-application that converts 3D-printable models into laser-cuttable equivalents. This section describes the implementation of *platener* within the *convertify* framework. Therefore, *platener* integrates with the system events and rendering engine of *convertify*. As proposed in the preceding section, it uses several plugins to bring its features to the WebGL scene. Section 4.3.1 platener Uses Plugins to Implement Its Features describes these plugins briefly. Section 4.3.2 The PlatenerPipeline Plugin Computes the Model Conversions shows

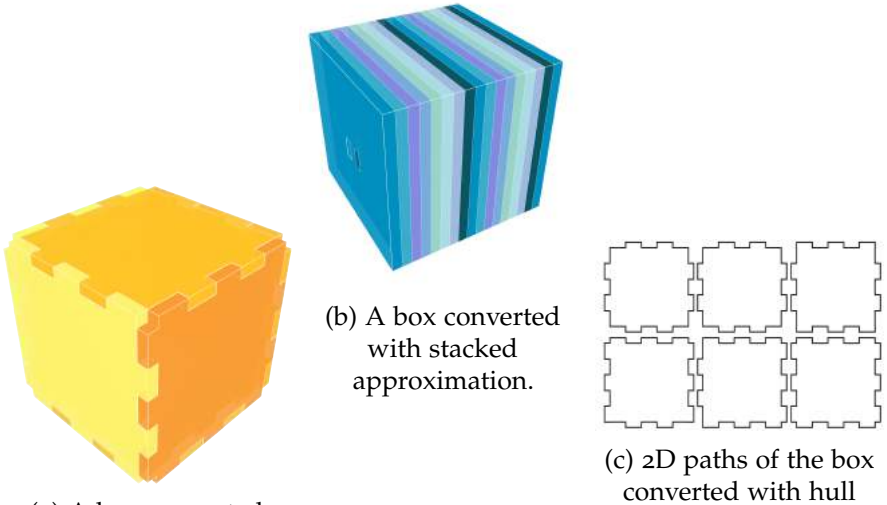
architectural details to the most important plugin: the *PlatenerPipeline* plugin. This plugin implements different conversion strategies for 3D models. In Section 4.3.4 The Frontend Package Connects Plugins and User Interfaces Into a Web Application we explain how the application combines user interfaces with the *Dispatcher* and the plugins.

4.3.1 *platener Uses Plugins to Implement Its Features*

The plugins composed into *platener* provide its computation logic and WebgGL scene rendering. We will give a brief introduction of each plugin in the following paragraphs. Figure 28 shows all available plugins of *platener*.

4.3.1.1 *platener Pipeline Plugin*

The *PlatenerPipeline* plugin does the major work on converting 3D models to 2D plates. The plugin defines multiple conversion approaches. A conversion is an approximation of the original model with plates. Figure 34a and Figure 34b show two conversion results. This plugin generates a set of 2D paths, so that the conversion can be produced with a laser cutter. The paths are depicted in Figure 34c. Section 4.3.2 The PlatenerPipeline Plugin Computes the Model Conversions explains the architecture of the *PlatenerPipeline* plugin in detail.



(a) A box converted with hull approximation.

(b) A box converted with stacked approximation.

(c) 2D paths of the box converted with hull approximation.

Figure 34: Results of the *PlatenerPipeline* plugin.

4.3.1.2 *Node Visualizer Plugin*

We visualize the results of the *PlatenerPipeline* plugin in the WebGL view. The *NodeVisualizer* plugin renders the results of each conversion and its intermediate computation steps respectively. With the help of this we debug the results of the computation visually. As explained in Section 2 USERINTERACTION, we select a single visualization at a time to inspect the results of the step associated with the visualization. Figure 35 shows a head-mounted display in the *Plate* step.

Figure 35: Visual debugging of intermediate conversion results. A head-mounted display, consisting of plates only.

4.3.1.3 *Scorer Plugin*

We run multiple conversion approaches sequentially and choose the best fitted conversion as output. Therefore, each conversion is scored by a scoring algorithm. This plugin provides such scoring algorithms.

4.3.1.4 *Solution Selection Plugin*

This plugin utilizes the *Platener Pipeline* plugin and the *Scorer* plugin to run and evaluate all conversion approaches. It outputs the result of the conversion with the best score.

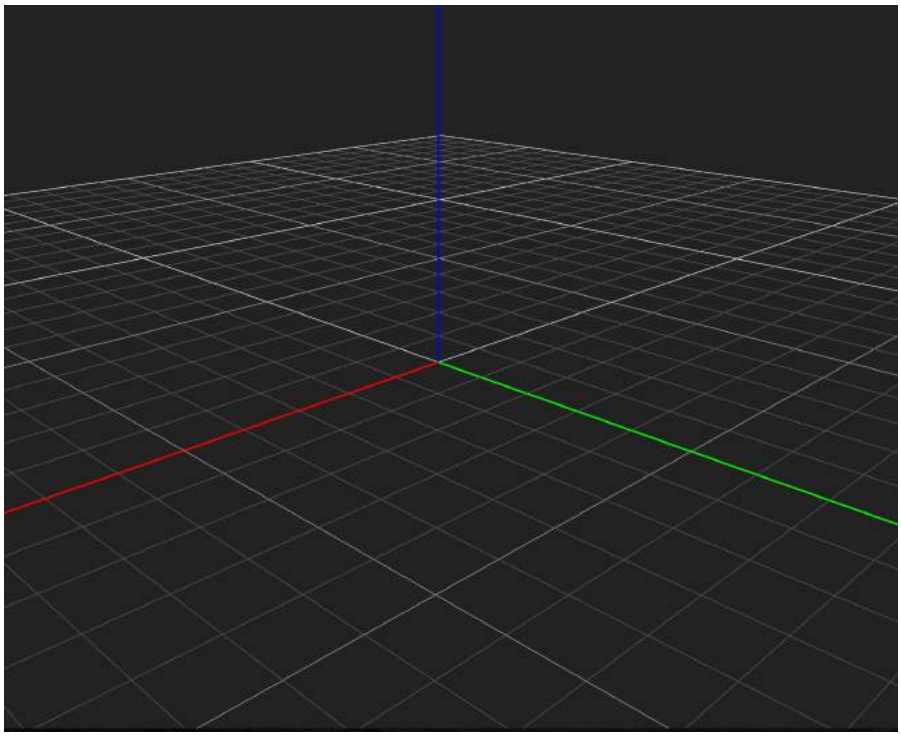


Figure 36: An empty scene showing the coordinate system.

4.3.1.5 *Coordinate System Plugin*

This plugin provides orientation enhancements for the WebGL scene. Rendering xyz-axes and an axis-aligned grid, users can grasp alignment and dimensions of 3D models. Figure 36 shows the coordinate system in the WebGL view. The Coordinate System is taken from *Brickify* as is[24, p. 92].

4.3.1.6 *Isolated Testing Plugin*

While we developed on different stages of a sequentially executed conversion approach in parallel, we needed a mechanism to test each component of the conversion before its preceeding or succeeding components were completely implemented. The *IsolatedTesting* plugin provides an isolated environment, which allows to execute a single intermediate computation step of a conversion approach with pre-defined input.

4.3.2 *The PlatenerPipeline Plugin Computes the Model Conversions*

The *PlatenerPipeline* is the main computation unit of *platener*. It contains algorithms and data structures to bring the 3D model into a 2D representation suitable for a laser cutter. Here we will outline the

details about the computation process. First we look at *platener*'s three conversion approaches. Then we explain the composition of conversion approaches from computation steps into a *Pipeline* data structure. Finally we present the *PipelineState*, a data structure which allows to take a snapshot of the computation state in between two computation steps. The *PipelineState* is used to render the visualizations of the *NodeVisualizer* plugin.

4.3.2.1 *platener Presents Three Conversion Approaches*

The *PlatenerPipeline* plugin facilitates conversion approaches. We call such a conversion approach *FabricationMethod*. It defines a linear process of analyzing a 3D model and thereby creates a suitable equivalent of the model, consisting of plates only. A *FabricationMethod* divides the conversion problem into smaller problems. Thus, it provides a set of algorithms, which are executed sequentially. Each algorithm solves a single subproblem. The algorithms work on results of previously executed algorithms. We refer to a sequence of algorithms as *Pipeline*. Each algorithm in the *Pipeline* is an intermediate computation step, called *PipelineStep*. The first *PipelineStep* works on the face-vertex mesh of the model. The last *PipelineStep* returns a directly exportable data format, in our case a ZIP file containing the 2D construction plans. The composition of *PipelineSteps* into a *Pipeline* is shown in Figure 37. *PipelineSteps* are independent from the *FabricationMethod* and thus, can be shared between different *FabricationMethods*. *platener* provides three *FabricationMethods*.

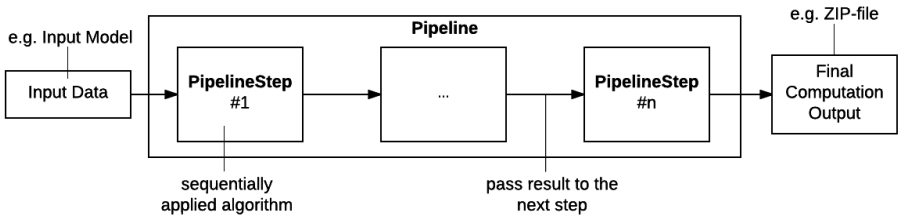


Figure 37: A *Pipeline* is composed from *PipelineSteps*

PLATE METHOD This *FabricationMethod* does hull and surface reconstructions of the input model. Plates are connected via finger joints to approximate the original 3D model. First, the outlines of all surfaces in the model are detected. Then, plates are formed from the outlines. For example two parallel planar surfaces form an *inher-*

ent plate, when they are about three to five millimeters apart. Next, plates are constructed from the remaining surfaces by *extruding* the planar shapes. Then, we connect the plates with finger joints. Figure 38 depicts the important *PipelineSteps* of the *Plate Method* and their visualizations.

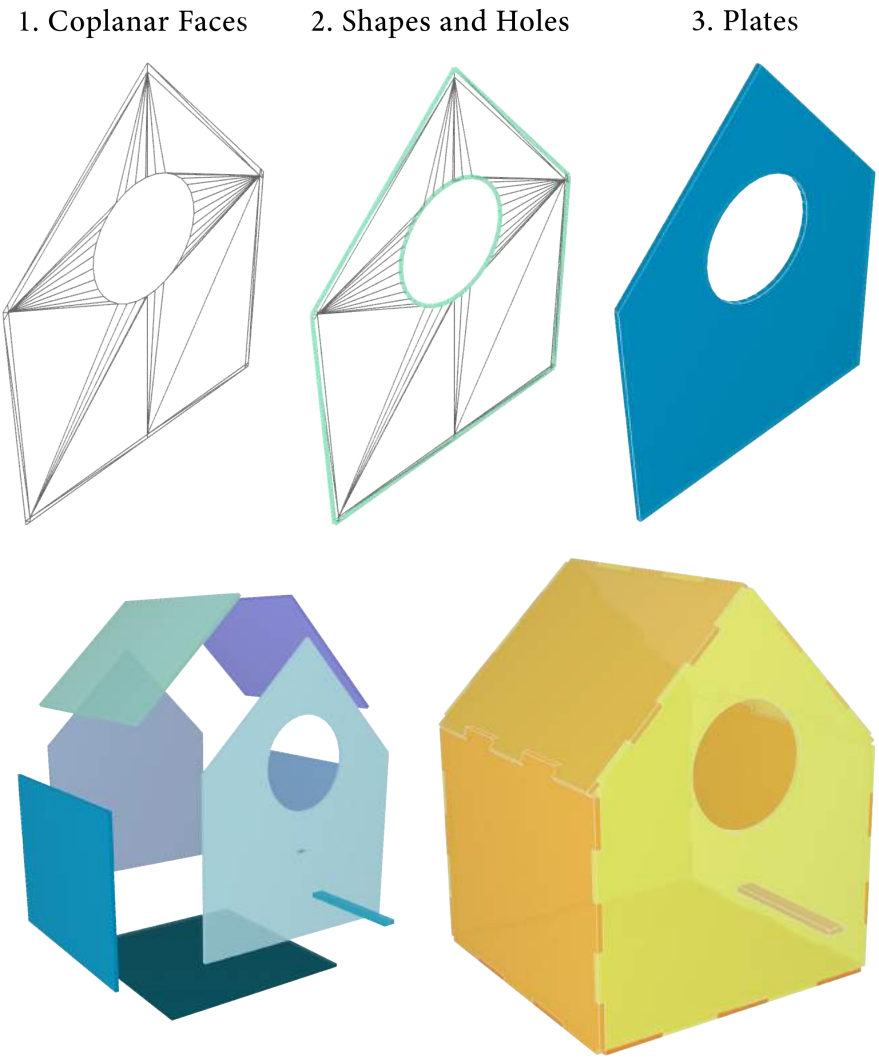


Figure 38: An overview of computation steps that are applied to the model, so that it can be built from plates.

STACKED PLATES METHOD This *FabricationMethod* does volume reconstructions of the input model. It stacks plates onto each other to approximate the shape of the original model, see Figure 39. This preserves the look and feel of the model. The model is sliced into equally

thick layers which form the plates. The plates are connected via shafts to simplify the assembly process.

Figure 39: A rabbit model converted to plates with stacking.

CLASSIFIER METHOD This is an advanced conversion approach, combining mesh analysis and construction techniques of known geometries. This *FabricationMethod* will be more robust when converting models with noisy mesh data, e.g 3D models with lots of texture on their surface. We analyze the model for primitive geometries algorithms. Primitive geometries are planes, prisms, boxes, cylinders or spheres. Figure 40 shows the classification of a cylinder in a model using a non-deterministic algorithm. These geometries can be combined to high-level representations of the input model, giving more information than a mesh of triangles. The model is structured into an hierarchical graph consisting of these primitive geometries only. It is similar to a scene graph. For each of these primitives we know a conversion approach to plates which will give better results than approximating a set of faces. The *Classifier Method* is a proposal and part of future work. We can currently classify a subset of the primitive geometries.

4.3.2.2 *Pipeline Steps Compute Cloneable States*

Each *FabricationMethod* assembles a *Pipeline* from *PipelineSteps*. *PipelineSteps* work on the data of previous computation steps and pass the data to the next compution steps. To improve the development and debugging experience we preserve a snapshot of the data. We call the state

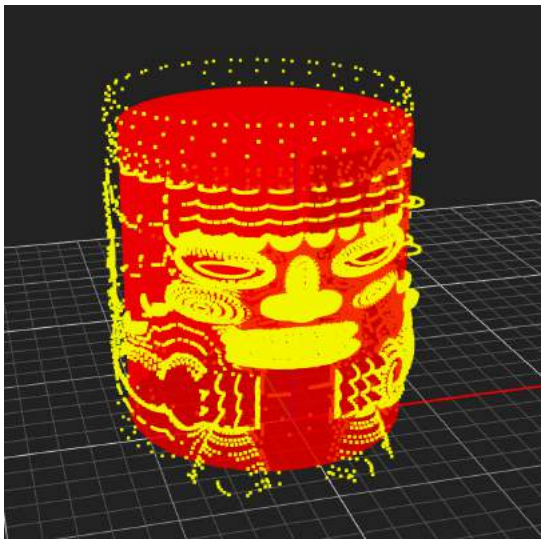


Figure 40: A classified cylinder in a model with textures. The yellow points, show the outline of the actual model.

of a given data structure at a given point in time a snapshot. A snapshot of the computed data is used to render a visual representation of the data. The visual representation gives a clear understanding of the data and thus, enhances the development and debugging experience. Note, that the *PipelineSteps* are merely computation units. All visual output is done in the *NodeVisualizer* plugin.

The data that is passed between the *PipelineSteps* is encapsulated in a *PipelineState*. The *PipelineState* contains all data structures that are ever to be computed by all *PipelineSteps*. It is essentially a container for every computation result.

The *PipelineState* is a cloneable data structure. A cloneable data structure can be copied. When the algorithm of a *PipelineStep* finishes, the *Pipeline* writes the results into the *PipelineState*. A snapshot is obtained by cloning the *PipelineState* after the execution of a *PipelineStep*. Figure 41 illustrates how snapshots are obtained between the *PipelineSteps*.

The encapsulated data in the state can contain multiple fields and can be hierarchical. We provide a fully cloneable data structure. That means, any nested data structure in the *PipelineState* implements a cloneable interface The clone of an instance of *PipelineState* is a deep copy.

A deep copy of a data structure is a clone of all fields and recursively the nested fields of all fields of that data structure. That means if a data structure references dynamically allocated memory, we do not clone the reference of the allocated memory to the copy of the data structure. Instead, we allocate new memory and duplicate the

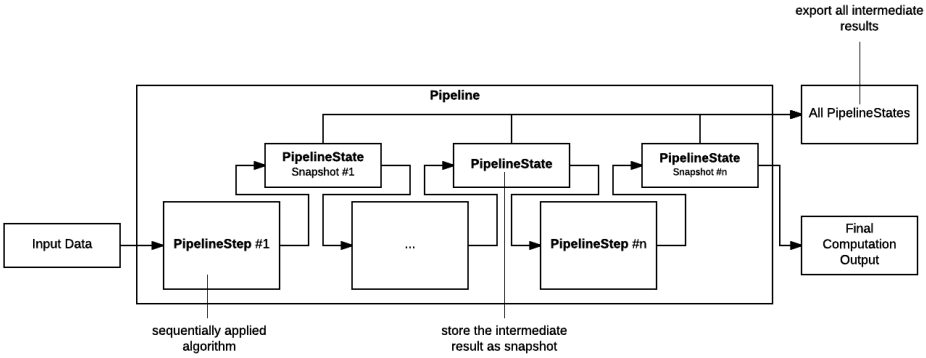
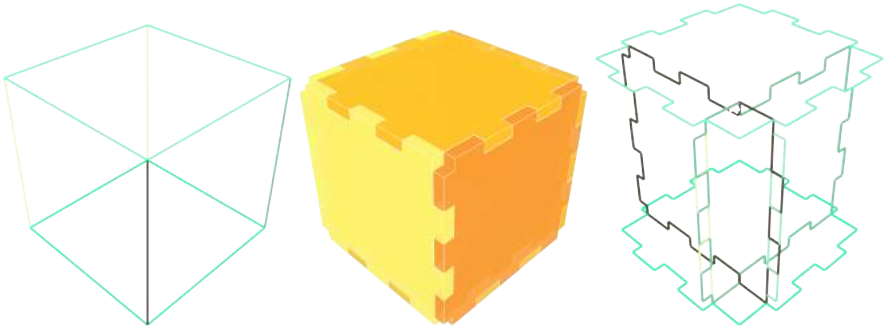


Figure 41: The pipeline preserves deep copies of intermediate computation results.

data to the newly allocated memory. Thus, we make sure modifications on the copy do not alter the original.

When we take a snapshot of the results of a *PipelineStep* we ensure that modifications of a subsequent *PipelineStep* will not alter the data in the snapshot of a previous *PipelineStep*. Otherwise, the *NodeVisualizer* plugin would render corrupted visuals for the previous step. Figure 42 shows how the surface reconstruction step would be effected by the creation of finger joints, when *PipelineSteps* are not deep clones. Figure 42a shows the edges of the cube. These edges are mutated in order to attach new finger joint geometry. The final finger joints are depicted in Figure 42b. When the edges data structure is not copied before adding the finger joints, the edge visualization will also show the finger joints (Figure 42c).



(a) The shapes of a box.

(b) A box with finger joints.

(c) Corrupted shapes of a box, showing the finger joints.

Figure 42: Modifications to shallow copies of the data corrupt the visualizations.

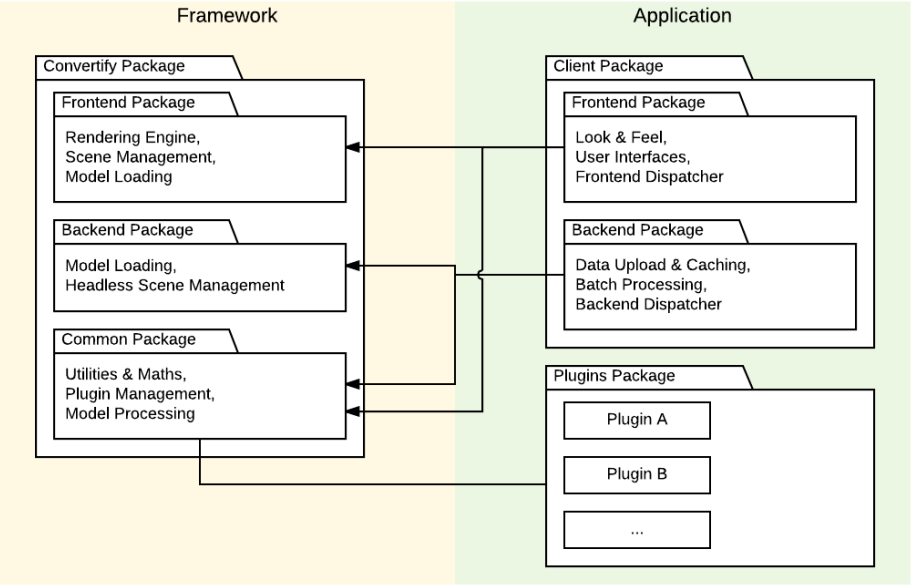


Figure 43: Main Packages of the Platener Architecture

4.3.3 *Code Packages of platener*

platener is divided into the code packages *convertify*, plugins and client. Figure 43 shows the detailed package diagram. Compared to Figure 23, the client package is now divided into two sub packages.

Each package takes over a set of distinct responsibilities. The *convertify* package is the framework, described above. The plugins package provides an exchangeable set of features for the WebGL scene. The client package gives the look and feel of *platener*. It provides user interfaces for the web application in the frontend package and a command line interface in the backend package.

The *convertify* code package was described in detail in Section 4.2 *convertify*.

The plugins package contains seven plugins, where each plugin is a separate subpackage. The plugins implement the computational and render logic of *platener*. We introduced these plugins in Section 4.3.1 platener Uses Plugins to Implement Its Features.

The client package of *platener* is twofold. It contains a subpackage for the frontend and backend application. The frontend application enables users to convert 3D models to laser cuttable files in their web browser. The web application is build with React components. React is a *JavaScript* library for building user interfaces. We will introduce React and our React setup in Section 4.3.4.1 Building User Interfaces With the React Library. The backend application is a com-

mand line interface running on *Node.js*. With the backend application users can batch-process a set of models without using a browser. We give details about the CLI-tool in Section 4.3.5 The Backend Package Provides a Command Line Interface for Batch Processing. Also, we provide a server which manages a repository of 3D models and their conversions.

4.3.4 *The Frontend Package Connects Plugins and User Interfaces Into a Web Application*

The frontend package bundles React components, building the web application's user interface. It accesses *convertify*'s *Bundle* to control the rendering engine and scene management as well as loading the plugins. We implement a *Dispatcher* in the frontend package which connects the plugins with the user interface.

4.3.4.1 *Building User Interfaces With the React Library*

React provides a powerful tool to build user interfaces for web and mobile applications [12]. As React is currently under active development the growing user base emphasizes its practical relevance[8].

React provides a virtual DOM consisting of lightweight representations of DOM nodes, called *Components* [12]. A Component is stateful and provides behavior, e.g. manages click interaction by a user. With focus on flexibility React allows composition of Components. Multiple Components may be nested into a parent Component. The parent Component passes state onto its children. Children can notify their parents by emitting events or executing callbacks. In short, in React data is passed from parent to children, while the event flow is directed from children to parent [13]. To enhance code readibility, React introduces the *JavaScript* Syntax Extensions (JSX)[9]. This code transpilation allows the usage of HTML-tag syntax to instantiate compositions of Components. As our setup is based on *CoffeeScript*, we use a *CoffeeScript* variant of JSX, called CJSX[10]. Listing 4 shows a React Component containing a CJSX snippet.

The real DOM is re-rendered as soon as React detects changes in the virtual representation of the DOM. This is done with reactive updates. This means the *render* method of a Component is invoked

8 http://facebook.github.io/react/blog/2015/03/30/community-roundup-26.html
9 https://facebook.github.io/jsx/
10 https://github.com/jsdf/coffee-react-transform

```
# instantiate a React Component
RootComponent = React.createClass
  # put this HTML-snippet into the DOM,
  # when displaying this component
  render: ->
    return (
      <section className="root">
        {@props.children}
      </section>
    )
```

Listing 4: A *CoffeeScript* example showing a root component, wrapping its children into an HTML-tag with a class name.

again, when the Component's data changes. This causes React to "throw away the entire UI and re-render it from scratch"[11] [12].

4.3.4.2 *Managing Data Flow of React Components with the Redux State Container*

Redux[12] is a *JavaScript* library which aims to reduce complexity when managing state of frontend components with React. As each React Component is stateful, keeping track of state and updating state in the asynchronous browser environment is cumbersome. With Redux we manage all state of our frontend application in one place, resulting in "a single source of truth" [5]. This state is a read-only object tree called *store*. All mutations to the store have to be applied by actions. An action is a plain object comprising the intent of mutating the state [5]. Listing 5 shows such an action.

```
store.dispatch({
  type: 'CHANGE_PLATEFINDING_ALGORITHM'
  algorithm: 'INHERENT_AND_EXTRUSION'
})
```

Listing 5: A state change to the store, indicating which plate finding algorithm is selected in the user interface.

The intended state change of an action is propagated by a reducer. Every reducer is a pure function. A pure function has no side effects. A reducer applies a data transformation on the state tree. Listing 6

11 http://www.quora.com/How-is-Facebooks-React-JavaScript-library/answer/Lee-Byron?srid=3DcX

12 http://redux.js.org/

shows a sample reducer which changes the state according to the selected algorithm. The pipeline reducer checks for the dispatched action and creates a new state object with the selected algorithm.

```
initialState =
  platefindingAlgorithm: DEFAULT_PLATEFINDING_ALGORITHM

pipeline = (state = initialState, action) ->
  switch action.type
    when CHANGE_PLATEFINDING_ALGORITHM
      return Object.assign(
        {}
        state
        { platefindingAlgorithm: action.algorithm }
      )
    else return state
```

Listing 6: A reducer applies state transformation of actions.

When the state is changed by a reducer each React Component that uses data from the changed portion of the state is updated. A re-render is triggered. Thus, we emphasize a purely data-driven update flow in our frontend application. This brings explicit dependencies and avoids race conditions [5].

4.3.4.3 *A Dispatcher Connects convertify With the User Interface*

convertify decouples any user interface specific code from the actual logic and rendering. We can observe the state of *convertify* by listening to events. In this section we describe how the events of *convertify* are propagated into our web interface. As all user interface is setup in a Redux store we have a purely data-driven approach to update the user interface. That means, changes from within *convertify* have to be propagated to the Redux store. Changes to the store are applied through actions. Each state change of *convertify* is mapped to a Redux action. This mirrors the change of state from *convertify* to the frontend. For example, when the *PlatenerPipeline* plugin finishes the computation of laser cutter conversions, the *Dispatcher*'s *evaluationDidFinish* callback is invoked. Figure 44 shows how the *Dispatcher* and Redux are connected. The *Dispatcher* has access to the Redux store. This is possible because the *Dispatcher* is implemented in the frontend package. It uses the store's *dispatch* method and an action to propagate the solutions to Redux. Now, we still have a fully data-driven approach connecting our framework with *platener*.

4.3.5 *The Backend Package Provides a Command Line Interface for Batch Processing*

The backend package of *platener* provides a command line interface (CLI). With the CLI of *platener* we can process multiple 3D models in a queue. Results are read from and stored in a directory. We receive detailed summaries and logs for each conversion. A walkthrough of our CLI is given in Section 2.4 *platener* as a Command Line Interface for Advanced Users.

A command line interface is an application solely consisting of textual input and output. A CLI is typically more flexible and powerful than a graphical user interface. Because a CLI has to be used in a terminal window, it targets mostly advanced users. Another upside of CLIs is that they can be integrated within other applications, e.g. when executing shell commands [20].

Performing a single conversion at a time is sufficient for most users. Using a web interface for that task makes it as easy as it can get. Though when dealing with a collection of objects the manual conversion of each model is cumbersome. Our CLI provides a batch processing method by reading data from the local file system. For each STL file the CLI starts the *PlatenerPipeline* and writes the output to a ZIP file into a target directory. Similar to the frontend package we implement a *Dispatcher* instance which connects the CLI commands and actions to *convertify*.

When a conversion finishes we log details of the progress. Figure 45 shows the variety of available reports. We only store the conversion's meta data into a *Report* instance. The garbage collector of the *JavaScript* engine frees the occupied memory of the 3D model. The *Reports* are printed to the console after all conversions finish. On top of the individual report per object we summarize all data and give a brief statistical overview.

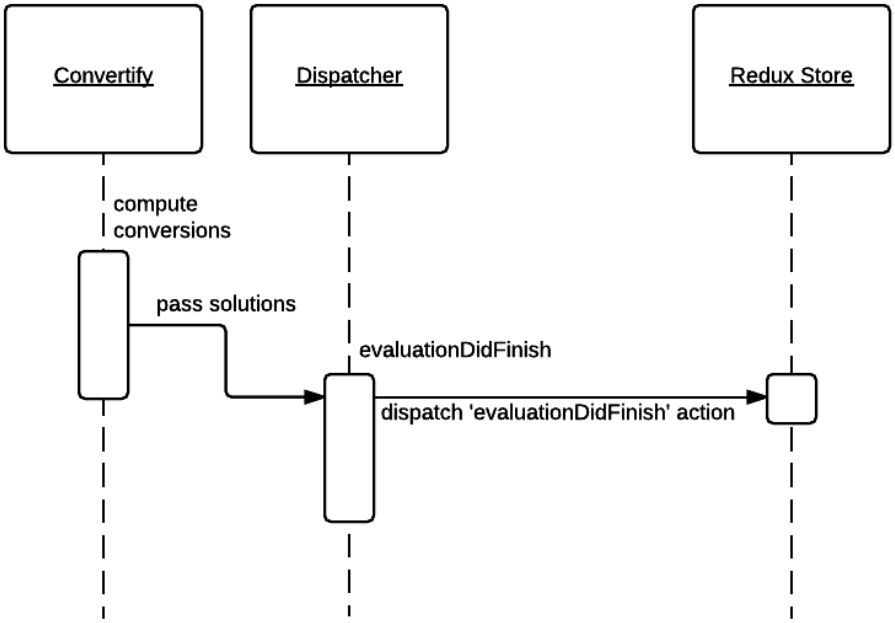


Figure 44: The *Dispatcher* connects *convertify* and the Redux store.

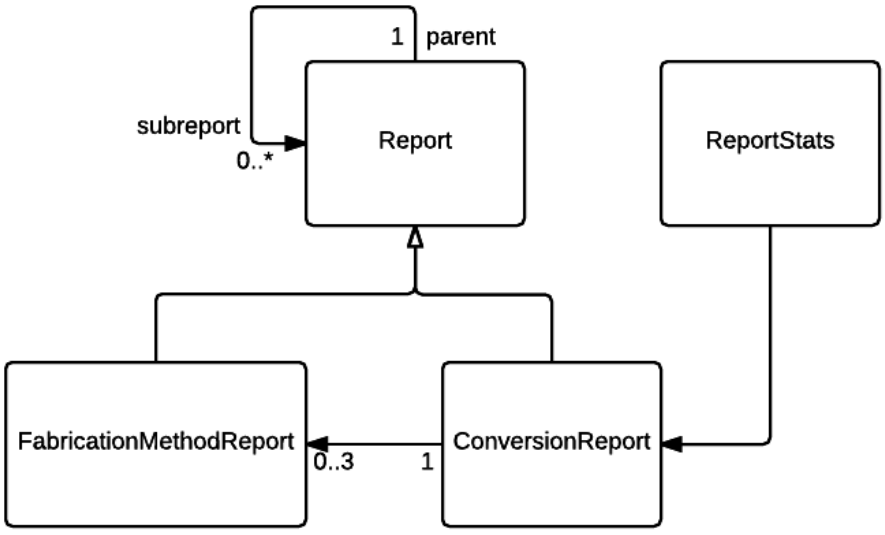


Figure 45: The CLI logs reports for each conversion.

5 PROCESSING PIPELINE

In this chapter we give an overview over the already mentioned processing pipeline and therewith associated data structures. The text serves as an reference point to fit the individual sections into the overall concept.

As described in chapter 4 ARCHITECTURE there are three fabrication methods that implement different pipelines: *Plate*, *StackedPlate* and *Classifier*. Each pipeline consists of pipeline steps that can be described as computation units. Each step performs an operation on the given data and provides its result for the following steps.

5.1 FABRICATION METHODS

5.1.1 *Plate Method*

Plate is the default fabrication method and converts the given model to an approximation consisting of plates. The pipeline is build up of following steps in the stated order:

- MeshCleanup
- ModelStorage
- Simplification
- MeshSetup
- CoplanarFaces
- ShapesFinder
- HoleDetection
- InherentPlates
- ExtrudedPlates
- RemoveContainedPlatesInherent
- RemoveContainedPlatesExtruded
- PlateGraph
- FingerJoints
- AssemblyInstructions

- Calibration
- ShapeLayouter
- MarkupGenerator
- ZipGenerator

5.1.2 *Stacked Plate Method*

StackedPlate converts the given model with plates that are stacked on top of each other. The pipeline is build up of following steps in the stated order:

- MeshCleanup
- ModelStorage
- MeshSetup
- CoplanarFaces
- ShapesFinder
- HoleDetection
- StackedPlates
- StackedPlatesAssemblyInstructions
- MarkupGenerator
- ZipGenerator

5.1.3 *Classifier Method*

Classifier does not produce a final conversion which has to be done in future work. It analyses the given model with several classifiers trying to identify the various primitives contained in the mesh. The pipeline is build up of following steps in the stated order:

- MeshCleanup
- ModelStorage
- MeshSetup
- CoplanarFaces
- ShapesFinder
- HoleDetection
- LabelledShapes
- GeometryClassification

5.2 PIPELINE STEPS

- MeshCleanup

  MeshCleanup searches the mesh for zero faces and removes them. This ensures the mesh's two-manifoldness.

- ModelStorage

  ModelStorage stores the input model for easy access within the pipeline.

- Simplification

  Simplification takes the loaded 3D model, reduces its complexity and removes unwanted features. Therefore the processing gets easier and faster.

- MeshSetup

  During *MeshSetup* the face-vertex mesh of meshlib is transformed into the *faceVertexMesh* data structure. Then the edges and faces are traversed to built a look-up table and the corresponding reverse look-up table. These tables greatly speed up algorithms that work with the mesh data, e.g. *CoplanarFaces*.

- CoplanarFaces

  CoplanarFaces groups faces which are both connected and coplanar, which allows the creation of more complex shapes.

- ShapesFinder

  ShapesFinder analyses faces to build continuous shapes. These are the base of the later generated plates.

- HoleDetection

  HoleDetection takes the found *shapes* and detects their *edgeLoops* either as outer contour or hole.

- InherentPlates

  InherentPlates searches the model for already modeled plates. These consist of both a top and a bottom side which are included in the mesh.

- ExtrudedPlates

  ExtrudedPlates creates plates by translating shapes along their normal, resulting in the six sides of a plate.

- RemoveContainedPlatesInherent

  RemoveContainedPlatesInherent deletes unecessary plates produced by the *InherentPlates* step.

- RemoveContainedPlatesExtruded

RemoveContainedPlatesExtruded deletes unecessary plates produced by the *ExtrudedPlates* step.

- PlateGraph

  The *PlateGraph* analyses plate adjacencies to find intersections between plates and prepare these for connecting them.

- FingerJoints

  The *FingerJoints* step creates the desired type of joints for each previously found plate intersection and adds them to the plates. This is required so the plates can be connected without glue or screws when cut and assembled.

- AssemblyInstructions

  Each plate is labelled with numbers for each connection to another plate. These numbers help the user to find plates that need to be assembled to each other.

- Calibration

  Calibration is done specificly for the used laser cutter. It offsets all outlines of all plates by half the laser cutter's kerf. Thus we incorporate that the plates will be smaller by a tenth of a millimeter after cutting. This is necessary so we can assemble all plates without using glue or skrew.

- ShapeLayouter

  The *ShapeLayouter* arranges the shapes in the cutting plan. It tries to pack them dense to avoid waste of material.

- MarkupGenerator

  The *MarkupGenerator* receives the *threejs* shapes of the *ShapeLayouter* step. These shapes represent the outlines of all plates in 3D space. The pipeline step converts the *threejs* datastructure into a tree of React Components. This Component tree is then rendered to SVG markup.

- StackedPlates

  StackedPlates slices the model in even intervals. By stacking the resulting plates on top of each other, the original model is approximated.

- StackedPlatesAssemblyInstructions

  StackedPlatesAssemblyInstructions enumerates all plates, telling the user in which order they should be assembled.

- LabelledShapes

Each shape is transformed into a *labelledShape*. *LabelledShapes* are used during *GeometryClassification* to assign classified primitives to their original faces and vertices in the mesh.

- GeometryClassification

  GeometryClassification looks for several types of primitive shapes within the model. Such shapes can be converted by considering its specific characteristics. This achieves a more appropriate conversion.

- ZipGenerator

  The *ZipGenerator* writes all previously generated SVG files into a ZIP file. Additionally, it collects meta data from the pipeline state and stores it as a JSON file. The meta data can be used by third party applications to reconstruct the internal data representation.

5.3 DATA STRUCTURES

Mesh

Mesh is the data that represents the complete 3D model which is loaded into our system. It contains the triangles a model is made of. Namely, the specific points and edges of the geometry.

FaceVertexMesh

FaceVertexMesh is the data class produced by *meshlib* which is responsible for the model import. We improve the data by adding indices which results in faster operations. Furthermore it contains the information to map altered or newly generated points and faces to the original geometry.

Shape

A *Shape* is a flat 2D surface in 3D space. It contains *EdgeLoops* that represent outer contour and holes. It is used to build up *Plates*. Furthermore it provides the functionality to represent its *EdgeLoops* in 2D space which is needed for several operations like joint generation.

EdgeLoop

An *EdgeLoop* contains a set of connected edges all lying in a plane. This 2D plane lies in 3D space. The edges are implicitly given by the stored vertices.

Plate

A *Plate* is a *Shape* with a thickness and therefore a three dimensional object instead of a 2D surface.

Polygon

A *Polygon* is a set of vertices in 2D space used in *PlateGraph* and finger joint generation. Generation of new geometry is done in 2D space, therefore 3D data is transformed into *Polygons* and retransformed into 3D space when the operations are finished.

Additionally we wrote the library *jsclipper Adapter* that handles clipping operations on *Polygons*.

6

APPROXIMATION

In this chapter we describe our methods to reduce the complexity of our data.

The pipeline step *Simplification* provides a simplified version of the 3D model for all subsequent processing steps. The operation is called mesh simplification. This processing step serves two purposes: Removal of unwanted details and runtime improvement.

The implemented mesh simplification is called vertex welding. It is also reused in other processing steps of our software to remove unnecessary vertices.

Furthermore we eliminate obsolete vertices during shape generation with point on line removal.

6.1 MESH SIMPLIFICATION

Mesh simplification reduces the complexity of a 3D model by decreasing the amount of vertices and faces. In this process the original object gets approximated with less information. While doing so the differences between the processing results of an original and an approximated object are kept as low as possible.

Our mesh simplification tackles three issues: Processing time, abstraction and elimination of redundant information.

PROCESSING TIME After the model is loaded and stored we decrease its complexity to speed up following processing tasks. Every pipeline step benefits from this mesh simplification because of the reduced data they are working on which implicates a much faster pipeline runtime.

ABSTRACTION Furthermore we eliminate beveled edges. These are created by 3D-modeling software because of aesthetic reasons. Therefore they do not serve any functionality and raise the complexity for conversion. The vertex welding approach gives us the needed shape abstraction to extract tiny details and minor curvatures we do not want to reproduce.

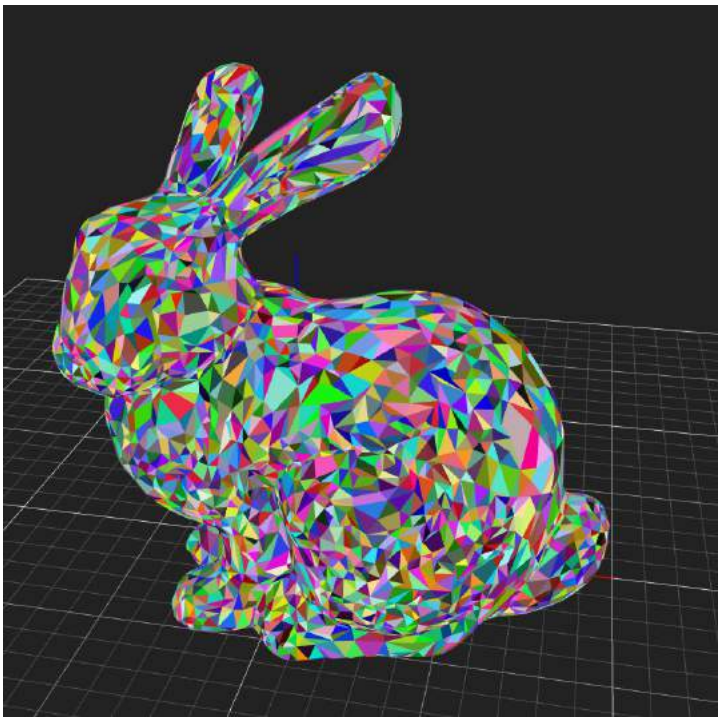


Figure 46: Stanford Bunny with 8662 faces

ELIMINATION OF REDUNDANT INFORMATION CREATED BY OUR PROCESSING PIPELINE The algorithm can be reused in further processing to ensure unambiguous vertices that may occur due to rounding differences during transformations.

The used technique is known as vertex welding in computer graphics. We merge two or more vertices that lie in a specified distance to each other. Then the mesh contains fewer vertices. As a consequence of the smaller vertex set, thin faces no longer consist of three distinct vertices. Two nearby vertices are just represented by one vertex and the face gets deleted.

To illustrate the method the 3D model "Stanford Bunny" shown in Figure 46 is simplified with an unusually high welding distance of 10mm in Figure 47. The images display each face with a different color to visualize the resulting face set. The loaded model consists of 8662 faces while the simplified bunny is reduced to 616 faces. Note, how smaller details like eyes and ears get lost with higher welding distance while the overall shape still remains. For effect illustration the welding distance is higher than the actual distance for processing.

6.1.1 *Vertex Welding*

Vertex Welding merges vertices and works as shown in Figure 48.

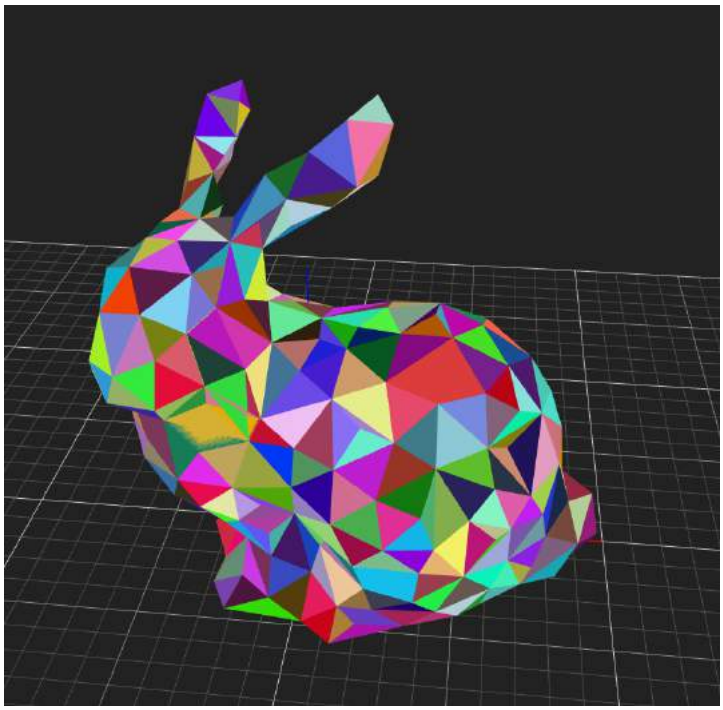


Figure 47: Simplified Stanford Bunny with 616 faces. Applied welding distance: 10mm

A set of triangles is given. We choose a welding distance so points that lie further away of each other will not get merged. This is also the maximum distance a merged vertex can be away from its original position. Therefore it describes the variance of input and resulting vertex set.

For each cluster of vertices that gets merged, a new vertex is created which is typically in the middle of all corresponding points. The original vertices are replaced by this new one in each triangle.

In case a triangle ABC has two points A and B in the same merge cluster, both get replaced by the new vertex V. As a result the triangle consists of the points VVC while it does not contain three distinct vertices anymore - it is a line. If all three of its points lie in a cluster the outcome triangle VVV is a single point. In these cases the triangle gets deleted. The initial area of these deleted triangles is now covered by their adjacent triangles.

The result is an approximation of the input with fewer vertices and faces while the maximum variance equals the specified welding distance. As illustrated in Figure 48, the resulting shape can also completely equal the original one without any differences: Only vertices inside the rectangle got merged which does not affect the outline of the object.

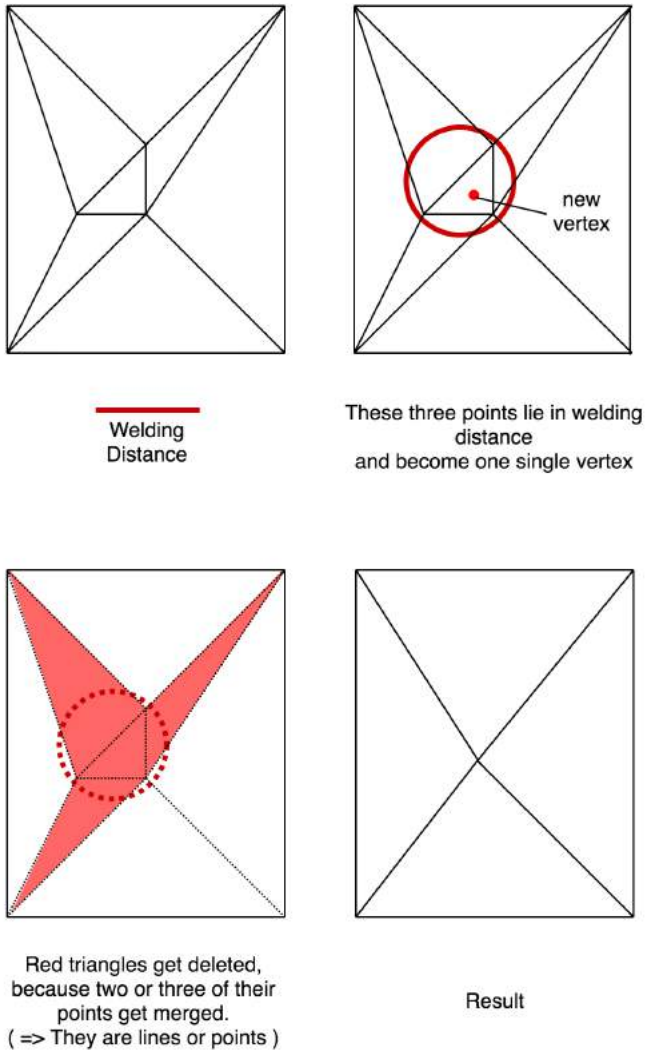


Figure 48: Vertex Welding

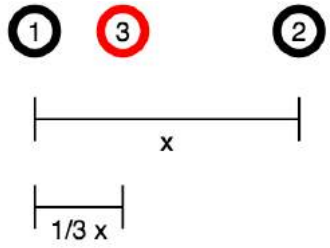


Figure 49: weighted vertex 3 is the result of merging weighted vertex 1 and point 2. Weighted vertex 1 has weight = 2. x is the distance between weighted vertex 1 and point 2.

6.1.1.1 *How the vertex algorithm is used in our software*

For each welding application a new instance of *VertexWelding* is created with the desired welding distance. Then, the data can be preprocessed before the actual welding which provides the best possible result. Each vertex is then replaced by the *VertexWelding* vertices. The points can be merged without preprocessing in case it is not needed, e.g. while using very small welding distances. Both preprocessing and replacement are explained in the following section *Implementation*.

6.1.1.2 *Implementation*

VertexWelding builds a list of weighted vertices during preprocessing. All original points are replaced with these *weightedVertices* during the welding process so this list contains all vertices that end up in the resulting dataset.

During list setup each new vertex is either merged with an existing *weightedVertex* or added to the list. Therefore all original vertices are represented by a *weightedVertex*. A *WeightedVertex* saves the number of represented points to merge new vertices correspondingly as shown in Figure 49.

In this figure point 1 is a *weightedVertex* with $weight = 2$. Point 2 is the next vertex to be preprocessed and is within welding distance to point 1. Therefore point 2 is merged with point 1. The resulting point 3 lies between point 1 and 2. Since point 1 has $weight = 2$ the distance between point 3 and 1 is half the distance between point 3 and 2.

After preprocessing all points of a given set have to be replaced while *getCorrespondingVertex* returns the respective vertex for each requested point.

In case a point is requested without preprocessing *VertexWelding* builds its list without putting the new vertex in between the merged points, but picks the first one as representative for all following.

PREPROCESSING Preprocessing is done with one of the following methods:

preProcessModel

preProcessVertices

preProcessVertex

They iterate over the weighted vertex list and weld the new vertex with an existing one if they are in welding distance. The vertex is just added to the list, if it was not merged after complete iteration. This process is explained by Listing 7 which shows simplified pseudo code.

```
1 preProcessVertex (newVertex) ->
2     for wv in weightedVertices
3         if wv.isInWeldingDistanceTo(newVertex)
4             wv.merge(newVertex)
5             return
6     weightedVertices.push( new WeightedVertex(newVertex) )
```

Listing 7: Algorithm for preprocessing a new vertex

Merging is done by adding a fraction of the vector between *WeightedVertex* and new vertex. The fraction is based on the weight of the *WeightedVertex* which is the amount of already welded vertices.

A *WeightedVertex* $\vec{v}_{weighted}$ is instanced with weight $w = 1$. When a point $\vec{v}_{new}$ is added, weight gets increased and the new coordinates are computed:

$$w = w + 1$$

$$\vec{v}_{weighted} = \vec{v}_{weighted} + \frac{\vec{v}_{new} - \vec{v}_{weighted}}{w}$$

VERTEX REPLACEMENT The actual vertex replacement is done manually for each point with *getCorrespondingVertex*, meaning *VertexWelding* just provides the new vertices and does not replace the old ones. It iterates over the weighted vertex list and returns a point if it is in welding distance. After iteration it adds the vertex to its weighted vertex list in case there was no corresponding one. This automatically handles welding without preprocessing: Each passed vertex is added

to the list as long there is no vertex in welding distance yet. Therefore the first vertex serves as representant for all following ones that lie nearby.

A complete model can be handled with *replaceVerticesAndDeleteNonTriangularFaces*. It takes a *meshlib* model and iterates over all faces. Each of the three points gets replaced and a face gets deleted if its points are not distinct anymore.

6.1.2 *Simplification Pipelinestep*

After the pipeline step *ModelStorage* saved the unmodified 3D model *Simplification* provides a simplified version for all subsequent processing steps. It is directly followed by *MeshSetup* and *CoplanarFaces* which analyses the given mesh to combine multiple faces.

This processing step serves two purposes: Removal of unwanted details and runtime improvement. Due to the capability of representing details with stacked plates, *Simplification* is not run in the fabrication method *StackedPlates*.

6.1.2.1 *Details with Stacked Plates*

If sliced in an advantageous direction tiny details of a 3D model can be obtained with stacked plates. These details may be textures or bump maps like an engraving or a rough surface.

In general *Simplification* removes such details as shown in Figure 51. With a welding distance of 3mm the result is a smooth surface while the original model shown in Figure 50 contains the text 'Make:'.

To simplify a model without loosing such a text is not possible with the implemented vertex welding due to the nature of these texts: The labeled surface differs barely from a smooth surface. Therefore the vertices are so close to each other that welding will occur with every chosen welding distance. To obtain those texts the algorithm has to know areas where it must not weld. If tried to outline the text with a smaller welding distance there will always be some kind of artifacts: missing or degenerated letters like in Figure 52.

In order to be able to represent such texts with stacked plates *Simplification* is not run in the fabrication method *StackedPlates*.

6.1.2.2 *Unwanted Details*

Most 3D editors offer to prettify objects by using curvatures instead of sharp edges. This can be done in various nuances to determine the smoothness of the curve.

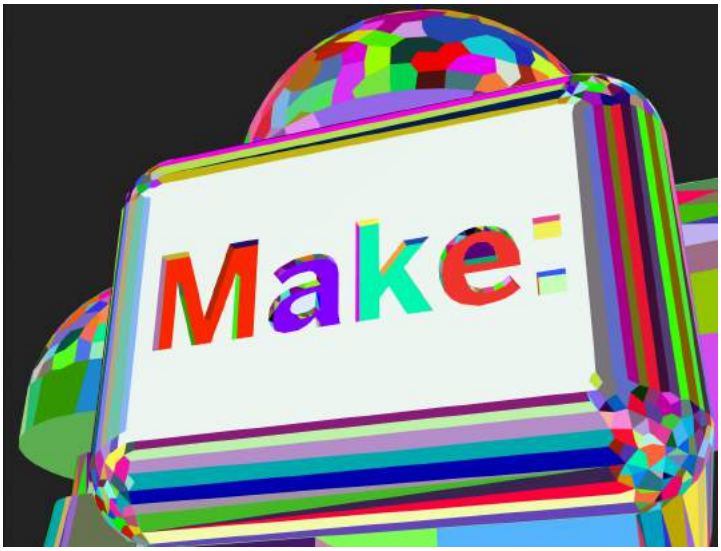


Figure 50: Original Makerbot letters

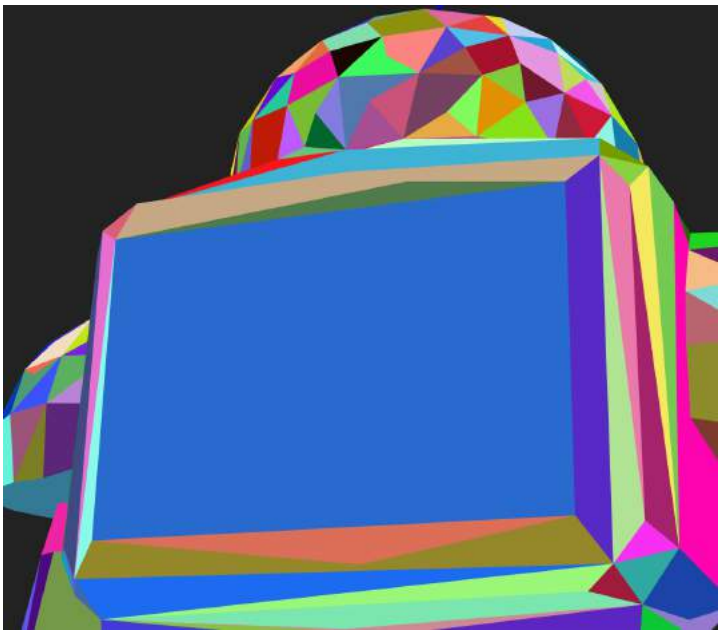


Figure 51: Simplified Makerbot with welding distance 3mm - letters completely removed

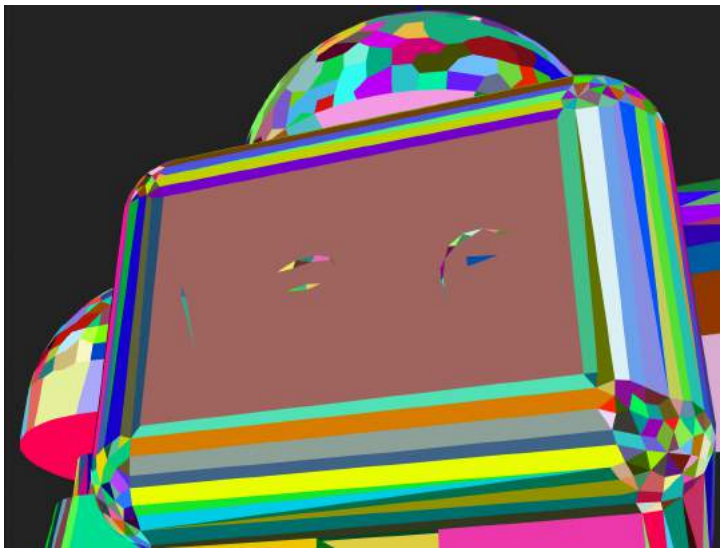


Figure 52: Simplified Makerbot with welding distance 0.8mm - artifacts

Obviously it is much easier to reproduce two connected plates without a beveled edge in the pipeline step *InherentPlates*. Since these curvatures are just for aesthetic reasons we revert the rounding without any loss of functionality.

Figure 53 shows three beveled cubes where the edge is subdivided into one, two and ten parts. All three result in the cube with straight edges after the *Simplification*. The granularity of the created beveled edge does not make any difference for the outcome: The higher the number of parts the denser the vertices while the absolute distance between start and end of a beveled edge remains the same. Therefore all vertices in this area get merged to the desired point independently from the edge segmentation.

In addition all sorts of surface modification like engravings and small attachments get removed which simplifies the plate recognition. In Figure 54 there is text on top of the actual surface and in Figure 55 there is text pushed into the surface while both text is removed in the resulting object.

6.1.2.3 *Process*

The pipeline step clones the model due to our clonable state approach so the original model can be accessed at any point in time. The clonable state approach is described in Chapter 4 ARCHITECTURE Architecture. Then the welding distance is calculated and the mesh gets simplified. Finally the 3D model is setup for the following pipelinge steps.

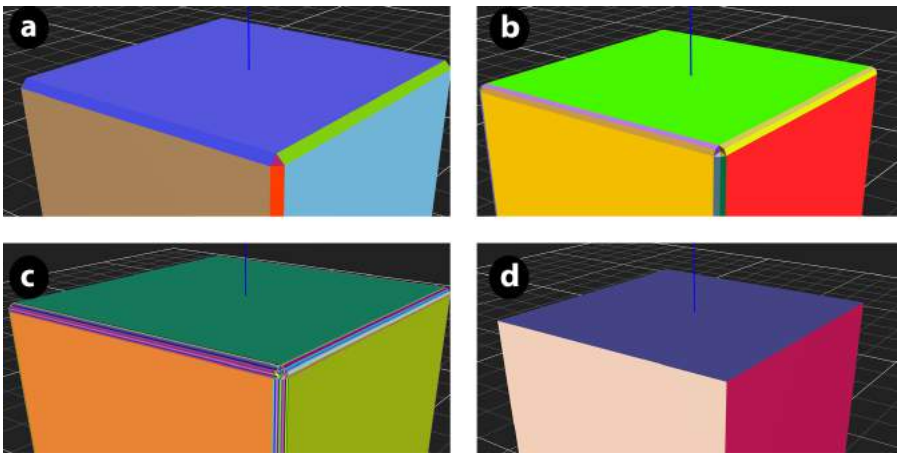


Figure 53: Beveled cube a) with 1 subdivision, b) with 2 subdivisions, c) with 10 subdivisions and their resulting cube d) without beveled edges

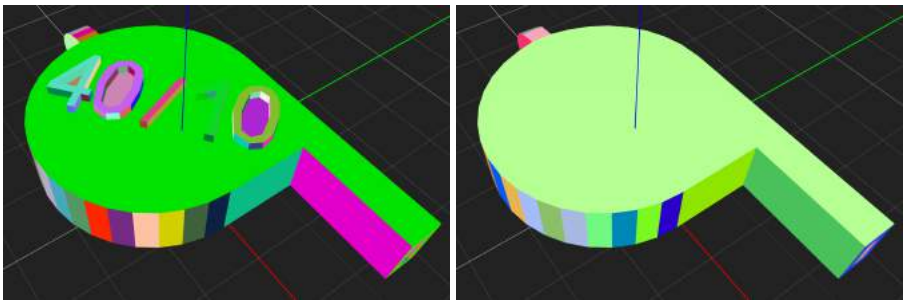


Figure 54: Extruded details get removed

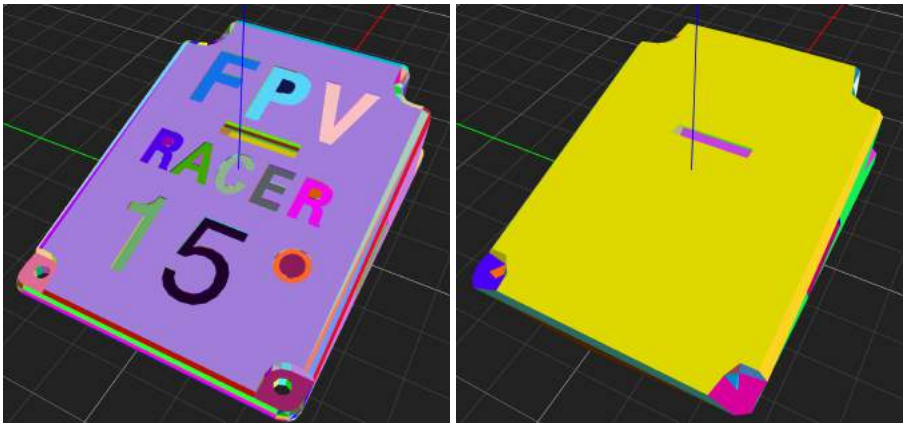


Figure 55: Pushed in details get removed

Figure 56: Vertex Welding UI Element

WELDING DISTANCE The welding distance is calculated based on the dimensions of the 3D model. Very small objects will not be welded at all and the welding distance increases with bigger models.

For most models a welding distance of 2mm is applied which removes all beveled edges created by common editors like Blender[1]. Due to the nature of beveled edges all beveling methods only modify small areas around corners which results in dense vertices that get merged by our vertex welding.

The user can modify the welding distance in the UI element shown in Figure 56. This is useful if the user only needs a rough approximation and does not care about small details so a greater welding distance can be chosen.

WELDING At first all vertices get preprocessed by the welding algorithm to create the vertex lookup-table. After that the model is passed to *VertexWelding* which iterates over all faces to replace its vertices with their corresponding vertex from the preprocessed table. Meanwhile it checks for invalid faces and deletes them.

PROVIDED MODEL After the actual vertex welding the 3D model is checked for an empty face set which might occur if the user picked a large welding distance. In this case the stored and unmodified mesh is returned in order to run the rest of the pipeline. The user gets reminded to pick a better welding distance.

The mesh preparation itself is extracted into another pipeline step: *MeshSetup*. It takes the simplified 3D model and makes it ready for all further processing: face vertex mesh building, normal calculation and mesh indexing.

6.1.3 *Reusage for Redundant Information Elimination*

The algorithm can be reused in later processing steps. This is done for two reasons:

1 https://www.blender.org

First a cluster of multiple points lying close together might not bring any additional usable information. So one single representative point of that cluster leads to the same result produced by our system. Therefore the data set can be simplified to improve its processing.

Second a set of points that were only one point in the first place. Different operations on a single vertex shared by multiple primitives lead to slightly different resulting vertices in each primitive. This happens because of floating point inaccuracies. Since these were actually just one point they get merged with our vertex welding.

For instance, this may occur during curve creation as described in Chapter 10 CURVES Curves where *VertexWelder* is used to eliminate these point cluster. In general whenever a new data object is generated welding can be applied to ensure unambiguous points.

6.1.4 *Alternative Methods for Vertex Welding*

There are different ways of implementing vertex welding. Three other appropiate approaches known in computer graphics are sorting vertices with an octree, via hashing or via binary space partitioning.

We chose our approach because it serves the simplification step with welding the complete 3D model as well as reusing it in later processing steps whenever we want to weld a data set. Therefore we do not have to implement different welding methods for various usages. Other implementations may not be applicable, may not serve all usages or will not bring any significant time improvement.

However, the welding can be split into different methods in future work if a runtime improvement is needed. Concretely the pipeline step *Simplification* could use an octree to speed up the model simplification and the remaining welding operations will be done with the current implementation.

OCTREE The space is discretised into blocks which dimensions equal the welding distance. At first all vertices are sort into these buckets. Then points are compared within each block along with its 26 surrounding ones and merged accordingly.

This would perform slightly faster for complete mesh simplification and slightly slower for fragmental welding of small vertex sets. Due to the vertex arrangement of most objects the number of comparisons would be lower and consequential the runtime too. On the opposite it performs slower for welding steps that take only a small amount of vertices: Most of them have to be compared so the space discretisation does not bring any advantages.

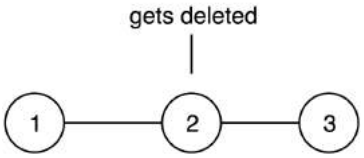


Figure 57: point 2 gets deleted because it lies on the line between point 1 and 3

The implementation of an octree based welding is a bit more complex than the current implementation. We did not chose the octree variant since the current implementation runs in adequate runtime, was faster to code and easier to test.

HASHING Hashing is similar to the Octree approach and sorts vertices into a hash list. The sorting is based on a hash value computed by a hash function. This method is usually used if only identical points are merged instead of vertices that lie close to each other.

Due to the nature of hash functions it is easy to identify identical points since they have the same hash vale. Contrary, different points will not end up with the same hash value even if they should be welded in case we use a higher welding distance. Therefore hashing is not applicable to our system.

BINARY SPACE PARTITION Binary space partitioning discretises the space into parts similiar to an octree. It divides the space into unequal parts containing just one point instead of dividing the space into equal parts containing various number of points. Therefore the amount of point comparisons can be reduced which leads to a faster algorithm.

However, the runtime improvement is marginal compared to an octree and is more complicated to implement, test and maintain. Therefore binary space partitioning is not appropiate for our system.

6.2 POINT ON LINE REMOVAL

During shape generation out of faces often more vertices are given than needed to define the shape. So a cleanup is reasonable.

PointOnLineRemover is used by the *ShapesFinder* to delete unnecessary vertices and therefore reducing their complexity. It checks every *EdgeLoop* of a found shape for vertices that lie on implicit lines given by other vertices as shown in Figure 57: Point 2 lies on the line between point 1 and 3 and gets deleted.

Shapes may represent only a line or a point instead of a 2D-shape if they do not contain enough vertices. An *EdgeLoop* gets deleted if it contains less than three vertices after point on line removal because it does not span an area any more. In case the contour of a *shape* gets deleted the complete *shape* is obsolete and gets deleted as well.

A threshold for the distance of a point to a line is given since minor deviations from the line are not relevant.

A simplified pseudo code version of the actual removal algorithm is shown in Listing 8: *getPreviousIndices()* returns the two previous indices that are not removed and *isPreviousOnLine(index, prev, prevprev)* returns true if $vertex_{prev}$ lies on the line between $vertex_{index}$ and $vertex_{prevprev}$. So we iterate over all vertices while deleting all its previous vertices until the previous is not on the line.

```
1 for index in [0...vertices.length] when index not in
  ↪ removedIndices
2     while true
3         {prev, prevprev} = @getPreviousIndices(index)
4         if @isPreviousOnLine(index, prev, prevprev)
5             removedIndices.push(prev)
6         else
7             break
```

Listing 8: Simplified point on line removal algorithm

When checking if a point lies on a line it is not sufficient to calculate the distance only as illustrated in Figure 58: Point 2 gets deleted in case point 3 lies within threshold distance to the line between point 1 and 2. But the actual point that should be deleted is obviously point 3 since it lies directly on the line between point 2 and 4.

Therefore it is needed to check if a point lies in between the points spanning the line and not outside. In Figure 58 point 2 lies outside of the line between point 1 and 3. With this additional check the resulting triangle consist of the points 1, 2 and 4 as obviously desired.

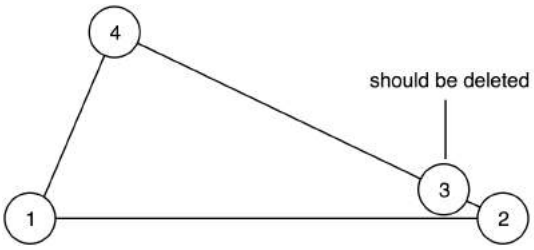


Figure 58: Point 3 gets deleted even if point 2 is within welding distance of the line between point 1 and 3. This is done because point 2 lies not in between point 1 and 3 but outside of them.

7 PLATES

7.1 PLATES CAN BE FOUND INHERENTLY, BY EXTRUDING OR BY STACKING

There are multiple approaches for finding plates in a 3D mesh. Two of these concepts, inherent plates [7, p. 32] and extrusion [7, p. 28], are based on existing algorithms. The third approach, stacking plates, was built from the ground up.

Inherent plates are modeled in the mesh, with both a top and a bottom side (see Figure 59). Between these, a plate is constructed. This is the most obvious approach, since it finds plates which a human would identify as well. It is discussed in Section 7.3 Inherent Plates Require Parallel Top and a Bottom Shapes. The second approach, extruding plates, uses the mesh surface to extrude plates into the object (Figure 60). It is inspired by the extrusion manufacturing process, where a malleable material is pressed through a die. Here, a flat part of the model's surface is translated along its normal, resulting in the opposite side of the plate. This is described further in Section 7.4 Extruding Plates Only Requires One Shape. Our algorithm uses a combination of these two concepts. First, the model is searched for inherent plates. Afterwards, all surfaces of the model which were not used for creating a plate yet are forwarded to the extruding algorithm. While the first step finds plates which were actually sculpted in the model, the second step approximates a filled objects hull. Thus, we can handle complex meshes which feature both types of plates.

The concept of stacking plates is described in Section 7.6 Stacking Plates Is Similar to 3D Printing. The model is sliced in regular intervals, with plates being created based on the cross sections. When these plates are put on top of each other, they approximate the original model (Figure 61). Additionally, special plates, called shafts, are added. These are perpendicular to the other plates and help with aligning them during assembly.

7.2 PLATES ARE GENERATED FROM PLANAR SHAPES

Before these plate finding algorithms can process the model, it has to be prepared properly. The surface has to analyzed, with the faces be-

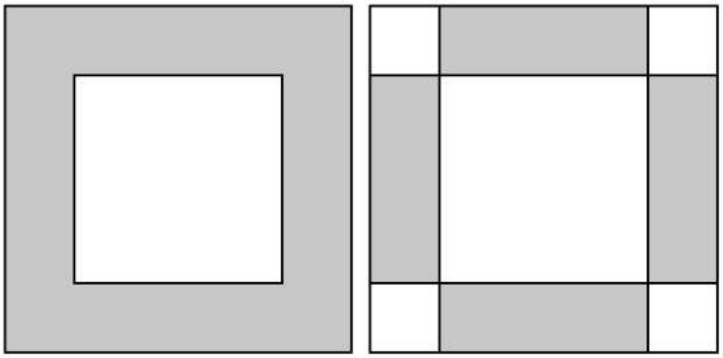


Figure 59: Finding inherent plates (Side view). The left image shows the hollow original model (grey), the right image shows the found plates (grey). The corner areas (white) are not found.

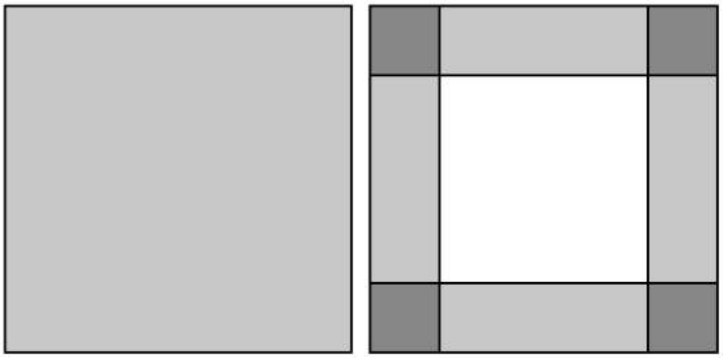


Figure 60: Finding extruded plates (Side view). The left image shows the filled original model, the right image shows the found plates (grey).The corner areas (dark grey) are found twice, while the center area (white) is not found.

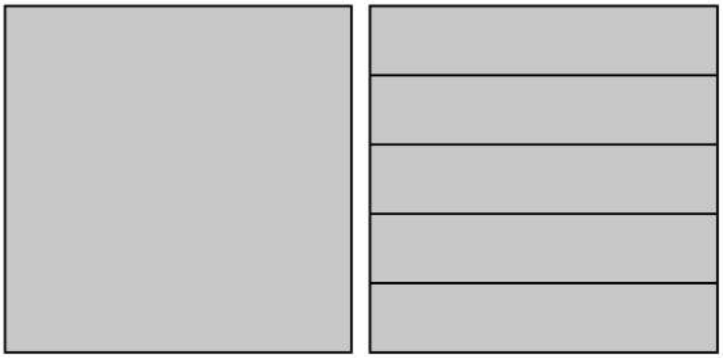


Figure 61: Finding stacked plates (Side view). The left image shows the filled original model, the right image shows the found plates.

ing grouped into planar shapes. These shapes can be used as surfaces of plates.

platener solves this by running multiple pipeline steps beforehand. The first, called *CoplanarFaces*, recreates planar shapes which have been divided when the model was triangulated by the creator. It groups the model's faces based on two test: The faces have to be connected and the angle between their normals must not exceed a given threshold. This step is described in Section 7.2.1 Coplanar Faces Are Grouped Based on Their Normal. In the next step, *edgeLoops* are created based on the previously generated groups of faces. An *edgeLoop* is a list of vertices, describing a shape which consists of multiple faces. Only the outer edges of the shape are stored, since the triangulation of the mesh is not relevant for *platener*. These *edgeLoops* are created by the *ShapesFinder*, which is discussed in Section 7.2.2 Shapes Finder. The *ShapesFinder* can create multiple *edgeLoops* for one shape. This means that the shape contains holes. In order to differentiate between the shapes and the contour, another pipeline step is run, which is called *HoleDetection*. It separates these by calculating the *edgeLoops'* area. This is further explained in Section 7.2.3 Hole Detection.

The result of these calculations is a list of shapes, which contain one or multiple *edgeLoops*. A label contains information about whether the *edgeLoop* is the shapes contour or a hole.

7.2.1 *Coplanar Faces Are Grouped Based on Their Normal*

By definition, faces are called coplanar if they lie in the same plane. For our purposes, we extend this by the requirement that the face have to be directly connected as well. This allows grouping them into planar shapes. An example mesh and the resulting planar shapes are shown in Figure 62. The algorithm for finding coplanar faces requires the models to be stored as a face-vertex mesh. This is calculated by *meshlib*, the library used for importing models. In a face-vertex mesh, each vertex and each face is assigned an index. Three lookup tables are created: These contain the faces, vertices and face normals of the mesh separately. Figure 63 shows an annotated mesh and the resulting lookup tables. The face lookup table contains the face indices and the indices of the vertices which compose the faces. Searching these in the vertex lookup table yields the coordinates of the vertices. The entries in the normal lookup table correspond with those in the face lookup table. For example, the normal of the face with the index 1 is located at index 1 in the normal lookup table. Face-vertex meshes enable faster adjacency checks than normal meshes, since vertex equality checks can be done by comparing the vertex indices. In fact, the *CoplanarFaces* step's computation time was reduced from 736 ms to

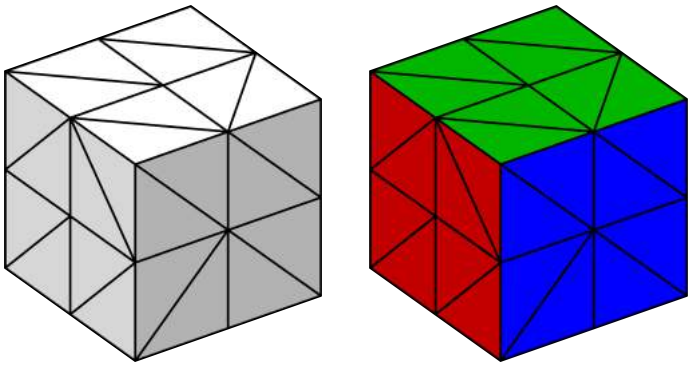


Figure 62: Finding inherent surfaces. The left image shows the original mesh, the right image shows the found planar shapes.

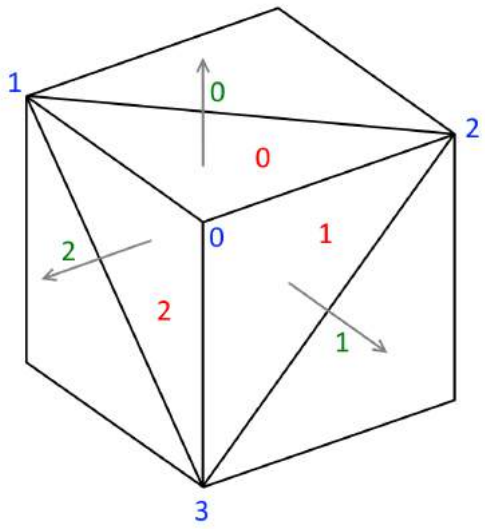
18 ms when processing a mesh with 2400 faces[1] by using a face-vertex mesh based implementation.

In order to enable further computations, two more lookup tables are added: One contains all edges belonging to each face, while the second one allows looking up the faces adjacent to an edge. The edges are stored as a sorted pair of vertex indices. In the face-vertex mesh shown in Figure 63 the face with the index 1 consists of the edges (0, 2), (0, 3) and (2, 3). Correspondingly, the edge (0, 2) refers to the faces 0 and 1. Both lists can be created in one single pass (see Listing 9).

```
setupFaceEdgeEdgeFaceLookup: ->
  for faceIndex in [0...faceCount]
    # Avoid double edges by sorting vertices
    { min_vertex
      mid_vertex
      max_vertex } = @findMinMidMaxVertex()
    # Register which edges this triangle uses.
    @addEntryToEdgeFaceMap(min_vertex, mid_vertex, faceIndex)
    @addEntryToEdgeFaceMap(mid_vertex, max_vertex, faceIndex)
    @addEntryToEdgeFaceMap(min_vertex, max_vertex, faceIndex)
    # Set the edges that make up this triangle.
    @faceVertexMesh.faceToEdges[faceIndex] = [
      [min_vertex, mid_vertex]
      [mid_vertex, max_vertex]
      [min_vertex, max_vertex]
    ]
```

Listing 9: Simplified lookup table generation.

1 This was measured on a Intel® Core™ i5-2400 CPU with 8 GB of RAM, running Windows 8.1 Pro and Node v0.12.7.



(a) Partly annotated mesh. Face are marked in red, vertices in blue and normals in green.

index	vertices
0	0, 1, 2
1	0, 2, 3
2	0, 3, 1
...	...

(b) Face lookup table.

index	coordinates
0	(1, 1, 1)
1	(0, 1, 1)
2	(1, 0, 1)
3	(1, 1, 0)
...	...

(c) Vertex lookup table.

index	direction
0	(0, 0, 1)
1	(1, 0, 0)
2	(0, 1, 0)
...	...

(d) Normal lookup table.

Figure 63: Example lookup tables of a face-vertex mesh.

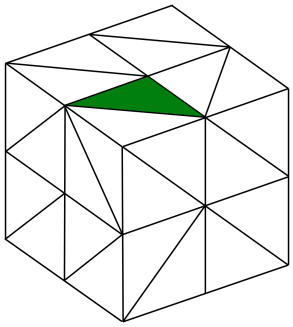
Using these lookup tables, the faces are grouped. A face group contains faces which are connected and are coplanar. This in done by iterating over all faces. When a face is found which has not been visited yet, a new face group is started. Now all of the face's edges are pushed to a queue, along with the current face index. Afterwards, we start working on the queue, using a breadth-first approach. The processing function is shown in Listing 10.

```
traverseAdjacentFaces: (...) ->
  if edgeQueue.length is 0
    return
  { edge, faceIndex } = edgeQueue.shift()
    faceNormal = @faceVertexMesh.getFaceNormal(faceIndex)
  # get the faces from the edge
  adjacentFaces = @faceVertexMesh.getFacesFromEdge(
    edge[0]
    edge[1]
  )
  continued = false
  for nextFaceIndex in adjacentFaces when nextFaceIndex
  ↪ isnt faceIndex
    # check if faces are coplanar
    # [...]
```

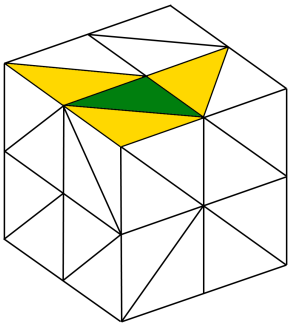
Listing 10: Function repeated for each edge in queue.

This is implemented as a *tail recursion*. First, the length of the queue is checked. If there a no more edges waiting to be processed, we can jump out of the recursion and continue searching for new groups. Otherwise, we first get the normal of the face from which we came. Next, we extract the new face from the edge and get the face's normal as well. Afterwards, we check the faces' coplanarity.

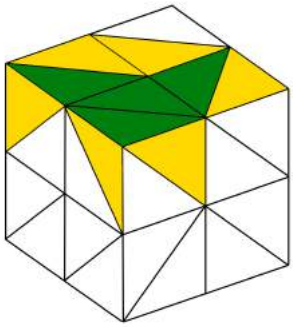
Listing 11 shows the coplanarity check. *isAngleZero* calculates the angle between the normals and compares it to zero. This is done even if the new face was already visited. If the faces are not coplanar, the edge is added to the group's outer edges. This list is used in the *ShapesFinder* when merging the faces to one shape. In case of coplanarity, we now check if the face was visited. If it was not, it is added to the face group. Additionally, an entry is added to the *faceToFaceGroup* lookup table and the face is marked as visited. After fetching the face's edges, they are pushed into the queue. The algorithm is visualized in Figure 64.



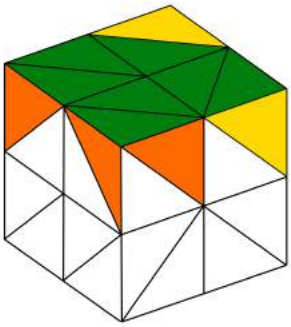
(a) An arbitrary face is selected as starting face (marked green).



(b) The adjacent faces are checked for coplanarity (marked yellow).



(c) Since they are coplanar, the faces are added to the group. Now, their adjacent faces are tested.



(d) The coplanar faces are added to the group (green). The orange faces fail the coplanarity check and thus are not added.

Figure 64: *CoplanarFaces* algorithm illustrated.

```
if @isAngleZero(faceNormal, nextFaceNormal)
  if not faceVisited[nextFaceIndex]
    faceGroup.push(nextFaceIndex)
    groupNormal.add(nextFaceNormal)
    @faceToFaceGroup[nextFaceIndex] = faceGroupIndex
    faceVisited[nextFaceIndex] = true
    adjacentEdges =
  ↪ @faceVertexMesh.getEdgesFromFace(nextFaceIndex)
    for nextEdge in adjacentEdges when not Util.arrayEquals
  ↪ edge, nextEdge
      edgeQueue.push({ edge: nextEdge, faceIndex:
  ↪ nextFaceIndex })
else
  outerEdgeGroup.push(edge)
```

Listing 11: Check for coplanar faces.

After all faces have been processed and assigned a group, the *faceGroups*, *faceGroupNormals* and *outerEdgeGroups* are injected into the face-vertex mesh, along with the *faceToFaceGroup* lookup table.

7.2.2 *Shapes Finder*

The shape finder creates shapes from the outer edge loops of the found coplanar faces. The outer edge loop is a set of edges in the format [vertex_index_1, vertex_index_2]. This is converted into a set of continuous edge loops of the form [vertex_index_1, vertex_index_2, ..., vertex_index_n, vertex_index_1].

To do so the *mergeEdges* function is called until no edge segments can be merged anymore (see Listing 12). This try to merge two edge segments which is possible if the starting or ending point of one of them matches the starting or ending point of the other (see Listing 13).

The resulting sets of continues edge loops are converted into *shapes* (see Listing 14). Therefore from each edge loop the last vertex is removed because it is the same as the first and our *edgeLoops* do not need this redundancy. Afterwards all vertex indices are mapped to there corresponding coordinated because we thus for our further calculations. This vertex sets are used to create *edgeLoops*.

With the resulting *edgeLoops* we create a *shape*.

```
mergeEdges: (inEdges, same) ->
  # take first edgeloop
  edges = [inEdges.shift()]

  # try to merge into other edgeloop
  merged = no
  for inEdge in inEdges
    added = no
    for edge, i in edges when not added
      newEdge = @mergeEdgeStrings edge, inEdge, same
      if newEdge
        added = yes
        edges[i] = newEdge
    if not added
      edges.push inEdge
    else
      merged = yes
  return { edges, merged }
```

Listing 12: Merging edge segments to continuous edge loops.

7.2.3 *Hole Detection*

The pipeline step *HoleDetection* is run after *ShapesFinder* and categorises every *edgeLoop* of each *shape* either as outer contour or as hole. This is done by computing their areas and comparing them: The *edgeLoop* with the biggest area is the outer contour and all others are holes.

7.3 INHERENT PLATES REQUIRE PARALLEL TOP AND A BOTTOM SHAPES

In order to find inherent plates in a mesh, all pairs of shapes have to be tested. The shapes have to be parallel to each other. Additionally, the distance between them is checked. In must fit one of the given plate thicknesses.

7.3.1 *Parallelism Is Checked Using Vector Functions*

The parallelism check uses vector functionality provided by *three.js*. After ensuring that the normals are parallel, we have to test if they are facing apart. Without this requirement, holes in objects could be recognized as plate. An example is shown in Figure 65. The vectors A and B in Figure 65a are facing towards each other. Using a vertex

```
mergeEdgeStrings: (
  edgeString,
  otherEdgeString,
  comparator = identityComparator
  ) ->
  lastIndex = edgeString.length - 1
  otherLastIndex = otherEdgeString.length - 1

  if comparator(edgeString[0],
  ↪ otherEdgeString[otherLastIndex])
    return otherEdgeString.concat(edgeString.slice(1))

  if comparator(otherEdgeString[0], edgeString[lastIndex])
    return edgeString.concat(otherEdgeString.slice(1))

  if comparator(edgeString[0], otherEdgeString[0])
    return
  ↪ edgeString.slice().reverse().concat(otherEdgeString.slice(1))

  if comparator(edgeString[lastIndex],
  ↪ otherEdgeString[otherLastIndex])
    return otherEdgeString.concat(edgeString.slice(0,
  ↪ -1).reverse())

  return null
```

Listing 13: Merging edge segments to continuous edge loops.

of vector A's shape and a vertex of vector B's shape, a new vector C is created. Since the angle between vector B and C is bigger than 90°, the shaped must face towards each other. Figure 65b shows a valid example: The angle between vector B and C is smaller than 90°, thus the shapes are facing apart. In this case, a plate can be created.

7.3.2 *Calculating the Best Plate Thickness*

The calculation of the optimal plate thickness is shown in Listing 15. First, the actual distance between the shapes is calculated. Next, all possible plate thicknesses are tested if they can approximate the actual thickness well enough. Lastly, the thickness with the lowest absolute deviation is chosen as the best thickness. Since the user selects which materials he has available, only those can be used. Thus, we do not use the actual thickness for our calculations, but rather the chosen best thickness.

```
createShape: (shape, groupIndex, faceVertexMesh) ->
  newEdgeLoops = []
  for _edgeLoop in shape
    edgeLoop = _edgeLoop.slice()
    edgeLoop.pop()

    for vertex, vertexIndex in edgeLoop
      vertex *= 3
      edgeLoop[vertexIndex] = new THREE.Vector3(
        faceVertexMesh.vertexCoordinates[vertex]
        faceVertexMesh.vertexCoordinates[vertex + 1]
        faceVertexMesh.vertexCoordinates[vertex + 2]
      )

    newEdgeLoop = new EdgeLoop(edgeLoop)
    newEdgeLoops.push newEdgeLoop

  shape = new Shape(
    newEdgeLoops,
    faceVertexMesh.faceGroupNormals[groupIndex],
    faceVertexMesh,
    groupIndex
  )
  { removalCount, shouldBeDeleted } =
   PointOnLineRemover.removeUnnecessaryVerticesInShape
  ↪ shape
  @_removedVerticesCount += removalCount
  if shouldBeDeleted
   return null
  return shape
```

Listing 14: Creating a *shape* out of a set of continuous edge loops.

7.3.3 *Plates Are Created by Intersecting the Shapes*

The plate creation uses the shapes' 2D representation, which is calculated when creating the shape. The vertices are rotated around the origin so that they have the same x-value. We use the shape's normal to build a rotation matrix, which is applied to a clone of each vertex. This rotation matrix is stored in the shape, while each of the shape's *edgeLoops* contains the corresponding rotated vertices. Since the rotation is performed deterministically, we can use the 2D representation to perform operations like inctersection and union on shapes.

After finding these plate candidates, the shape which plane's distance to the origin (the z-value of all vertices when laid into the x-y-plane) is smaller is selected as the base shape of the plate, as shown

(a) The normals are facing towards each other. No plate should be created.

(b) The normals are facing apart. The test is successful.

Figure 65: Checking if the shapes' normals are facing apart avoids creating unwanted plates.

in Listing 16. We only use one shape to represent the plate, since both shapes have to be congruent anyway. The other shape can be restored by translating the base shape using its normal and the plate's thickness. Now, the intersection of both shapes is calculated. This is done by using the 2D representation. The resulting intersection is transformed back into 3D space using the rotation matrix of the base shape. With the resulting shape, the plate is created.

The intersection between the shapes is done using a customized branch[2] of the Javascript Clipper library, which adds floating point coordinates support, as well as automatic data type conversion. After parsing them into the library's polygon class (see Section 5.3 Data Structures), they can be easily clipped, resulting in a list of intersections which can be parsed back into shapes. To obtain the vertices in 3D space, we use the previously calculated rotation matrix. The plate creation is based on the previously selected base shape. While the calculated intersection is used as the shape of the plate, the thickness is computed by subtracting the base shape's z-value from the other shape's z-value. Additionally, the base shape's z-value is used as plane constant.

7.4 EXTRUDING PLATES ONLY REQUIRES ONE SHAPE

The extrusion of plates is a simpler approach in terms of calculation, which is shown in Listing 17. The selected plate thickness has to be inverted, due to the plate being extruded in the opposite direction of the face's normal. The plane constant of the plates base plane is the same as the z-value of the shape's 2D representation. We compare the face's area to a threshold, which filters plates which would be to small to fabricate. Afterwards, the new plate is created.

2 https://github.com/platener/jsclipper

```
checkPlateThickness: (shape1, shape2) ->
  actualThickness = @distanceBetweenPlanes shape1, shape2
  okThicknesses = []
  # check which thicknesses are ok factor-wise
  for plateThickness in @plateThicknesses
    if plateThickness * @minThicknessDeviationFactor <
  ↪ actualThickness < plateThickness *
  ↪ @maxThicknessDeviationFactor
      okThicknesses.push plateThickness
  bestThickness = null
  bestDeviation = null
  for plateThickness in okThicknesses
    deviation = Math.abs(plateThickness - actualThickness)
    if bestDeviation is null or deviation < bestDeviation
      bestDeviation = deviation
      bestThickness = plateThickness
  return bestThickness
```

Listing 15: Finding the best plate thickness.

```
shape2CloserToOrigin = abs(shape2.zValue) < abs(shape1.zValue)
polygon1 = create2DPolygon(shape1)
polygon2 = create2DPolygon(shape2)
intersection = polygon1.intersect polygon2
shapes = parseToShapes(intersection)
plates = parseToPlates(shapes)
return plates
```

Listing 16: Face intersection for creating inherent plates.

7.5 REMOVING CONTAINED PLATES

RemoveContainedPlates is executed after plate generation and removes unnecessary ones that appear with our software.

There are two types of plates that get deleted: First as already described in the master thesis *Platener* [7] in Section '4.4.4 Removing contained plates', small plates can be created that completely lie inside another plate. Second there may be parallel plates that lie inside each other. That occurs due to extrusion if the the 3D object is modeled with thicker blocks than the available material thickness.

We iterate over all plates and compare them to all other plates. The comparison is based on their normals while further checks are not necessary most of the time because the normal check excludes most of them. Therefore the comparison is fast enough and the amount of comparisons must not be reduced with a different iteration algorithm.

```
createPlateFromShape: (shape) ->
  thickness = -@plateThicknesses[0]
  planeConstant = shape.edgeLoops[0].xyPlaneVertices[0].z
  if shape.getContour().computeArea() > @areaThreshold
    return new Plate shape, thickness, planeConstant
  else
    return null
```

Listing 17: Extruding a plate from a shape.

7.5.1 *Plate Contains Other Plate Completely*

As described in the master thesis the algorithm may produce small plates that lie completely inside the actual modeled plate. These plates are orthogonal.

So if two plates have orthogonal normals we check if one of them lies completely in the other and remove it.

7.5.2 *Overlapping Parallel Plates*

A 3D block can not be represented by one single plate if it is thicker than a plate. The software creates two plates instead. The resulting plates will overlap in case this block is thinner than those two plates together.

So if two plates are parallel we compare their plane constants which specify the distance to the origin. In case the plane constants hint at overlap the actual *EdgeLoops* are compared to identify intersections. In case they overlap one of the plates gets deleted.

7.6 STACKING PLATES IS SIMILAR TO 3D PRINTING

As an alternative approach, plates can be stacked. An example is shown in Figure 66. By putting multiple plates on top of each other, the model is approximated. In order to achieve this, the model is sliced in regular intervals, with plates being created based on the cross sections. This technique is similar to 3D printing, since it puts multiple layers of material on top of each other. In order to enable easy assembly, the plates are connected with shafts. These are additional plates, which are perpendicular to the other plates and help to align them.

The main function for stacking is shown in Listing 18. First, the model is rotated in order to optimize the stacking direction. Afterwards, the clipping planes are calculated. These plates are parallel to

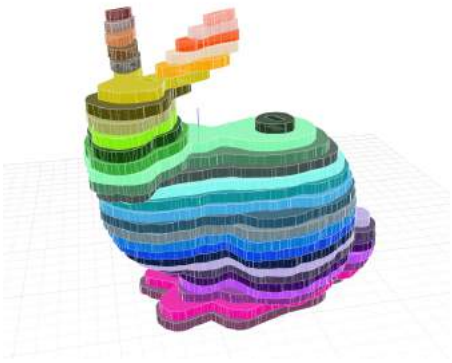


Figure 66: Stacked plates can be used to approximate the model.

the x-y-plane. They are used to create cross-sections of the model. This is done by intersecting each of the model's faces with these plates. Afterwards, the resulting edges are merged into edge loops, which are used for creating shapes and, as helper objects, polygons. With these, shafts can be added, which connect plates for easier assembly. Afterwards, plates are created. These have to be rotated back based on the original rotation in order to align them with the model.

```
findStackedPlates: (model, shapes) ->
  return new Promise (resolve) =>
    rotationMatrix = @findRotation shapes
    model.getClone().then((clone) =>
      @model = @rotateModel clone, rotationMatrix
      @planes = @getClippingPlanes()
      @clipFacesAgainstPlanes @model.model.mesh.faces
      @mergeEdgesInPlanes()
      @shapeGroups = @createShapes()
      @polygonGroups = @createPolygons()
      @shafts = @findShafts()
      @shapes = @clipShafts()
      plates = @createPlates().filter((p) -> p?)
      shaftPlates = @createShaftPlates()
      plates = plates.concat shaftPlates
      plates = @rotatePlatesBack plates, rotationMatrix
      resolve plates
    )
```

Listing 18: Plate stacking main function.

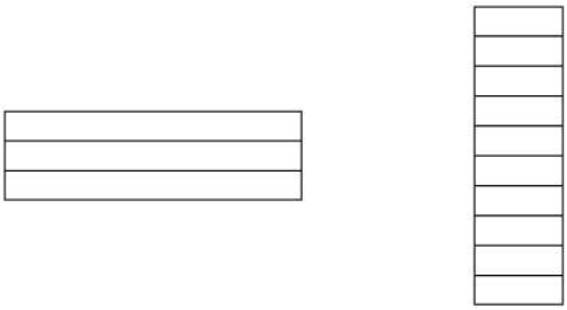


Figure 67: Different stacking directions yield different results.

7.6.1 *Rotating the Model Optimizes The Results*

In order to find a good rotation, the model's biggest surface is aligned to the x-y-plane. A good rotation results in fewer plates which are preferably all connected. Figure 67 shows how different rotations yield different results. While the approach of stacking plates doesn't require information about the model's coplanar surfaces, this optimization does. By iteration over them, the biggest surface's rotation matrix can be found (see Listing 19). This matrix can be used for transforming all vertices so that the chosen surface is parallel to the x-y-plane.

```
1  findRotation: (shapes) ->
2    maxArea = 0
3    rotationMatrix = new THREE.Matrix3()
4    shapes.forEach((shape) ->
5      area = shape.area || shape.getContour().computeArea()
6      if area > maxArea
7        maxArea = area
8        rotationMatrix = shape.rotationMatrix
9    )
10   return rotationMatrix
```

Listing 19: Finding an optimal rotation.

7.6.2 *The Clipping Planes Are Created in Regular Intervals*

Due to the model being rotated, it can be sliced using planes parallel to the x-y-plane. In order to calculate these, the model's bounding box is calculated first. Using the minimal and maximal z-value,

planes are creates with an distance equal to the selected plate thickness. These planes are constructed from a *three.js* plane object and a (initially empty) list of edges located in this plane. The planes are displaced by half the plate thickness. Thus, the sampling happens in the middle of the plates, which is an approximation which works for most applications. The plane creation is shown in Listing 20.

```
getClippingPlanes: ->
  planes = []
  for i in [@minZ..@maxZ] by @thickness
    planes.push {
      plane: new THREE.Plane(
        new THREE.Vector3(0, 0, 1),
        -(i + @thickness / 2)
      )
      edges: []
    }
  return planes
```

Listing 20: Clipping plane generation.

7.6.3 *The Model's Faces are Intersected with the Clipping Planes*

Clipping each face with each plane would be very slow. Thus, we first find one plane which clips the face and check the adjacent planes afterwards (see Listing 21). I order to quickly find this plane, binary search is used, as shown in Listing 22. Starting at the median plane, whose index is calculated with *@planes.length // 2*, we search the planes until one plane clipping the face is found or it is certain that no plane clips the face.

```
clipFaceAgainstPlanes: (face) ->
  planeIndex = @findClippingPlane(face)
  if planeIndex > -1
    @checkAdjacentPlanes(face, planeIndex)
```

Listing 21: Clipping a face against all planes.

```coffee
findClippingPlane: (face) ->
  stepWidth = @planes.length / 4.0
  index = -1
  currentIndex = @planes.length // 2
  oldIndex = -1
  while currentIndex isnt oldIndex and 0 <= currentIndex <
  ↪  @planes.length and index is -1
    direction = @clipFaceAgainstPlane(
      face,
      @planes[currentIndex]
    )
    # found clipping plane
    if direction is 0
      index = currentIndex
    # continue search
    else
      oldIndex = currentIndex
      currentIndex += Math.round stepWidth * direction
      stepWidth /= 2
  return index
```

Listing 22: Finding a plane which clips the face.

The search direction is calculated by clipping the face against the plane (Listing 23). After clipping each edge with the face, the number of intersection decides the result.

- No intersections: The face does not clip the plane.
- One Intersection: Invalid. It is not possible to get only one intersection when clipping a valid face.
- Two intersections: If both intersection points are the same, one vertex of the face clips the plane. Otherwise two edges clip the plane.
- Three intersections: If any two of the intersection points are the same, a vertex and the opposite edge clip the plane. Otherwise the whole face lies in the plane.

If the face does not clip the plane or only intersect with one vertex, we check if it is below or above the plane and return either -1 or 1 as direction (calculation shown in Listing 24). In all other cases, 0 is returned, because the face clips the plane and we don't have to search further.

Additionally, if either two edges, an edge and a vertex or the whole face clips the plane, these intersections are stored in the plane.

```
clipFaceAgainstPlane: (face, plane) ->
  intersections = []
  face.vertices.forEach((vertex, index) =>
    start = vector(vertex)
    end = vector(face.vertices[(index + 1) % 3])
    line = new THREE.Line3(start, end)
    intersection = plane.plane.intersectLine line
    if intersection? then intersections.push intersection
  )
  # handle intersections
  # [...]
  return direction
```

Listing 23: Clipping a face against a plane.

```
getDirectionFromPlaneToFace: (face, plane) ->
  sum = 0
  face.vertices.forEach((vertex) ->
    sum += vertex.z + plane.plane.constant
  )
  if sum is 0 then throw new Exception()
  return sum / Math.abs sum
```

Listing 24: Calculating the direction from a plane to a face.

After one plane which intersect the face has been found, the adjacent ones have to be checked as well, since a face can span over multiple planes. This is done by iterating over the planes, starting from the next one and moving away. If the returned direction is not 0, we can stop because the plane and all following ones do not clip the face. Both directions, upwards and downwards, are checked separately (see Listing 25).

7.6.4 *The ShapesFinder Is Used to Create the Cross-Sections*

Using the intersections stored in each plane, shapes can be created. This is done using the *ShapesFinder*, which is described in Section 7.2.2 Shapes Finder. Instead of the outer edges of face groups (see Section 7.2.1 Coplanar Faces Are Grouped Based on Their Normal), the intersections between the faces and the clipping planes are used. The resulting shapes represent the cross-sections of the model. Additionally, the Javascript Clipper library is used to create 2D polygons, which are used in the next step.

```
checkAdjacentPlanes: (face, index) ->
  runIndex = index + 1
  direction = 0
  while(runIndex < @planes.length and direction is 0)
    direction = @clipFaceAgainstPlane(
      face,
      @planes[runIndex++]
    )
  runIndex = index - 1
  direction = 0
  while(runIndex >= 0 and direction is 0)
    direction = @clipFaceAgainstPlane(
      face,
      @planes[runIndex--]
    )
  return
```

Listing 25: Checking if adjacent planes are clipping too.

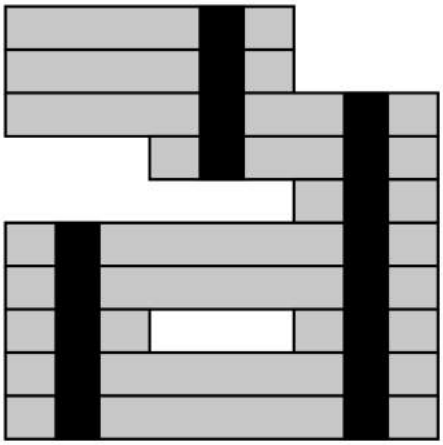


Figure 68: Side view of stacked plates (grey) connected by shafts (black).

7.6.5 *Shafts Are Added to Ease Assembly*

In order to assemble the stacked plates, we connect them with shafts (Figure 68). Our approach for this: if two shapes which are located in adjacent planes intersect, they have to be connected by at least one shaft. The shaft finding algorithm has three steps: First, we iterate over all shapes and connect as many as possible. Next, we fix shapes which are not yet connected. Afterwards, we clean up shafts which are only connected to one shape. The shaft object contains a list of the polygon it connects, as well as the intersection of all these polygons.

7.6.5.1 *Adjacent Layers Are Intersected to Add Shafts*

Instead of the actual shapes, the corresponding polygons are used, which enable working with the Javascript Clipper. These are organized in layers, matching the clipping planes. Each layer contains a list of polygons, since there can be multiple shapes in one layer. We have to iterate over both lists, the inner and the outer, as shown in Listing 26.

When processing a new polygon, we first mark that it has not been added to any shaft yet. Now, we iterate over all shaft candidates. If the current level (the layer) is bigger than the shaft's last added polygon's level plus one, there was no polygon from the last layer added to the shaft. Since we do not want shafts to create "bridges" spanning between unconnected layers, the shaft is marked as completed. If that is not he case, we try to add the polygon to the shaft.

Listing 27 shows the calculations run when adding a polygon to a shaft. First, the shaft's intersection and the new polygon are intersected. The resulting intersection is checked if it is big enough to fit the shaft. If that is the case, the shaft's intersection is updated and the polygon is added to the shaft.

If the polygon can not be added to any of the shafts, we create a new one - if the polygon is big enough (Listing 26). Additionally, the shaft is overlapped "backwards", up to three layers. This improves stability in the assembled model.

7.6.5.2 *Unconnected Plates Are Connected by Expanding Shafts*

In some cases, not all polygons - and thus not all plates - are connected to the others yet. Additionally, shafts should be expanded in both directions as far as possible. A fix is shown in Listing 28.

Each polygon is checked how well it is connected to other polygons. If it is connected to both a polygon below and above (*connection.isConnected* is true), we do not have to connect it further. Otherwise, the shafts connecting it from above and below are checked. We try to expand these to connect more polygons in the given direction. If this succeeds, handling for this polygon is completed. Otherwise, we create a new shaft at this polygon and try to expand it in both directions.

7.6.5.3 *Unused Shafts Are Removed Again*

As a result of this method, polygons which are not adjacent to any other polygon get assigned an own shaft. Since this does not improve the models stability, these are removed again, as shown in Listing 29.

7.6.6 *The Shafts Are Clipped From the Polygons While Creating Plates*

After the shafts are found, their cross section is cut from the connected polygons. Next, these polygons are parsed into shapes and plates. Additionally, the shafts' contours are calculated, with additional plates being created from them. Together, all plates are returned.

```
findShafts: ->
  shaftCandidates = []
  @polygonGroups.forEach((polygonGroup, level) =>
    polygonGroup.forEach((polygon) =>
      added = false
      shaftCandidates.forEach((shaftCandidate) =>
        if level > shaftCandidate.lastlevel + 1
          shaftCandidate.finished = true
        if not shaftCandidate.finished
          if @tryAddingPolygonToShaftCandidate(
                polygon, shaftCandidate
              )
            shaftCandidate.lastlevel = level
            added = true
      )
      if not added
        if Shaft.isIntersectionBigEnoughForShaft(
              polygon.polygon, @shaftData
            )
          newShaftCandidate = Shaft.fromPolygon(
            polygon,
            level,
            @shaftData
          )
          newShaftCandidate.polygons[0].shafts.push(
            newShaftCandidate
          )
          @tryOverlappingShaftBackwards newShaftCandidate
          shaftCandidates.push newShaftCandidate
    )
  )
  @fixUnconnectedPlates shaftCandidates
  @cleanUpShafts shaftCandidates
  shaftCandidates.forEach((shaft) ->
    shaft.createShaftContourAndCrossSection()
  )
  return shaftCandidates
```

Listing 26: Finding shafts.

```
tryAddingPolygonToShaftCandidate: (
    polygon, shaftCandidate, direction
  ) ->
    clip = polygon.polygon.intersect
    shaftCandidate.intersection
    if clip.length > 0 and
        Shaft.isIntersectionBigEnoughForShaft clip[0],
    @shaftData
      newIntersection =
        new jsclipper.Polygon clip[0].getShape(),
    clip[0].getHoles()
      shaftCandidate.intersection = newIntersection
      if direction is "DOWN" then
    shaftCandidate.polygons.unshift polygon
      else shaftCandidate.polygons.push polygon
      polygon.shafts.push shaftCandidate
      return true
    return false # no intersection
```

Listing 27: Adding a polygon to a shaft.

```
fixUnconnectedPlates: (shaftCandidates) ->
  @polygonGroups.forEach((polygonGroup, level) =>
    polygonGroup.forEach((polygon) =>
      connection = @isPolygonConnected polygon, level
      if not connection.isConnected
        expanded = false
        direction = "DOWN"
        if connection.shaftsFromAbove.length > 0
        expanded = @tryExpandingShaftsInOneDirection(
          connection.shaftsFromAbove, level, direction
        )
        if connection.shaftsFromBelow.length > 0
          direction = "UP"
          expanded = @tryExpandingShaftsInOneDirection(
            connection.shaftsFromBelow, level, direction
          )
        if not expanded and
            Shaft.isIntersectionBigEnoughForShaft(
              polygon.polygon, @shaftData
            )
          newShaftCandidate = Shaft.fromPolygon(
            polygon
            level
            @shaftData
          )
          newShaftCandidate.polygons[0].shafts.push(
            newShaftCandidate
          )
          @tryExpandingShaftAroundLevel(
            newShaftCandidate
            level
            direction
          )
          shaftCandidates.push newShaftCandidate
    )
  )
```

Listing 28: Fixing unconnected plates.

```
cleanUpShafts: (shafts) ->
  if shafts.length > 0
    for i in [shafts.length - 1..0]
      if shafts[i].polygons.length < 2
        shafts[i].polygons.forEach((polygon) ->
          polygon.shafts.splice polygon.shafts.indexOf(shafts[i]), 1
        )
        shafts.splice i, 1
```

Listing 29: Cleaning up shafts.

8 ADJACENCY PLATEGRAPH

On the basis of the plategraph we create connectors for plates in the later step 9 JOINT COMPUTATION. The algorithms picks an adequate connector type based on angles and neighborhood relationships.

In this step we analyse the spatial arrangement of plate objects in 3D space to create a graph structure which tracks the adjacency. The plates that are supported to be analysed are detected in the previous step 7 PLATES. Two or more plates are adjacent to each other when at least one side touches or overlaps with the other plate. In addition, the angles in between the plates are calculated.

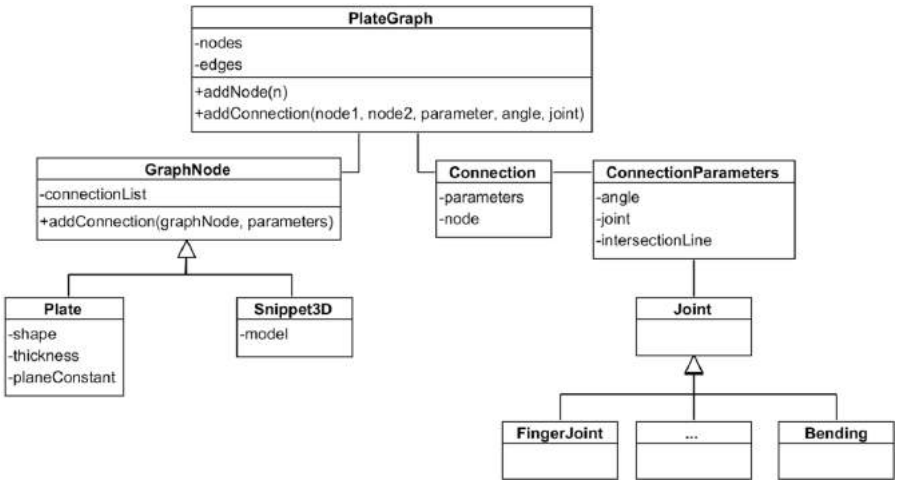


Figure 69: Class diagram of the plategraph structure. Nodes and Connections can be added to the graph when a new intersection has been found.

The graph, shown in Figure 69, holds two lists. One tracks the parts which have been found by the previous step 7 PLATES. Such a part is called graph node. And the other list tracks the connections of the nodes which we call edge.
Each node saves all edges it belongs to in a connection list. Each connection exists twice, one for each node it connects. Therefore we can traverse from one node to another via such a connection. Additionally, a connection saves properties like the angle and the intersection line at which the current node will need to add its joints in the next step (9 JOINT COMPUTATION). Which type of joint is created there

depends on the type which is saved in the connections property *joint*. The parts which can be connected are plates. But we support the existence of other node types such as a 3D-snippet which is a part of the model which has not been converted to plates.

8.1 DETECTING SPATIAL ARRANGEMENT

As a prerequisite to building up a plate graph the step 7 PLATES needs to find inherent plates or create extruded plates first. Afterwards the plane-plane intersections of all combinations of both sides of the plates are computed. This results in up to four intersection lines. In preparation for the step 9 JOINT COMPUTATION we truncate the inner intersection lines that would otherwise overlap with adjacent plate intersections.

8.1.1 *Finding Intersections*

When two planes intersect there is an intersection line. Since we work with plates which can be interpreted as two equal parallel planes we expect to find up to four intersection lines.
First, we retrieve the direction vector *dir* of an intersection line between two plates by calculating the cross vector of both normals. In order to retrieve all possible intersection lines of two plates we then calculate four possible plane-plane intersections of

- the two main sides of the plates
- the two parallel sides of the plates
- one main and one parallel side
- one parallel side and one main side

First, a possible position vector p that lies on both planes is calculated [4].

Plane constants: d_1, d_2
Normals: $\vec{n_1}, \vec{n_2}$

$$p = \frac{d_1 * \vec{n_2}^2 - d_2 * (\vec{n_1} * \vec{n_2})}{\vec{n_1}^2 * \vec{n_2}^2 - (\vec{n_1} * \vec{n_2})^2} * \vec{n_1} + \frac{d_2 * \vec{n_1}^2 - d_1 * (\vec{n_1} * \vec{n_2})}{\vec{n_1}^2 * \vec{n_2}^2 - (\vec{n_1} * \vec{n_2})^2} * \vec{n_2}$$

On the basis of the position vector p and the direction $\vec{dir}$ the intersection line can be computed: $line = p * x + \vec{dir}$
For the four possible plane-plane intersections we calculate four position vectors which yields us four lines.

Now that all lines are found we have to test if the lines actually go through both plates and not only the according planes and that the plates touch. Figure 70 shows all infinite intersection lines and has the actual plate intersections highlighted. In addition, this step retrieves

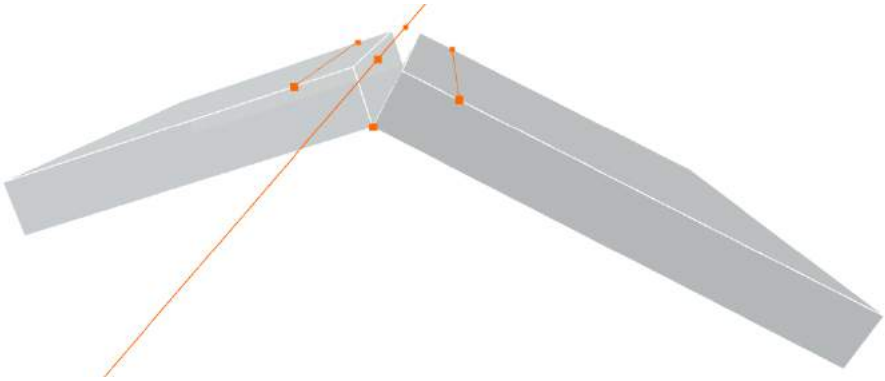


Figure 70: Single plate pair with all of its intersection lines. The infinite line will be clipped to the length of the other lines.

the exact start and end points of the line segment that defines the intersection of both plates. In order to find these boundaries we calculate the intersections of the lines with all boundary edges of the plates. The result is shown in Figure 71.

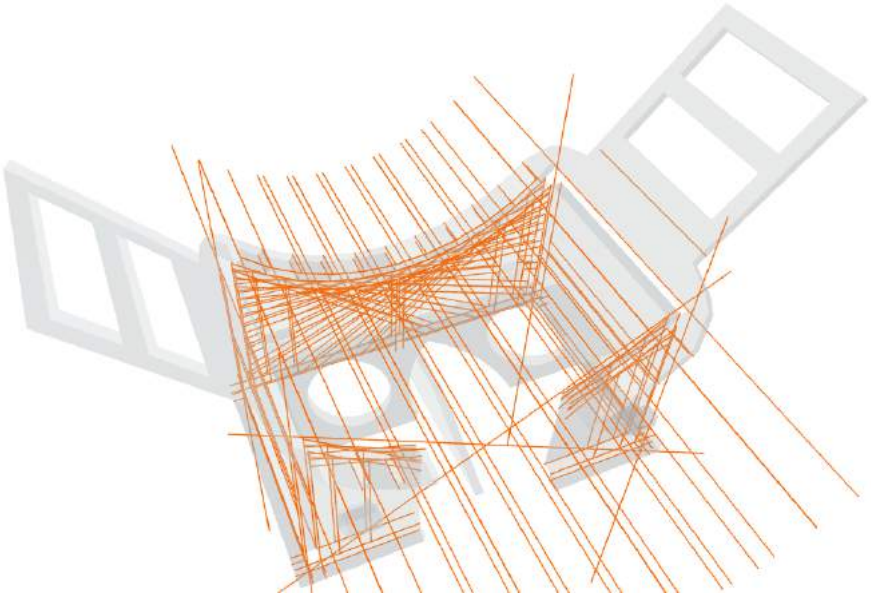


Figure 71: All intersections that were found in the model of a head mounted display.

8.2 PREPARING PLATES FOR CONNECTORS

8.2.1 *Volume Based Clipping*

Before joints can be added to the plates we have to prepare them. We cut the shapes so that the joints do not overlap with the other plate. As seen before, up to four intersection lines have been calculated per intersection. Two of them lie on one side of the plate and the other two belong to the second side. The two lines on one side build a rectangle when their ends are connected, see Figure 72. We use those two rectangles to remove the parts of the plates where we will place finger joints. The shape is rotated in the xy-plane, as well as the rectangles. A 2-dimensional clipping algorithm is now used to create the new shape. When the shape is rotated back into the 3D scene, see Figure 74, the overlapping parts of the plates are removed.

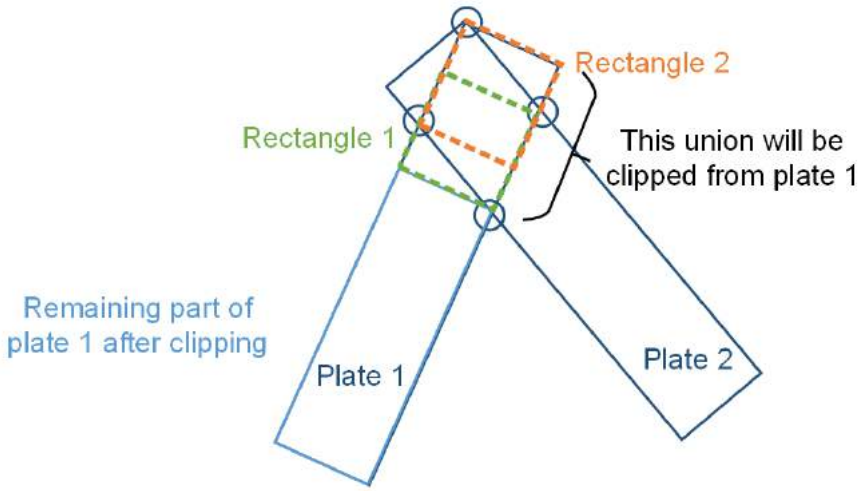


Figure 72: Profile view of a single plate pair. The circles mark the positions of intersection lines. The rectangles show what is clipped off plate 1.

In some cases not all four lines intersect within the plates. Therefore the algorithm does not derive four intersection points, but one to three as well, see Figure 73.

In these cases the infinitely long plane intersection line will be clipped to match the length of the existing plate intersections. This yields two rectangles as well, which can then be used for clipping the shapes of the plates.

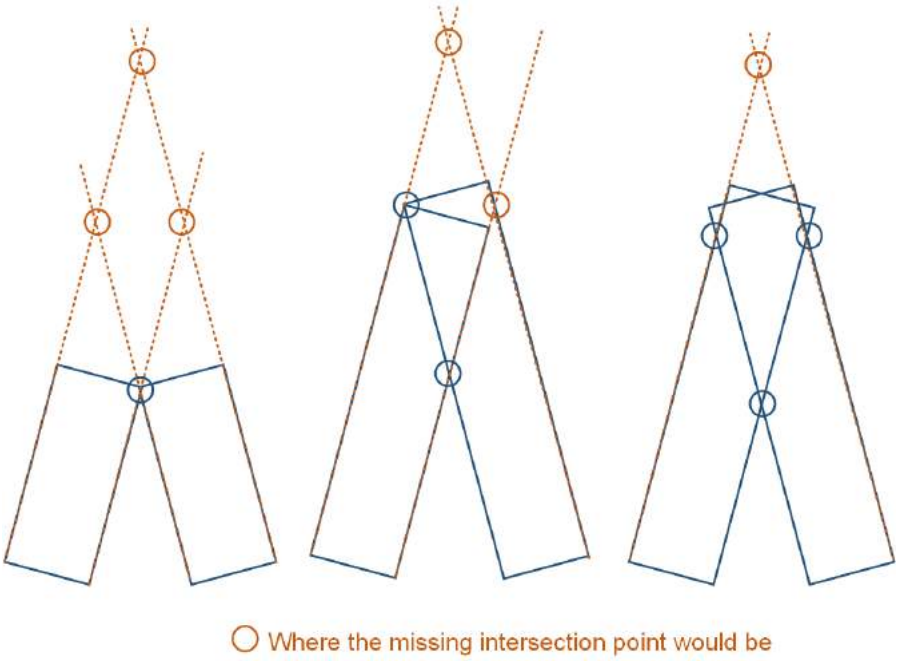


Figure 73: There can also be less than 4 actual intersections. In this case we have to calculate the projected intersection lines as well as the ones which are only a correct plane-plane-intersection but not actually going through a part where the plates overlap

8.3 ANALYZING SPATIAL ARRANGEMENT

8.3.1 *Angle Calculation*

In order to generate fitting joints in the upcoming step and for grouping plates to bent plates (see Chapter 10 CURVES) we have to measure the angles of the plates.
First, we determine the angle between the according planes.
plane normals: $\vec{u}, \vec{v}$
angle between planes: θ

$$cos(\theta) = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}| * |\vec{v}|}$$

This angle would be correct if we were working with infinitely large planes. Instead, we need the angle which is enclosed by the plates. Therefore, we need to adjust the angle whenever the plane angle does not correspond with the plate angle. This depends on where the normals are faced, see Figure 75. Thanks to the previous clipping step we can find out when the angle has to be adjusted.
As seen in Figure 76, plates with an acute angle still touch when cut back and plates with an obtuse angle do not.

(a) Two plates before clipping. (b) Two plates after clipping.

Figure 74: When the shapes have been cut in 2D they are rotated back to form a 3-dimensional plate again. This figure depicts the volume which was cut off by the algorithm leaving the remaining plates p_1 and p_2

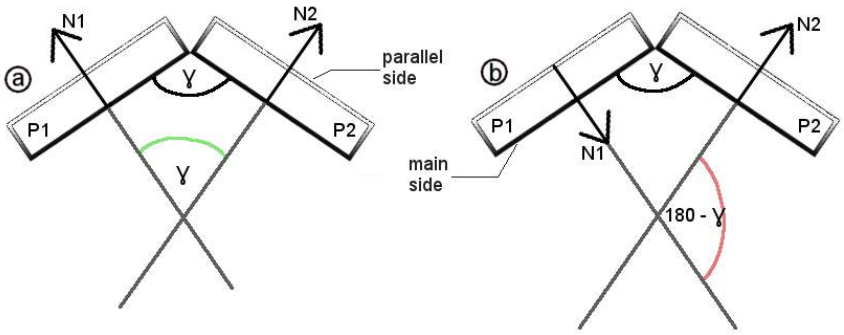
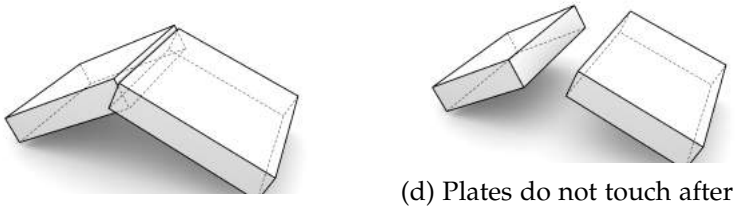


Figure 75: (a) The angle between the plates' normals correspond with the angle γ between the plates. (b) The angle between the plates' normals is in this case an adjacent angle to the requested angle γ.

(a) Plates with an acute angle.

(b) Plates still touch after clipping.



(c) Plates with an obtuse angle.

(d) Plates do not touch after clipping anymore.

Figure 76: After the clipping process the plate pair might be not touching anymore. This means that the angle between the plates has to be larger than 90°.

Finally, we test if the plates still touch. This is how we know if the angle is acute or obtuse. If the originally calculated angle does not correspond with this we adjust the plate angle by subtracting it from 180°.

8.3.2 *Finding New Main Lines*

In order to know where to add the joints to the plate we have to identify one line segment for each plate whithin a connection.
For that we compare the distances of the new edges of the two plates which are shown in Figure 77. The lines belonging to the smallest distance determine where to place the joints.

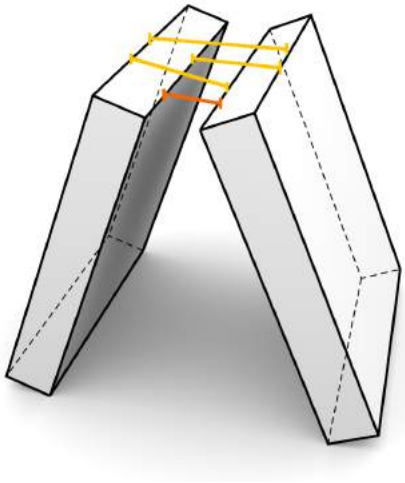


Figure 77: The distances between the new edges after the clipping process reveals which of them are the inner lines. Those lines will be used for adding joints in the next step.

The segments are called main line for that connection and plate.

8.3.3 *Truncating Intersection Lines*

Now that all main lines have been detected we need to shorten the lines so that no other lines overlap with it. This step is crucial because the following step 9 JOINT COMPUTATION would run into problems because the joints overlap each other.
If we now have a look at the intersections of plates in a model (see Figure 71) we can identify the overlaps. For truncating the lines we use the following algorithm 30 to achieve the result shown in Figure 78.

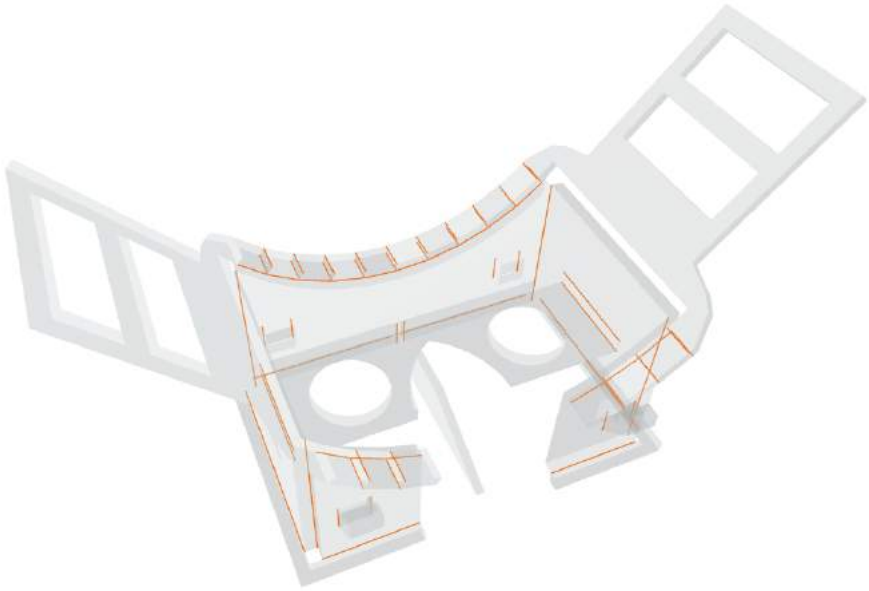


Figure 78: All found intersections have been overlaping. After the truncation they only cover exactly the line which is needed for adding joints later.

First, we take two of all the found lines and check if they intersect. They only intersect when their intersection point is on both of their line segments. Then, the two parts of both lines can be created since we calculated the intersection point which splits the lines. We assume that the part of the line we want to keep will always be the larger part. Therefore, we discard the shorter lines and return the others.

Finally, we derived all plate adjacencies and necessary information on the connected parts. We also adjusted the plates by cutting the plates where they intersected with another. Therefore the next step explained in Chapter 9 JOINT COMPUTATION can add joints.

8.4 ALTERNATIVE APPROACHES

The solutions presented in this chapter may not be the only possible solutions. Therefore, we want to give an overview of other algorithms we tried or are aware of and why picked our current solution.

8.4.1 *Possible Data Structures for Intersection Tracking*

Beyer et al. used an adjacency matrix for keeping track of neighboring plates. All the plates were kept in a plate array. Their indices corresponded to the rows and columns of the matrix. Each cell contained a set of plate adjacency objects. This made it possible that one plate could have more than one connection.

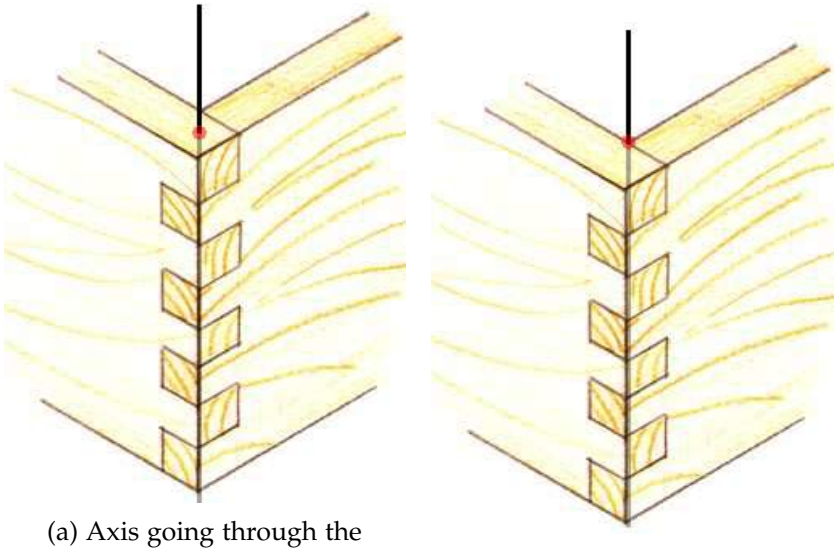
```
for line_pair in lines
  line1 = line_pair[0]
  line2 = line_pair[1]
  point = lineLineIntersection(line1, line2)
  if isPointOnLine(point, line1) and isPointOnLine(point, line2)
  {
    # linesegments actually cross
    startSegmentLine1 = new Line(point, line1.start)
    endSegmentLine1 = new Line(point, line1.end)
    startSegmentLine2 = new Line(point, line2.start)
    endSegmentLine2 = new Line(point, line2.end)
    # the shorter part of each line is discarded
    if startSegmentLine1.distance() > endSegmentLine1.distance()
      if startSegmentLine2.distance() > endSegmentLine2.distance()
        return {
          endSegmentLine1
          endSegmentLine2
        }
      else
        return {
          endSegmentLine1
          startSegmentLine2
        }
    else
      return {
        startSegmentLine1
        startSegmentLine2
      }
  }
```

Listing 30: Truncating lines

Instead of an adjacency matrix we used the previously explained plategraph class structure because it allows for traversal along the plates and additionally the found edges.
Moreover, our structure is open for adding additional objects beside plates. For example 3D printable snippets which were not converted to plates by our software because they were either too small or contained aesthetical objects important to the user.

8.4.2 *Approaches for Finding Intersections*

In the master thesis on platener[7] a connection is calculated by finding all four possible connections between both sides of two plates. Then the edges of the plates are tested against the line. The part of



(a) Axis going through the middle of the joints.

(b) Plate specific intersection lines

Figure 79: Two possible representations of where to place joints.

Source: http://www.technologystudent.com/images/fingjt1.jpg (visited on July 25, 2016)

the line overlapping with an edge was kept for further calculations.

When looking for plate intersections we also calculate all possible intersections. But plates do not always touch each others edges. Therefore we replaced this algorithm by a different approach explained in the previous section (8.1.1 Finding Intersections).

8.4.3 *Representation of the Joint Location*

As seen before in the Section 8.3.2 Finding New Main Lines we calculate one line per intersection and plate which determines where we will place the joints. The solution is simple because the joints only have to be moved onto the line and can directly be added to the plates.
A different, more future oriented, approach is to work with the axis going through the middle of where the joints will be. This means it also defines the axis around which the plates could rotate when their angle changes, see both options shown in Figure 79. In the case that our software will be extended with an editor the user might want to select an edge and define that the connection should be similar to a hinge. For this you would want rather one axis than two lines specific to a plate. On the other hand then the calculation for where to place the joints at the plate becomes more expensive.

In our use case the single line approach for an edge of a plate is completely sufficient.

8.4.4 *Finding Main Lines Based on Shape Corners*

We already mentioned that when plates intersect there can be up to four intersection lines. We need to define which of them is the most important one in order to know where to put joints. The line we wanted to choose had to touch both plates.
This means all lines which intersected with any points of the shapes of the plates were ignored. The other line became main line.
The assumption we adopted (see Figure 80) from the platener thesis [7] for finding the main line is the following:
Two plates always touch one another edge-to-edge. Therefore, if an intersection line does not lie on an outer boundary of the shape it is the main line.

(a) The only intersection line was immediately marked as main line.

(b) The orange square marks the inner intersection line which we chose as main line.

Figure 80: Our assumption was that we only need to define one main line per plate pair. If there was just one intersection at all it was definitely the correct one. And when there were more than one intersection we thought the inner intersection should be the main line.

The problem about this approach is that plates can intersect in many different ways, see Figure 81, which do not all verify our first assumption. This is why we replaced this algorithm with the one from the previous section (8.3.2 Finding New Main Lines).

8.4.5 *Adjusting Angles Based Where Plates Touch*

Our previous version of finding a main line was supposed to yield one line. Based on this line the angle was calculated with the following procedure:

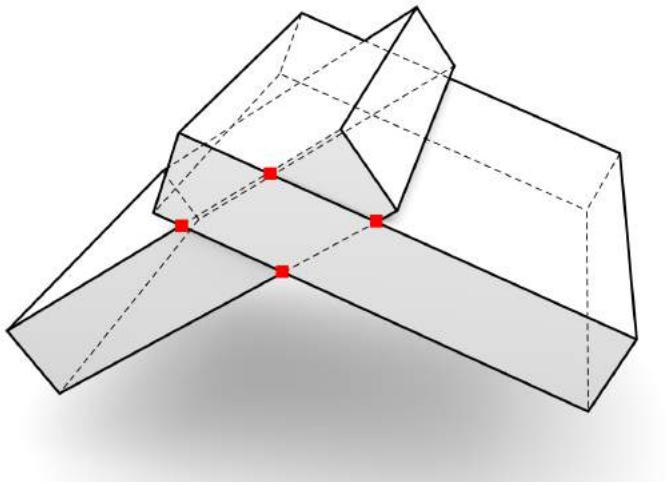


Figure 81: In this case the angle between the plates is obtuse. If we just took the inner intersection here without first clipping of the plates there would be no space for joints. Additionally, none of the intersection lines marked here in red lie on an outer boundary point. Therefore they would all have been marked as main.

After the angle of the planes had been calculated we adjusted the angle in some cases dependent on the direction in which the normals were pointing.
We know which sides of the plates are enclosing the angle thanks to the previous main line calculation.
In order to find out in which cases the angle has to be adjusted we check for two properties.
First, there is the question which of the sides of the plates intersect. A plate is defined by a 2D-shape which is called main side. The other side which exists due to a specified thickness of the plate is called parallel side.
Additionally, we take the direction of the normals into account. The direction is positive when it is directed from the main to the parallel side and negative otherwise.
Angles that fit the following conditions need to be adjusted.

- The two plates touch with the same type of side (main or parallel)

  and

- Their directions are both positive or both negative

But, as mentioned before, all of these calculations were based on the assumption that the previous main line calculation only yielded one line.
If that is not the case the previous algorithm fails which is why we developed a new approach for finding the correct angle as already explained in the Section 8.3.1 Angle Calculation.

8.5 FUTURE WORK

A few more things have to be accomplished to allow for a smooth conversion.

8.5.1 *Multiple Line Segments per Edge*

Currently, we only calculate one line per edge but as can be seen in Figure 82 this is not enough. To solve this the line segments should be found and for each segment a new edge should be created to be handles seperately.

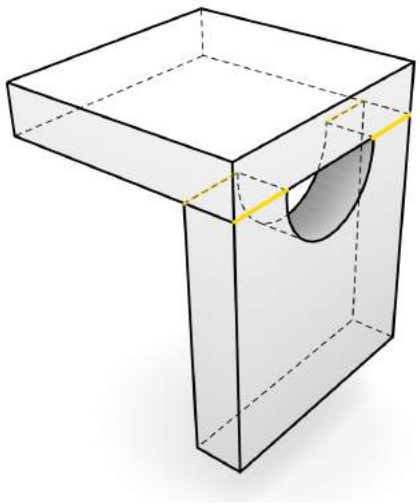


Figure 82: The vertical plate touches the horizontal plate twice. This should result in two edges and intersections which should be handled seperately.

8.5.2 *Detect if the Model is Producible*

A very important information about the final converted model is if it can actually be build. A user may not only need assembly instructions on how to build the model when cut but also if it is possible to do so. There may be a conflict when plate a needs to be added before plate b but plate b also needs to be put up before plate a. This should never happen or at least be mentioned to the user.

8.5.3 *Rejoining Broken up Plates*

In the master thesis [7] it was mentioned that plates can be split by another one standing perpendicular to it. This should result in a t-junction. Therefore the plate graph has to recognize those broken up plates and reunion them. The suggested solution is to group two plates together again when the following statements are true for three plates *P1*, *P2*, *P3*:

1. the main sides of P1 and P2 are coplanar
2. P1 and p2 have the same thickness
3. P1 and P2 each have a shared edge with p3 that is parallel and which overlaps when projected onto each other

In order to reunion such plates the following steps are necessary:

1. Iterate over all nodes and check if the current node has two or more neighbors.
2. Test if any of the neighbors are coplanar to each other. If so they are labeled as plates P1 and P2. Any other neighbor is labeled P3 or higher.
3. Check if P1 and P2 have the same thickness. If not start over with the next node
4. If they have the same thickness the intersection lines need to be checked to be parallel and overlaps between P1 and P3, and P2 and P3.
5. When determined that P1 and P2 should actually be one plate which has been parted by P3 then they need to be unioned and marked for receiving a t-junction in the next step.

9

JOINT COMPUTATION

9.1 JOINT COMPUTATION

A laser cutted object needs connectors if it consists of multiple parts. Those can be elements like screws, nails and glue. Or the plates already come along with connectors. Those can be for example finger joints which, when well calibrated, do not need any external material to hold together.
Whenever possible we want to provide connections like that because a user needs less effort when components work directly out of the laser cutter. This is why we provide three types of joints which are added to the determined plates.
The previous 8 ADJACENCY PLATEGRAPH step provides all edges that need to receive joints, the angle of each edge and the line along which each plate receives the joints.

9.2 BUILDING JOINTS

The general procedure of creating joints consists of three steps: First we calculate how many female and male joints fit onto the intersection. Then the retrieved joints are placed at the intersection line and the shapes are merged to result in the original plate with joints.

Building the Joints for a Specific Intersection Line

First we find out how many joints, males and females, fit along the length of the line. Since all joint types have equal widths for female and male joints we divide the length of the line by the width of a joint.

But the result from this calculation has to be rounded because the number of joints just found do not necessarily work out evenly with the length of the line. This yields us a whole number which defines how many joints we need to create. Then we adjust the width so that the joints will be evenly spread without leaving space on either end of the line.
Now that we know how the joints will look like and how many we need it is time to find out how to distribute them.

In order to always create a well defined female and male part we place a thick joint in the center of the line if there are an even number of joints to be distributed, see Figure 83.

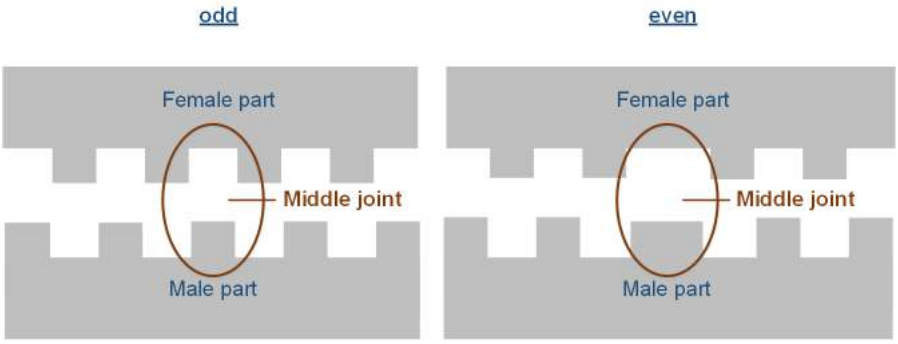


Figure 83: Distribution of joints depending on the number of joints. An even number results in a thick inner joint.

On both sides of the middle joint will be an equal number of male joints. How many depends on the jointCount. An even jointCount means 2 joints less and odd means one joint less on the sides because the middle joint will be created seperately. Since the jointCount specifies the number of all joints, male and female we have to divide by two to get the number of male joints only and then divide by two once more to achieve the seperation into the two sides.

$$\text{maleJointsPerSide} = \begin{cases} (jointCount - 2)/4, & jointCount \ \% \ 2 == 0 \\ (jointCount - 1)/4, & jointCount \ \% \ 2 == 1 \end{cases}$$

Finally, we can start placing the middle joint and distributing as many joints as just calculated on either side of it evenly with leaving enough space in between the joints for a female one to fit in as can be seen in Figure 84. The computation for the number of female joints on one side of the middle joint is very similar to the previous computation for male joints. Only that the middle joints do not have to be subtracted.

$$\text{femaleJointsPerSide} = \begin{cases} (jointCount)/4, & jointCount \ \% \ 2 == 0 \\ (jointCount + 1)/4, & jointCount \ \% \ 2 == 1 \end{cases}$$

When creating *finger joints* the female joints are the exact negative of the male joints. Therefore we do not create these joints one by one. Instead we retrieve the boundingbox of the male joints along the length of the intersection line and calculate the difference of this rectangle and the male joints. The result are the female joints.

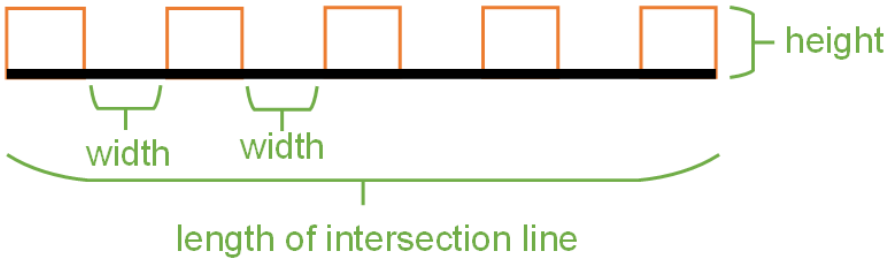


Figure 84: The joints are spreaded evenly along the line leaving space of a joint with in between for the other plates joints to fit in.

In the case of *dovetail-* and *jim-joints* we have to create the female joints by placing each single joint along the intersection line.
This is achieved by distributing them in the same way as the male joints are evenly spread. Except that for the females we leave the space for the middle joint empty.
Finally, we have created female and male joints which are aligned along a line with the length of the intersection.

Placing the Joints at the Correct Edge of the Plate

This step moves the joints to the correct position in space.
To do shape union functions we need to move the shape of the plate and the intersectionline into the xy-plane.
The joints which already lie in the xy-plane are rotated and translated onto the intersectionline.

To make sure that the joints will be appended to the plates we test if the transformed joints lie inside the plate. If so they are moved towards the outside of the plate.

Merging the Joints with the Plate

The last step unions the now aligned shapes of the joints and plates and rotates them to the correct position in space where the plate belongs to within the model.

9.3 DIFFERENT FINGER JOINT TYPES

We support three types of joints. Each have benefits when used for specific material. For example acylic is not flexible which means it is very sturdy, but bendable when heated.

We allow the user to choose the type of material which then affects the choice of joints in the converted model.

9.3.0.1 *Finger Joint Template*

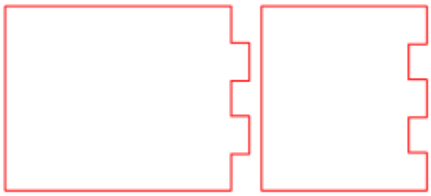


Figure 85: Path which a laser cutter has to follow to cut our plates with finger joints.

Finger joints, see Figure 85, only consist of 90 degrees angles. The female joints are the exact inverse of the male joints. This means that the joints can slide directly into each other.

But that only works when the sizes are measured correctly so that they create a tight fit. Otherwise plates connected by finger joints do not fit into each other at all or fall apart and have to be glued. When using finger joints one typically has 90 degree angled plates because it is the easiest way to connect them. Other angles can be created but it is hard to know when the plates form the exact angle that is wanted unless the plates are part of a construction which gives them no other choice than to form the given angle.
Regarding the material finger joints are useless when flexible materials are connected with it. The problem is that a tight fit cannot be created in most cases. Also very thin material is not working well since finger joints hold up due to the friction of one plate to the other. The fewer material the less friction.
We usually use finger joints for acrylic and wood.

9.3.0.2 *JimJoint Template*

JimJoints, see Figure 86, are named after Jim McCann who thankfully showed us the design for these connections. The male and female joints are the same but they grow wider with its height.

This means these joints cannot slide into each other but they rather snap together. This means once they are connected they cannot be taken apart just by pulling on the plates.

In order for the snapping to work the material has to be flexible or very thin. McCann already used it for foam. We now use it especially

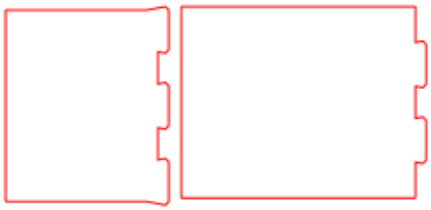


Figure 86: Path which a laser cutter has to follow to cut our plates with JimJoints.

for paper because this material would be too thin for creating enough friction for finger joints to work.

9.3.0.3 *Dovetail Template*

Figure 87: Original dovetails created with a mill in wooden plates.

Source: http://www.startwoodworking.com/sites/default/files/uploads/1/1051/dovetails%201.jpg (visited on July 25, 2016)

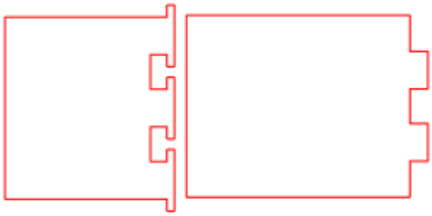


Figure 88: Path which a laser cutter has to follow to cut our plates with dovetails.

Typically a dovetail joint is used in woodwork, see Figure 87. It helps to ensure that when there is a pulling force on both wooden plates that the joints all hold onto the other plate.
But in order to create the female joints the wood needs to be cut in an angle. This is not possible without inadequately high time and power consumption. See the upcoming Section 9.5 Future Work for information on how to laser cut the female part of a dovetail.
Instead, we developed a joint type, see Figure 88, very similar to the original dovetail. This can be easily cut with a laser cutter. The males and females are a lot like typical finger joints. But the females have an additional part which will fixate the other plate from one direction.
These joints now withstand a force pulling away the plate with the female joints. It works with any material with at least around one millimeter of thickness when the material is not very flexible.

9.3.1 *Adjusting the Finger Joint Length when Plates are Angled*

Not only the width has to be adjusted but also the height needs to be adapted to the plates connection. Depending on the angle of the plates the joints need to be longer or shorter accordingly.
This problem can be solved by using trigonometry within the geometry of the overlap of the plates.
If the plates overlap they create a parallelogram, see Figure 89. The length of the long diagonal d is the key to finding the appropriate length for the joints.

We have already calculated the angle between the plates earlier and can now use it to find the lengths of the sides a and b of the parallelogram by the help of trigonometry:
Thickness of plate p_1: t_1
Thickness of plate p_2: t_2

$$a = t_1 / sin(180^\circ - \alpha)$$

$$b = t_2 / sin(180^\circ - \alpha)$$

This helps us to calculate the length of the diagonal d:

$$d = \sqrt{a^2 + b^2 - 2ab * cos(\alpha)}$$

Finally, the triangle with the sides d, a, b which helps us retrieve the height h for the finger joints can be constructed.

$$h = \sqrt{d^2 - a^2}$$

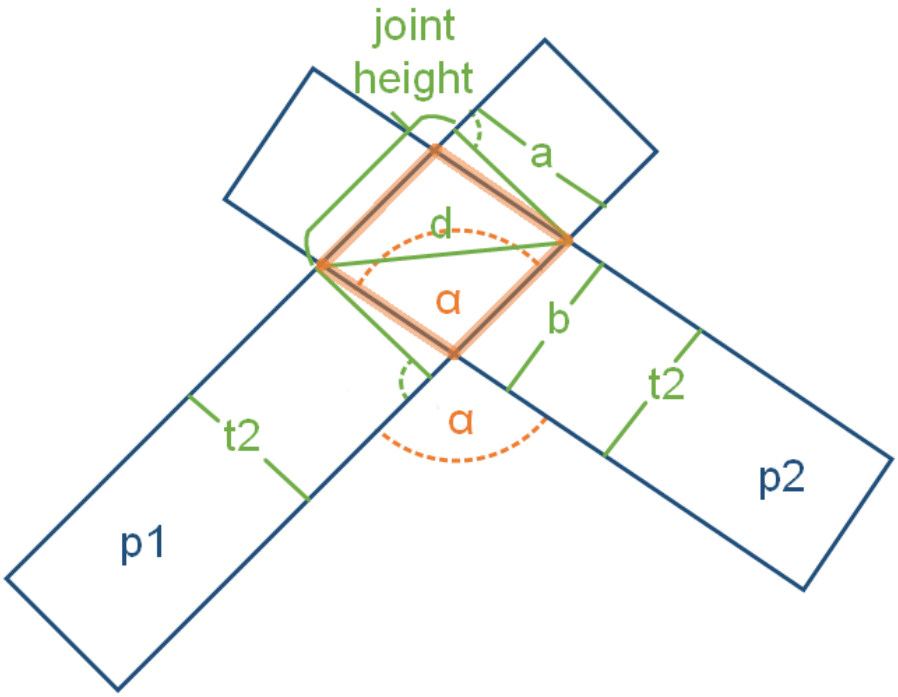


Figure 89: The parallelogram spanned by the overlap of the plates allows us to find the appropriate height for a joint.

Finally, this height can be applied to all joints of the current intersection line. This achieves that, when cut, the joints slide into each other, allowing to create the wanted angle.

9.4 ALTERNATIVE SOLUTIONS

9.4.1 *Angle Approach from Beyer et al.*

According to the master thesis on platener [7] connections of plates can be seperated into four different angle types, see Figure 90. This is based on the assumption that plates always overlap with an edge which is not the case as we found out. He computed different angle types. They are based on the assumption that plates would only meet at edges. But plates can lie in one another. Therefore we do not use those angle type but we clip both plates with their four intersection lines.
Then due to angle and trigonometry etc. we can calculate the correct length for the joints which connects the plates again.

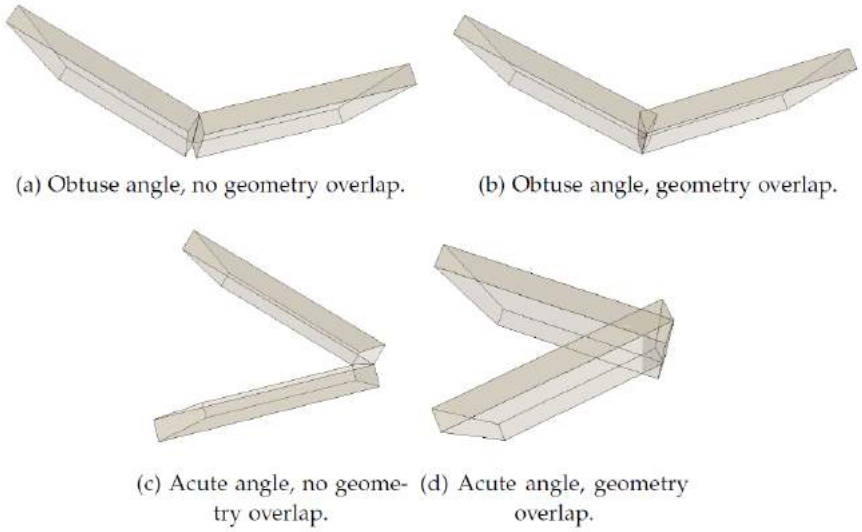


Figure 90: Figure from the thesis [7] showing the different angle types.

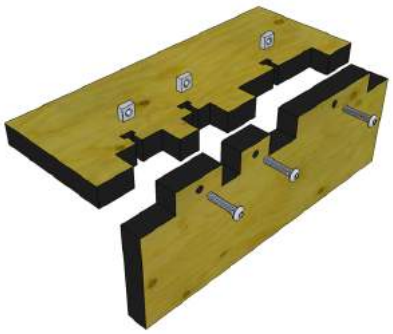
9.5 FUTURE WORK

9.5.1 *Additional Joint Types*

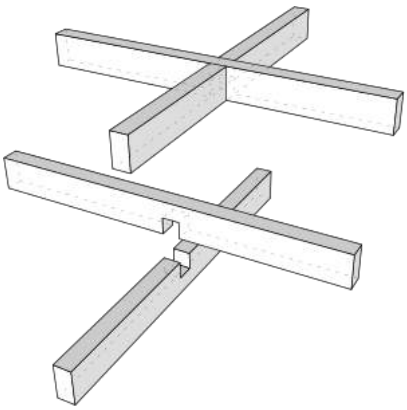
In Figure 91 we show different types of joints which could be added to our repertoire.

9.5.1.1 *Joints Should not Stick out over the End of the Intersection*

Currently, when using JimJoints or dovetails they might overlap the end of the line because the width value which is used for spreading the joints does not represent the with of the joint at its heighest point. This is because JimJoints and the females of the dovetails grow wider apart with its height.
In order to solve this the joint number should be calculated based on a joints maximum width, but they still have to be spread based on the minimum width.

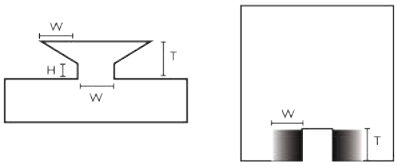


(a) Pettis joint: finger joints with screws for more stability. Only 90° angles can be achieved.



(b) Halved joint: slot joinery technique for plates which have an intersection through its middle.

(c) Snap fit: Using a Spring to snap into place. Holds tight, but takes up alot of space on the plate.



(d) Real dovetail: Using in a gradient [2] from white to black allows to cut different depths and achieve an angle.

Figure 91: Additional possibilites for connecting plates.

Source: http://1abxf1rh6g01lhm2riyrt55k.wpengine.netdna-cdn.com/wp-content/uploads/2012/04/biased-box-corner-with-pettis-joints.png (visited on July 25, 2016)

Source: https://upload.wikimedia.org/wikipedia/commons/thumb/f/ff/Joinery-SimpleHalved.svg/300px-Joinery-SimpleHalved.svg.png (visited on July 25, 2016)

10

CURVES

10.1 CUTTING CURVED SHAPES

Approximating curved shapes with parts created by the laser cutter is a special challenge because only 2D shapes can be cut. There are several approaches to make the cut material bendable, for example, paper can be bent because of the material properties. Acrylic becomes bendable when heated, while bending wood is made possible by the use of a special cutting pattern, the so-called living hinges (see Section 10.4 Living Hinges Make Rigid Materials Flexible).

Theoretically, this only enables the possibility to create shapes from developable surfaces, like cylinders, but no doubly curved ones, like spheres. But since our meshes consists of triangles, which are themselves two-dimensional, our meshes are always developable.

10.2 BEND JOINTS ARE USED AS ALTERNATIVE FOR FINGER JOINTS

In our implementation, we use the *bend* joint type as an alternative to, for example, finger joints. Like the finger joint generation, the calculation of bends is based on the plate graph. The bent plate generation is separated into two steps. First, our implementation analyses which parts can be bent. Second, it creates a flat shape from the curved parts, which can be cut with a laser cutter.

10.3 PREREQUISITES CREATING BENT PLATES

The algorithm uses the plates and connections from the plate graph. The plates provide their shape, normal and rotation matrix, which is used for laying it into the xy-plane. Additionally, the connections intersection line and angle are used. Before calculating bends, the finger joints have to be created, with the resulting shapes being stored in the corresponding connection.

10.4 LIVING HINGES MAKE RIGID MATERIALS FLEXIBLE

Living hinges are segments of a rigid material that are made flexible locally. There are several techniques to do this, e.g. thinning the material in this area. Using the laser cutter a living hinge can be created by dense parallel cuts. They are disconnected every few centimeters and the breaks are offset to the next parallel cut (see Figure 92).

(a) A sealing clip where a thinned part acts as living hinge.

(b) Laser cut living hinges in a wooden smart phone stand.

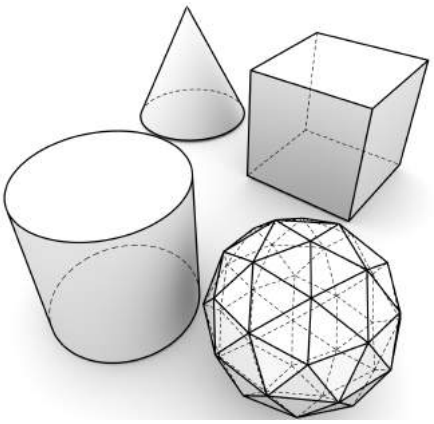
Figure 92: Two typical types of living hinges.

10.5 DEVELOPABLE SURFACES CAN BE BEND

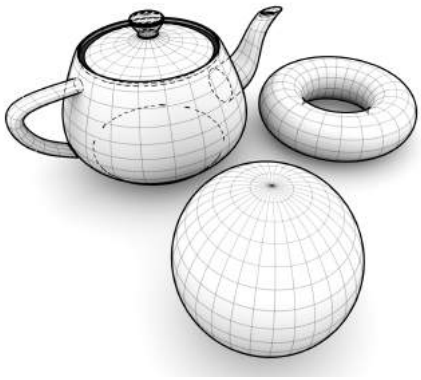
Formally developable surfaces are defined as surfaces with zero Gaussian curvature. Practically this means they can be created from a flat surface by cutting, bending and folding and without compressing or stretching. Examples for objects with and without developable surfaces are shown in Figure 93.

10.6 THE BENT PLATE OBJECT

Bent plate is a data structure we use to represent a set of plates which are connected directly or indirectly with bend connections. Thus a bent plate can be cut as one piece. A *bent plate* consists of a set of *plates*. It provides the method `generateShape` which develops the surface of the connected shapes (see Section 10.6.1 The GenerateShape Function Creates a Shape of the Developed Surface). Another method implemented by *bent plate* is `generateBendLines` which creates cuttable marks where two plates are merged (see Section 10.6.2 Bend Lines Help to Bend the Plate).



(a) Objects like cubes, cylinders, cones and also sphere approximations like an icosphere have developable surfaces.



(b) So called doubly curved surfaces like from a sphere or torus are not developable.

Figure 93: Some examples of models with and without developable surface.

10.6.1 *The GenerateShape Function Creates a Shape of the Developed Surface*

To generate the *shape* of the *bent plate* we first check how many *plates* this *bent plate* contains. If none are contained, the generated *shape* is *null*. If there is just one *plate* then its *shape* is used.

If there are multiple plates in the bent plate the generation is done the following way: For each of the plates in the bent plate, the corresponding shape is transformed using the bend matrix (see Section 10.8.1 Traversing Along Bend Connections). Afterwards, they are converted into polygons suitable for using with the Javascript Clipper and merged using the *union* function.

To avoid unconnected polygons after merging caused by floating point inaccuracy we use vertex welding before merging (see Section 6.1.1 Vertex Welding). The resulting polygon is converted back to a shape (see Listing 31).

10.6.2 *Bend Lines Help to Bend the Plate*

This method generates a dashed line located at the connection line between two plates of the bent plate. This marks where the user must bend the material. Additionally, it weakens the material and allows for easier bending.

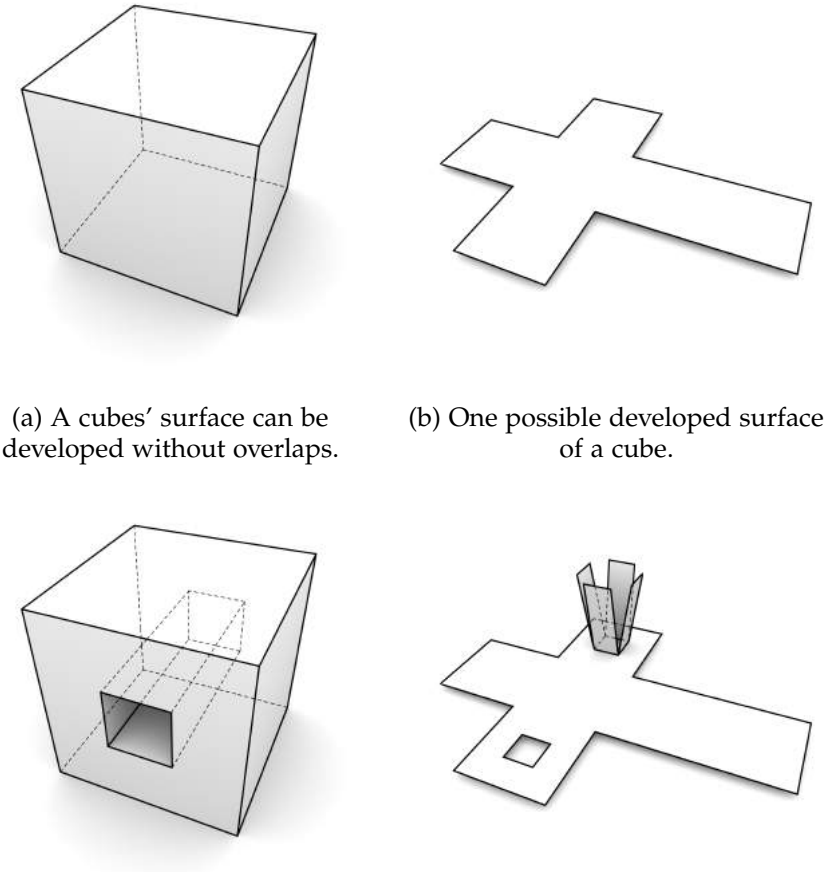
To locate the general placement of the bend line all intersection lines of the bent plate are processed. They are checked whether they

belong to a bend connection and are not used for creating a bend line yet. To place them into the shape they are transformed by the corresponding bend matrix (see Listing 32).

At this position we create a dashed cutting line.

10.7 SETTING THE JOINT TYPE

In this step, we try to find out which connections can be implemented as a bending joint. The resulting shape of the connected plates is developable without overlaps. The difference between a model which surface is developable without overlaps and one which in not is shown in Figure 94.



(a) A cubes' surface can be developed without overlaps.

(b) One possible developed surface of a cube.

(c) The surface of a cube with a hole can not be developed without overlaps.

(d) The shapes inside the hole will always overlap some part of the developed surface.

Figure 94: Two objects with a developable surface, but one without and one with overlaps.

To do so, we start with one plate and run a series of tests for all of its connections:

- Is the type of this connection not yet set (it is set to *unset* instead of e.g. *bend joint*)?
- Is the connection angle close enough to 180°, allowing the material to be bent this far? (What near enough means depends on the used material see Section 10.9.4 Material Dependant Bend Angle)
- Is it possible to add the to-be-connected plate's shape to the currently processed plate's shape without overlapping?

If all checks succeed, the connected plate is added to this plate and they form a bent plate. If the connection type is not yet set but one of the other tests fails a finger joint is used.

This is repeated for all the connections of the bent plate until we assigned a joint type to each of them. This process is repeated until all plates are assigned to a bent plate.

When checking if a plate could be added to an existing bent plate, two shapes are created. The first one is the union of the bent plate's shape and all of its finger joint shapes, excluding the connection with the new plate. The second is the union of the shape of the new plate and its finger joint shapes, excluding the connection with the bent plate.

Afterwards, we calculate the intersection of this two. If the result is empty, there are no overlaps, the plate can be safely added to the bent plate. The connection is annotated as a bending joint.

10.8 BUILDING THE BENT PLATES

For building the bent plates our implementation starts with one plate of the plate graph and traverses the graph along the marked bend connections. This way all directly or indirectly connected plates (via bend connections) are collected into a array. From these plates one bent plate is created. If there are plates in the plate graph which are not added to a bent plate, the process is repeated until no plate is left.

10.8.1 *Traversing Along Bend Connections*

While traversing along the bend connections, the transformation matrices for the plates are calculated.

The first plate's matrix is already known. It is the rotation matrix of its shape to rotate it into the xy-plane.

When laying the other plates into the xy-plane, it is important to make sure that they keep touching with the connected plate while laying them into this plane. Therefore the bend angle must be calculated. It is 180° minus the angle between the connected plates, which is already known from the plate graph creation.

The bend axis is the axis to rotate around to develop the surface. It is determined by calculating the cross product of the two normals of the connected plates. This way the axis does not only lie at the right place but also points in the right direction (This would not be the case if only the connection line between the two plates is used).

The axis has to be transformed into the xy-plane, onto the edge of the previous plate. To do so, our implementation uses the starting point of the intersection line and ending point, which is created by adding the axis vector to the starting point. These two points are transformed using the transformation matrix of the previous plate. The final axis is determined from the transformed points by subtracting the ending point form the staring point.

The angle and the axis are used to create the rotation matrix used to lay the plate in the same plane like the previous one.

Because the rotation matrix only rotates around the point of origin, two translation matrices are created. The first one translates the plate to the origin after which the rotation is performed to lay the plate into the xy-plane. The second one translates the plate back to the connection edge.

For calculating the final transformation matrix for this plate, the transformation matrix of the previous plate is multiplied by the first translation matrix, the rotation matrix and finally the second translation matrix. This will also be used for creating the transformation matrices of the following plates.

10.9 ALTERNATIVE SOLUTIONS

10.9.1 *Search for Best Surface Development*

The current implementation does not always find the best solution. While traversing the graph it is possible that there multiple different paths through the graph. This results in unequally good sets of bent plates. For example one way of traversing could create more but smaller bent plates then another. Because currently only one attempt to traverse the graph is made, the second and probably better option would not be found (see Figure 96).

We propose following solution to avoid this problem: If a plate which should be added to a bent plate intersects a already added

plate, we try removing the present plate and check if adding the new one yields better results.

This will increase the run time and results in better solutions for models where the bendable parts of the 3D model form a highly connected graph.

10.9.2 *Joints at Curved Edges*

Currently the finger joints are generated just based on the plates but not on the bent plates. If multiple small plates which each have to small edges to place finger joints on them are merged into a bent plate, the result would not have finger joints added to it as well.

To avoid this, a new method must be found to generate finger joints at curved edges. Since the edge of the plate that connects to the bent plate will be curved, the current finger joint algorithm can not be used here. Because adding a plate to the bent plate changes the finger joints, we have to check for overlaps constantly. Thus, the finger joints have to be generate for every attempt to add a new plate.

10.9.3 *More Bend Line Types*

Independently of the used material, a dashed line is added as the bend line. This is good for folding paper. While acrylic can not be bend as easily, we propose engraving a continuous line, which tells the user where to bend the material without weaken the material to much. For wood, on the other side, living hinges should be generated to make the wood bendable at all.

10.9.4 *Material Dependant Bend Angle*

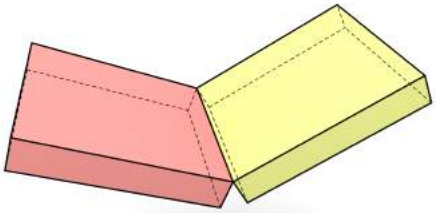
Currently the the maximal allowed bending angle is fixed. But because the flexibility of the cut plates differs depending on the used material and thickness, thus should also influence the maximal bend angle. It is also influenced by the bend lines, especially by living hinges, which are designed to change the flexibility (see Section 10.9.3 More Bend Line Types and 10.4 Living Hinges Make Rigid Materials Flexible).

```coffee
generateShape: ->
  if @plates.length < 1
    @shape = null
    return
  if @plates.length is 1
    @shape = @plates[0].shape
    return
  # lay shapes into xy-plane
  polygons = @plates.map (plate) =>
    bentMatrix = plate.bentMatrix
    shape = plate.shape
    contour = @applyMatrices(
      shape.getContour(), bentMatrix)
    holes = shape.getHoles().map (hole) =>
      @applyMatrices hole, bentMatrix
    return {contour, holes }
  # remove-doubles-fix
  weldingDistance = 0.02
  welder = new VertexWelding(weldingDistance)
  polygons.forEach (polygon) ->
    polygon.contour.forEach (vertex) ->
    correspondingVertex = welder.getCorrespondingVertex(
      {x: vertex[0], y: vertex[1], z: 0})
    vertex[0] = correspondingVertex.x
    vertex[1] = correspondingVertex.y
  # create clipping polygons
  polygons = polygons.map (polygon) ->
    return new Clipper.Polygon(
      polygon.contour, polygon.holes)
  # merge polygons
  mergedPolygon = polygons[0]
  polygons.shift()
  mergedPolygon = mergedPolygon.unionMultiple polygons
  # create shape from clipping polygon
  contour = mergedPolygon[0].getShape().map (vertex) ->
    return new THREE.Vector3 vertex[0], vertex[1], 0
  contour = new EdgeLoop contour
  holes = mergedPolygon[0].getHoles().map (hole) ->
    hole = hole.map (vertex) ->
    return new THREE.Vector3 vertex[0], vertex[1], 0
    hole = new EdgeLoop hole
    hole.hole = true
    return hole
  shape = holes
  shape.push contour
  shape = new Shape shape, new THREE.Vector3 0, 0, 1
  # set shape as shape of the bent plate
  @shape = shape
```

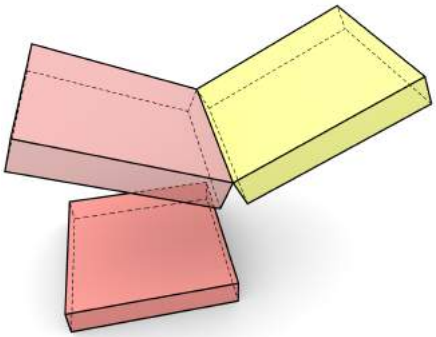
Listing 31: Generating the shape for a bent plate.

```
generateBendLines: ->
  seenConnections = []
  @bendLines = []
  polygons = @plates.forEach (plate) =>
    bentMatrix = plate.bentMatrix
    plate.connectionList.forEach (connection) =>
      parameters = connection.parameters
      if parameters.joint is 'bend' and not (parameters in
↪ seenConnections)
        seenConnections.push parameters

        intersectionLine =
↪ parameters.intersectionLine.clone()
        intersectionLine.applyMatrix4 bentMatrix
        intersectionLine.start.z = 0
        intersectionLine.end.z = 0
        @generateDashedLines intersectionLine

        @bendLines.push intersectionLine
```

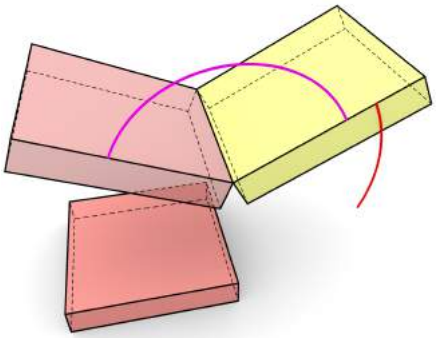
Listing 32: Generate the bend lines for a bent plate.



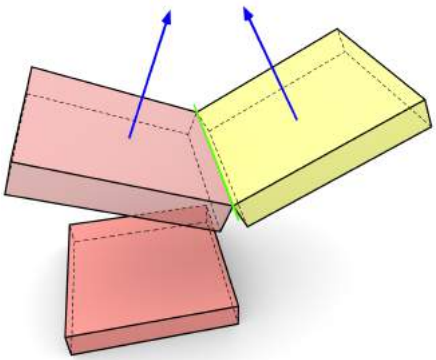
(a) Two plates which should be connected with a bend.



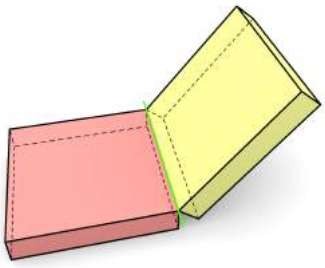
(b) For the first one the transformations matrix is known already.



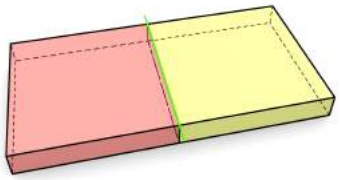
(c) The bend angle (red) is 180° minus the angle between the plates (magenta).



(d) The bend axis is the cross product of the bent plate's normals.



(e) The bend axis and the second plate are transformed with the matrix of the first plate.

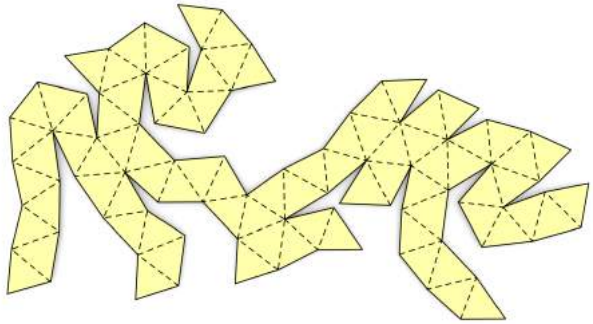


(f) With the new generated transformation matrix the second plate lies in the xy-plane next to the first plate.

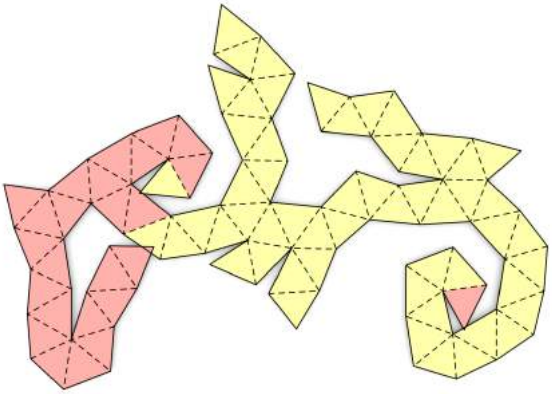
Figure 95: Creation steps of the bend matrix.

```coffee
angle = (180 - connection.parameters.angle) * Math.PI / 180
intersectionLine = connection.parameters.intersectionLine
start = intersectionLine.start.clone()

lineDir = plate.shape.normal.clone().cross
 ↪  connection.node.shape.normal
end = start.clone().add lineDir

start = start.applyMatrix4 rotMat
end = end.applyMatrix4 rotMat

axis = start.clone().sub end
axis.normalize()

moveMat1 = new THREE.Matrix4().makeTranslation(
  -start.x,
  -start.y,
  -start.z
)
moveMat2 = new THREE.Matrix4().makeTranslation start.x,
 ↪  start.y, start.z
bendMat = new THREE.Matrix4().makeRotationAxis axis, angle

newRotMat = moveMat2.clone().multiply bendMat
newRotMat = newRotMat.multiply moveMat1
newRotMat = newRotMat.multiply rotMat.clone()
```

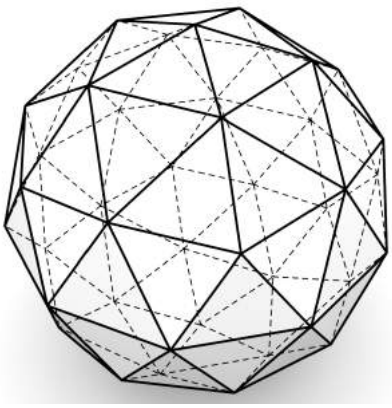
Listing 33: Creating the transformation matrix for a plate as part of a bent plate.



(a) A surface development of a icosphere without overlaps is possible.



(b) If the plate graph is not traversed in a optimal way, overlaps force the algorithm to split the surface into multiple parts.



(c) The model from which the developments where created.

Figure 96: The model from which the developments where created.

11

ASSEMBLY

Assembling the laser cut plates to form the desired object is not trivial. Since multiple plates may be very similar to each other, it is not always clear which have to be put together. Thus, we decided to add assembly instructions, which allow the user to assemble the model hassle-free. Section 11.1.1 Plates Connected With Finger Joins Are Annotated on Each Edge describes how these instructions are added to plates which were found either inherently or by extruding. The assembly instructions for stacked plates are discussed in Section 11.1.2 Stacked Plates Are Enumerated.

11.1 ADDED NUMBERS HELP ARRANGING PLATES

This section describes the currently used methods for adding assembly instructions. While these simplify assembly, they are not optimal. Ideas for improvement are discussed in Section 11.2 Possible Improvements.

11.1.1 *Plates Connected With Finger Joins Are Annotated on Each Edge*

Plates created by the *PlateMethod* are connected by finger joints. In order to show which plates have to be put together, the corresponding edges are annotated accordingly.

These annotations are created for each edge separately, by calling the *addInstruction* function for each. This function is shown in Listing 34. The connection data is passed by setting this property beforehand. Thus, the nodes (which contain the plates) and the connection parameters can be extracted. Using these, the instructions are built for both plates.

```
1 addInstructions: ->
2   { node1, node2, parameters } = @connection
3   @buildInstruction(node1, parameters.node1Direction)
4   @buildInstruction(node2, parameters.node2Direction)
```

Listing 34: Adding instructions to connections.

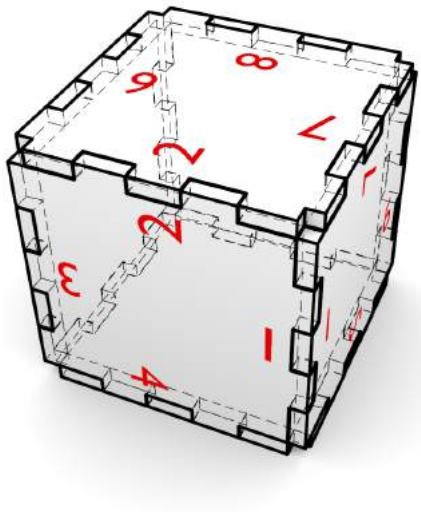


Figure 97: Assembly instructions for plates connected by finger joints.

Listing 35 shows how this is done. First, we ensure that the data for the direction and the intersection line is valid. The intersection line tells us where to place the annotation, with the direction being the direction from the intersection line towards plate. This helps to place the annotation on the plate instead of placing it directly on the edge (with half of it not actually being located on the plate). The index of the connection is the number we want to annotate the edge with. Thus, we build a text box containing this number. Afterwards, the box is moved onto the intersection line. Using the direction, we can place it on the plate. Lastly, the created text box is added to the node.

```
1  buildInstruction: (node, direction) ->
2    if direction is null then return
3    { index, intersectionLine } = @connection.parameters
4    if not intersectionLine? then return
5    box = @buildInstructionBox(index)
6    @moveObjXYToLineXY(box,
     ↪  @layLineIntoXY(intersectionLine[0], node.shape))
7    box.translateOnAxis(direction, 0.75 * jointSpecs.height)
8    node.assemblyInstruction.add(box)
```

Listing 35: Building the assembly instruction for one plate.

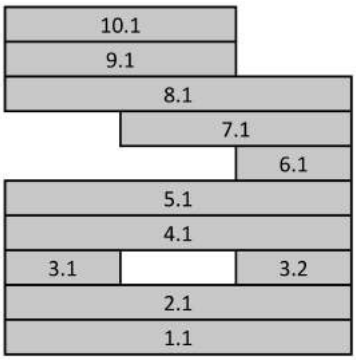


Figure 98: Side view of assembly intructions for stacked plates. In reality, the annotations are placed on the main side of the plate.

11.1.2 *Stacked Plates Are Enumerated*

The annotations added to stacked plates differ from the ones discussed above. Since the shafts already simplify the assembly of the model, we just enumerate the layers of plates. Sorting the plates by the number written on them tells the user how to assemble them. Figure 98 shows a side view of how the plates are annotated. While iterating over all layers and all plates contained in these layers, we add an annotation to each plate.

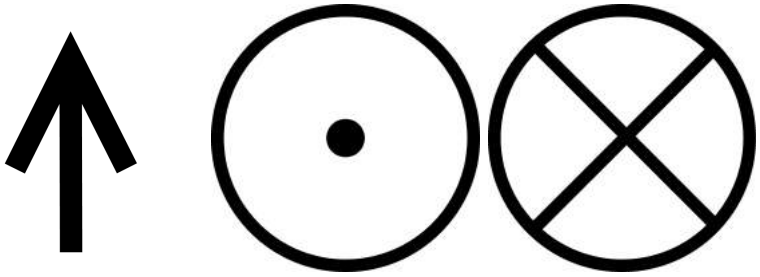
11.2 POSSIBLE IMPROVEMENTS

11.2.1 *Icons Can Show the Orientation of Plates*

In addition to the annotations described in Section 11.1.1 Plates Connected With Finger Joins Are Annotated on Each Edge, we propose adding icons which tell the user where the plate is located. Thus, the distinction between horizontal and vertical plates is made easy. The proposed icons are shown in Figure 99. In addition to the arrow, which signalizes the arrangement of vertical plates, two icons describing horizontal plates are shown. These are based on the mathematical symbols indicating if an vector is pointing into or out of a diagram. All three symbols tell the user where - from the plates perspective - the top side of the model is located.

11.2.2 *Annotating Plates Only Once Reduces Cutting Time*

Engraving the annotations onto plates usually takes longer than the actual cutting process. In Order to reduce this time, we propose to not annotate each edge individually. Instead, each plate should be



(a) Vertical plate. The arrow points to the top.

(b) Horizontal plate, located at the top of the model.

(c) Horizontal plate, located at the bottom of the model.

Figure 99: Icons signalizing the orientation of a plate.

engraved with a number, which clearly identifies it. Coupled with assembly instructions shown on screen, this still enables fast assembly, while reducing cutting time.

12

CLASSIFIERS

12.1 CLASSIFIYING IDEA

[4] In the previous chapters we covered the approach of solely finding plates within 3D models. This approach needs the model to be 'boxy'. Therefore sometimes the algorithm fails because of too many round edges or other noise.
This is why we tried another approach, called RANSAC [21], which is also being used by CGAL [1]. We look for all kinds of primitive shapes in addition to plates. This allows us to split the model into separate objects that can then be independently converted to laser cutable plates. During the conversion we choose a specific method to realise this part with plates which often helps to achieve a more appropriate solution than to directly start finding plates. Due to the tolerance of the algorithm to noise we can even detect surface when there is a lot of texture. Currently, our software system does not finding the classifiers for a conversion. Instead we wanted to find out how well the algorithm works for our use case.
In order to properly include the classifiers we built a theoretical structure to make the most of the findings.
A Classifier finds a particular shape like a plane, plate, box, sphere, cylinder, prism, etc. Classifiers can use findings of others for example the box finder depends on the previously found planes. As soon as all classifiers have finished their process all findings are hierarchichally grouped based on the depencies of the classifiers. Each finding is represented as a *graphObject* which can either be of the type volume, like a box or cylinder for example, or a freeform. Freeforms are objects that could not be classified. Nevertheless we check if this object is some kind of connector of volumes. This is valuable information for the conversion step because it means there needs to be something between specified volumes this can be anything that the conversion method might choose. Then a graph is created which keeps all objects and their connections. Finally, the reconstruction starts: Each volume will be converted by its specific converter. The box converted is an example. Connectors will result in plates which are connected to all volumes it belongs to. And from unclassified objects we create plates based on the shapes found in this part of the mesh.

Usually, RANSAC is used with pointclouds. This means we need a lot of points and the more noise there is within the points the better for

the algorithm. We work with meshes where a model of a box might only have 8 points. This is a problem because the diagonals of the box look just like another plane to the algorithm because the sides and the diagonals are supported by 4 points. On the other hand, when this box consisted of thousands of points they would all be somewhere on the sides. Meaning that the diagonal would not be found as a plane because it is maybe only supported by up to 10 points. In contrast, each side is supported by up to hundreds of points. Therefore, the RANSAC approach can not be used with every 3D model. Instead it should be used as an additional option to split the model.

12.2 RANSAC - RANDOM SAMPLE CONSENSUS

The RANSAC-approach firstly chooses a defined number of random points. These points are a subset from the point data and its number is defined by the shape that has been classified. This subset should be choses as small as possible.
On the basis each of these minimal sets a candidate shape is generated and tested against all points in the data set. The candidate gets a score which tells how well the randomly chosen points represent the shape. This score can result from counting the points which lie within the candidate.
Based on this score the best model is saved or overwritten after several candidate attempts.

12.3 PRIMITIVES

In order to classify primitives, a minimum number of points has to be defined to enable a shape reconstruction.

12.3.1 *Plane Classifier*

The minimal set for a plane consists of three points {p1, p2, p3} because three points uniquely identify a plane.
Once a plane candidate has been found it is necessary to check its plausibility. The deviations of the plane normal to the according point normals of p1, p2 and p3 should be less than an angle α, currently set to 0.1 .

After the detection of the best model it may be necessary to refit the candidate to all its inliers. We use the Least Squares method [3]. We are aware that this method can only compute planes where its z-values are dependent on the x- and y-values this is not the case

the the plane is perpendicular to the x-y-plane. Therefore we ignore planes to that are affected by this circumstance. Another possibility would be to work with eigenvectors where this would not be an issue. When using the least squares method the problem can be transformed into an equation of the form Ax = b. Where A is a matrix consisting of the sum of all x values of the points, y values, x times y and x squared and y squared.

$$\begin{bmatrix} \sum_{i=1}^{m} x_i^2 & \sum_{i=1}^{m} x_i y_i & \sum_{i=1}^{m} x_i \\ \sum_{i=1}^{m} x_i y_i & \sum_{i=1}^{m} y_i^2 & \sum_{i=1}^{m} y_i \\ \sum_{i=1}^{m} x_i & \sum_{i=1}^{m} y_i & \sum_{i=1}^{m} 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{m} x_i z_i \\ \sum_{i=1}^{m} y_i z_i \\ \sum_{i=1}^{m} z_i \end{bmatrix}$$

The least squares solution is the following plane equation: $z = Ax + By + C$.

12.3.2 *Cylinder Classifier*

The minimal set for a cylinder consists of two points $\{p_1, p_2\}$. In Figure 100 the reconstruction process is sketched. Those randomly picked points are assumed to lie on the shell of the cylinder. If they are not actually lying on the shell then later there will not be enough candidates to support the cylinder which means it will be discarded. The axis can be calculated by calculating the cross product of their normals n_1 and n_2. Based on the axis a projection plane is formed which is perpendicular to the axis. The two lines $l_1 = p_1 + x * n_1$ and $l_2 = p_2 + x * n_2$ are projected onto this plane and should have an intersection. If they do not intersect the candidate is invalid. Otherwise, the intersection point is marked as the center of the cylinder-candidate. The radius is the mean value of the distance of both points to the center on the axis.

After a valid candidate has been formed we have to check the plausibility. For this we look for three indicators. Firstly, the randomly chosen points p1 and p2 should not equal, secondly, the calculated radius has to be larger than zero and lastly, the distances of the two points to the center should not exceed an epsilon value ϵ.

12.3.3 *Prism Classifier*

A Prism is a base element for many other primitives which are characterized by two parallel shapes. Therefore a prism datastructure contains two shapes and the lateral surface as a set of primitives which are usually faces or shapes.

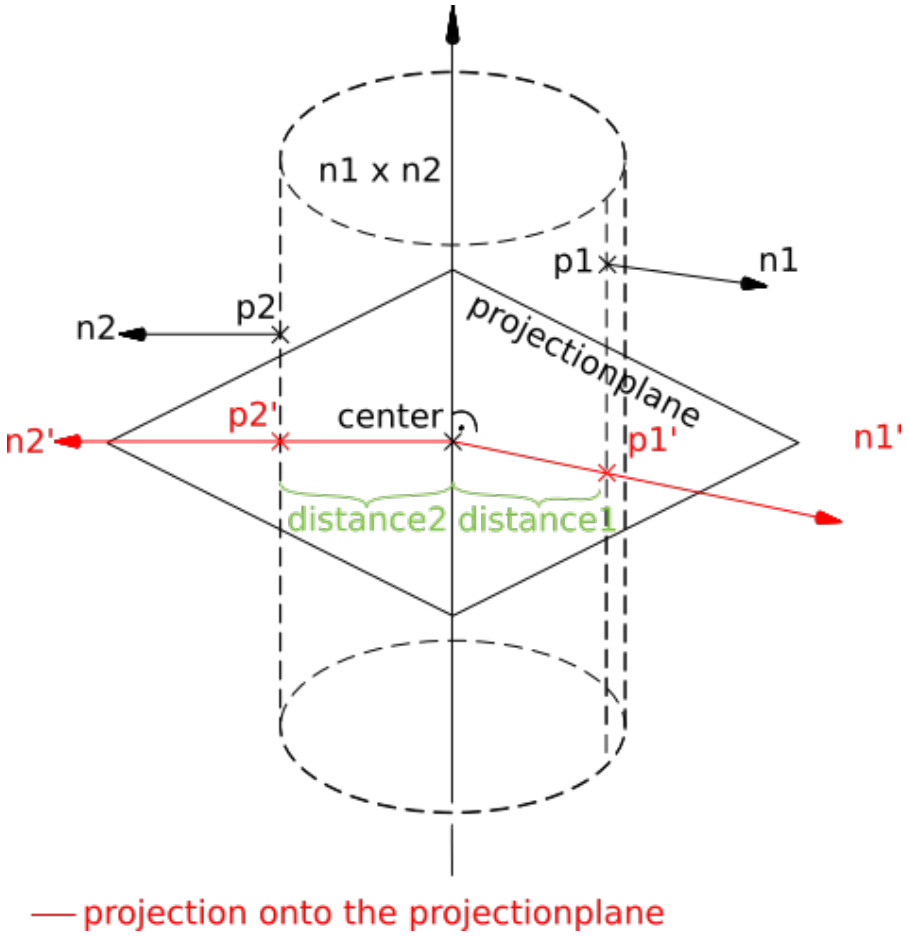


Figure 100: Two points are enough to represent a cylinder. The axis can be reconstructed, as well as the radius.

To find the prisms all shapes have to be checked for parallelism. The comparison is based on normals which are sorted into buckets to speed up the process. The attached lateral surface can be identified by flood fill.

12.3.3.1 *Bucket System*

The surface normals $\vec{n}$ are sorted into buckets to reduce the amount of comparisons. Treating slightly different normals as equal considering a given threshold ϕ each bucket covers this angle. Therefore to check for similarity only normals within a bucket and the surrounding ones have to be considered: The angle between two normals is always greater ϕ if their buckets are not neighbors. The corresponding bucket for a normal is determined by two values α and β. Their calculation is explained below.

$$\vec{n} = (x, y, z)$$

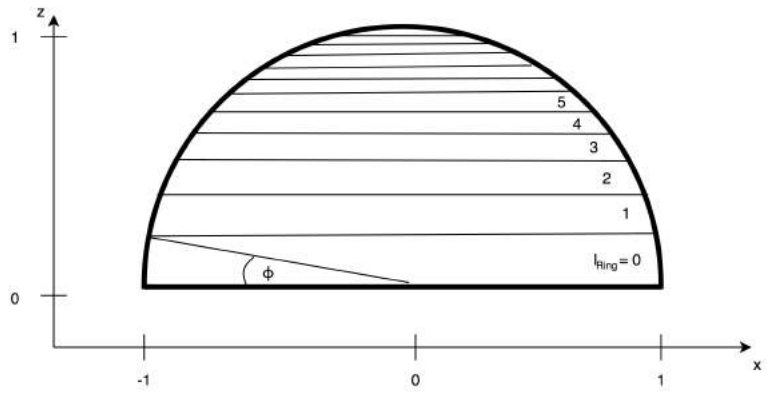


Figure 101: Rings visualised from side

All normals lie on the surface of the unit sphere because of their normalisation. Considering opposite normals as equal like planes with normals $(1,0,0)$ and $(-1,0,0)$ are parallel, all normals are transformed into a hemisphere as shown in Listing 36: A normal is negated in case its z value is negative.

```
1 _normalInHemisphere: (normal) ->
2     if normal.z < 0
3         return normal.clone().negate()
4     else
5         return normal
```

Listing 36: Normals are transformed into hemisphere

This hemisphere is separated into rings. Each ring covers the threshold angle ϕ in z-direction and is visualized in Figure 101 and Figure 102. Each normal is sorted into a ring by its correlating angle α which is used to determine the ring index. It is proporional to its coordinates as demonstrated in Figure 103: the angle between $\vec{n}$ and $(x,y,0)$:

$$\sin\alpha = \frac{z}{\sqrt{x^2+y^2}}$$
$$=> \alpha = \arcsin\frac{z}{\sqrt{x^2+y^2}}$$

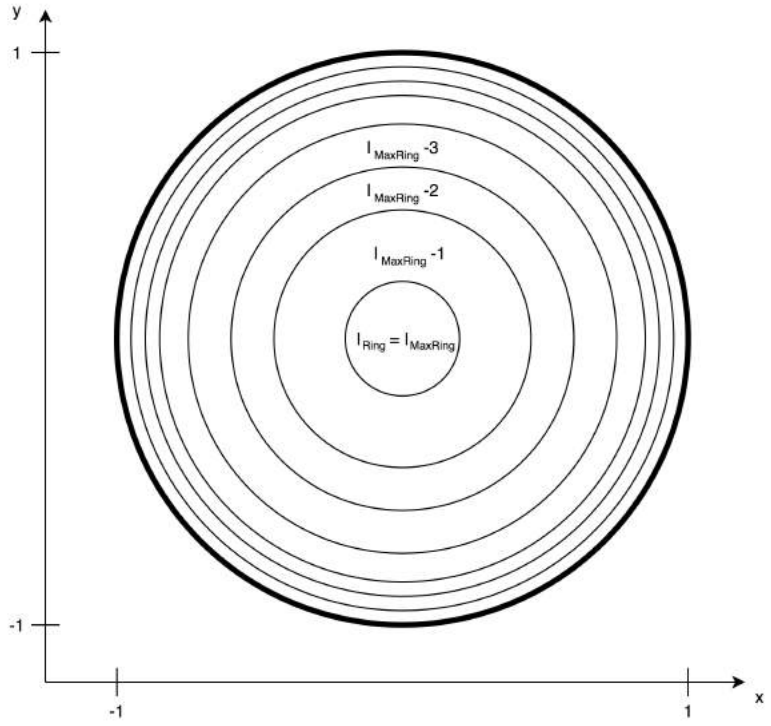


Figure 102: Rings visualised from top

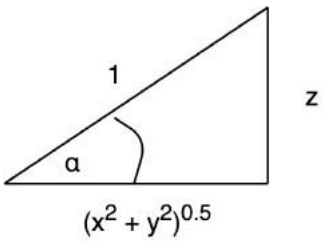


Figure 103: Angle of a normal which is used for sorting it into a ring

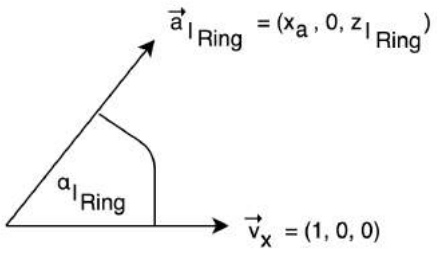


Figure 104: Geometric visualization of the z value in the middle of a ring

The angle α is represented as a multiple of the threshold angle ϕ to get the actual ring index I_{Ring}:

$$I_{Ring} = \left\lfloor \arcsin \frac{z}{\sqrt{x^2 + y^2}} / \phi \right\rfloor$$

The maximum angle is 90°. Consequently the maximum ring index $I_{MaxRing}$ is calculated as follows:

$$I_{MaxRing} = \left\lfloor \frac{\pi}{2} / \phi \right\rfloor$$

The angle in the middle of each ring $\alpha_{I_{Ring}}$ consequently depends on index and threshold:

$$\alpha_{I_{Ring}} = (I_{Ring} + 0.5) * \phi \qquad \forall I_{Ring} \in [0, I_{MaxRing}] \tag{1}$$

Figure 104 implies the calculation of the corresponding middle z value $z_{I_{Ring}}$: We create a normal $\vec{a}_{I_{Ring}}$ with a fixed y value of 0 that forms the angle $\alpha_{I_{Ring}}$ with the unit vector in x direction $\vec{v}_x$. Consequently all normals that lie in the middle of a ring ($\alpha = \alpha_{I_{Ring}}$) have the z value $z_{I_{Ring}}$.

$$\vec{v}_x = (1, 0, 0) \tag{2}$$

$$\vec{a}_{I_{Ring}} = (x_a, 0, z_{I_{Ring}}) \qquad \begin{array}{l} \forall I_{Ring} \in [0, I_{MaxRing}] \\ with |\vec{a}_{I_{Ring}}| = 1 \end{array} \tag{3}$$

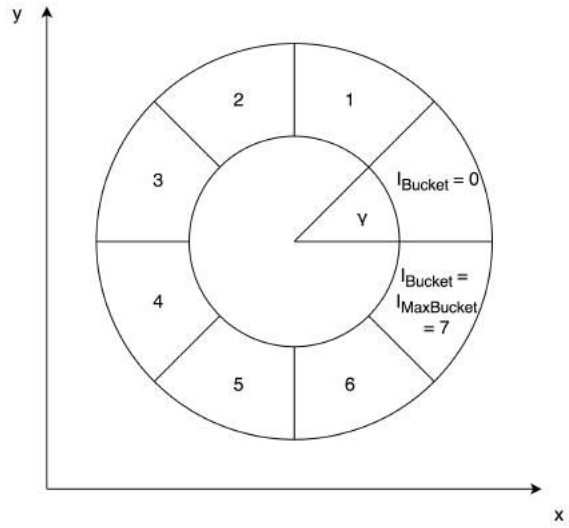


Figure 105: Buckets in a ring span the angle ϕ . Its projection is the angle γ.

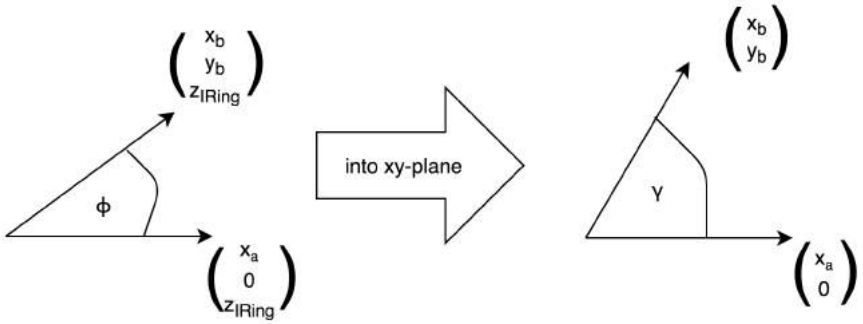


Figure 106: Projection of the angle ϕ

$$\cos\alpha_{I_{Ring}} = \frac{\vec{v}_x * \vec{a}_{I_{Ring}}}{|\vec{v}_x| * |\vec{a}_{I_{Ring}}|} \overset{(2)+(3)}{=} \vec{v}_x * \vec{a}_{I_{Ring}} \overset{(2)+(3)}{=} x_a \tag{4}$$

$$\begin{aligned} 1 = |\vec{a}_{I_{Ring}}| &\overset{(3)}{=} \sqrt{x_a^2 + z_{I_{Ring}}^2} \\ => z_{I_{Ring}} = \sqrt{1 - x_a^2} &\overset{(4)}{=} \sqrt{1 - cos^2\alpha_{I_{Ring}}} \end{aligned} \tag{5}$$

Each ring is seperated into buckets which cover the threshold angle ϕ as shown in Figure 105. Therefore the onto the xy-plane projected angle γ needs to be calculated which differs from ring to ring. This is shown in Figure 106 and leads to the following calculation.

$\vec{a}'_{I_{Ring}}$ is the projection of $\vec{a}_{I_{Ring}}$:

$$\vec{a}'_{I_{Ring}} = (x_a, 0) \tag{6}$$

$\vec{b}_{I_{Ring}}$ spans the threshold angle ϕ with $\vec{a}_{I_{Ring}}$ and has the same z value $z_{I_{Ring}}$:

$$\vec{b}_{I_{Ring}} = (x_b, y_b, z_{I_{Ring}}) \qquad with|\vec{b}_{I_{Ring}}| = 1 \tag{7}$$

The corresponding projected vector $\vec{b}'_{I_{Ring}}$:

$$\vec{b}'_{I_{Ring}} = (x_b, y_b) \tag{8}$$

Due to the mentioned conditions the coordinates of $\vec{b}_{I_{Ring}}$ are calculated as follows:

$$\cos\phi = \frac{\vec{a}_{I_{Ring}} * \vec{b}_{I_{Ring}}}{|\vec{a}| * |\vec{b}|} = \vec{a}_{I_{Ring}} * \vec{b}_{I_{Ring}} \overset{(3)+(7)}{=} x_{a\,I_{Ring}} * x_{b\,I_{Ring}} + z^2_{I_{Ring}} \tag{9}$$

$$(9) => x_b = \frac{\cos\phi - z^2_{I_{Ring}}}{x_a} \tag{10}$$

$$\begin{aligned} 1 &= |\vec{a}_{I_{Ring}}| = |\vec{b}_{I_{Ring}}| \\ &=> \sqrt{x_a^2 + z^2_{I_{Ring}}} = \sqrt{x_b^2 + y_b^2 + z^2_{I_{Ring}}} \\ &=> y_b = \sqrt{x_a^2 - x_b^2} \end{aligned} \tag{11}$$

The angle $\gamma_{I_{Ring}}$ which is used to separate a ring into buckets is calculated with the projected vectors $\vec{a}'_{I_{Ring}}$ and $\vec{b}'_{I_{Ring}}$:

$$\begin{aligned}
\cos\gamma_{I_{Ring}} &= \frac{\vec{a}'_{I_{Ring}} * \vec{b}'_{I_{Ring}}}{|\vec{a}'_{I_{Ring}}| * |\vec{b}'_{I_{Ring}}|} = \frac{x_a * x_b}{x_a * \sqrt{x_b^2 + y_b^2}} \\
&\overset{(11)}{=} \frac{x_a * x_b}{x_a * \sqrt{x_b^2 - x_b^2 + x_a^2}} = \frac{x_a * x_b}{x_a * x_a} = \frac{x_b}{x_a} \\
&\overset{(10)}{=} \frac{\cos\phi - z^2_{I_{Ring}}}{x_a^2} \overset{(4)}{=} \frac{\cos\phi - z^2_{I_{Ring}}}{\cos^2\alpha_{I_{Ring}}} \\
&\overset{(5)}{=} \frac{\cos\phi - (1 - cos^2\alpha_{I_{Ring}})}{\cos^2\alpha_{I_{Ring}}} = \frac{\cos^2\alpha_{I_{Ring}}}{\cos^2\alpha_{I_{Ring}}} + \frac{\cos\phi - 1}{\cos^2\alpha_{I_{Ring}}} \\
&\overset{(1)}{=} 1 + \frac{\cos\phi - 1}{\cos^2\left((I_{Ring} + 0.5) * \phi\right)} \\
=> \gamma_{I_{Ring}} &= \arccos\left(1 + \frac{\cos\phi - 1}{\cos^2\left((I_{Ring} + 0.5) * \phi\right)}\right)
\end{aligned} \tag{12}$$

The corresponding angle β of a normal $\vec{n}$ is calculated with use of the unit vector in x direction $\vec{v}'_x$. This angle needs to be adapted in case $y < 0$ since the maximum angle between two vectors is 180° and we want to cover a complete circle:

$$\begin{aligned}
\cos\beta &= \frac{\vec{n}' * \vec{v}'_x}{|\vec{n}'| * |\vec{v}'_x|} = \frac{x}{\sqrt{x^2 + y^2}} \\
=> \beta &= \begin{cases} \arccos\frac{x}{\sqrt{x^2+y^2}}, & y \geq 0 \\ 2\pi - \arccos\frac{x}{\sqrt{x^2+y^2}}, & y < 0 \end{cases}
\end{aligned} \tag{13}$$

The angle β is represented as a multiple of the γ angle to get the bucket index I_{Bucket}:

$$I_{Bucket} = \lfloor \frac{\beta}{\gamma_{I_{Ring}}} \rfloor \tag{14}$$

Consequently the maximum bucket index is calculated with the maximum angle of 360°:

$$I_{MaxBucket} = \lfloor \frac{2\pi}{\gamma_{I_{Ring}}} \rfloor \tag{15}$$

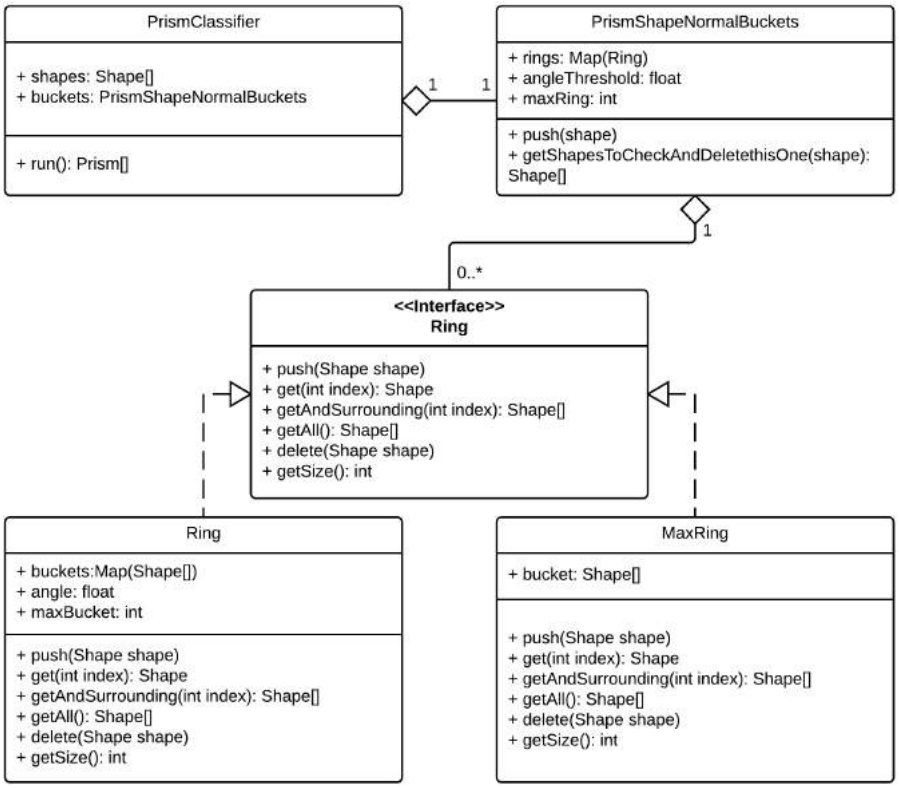


Figure 107: Class diagram of the PrismClassifier

12.3.3.2 *Implementation*

The *PrismClassifier* works on *shapes* from the *ClassifierGraph* and should also take account of *planes* in future development. Therefore it depends on the *PlaneClassifier*. It returns a set of *prisms* and uses *PrismShapeNormalsBuckets* to check shapes for parallelism. Its structure is represented in the class diagram Figure 107.

PRISMCLASSIFIER Listing 37 shows the implementation of the *PrismClassifier* run method:

At first the normal buckets are initialised with an angle threshold as described in section *Bucket System*. These values are stored in the global config with $\_angleThreshold = 0.15 = 8.6°$, previously labeled ϕ, so shapes with an angle of 8.6° will be considered parallel.

Then all shapes are put into the corresponding buckets. This is done by iterating over all shapes and passing them to the bucket system which handles the classification on its own.

Next we iterate over all shapes again while requesting the potential parallel shapes from the bucket system. The shapes are checked with the given threshold and a *prism* is created in case of parallelism.

```
run: ->
    return new Promise( (resolve) =>
        buckets = new
↪   PrismShapeNormalsBuckets(_angleThreshold)

        @_putIntoBuckets @shapes, buckets

        prisms = @_detectPrisms @shapes, buckets

        log.debug("PrismClassifier found " + prisms.length +
↪   " prisms")

        resolve(prisms)
    )
```

Listing 37: run method of the PrismClassifier

At the end a set of all prisms is returned which can be used for further primitive classification.

PRISM SHAPE NORMALS BUCKETS *PrismShapeNormalsBuckets* is based on the specification in Section 12.3.3.1 Bucket System Bucket System: It divides a hemisphere into rings with buckets and is initialised with the angle threshold.

The constructor saves the angle threshold, instantiates the rings as a javascript map and calculates the maximum ring index $I_{MaxRing}$.

It provides two functions used by the classifier:
push and *getShapesToCheckAndDeleteThisOne*.

push takes a *Shape* and puts it in the corresponding ring. At first it transforms its normal into the hemisphere. Then the ring index I_{Ring} is calculated. If the ring is not initialised yet it gets created and *ring.push(shape)* is called to sort the *Shape* into the corresponding bucket.

getShapesToCheckAndDeleteThisOne takes a *Shape* and removes it from the bucket system because it will be checked for all possible prisms by the classifier and should not be returned as a potential parallel shape in another pass. Then it returns an array of all *Shapes* lying in the same and the surrounding buckets: It requests the corresponding buckets from the ring with the shape's ring index I_{Ring} and the ring above and beneath.

While determining the surrounding rings following special cases need to be considered:

First: The ring with the maximum index $I_{MaxRing}$ is only adjacent to the ring underneath, because it is the highest ring on top of the hemisphere. It is not separated into different buckets and serves as a single bucket which is adjacent to all buckets in the ring underneath.

Second: The ring with index 0 "is adjacent to itself" because the unit sphere got transformed into a hemisphere. Therefore the buckets 180° around have to be taken into account for parallelism checks. Take two normals with the same x, y coordinates and a negated z value that span an angle smaller ϕ: $\vec{n_1} = (x, y, z)$ and $\vec{n_2} = (x, y, -z)$ with $z > 0$. Consequently $\vec{n_1}$ naturally lies in ring 0 and $\vec{n_2}$ gets negated to fit into the hemisphere. The negated normal is $\vec{n_2}' = (-x, -y, z)$ and lies also in ring 0 but in the opposite bucket 180° away.

RING A *Ring* contains buckets as a javascript map and is initialized with an index I_{Ring} and the angle threshold ϕ. With these the angle γ and the maximum bucket index $I_{MaxBucket}$ are calculated as described in Section 12.3.3.1 Bucket System. It provides the methods *push*, *get*, *getAndSurrounding*, *getAll*, *delete* and *getSize*.

The method *push* sorts a given shape into the corresponding bucket, *get* returns all shapes of a bucket with a given index, *getAndSurrounding* returns the same and additionaly the two neighboring buckets while *getAll* returns all shapes that lie in that ring. With *delete* a given *Shape* is removed from the ring and *getSize* returns the amount of *Shapes* that lie in that ring.

MAXRING The *MaxRing* is only instanced once and is the ring with index $I_{RingMax}$. It provides the same methods as a *Ring* but with adapted behaviour.

PRISM A *Prism* contains two parallel shapes that span a prism. In future work this should be extended by the lateral surface which is not handled yet.

12.3.3.3 *Future Work*

PLANES *Planes* from the *PlaneClassifier* should be taken into account in addition to the shapes. Ideally the *PlaneClassifier* detects noisy planes that are not represented as *Shapes*. Therefore a better approximation of the original model could be achieved.

LATERAL SURFACE The lateral surface of the prism has yet to be identified and added to the prism structure. This is needed for further operations and more specific classifications of the prism. For example for classifying a cube the surrounding sides have to be known.

SCORING Prism scoring is not implemented yet. To rate a prism the actual angle between the two nearly parallel shapes have to be considered. If scoring is implemented the angle threshold for parallelism can be adjusted to a much higher value while big variations to 0° will be reflected in the score.

12.3.4 *Other Primitives*

In addition to the just mentioned objects any other primitive can be included in the search with RANSAC.
For this a number of minimum support points is needed. This number of points needs to suffice to reconstruct the object.
Other points from the dataset can then be tested if they support the constructed object.

12.4 PROBLEMS WITH NON-POINT-CLOUD INPUT

In our use case we do not get a point-cloud as input. Instead we operate on the much fewer points in a polygon mesh.
This has large influence on the threshold values that are used. The values taken from CGAL almost never achieve the desired results. We tried to find new values for any model which turned out to be infeasible. Therefore we tried adjusting the thresholds based on the number of vertices in the mesh or the volume of the bounding box. We have not encountered a working combination yet. This is a problem that needs to be tackled in future work.

13

CONCLUSION AND FUTURE WORK

13.1 LASER CUTTING 3DMODELS

People have always come up with brilliant ideas. Currently, the way people fabricate their prototypes is changing. With a 3D printer you can create objects without having any knowledge on the fabrication process. All it takes is the push of a button.
Thanks to the wide popularity of 3D printing there are a lot of 3D models available on the internet.
We have shown that such models can be converted to a 2D plan which can be laser cut. Originally, one needed a lot of know how for creating such plans. The transition from the 3D world to a flat surface is not an easy task. Especially since the flat objects need connectors which have to fit into the object when assembled.
Our software automatically generates a 2D plan for laser cutting a 3D model. This supports the progress of automation of fabrication techniques.

13.2 MAKER FAIRE RUHR, VIENNA, HANOVER

For receiving feedback on our application we participated in three Maker Faires.
Those are events where people who have had an idea came up with a solution or a prototype and want to exhibit it to share their learnings and obtain feedback.
Visitors at our booth were very interested in our work. Many told us they built or bought a laser cutter for their home. But most of them said they never thought of making such 3 dimensional objects which we presented to them.

We want to open up new possibilities to have anyone, even without a lot experience with a laser cutter, be able to put their ideas into practice.

BIBLIOGRAPHY

[1] The computational geometry algorithms library. URL http://www.cgal.org/.

[2] Lasercut like a boss. URL http://digitalcommons.olin.edu/cgi/viewcontent.cgi?article=1003&context=clare_boothe_luce.

[3] Least squares fitting of data. URL http://www.geometrictools.com/Documentation/LeastSquaresFitting.pdf.

[4] Schnitt zweier ebenen in normalenform. URL https://de.wikipedia.org/wiki/Schnittgerade#Schnitt_zweier_Ebenen_in_Normalenform.

[5] Dan Abramov. Redux - three principles. 2016. URL http://redux.js.org/docs/introduction/ThreePrinciples.html.

[6] Wohlers Associates. 3d printing and additive manufacturing industry expected to quadruple in size in four years. http://wohlersassociates.com/press65.html.

[7] Dustin Beyer, Stefanie Mueller, and Patrick Baudisch. *Platener: Low-Fidelity Fabrication of 3D Objects by Substituting 3D Print with Laser-Cut Plates*. Master thesis, Hasso Plattner Institute, Potsdam, Germany, September 2014.

[8] F. Durand. A short introduction to computer graphics. http://people.csail.mit.edu/fredo/Depiction/1_Introduction/reviewGraphics.pdf. Accessed: 2016-06-21.

[9] Garret Foster. Understanding and implementing scene graphs. http://archive.gamedev.net/archive/reference/programming/features/scenegraph/index.html. Accessed: 2016-06-21.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[11] H. Heckner and M. Wirth. Vergleich von dateiformaten für 3d-modelle. http://cedifa.de/wp-content/uploads/2014/05/07_3D-Modell-Formate.pdf. Accessed: 2016-06-22.

[12] Pete Hunt. Why did we build react. 2013. URL http://facebook.github.io/react/blog/2013/06/05/why-react.html.

[13] Pete Hunt. React documentation - multiple components. 2015. URL http://facebook.github.io/react/docs/multiple-components.html.

[14] Jim McCann. URL http://www.cs.cmu.edu/~jmccann/.

[15] Stefanie Mueller, Bastian Kruck, and Patrick Baudisch. Laserorigami: Laser-cutting 3d objects. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 2585–2592, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1899-0. doi: 10.1145/2470654.2481358. URL http://doi.acm.org/10.1145/2470654.2481358.

[16] Stefanie Mueller, Tobias Mohr, Kerstin Guenther, Johannes Frohnhofen, and Patrick Baudisch. fabrickation: Fast 3d printing of functional objects by integrating construction kit building blocks. In *CHI '14 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '14, pages 187–188, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2474-8. doi: 10.1145/2559206.2582209. URL http://doi.acm.org/10.1145/2559206.2582209.

[17] R. Nystrom. Game programming patterns - game loop. http://gameprogrammingpatterns.com/game-loop.html. Accessed: 2016-06-21.

[18] Addy Osmani. *Learning JavaScript Design Patterns - a JavaScript and jQuery Developer's Guide.* O'Reilly, 2012. ISBN 978-1-449-33181-8.

[19] Elizabeth Palermo. Fused deposition modeling: Most common 3d printing method. http://www.livescience.com/39810-fused-deposition-modeling.html. Accessed: 2016-07-21.

[20] The Linux Information Project. Command line interface definition. URL http://www.linfo.org/command_line_interface.html. Accessed: 2016-07-19.

[21] Reinhard Klein Ruwen Schnabel, Roland Wahl. Efficient ransac for point-cloud shape detection. URL http://cg.cs.uni-bonn.de/aigaion2root/attachments/schnabel-2007-efficient.pdf.

[22] Greg Saul, Manfred Lau, Jun Mitani, and Takeo Igarashi. 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, pages 73–80, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0478-8. doi: 10.1145/1935701.1935717. URL http://doi.acm.org/10.1145/1935701.1935717.

[23] A. Schatten, M. Demolsky, Winkler D., Biffl S., Gostischa-Franka E., and Östreicher T. Delegation pattern - best practice software-engineering. http://best-practice-software-engineering.

ifs.tuwien.ac.at/patterns/delegation.html. Accessed: 2016-06-21.

[24] Arthur Silber, Stefan Neubert, Johannes Deselaers, Yannis Kommana, and Adrian Sieber. *Brickify: Fast 3D printing through interactive conversion of 3D geometry to LEGO-Bricks.* Bachelor thesis, Hasso Plattner Institute, Potsdam, Germany, June 2015.

[25] Dan Williams. Overview of build systems. URL http://www.cs.virginia.edu/~dww4s/articles/build_systems.html. Accessed: 2016-07-21.

DECLARATION

I certify that the material contained in this thesis is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Ich erkläre hiermit weiterhin die Gültigkeit dieser Aussage für die Implementierung des Projekts.

Potsdam, July 25, 2016

Daniel-Amadeus J. Glöckner
Chapter 2 USERINTERACTION, Section 7.2.2 Shapes Finder in Chapter 7 PLATES, Chapter 10 CURVES

Sven Mischkewitz
Chapter 1 INTRODUCTION, Section 2.4 *platener* as a Command Line Interface for Advanced Users in Chapter 2 USERINTERACTION, Chapter 3 IMPLEMENTATION GOALS AND TOOLCHAIN, Chapter 4 ARCHITECTURE

Dimitri Schmidt
Chapter 5 PROCESSING PIPELINE, Chapter 6 APPROXIMATION, Section 7.2.3 Hole Detection in Chapter 7 PLATES, Section 7.5 Removing Contained Plates in Chapter 7 PLATES, Section 12.3.3 Prism Classifier in Chapter 12 CLASSIFIERS

Klara Seitz

Chapter 8 ADJACENCY PLATEGRAPH, Chapter 9 JOINT COMPUTATION, Chapter 12 CLASSIFIERS, Chapter 13 CONCLUSION AND FUTURE WORK

Lukas Wagner

Chapter 7 PLATES, Chapter 11 ASSEMBLY