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

PLATENER: GENERATING 2D LASERCUTTABLE CONSTRUCTION PLANS FROM 3D-MODELS

PLATENER: GENERATING 2D LASERCUTTABLE CONSTRUCTION PLANS FROM 3D-MODELS

DANIEL-AMADEUS J. GLOECKNER, 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

Human-Computer Interaction Group
Hasso Plattner Institute
University of Potsdam

July 2016

Daniel-Amadeus J. Gloeckner, Sven Mischkewitz, Dimitri Schmidt,
Klara Seitz, Lukas Wagner :
Interactive Construction
July 2016

ADVISOR:
Prof. Dr. Patrick Baudisch

To Deep Thought

ABSTRACT

Paragraph 1

Paragraph 2

ZUSAMMENFASSUNG

Paragraph 1

Paragraph 2

PUBLICATIONS

This thesis is NOT based on the following publications:

Papers

Mueller, S., Lopes, P., Baudisch, P. Interactive Construction: Interactive Fabrication of Functional Mechanical Devices. In *Proceedings of UIST'12,* pp. 599-606. (Fullpaper)

Mueller, S., Kruck, B., Baudisch, P. LaserOrigami: Laser-Cutting 3D Objects. In *Proceedings of CHI'13,* pp. 2585-2592. (Fullpaper)
[Best Paper Award]

Demonstrations

Mueller, S., Lopes, P., Kaefer, K., Kruck, B., Baudisch, P. constructable: Interactive Construction of Functional Mechanical Devices. *Invited demo at TEI'13*.

Mueller, S., Lopes, P., Kaefer, K., Kruck, B., Baudisch, P. constructable: Interactive Construction of Functional Mechanical Devices. In *Extended Abstracts of CHI'13*, pp. 3107-3110.

Mueller, S., Kruck, B., Baudisch, P. LaserOrigami: Laser-Cutting 3D Objects. In *Extended Abstracts of CHI'13*, pp. 2851-2852.

Talks

Mueller, S., Lopes, P., Kaefer, K., Kruck, B., Baudisch, P. constructable: Interactive Construction of Functional Mechanical Devices. In *Proceedings of SIGGRAPH '13,* ACM SIGGRAPH 2013 Talks, Article No. 39.

Disclaimer
All projects are based on a group effort with the primary investigator being Stefanie Mueller under supervision of Prof. Dr. Baudisch. In the process of preparing constructable and LaserOrigami for publication and demonstration, Pedro Lopes built constructable's laser pointer toolbox and the foot switch control. Konstantin Kaefer and Bastian Kruck reengineered constructable for demonstrations and integrated LaserOrigami into constructable. The implementation described in this thesis is the original architecture implemented by Stefanie Mueller.

ACKNOWLEDGMENTS

So long and thanks for all the fish.

CONTENTS

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 **??**). There are ideas concerning creative processes, like designing a fashion line for shoes which offer correct fit to the the feet of its wearer (Figure **??**). 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], see Figure **??**.

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[**?**]. 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)[]source that FDM is common. 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[]source thats how FDM works.

measure bird house dimensions

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 (4x4x5cm) in Figure **??** printed for three

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

hours[2]. In its original dimensions (30x30x45cm) it requires 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. These characteristics are aesthetics or haptics, for example the feeling when touching the material surface.

Another fabrication machine is the laser cutter, shown in Figure **??**. A laser traces the outlines of a planar cutting plan, producing plates which are cut out from flat materials (Figure **??**). Such materials are wood or acrylic. High-performance devices can cut textiles, metals, and stone. The cutting process requires very short time, compared to the printing time of a 3D printer. Figure **??** and Figure **??** 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 **??**.

With a laser cutter users can design objects with decorative aspects (Figure **??**). The laser cutter preserves the original structure of the used materials. The fabricated object benefits from the unique characteristics of the material.

When creating three dimensional 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 **??** 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 users to produce three dimensional objects with a laser cutter. We produce cutting plans from 3D models automatically. We improve the creation

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

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

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 the model with plates. Users are not required to map the 2D cutting plan into 3D space manually. 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.

cross refs

We present our system architecture in Chapter **??** and give an overview of the most important data structures in Chapter **??**. From Chapter **??** to Chapter **??** we explain the conversion process of 3D models to 2D cutting plans.

1.2 RELATED WORK

PLATENER **?** 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.

how to keep previous and now platener apart?

BRICKIFY **?** present a low-fidelity fabrication approach using 3D printers and *LEGO*^{TM} 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 **?** .

add bib for fabrickation

LASERORIGAMI **?** 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. **?** 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.

add bib for laser origami

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

add bib for sketchchair

123MAKE Autodesk[4] ships a 3D editor which converts 3D models into slices. 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.

some source for 123make

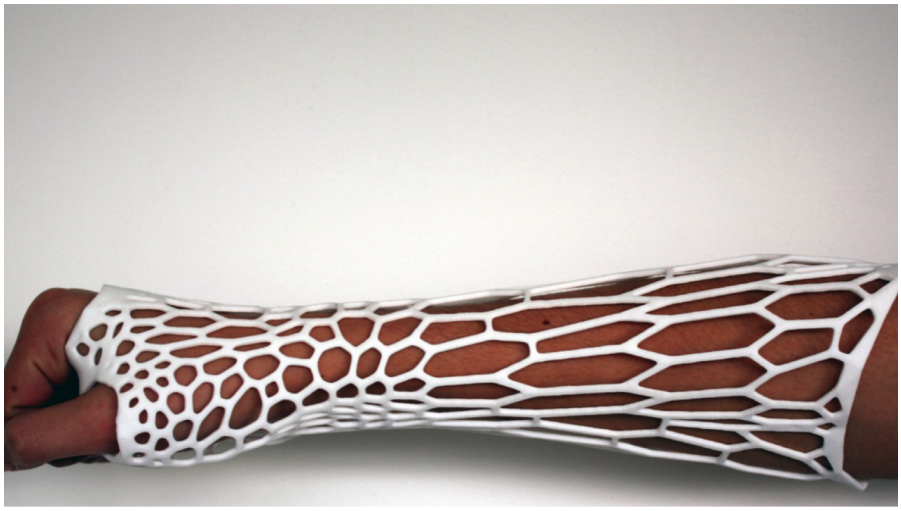
4 http://www.autodesk.de/

(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 21, 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 21, 2016)



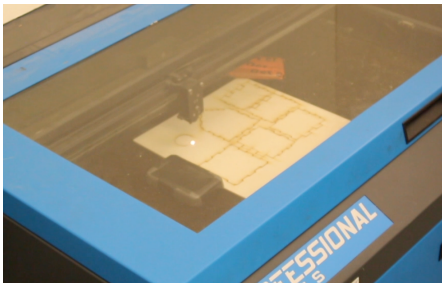
(c) *Cortex*, a fully ventilated plaster.

Source: ©Jake Evill, http://www.evilldesign.com/image/41499/2000x0-5/p18qh3vpntsq91sg21a261dr114dc6.jpg (visited on July 21, 2016)

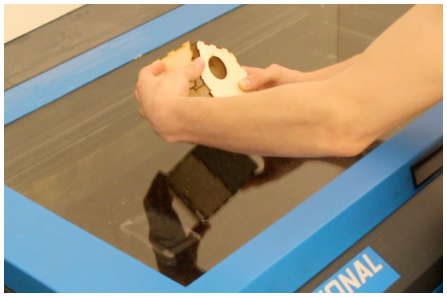
Figure 1: With fabrication machines, broad ideas come to life.

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.

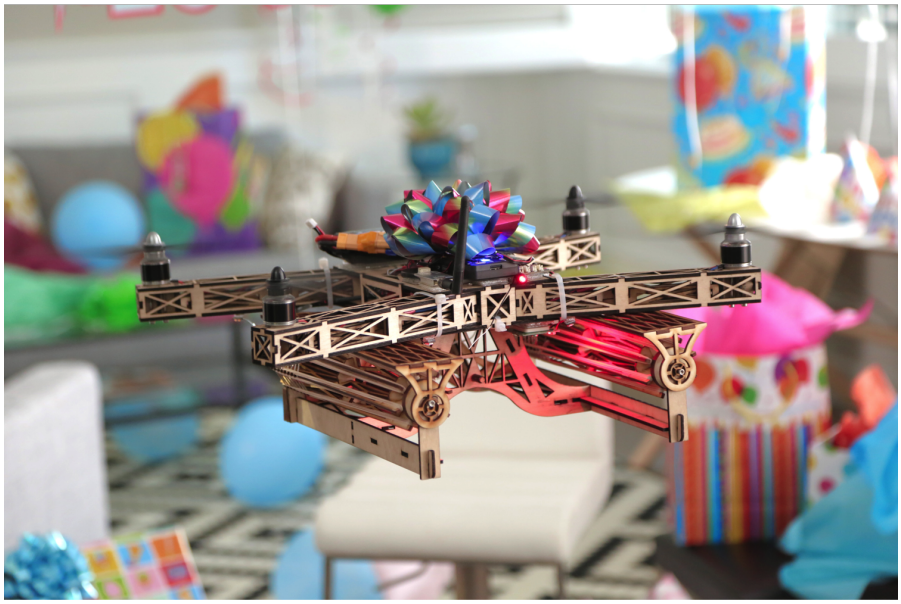


Figure 4: A laser cutted quadcopter.

Source: http://wedreamabout.com/wp-content/uploads/2016/01/drone-plywood-mixedbg-2.jpg (visited on July 21, 2016)

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

Source: https://i.ytimg.com/vi/q0dDhesoC08/maxresdefault.jpg (visited on July 21, 2016)

2

USERINTERACTION

2.1 DRAG N DROP MAIN INTERACTION

2.2 CUSTOMIZE VIEW

2.2.1 *Material Parameters*

2.2.2 *Device Parameters (Calibration)*

2.3 DEBUG VIEW

2.3.1 *Pipeline Visualizations*

2.3.2 *Algorithm/ Method Selection*

2.3.3 *Testables (Operating Mode)*

2.3.4 *Console Debug Output*

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. We intent to maximize scalability and user-friendliness by providing a web application. 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. Also, the web solution focuses on the results of the conversion. So far, we have no 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 **??** and we demonstrate the logging capabilities in Section **??**.

2.4.1 *Usage Instructions of platener's CLI*

A CLI is self-documenting. Listing **??** 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. Also, we can recursively read input directories to convert each STL file for the laser cutter. Both commands are given in Listing **??**.

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

```
# 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.

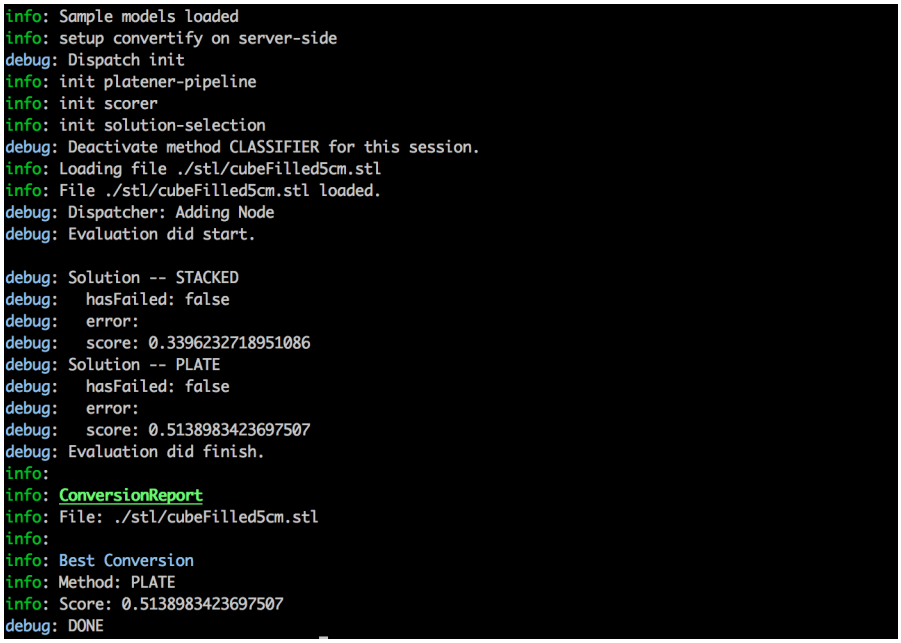


Figure 6: A *Report*, showing a summary of the conversion.

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 where and under what circumstances the software failed. With reports, we document the conversions 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 **??** shows a summary report. A report itemizing the results of our three conversion approaches can be seen in Figure **??**. In Figure **??** 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 **??**.

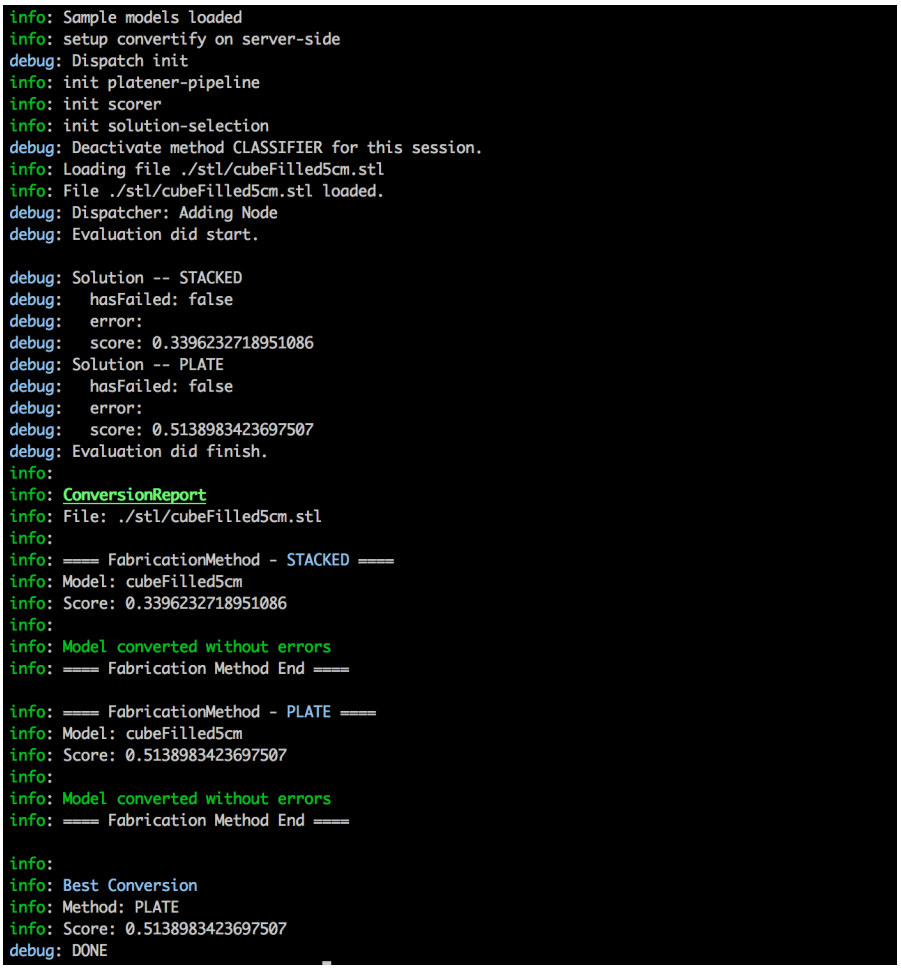


Figure 7: A *Report*, showing a separate summary for each fabrication method of the conversion.

```
info: ConversionReport
info: File: ./stl/cubeFilled5cm.stl
info:
info: No Filesystem Error and no Unexpected Error occured
info:
info: ==== FabricationMethod - STACKED ====
info: Model: cubeFilled5cm
info: Score: null
info:
error: Pipeline Error
error: Pipeline was cancelled during computation. Could not save any data for method STACKED
on model cubeFilled5cm name=HasFailedError, message=Pipeline was cancelled during computation
. Could not save any data for method STACKED on model cubeFilled5cm, stack=HasFailedError: Pi
peline was cancelled during computation. Could not save any data for method STACKED on model
cubeFilled5cm
    at hasFailedError (webpack:///./src/server/convertHelper.coffee:43:1)
    at Conversion.saveToFile (webpack:///./src/server/Conversion.coffee:54:1)
    at webpack:///./src/server/convert.coffee:65:1
    at tryCatcher (/Users/sven/Documents/workspace/platener/node_modules/bluebird/js/release/
util.js:16:23)
    at Object.gotValue (/Users/sven/Documents/workspace/platener/node_modules/bluebird/js/rel
ease/reduce.js:145:18)
    at Object.gotAccum (/Users/sven/Documents/workspace/platener/node_modules/bluebird/js/rel
ease/reduce.js:134:25)
    at Object.tryCatcher (/Users/sven/Documents/workspace/platener/node_modules/bluebird/js/r
elease/util.js:16:23)
    at Promise._settlePromiseFromHandler (/Users/sven/Documents/workspace/platener/node_modul
es/bluebird/js/release/promise.js:504:31)
    at Promise._settlePromise (/Users/sven/Documents/workspace/platener/node_modules/bluebird
/js/release/promise.js:561:18)
    at Promise._settlePromiseCtx (/Users/sven/Documents/workspace/platener/node_modules/blueb
ird/js/release/promise.js:598:10)
    at Async._drainQueue (/Users/sven/Documents/workspace/platener/node_modules/bluebird/js/r
elease/async.js:143:12)
    at Async._drainQueues (/Users/sven/Documents/workspace/platener/node_modules/bluebird/js/
release/async.js:148:10)
    at Immediate.Async.drainQueues [as _onImmediate] (/Users/sven/Documents/workspace/platene
r/node_modules/bluebird/js/release/async.js:17:14)
    at processImmediate [as _immediateCallback] (timers.js:371:17)
info: ==== Fabrication Method End ====
```

Figure 8: When a conversion fails the *Report* shows the error message.

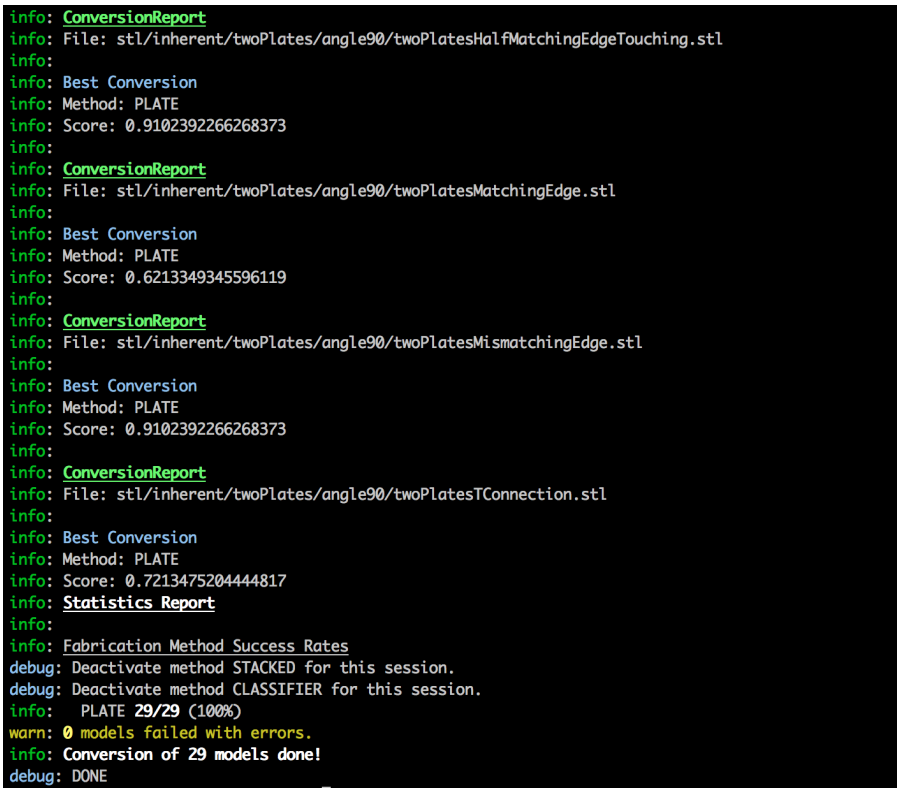


Figure 9: A sequence of conversions is summarized when all computation completed.

3 IMPLEMENTATION GOALS AND TOOLCHAIN

check all footnotes

3.1 THE SOFTWARE IS A WEB SERVICE

Our application platener is a web service. platener can be used without installation on desktop and mobile devices. We aim to enable any user to produce 3D objects with a laser cutter. We minimize the system requirements of our software by providing a cross-platform web application. platener is based on HTML 5, CSS, CoffeeScript and WebGL. The web server runs in Node.js. All code 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[1] hosts about 250.000 code packages. By reusing modular code 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* is a JavaScript library for building modular user interfaces.
- *Redux* is a predictable state container which helps us to manage the application state in a data-driven way.
- *stylus* 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* CSS framework.
- *three.js* provides high-level abstractions of the WebGL API.
- *webpack* is a module bundler. It prepares all source code and assets into an deployable package.

1 https://www.npmjs.com/

- *Grunt* is a JavaScript task runner with which we automate our setup and build process.

3.3 DEVELOPMENT WITH A BUILD SYSTEM

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 [**?**]. Figure **??** gives an overview of our toolchain.

Figure 10: Toolchain and build system of platener.

With our setup the code automatically rebuilds when files are changed by a developer. Once the new code is built the sources are loaded into the browser. Loading sources from the server can take several seconds. We reduce the waiting 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. Our software 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 **??**. The design of convertify is explained in Section **??**. Finally, we give details of the structure of platener in Section **??**.

4.1 COMPUTER GRAPHICS IN WEB ENVIRONMENTS

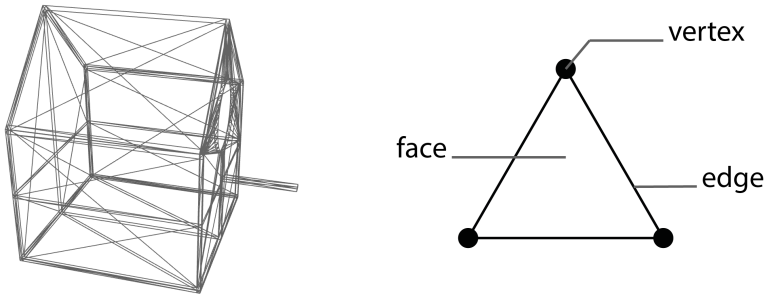
In this section we give a brief overview of graphics programming fundamentals in general and in web environments. These fundamentals are the concepts of 3D model data representation in Section **??** and render loops and scene graphs in Section **??**.

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 connected polygons, approximating the model surface. These connected polygons form a mesh. Figure **??** shows a mesh. We use triangles as polygons, because then we benefit from hardware acceleration. A triangle in the mesh is called face. Figure **??** illustrates the terminology. Each face is described by a list of three 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. An edge is the line between two connected vertices [**?** , p. 3].

platener supports all geometry which is arranged in two-manifold meshes. Two-manifoldness is a constraint on the mesh, which requires each egde to exactly touch two neighboring faces. Figure **??** shows a non-manifold part of a mesh. Each triangular face of a two-manifold mesh has to have three adjacent faces. This constraint en-



(a) A triangle mesh of a bird house.

(b) A single face with vertices and edges.

Figure 11: Terminology of a Mesh

forces the mesh to be a fully connected graph without holes [**?** , p. 28]. In order to make sophisticated assumptions when designing the conversion algorithms, the software requires two-manifold meshes.

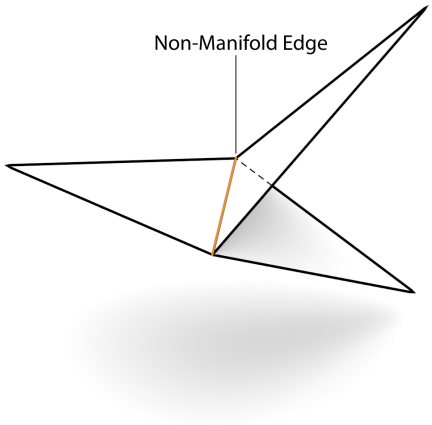


Figure 12: A non-manifold cutout of a mesh.

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 **??** shows an STL file in ASCII encoding. An STL file can also be stored in a bi-

nary format. The binary format is used commonly, because it requires less disk-space than the ASCII encoding [**?** , 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. Two faces that share the same vertex do refer to that vertex by floating point numbers as well. 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 distinguish fail-safe. We have to assume that two overlapping points represent an identical vertex. We use the library Meshlib 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. The section below explains how the three.js representation is displayed on the screen.

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 [**?** , p. 2]. A 3D model 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 [**?**]. Figure **??** shows an exemplary flow.

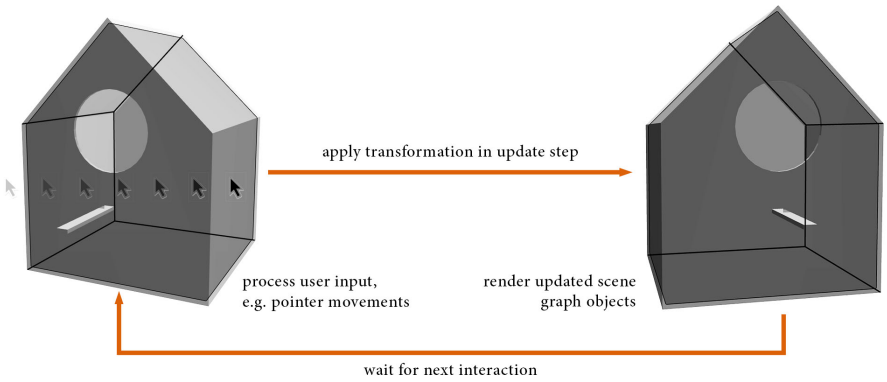
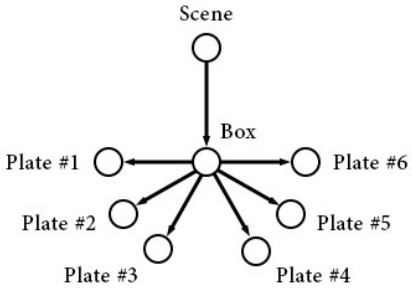
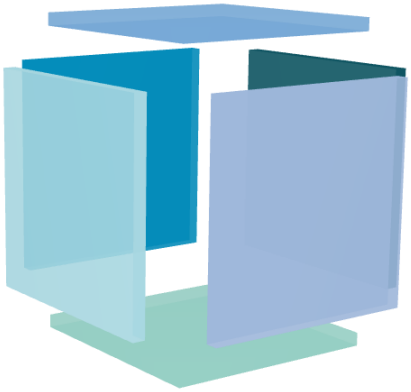


Figure 13: Pointer input is processed by the render loop.

Each object is part of a scene. A scene is the visual space, in which the rendered objects can be seen. We use representations of objects which are organized in a hierarchical tree data structure. We refer to this hierarchical structure as scene graph**??**reference scene graph data structure. A simple scene graph is depicted in Figure **??**. 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 object that is recognized by convertify, is part of a scene graph. Finally, the vertices of an object 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 14: 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.

cross-platform/ headless capabilites chapter missing - then we can introduce the separated client/ server package in the platener chaper

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 **??** 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 **??**. 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.

The plugins package and the client package are not part of the convertify framework. These packages are implemented by an application like platener, which is based on convertify. The plugins package provides an exchangeable set of features. 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 **??**. 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.

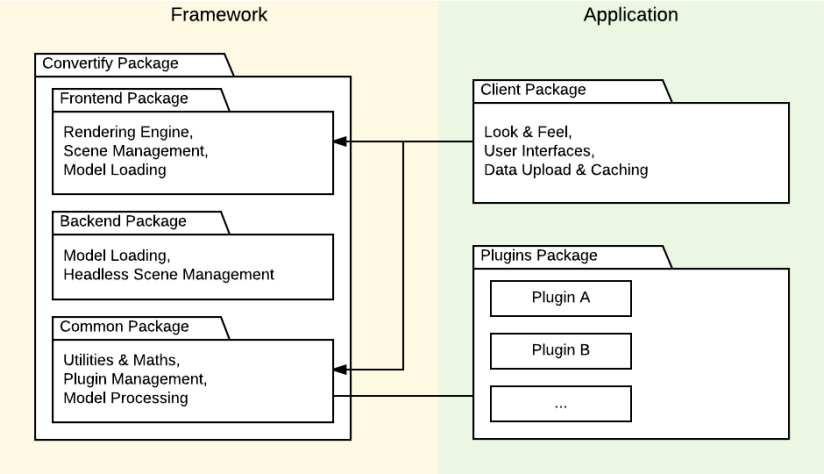


Figure 15: Packages of convertify

We will explain core features of convertify in Section **??**. Then we will dive into its plugin system in Section **??**. convertify is based on previous work by **?** . We will give a short comparison of their work with convertify in Section **??**. We elaborate on the client package when we talk about our application platener in Section **??**.

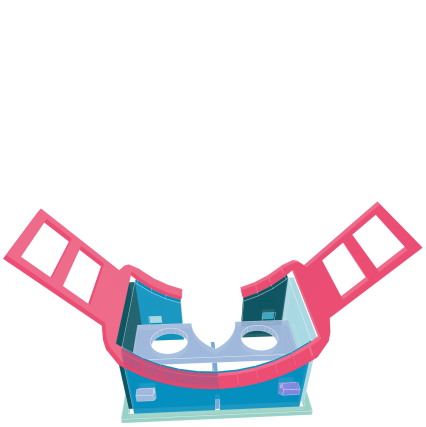
4.2.1 *Introduction to the Core System of convertify*

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.

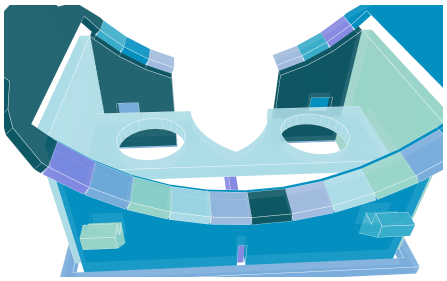
4.2.1.1 *3D models Are Managed in a Flat Scene Graph*

We use a flat scene graph. A flat scene graph has only one level of hierarchy. Such a graph is sufficient, because the loaded 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 **??** 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 **??** illustrates computed plates of the virtual reality glasses.



(a) The bended area is highlighted in red.



(b) Plates are computed by an algorithm.

Figure 16: The input model are virtual reality glasses.

All *Nodes* are part of a *Scene*. 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*. Figure **??** shows how these classes relate.

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

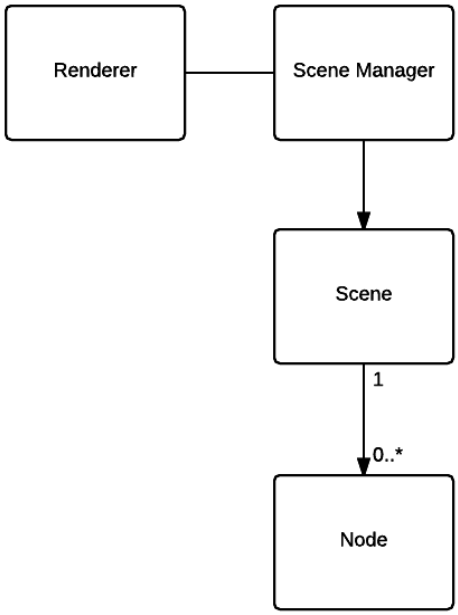


Figure 17: Relation of scene graph components

be rendered. Such three.js objects are instances of *THREE.Object3D*[1]. This is a generic object, which can be rendered into a WebGL scene. These *THREE.Object3D* will be displayed on the screen. The scene graph of three.js is not flat. A *THREE.Object3D* can have a hierarchy of objects of any size. Figure **??** 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 **??**. 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 **??**. 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 multiple independent plugins, these plugins are integrated with the core

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

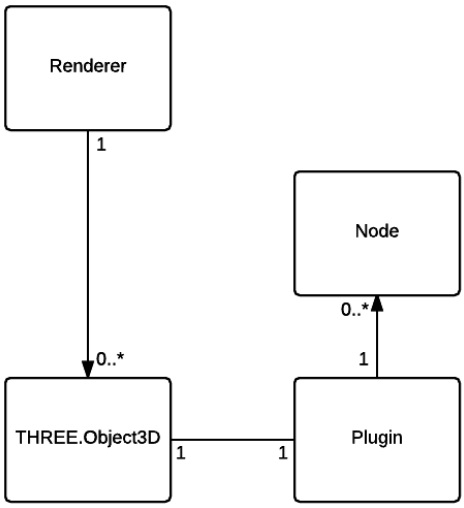


Figure 18: *Nodes* and indirectly associated *THREE.Object3D*

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 **??** 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 adds features 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. We will look at the details of plugins and plugin communication in the next section.

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.

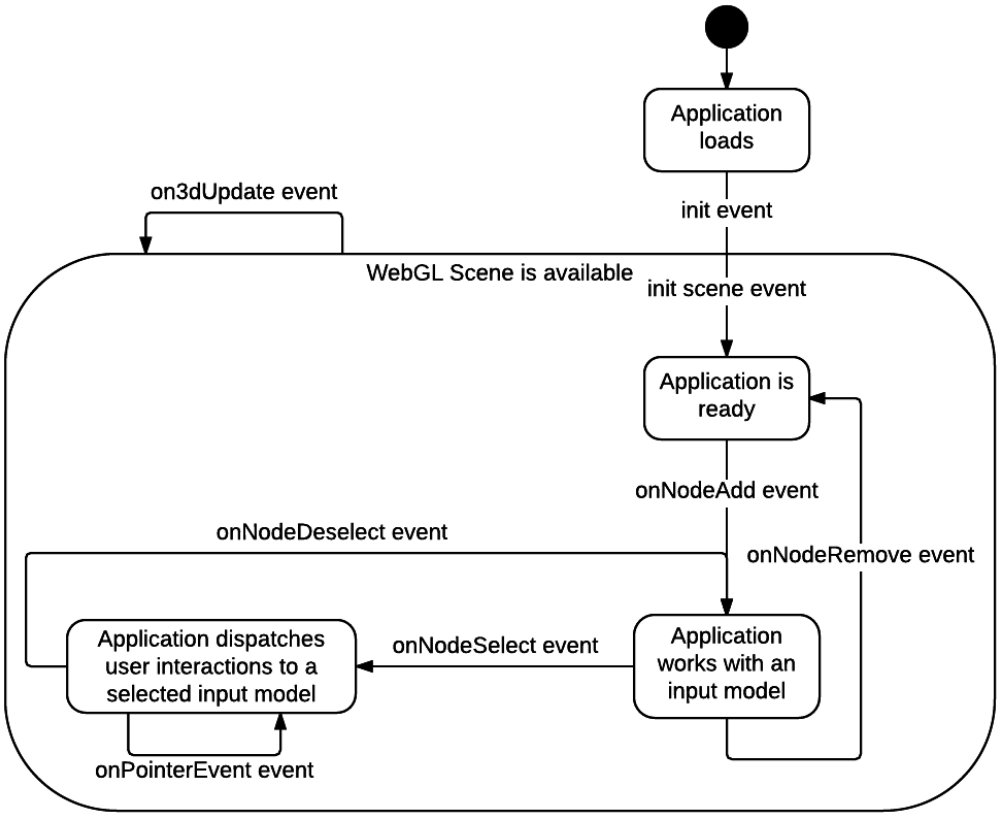


Figure 19: The complete lifecycle of convertify

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 **??** shows our seven plugins and how they relate to each other. We give an overview for each plugin in Section **??**.

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

convertify emits system events from internal components. Figure **??** 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

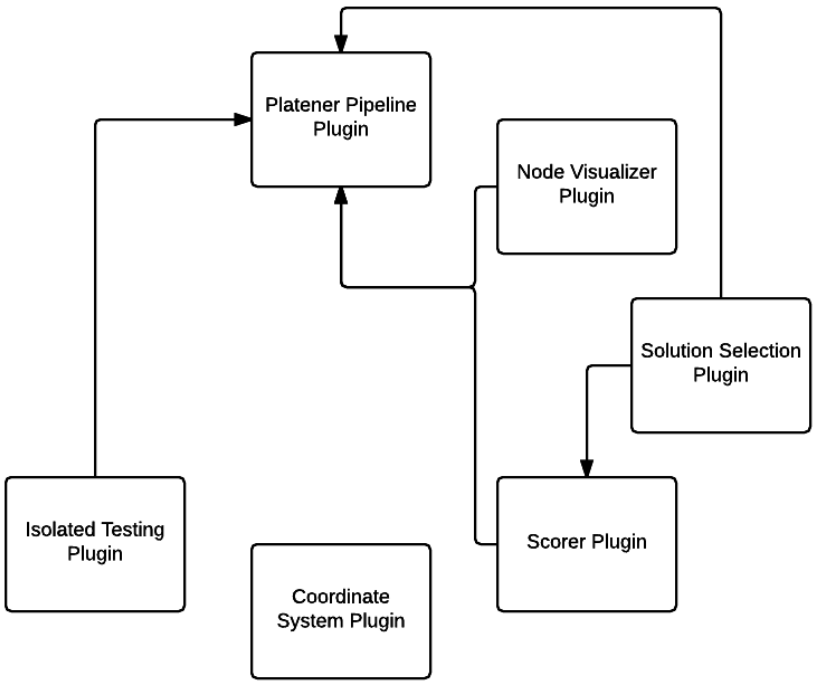


Figure 20: Platener's plugins and their dependencies.

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 **??** 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 **??** shows, the dependencies between plugins are hard to oversee. We propose a *Dispatcher* component, behaving similar to the *mediator* pattern. Figure **??** shows how the dependencies between plugins are managed with a *Dispatcher*.

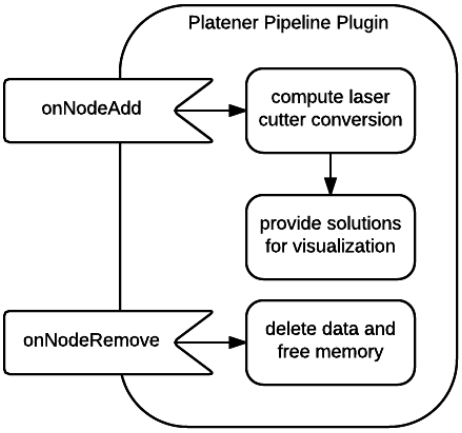


Figure 21: Workflow of the *PlatenerPipeline* plugin.

According to **?** , 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 [**?** , 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 **??**. 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 **??** shows how the mediator connects two plugins. A *Protocol* is a mixin implemented by the *Dis-*

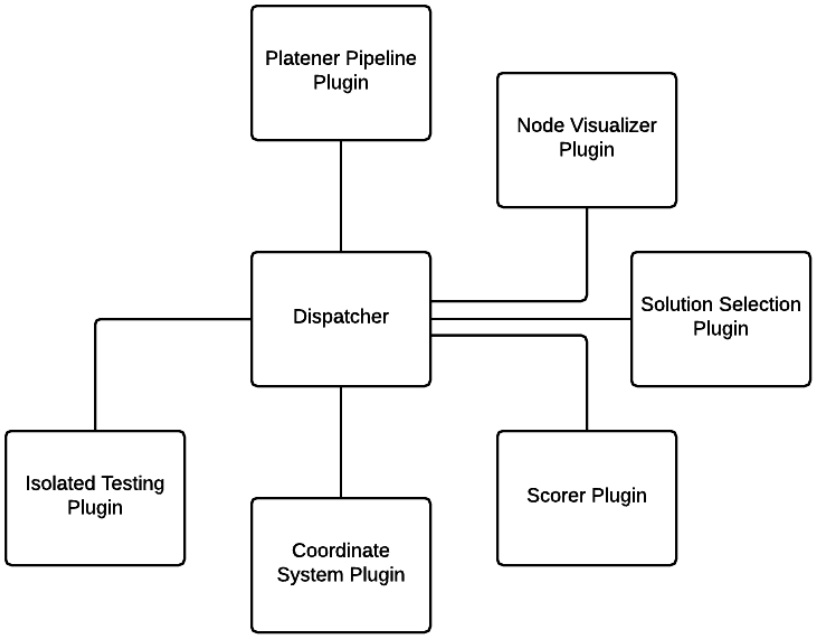


Figure 22: Platener's plugins, managed by a *Dispatcher*.

patcher. A mixin adds functionality to an object by object composition[**?** , 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 [**?**]. 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 we can animate the scene programmatically or sync scenes of mul-

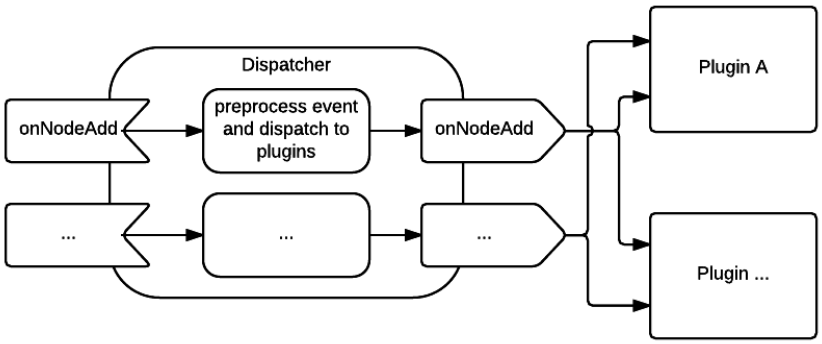


Figure 23: The *Dispatcher* remits system events in an explicit order.

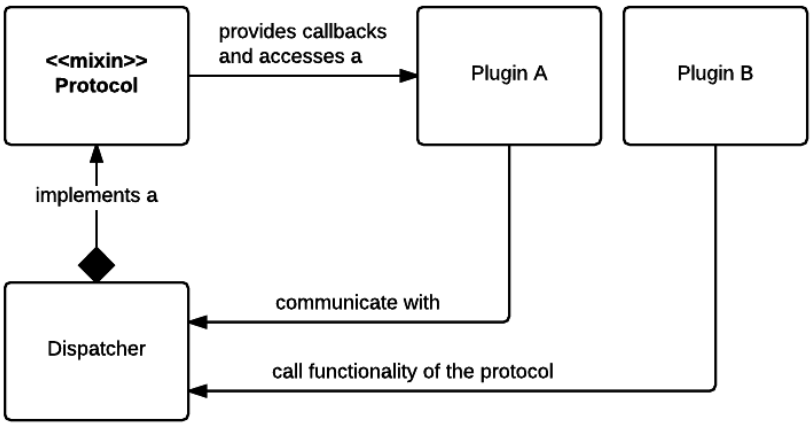


Figure 24: Plugin B can use features of Plugin A by calling functionality of the *Protocol* implemented by the *Dispatcher*.

tiple 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 **??** shows the *Bundle* in the context of an application.

4.2.3 *convertify is a cross-platform isomorphic framework*

convertify is a cross-platform framework. Cross-platform JavaScript code runs seamlessly in multiple browsers being aware of their dif-

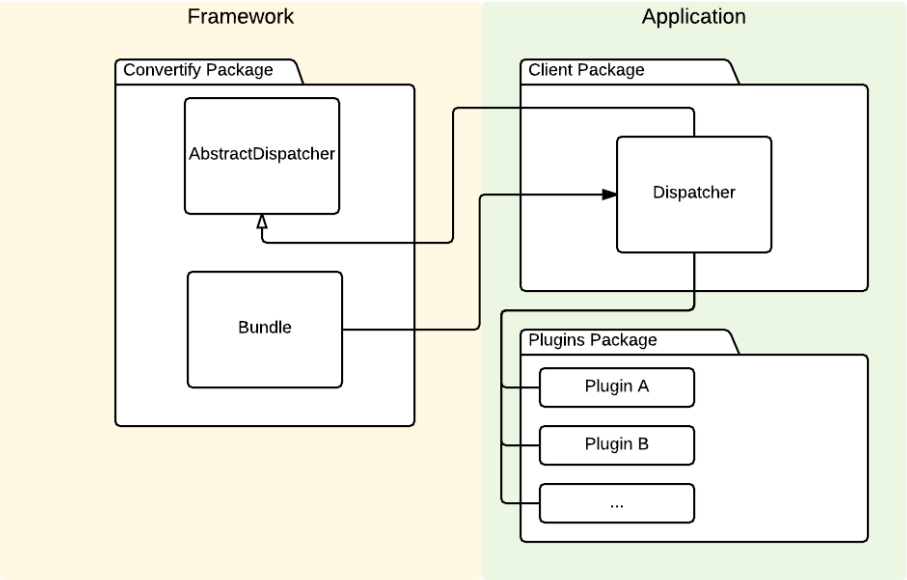


Figure 25: A *Bundle* is configured with a *Dispatcher*.

ferent implementations and APIs. This allows convertify to run on almost any device.

To enable cross-platform JavaScript support we use polyfills and libraries that are cross-platform as well.

A polyfill is "a piece of code or plugin that provides technology that [we] expect the browser to support natively"[2]. For example legacy JavaScript engines do not support data structures like *Map*, *Set* or *Promise*. We provide these features with the *es6-shim*[3] polyfill.

Using libraries that have cross-platform support themselves enables us to maintain convertify as a cross-platform framework. Such libraries are for example *lodash*[4] 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 [5].

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[6]. The most common non-browser JavaScript environment is Node.js[7]. convertify can use its full feature set in Node.js. This enables us to write applications

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

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 **??**.

4.2.4 *convertify is built on brickify*

The web application brickify was introduced in Section **??**. 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 their 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

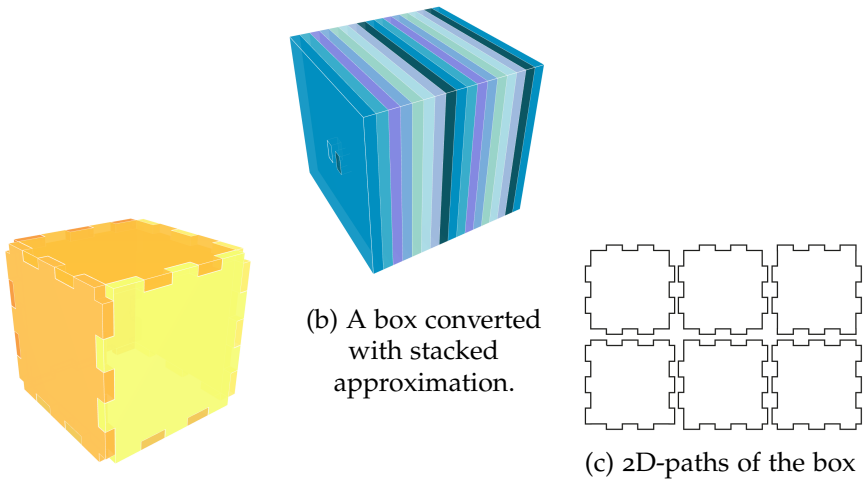
platener is a web-application that converts 3D-printable models into laser-cuttable equivalents. This section describes the implementation of platener within the convertify framework. Therefore, platener integrates with the system events and rendering engine of convertify. As proposed in the preceding section, it uses several plugins to bring its features to the WebGL scene. Section **??** describes these plugins briefly. Section **??** shows architectural details to the most important plugin: the *PlatenerPipeline* plugin. This plugin implements different conversion strategies for 3D models. In Section **??** 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 **??** 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 **??** and Figure **??** 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 **??**. Section **??** explains the architecture of the *PlatenerPipeline* plugin in detail.



(a) A box converted with hull approximation.

(b) A box converted with stacked approximation.

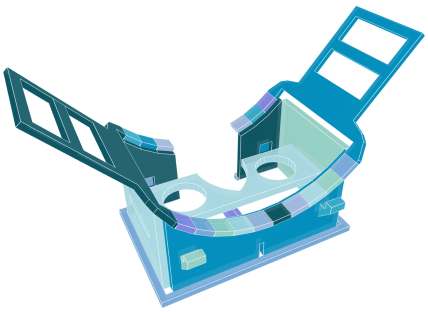
(c) 2D-paths of the box converted with hull approximation.

Figure 26: Results of the *PlatenerPipeline* plugin.

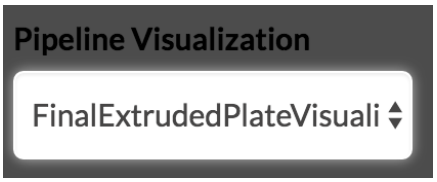
4.3.1.2 *Node Visualizer Plugin*

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 **??**section walkthrough, we select a single visualization at a time to inspect the results of the step associated with the visualization. Figure **??** shows a head-mounted display in the *Plate* step. Figure **??** shows the selection of that visualization in the user interface.



(a) A head-mounted display, consisting of plates only.



(b) The selection of the intermediate plate step in the user interface.

Figure 27: Visual debugging of intermediate conversion results.

4.3.1.3 *Scorer Plugin*

We run multiple conversion approaches sequentially. Then, we choose the best fitted conversion as output. Thus 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.

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 **??** shows the coordinate system in the WebGL view. The Coordinate System is taken from *Brickify* as is[**?** , p. 92].

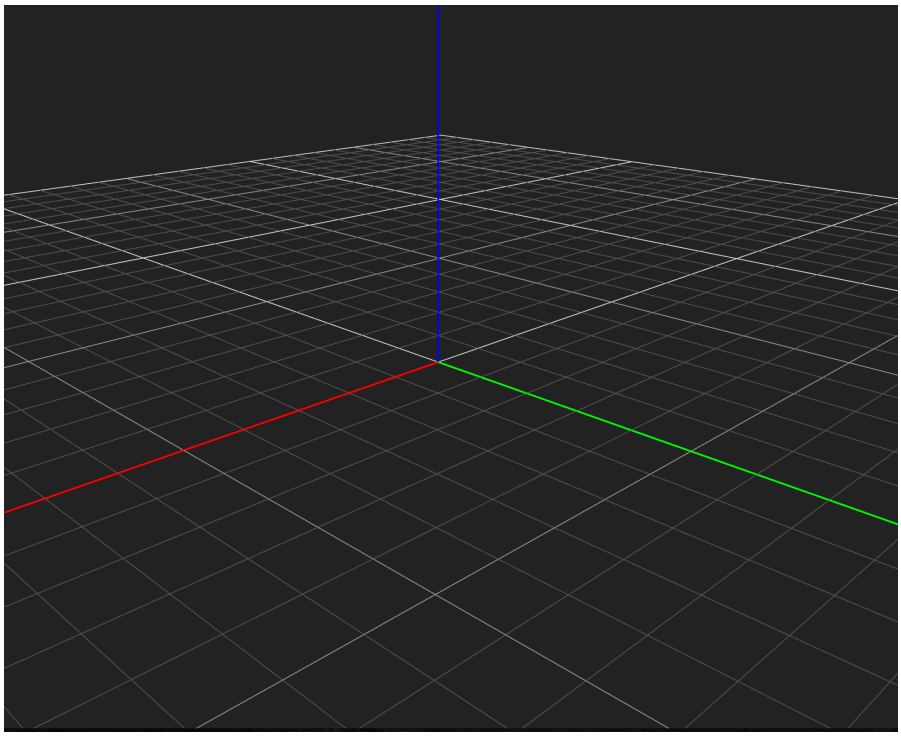


Figure 28: An empty scene showing the coordinate system.

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 **??**. *PipelineSteps* are independent from the *FabricationMethod* and thus, can be shared between different *FabricationMethods*. platener provides three *FabricationMethods*. The following paragraphs present each method briefly.

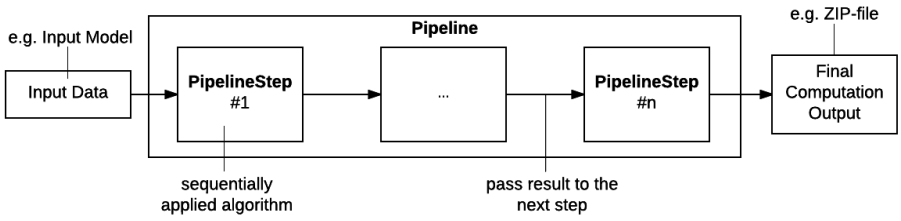


Figure 29: A *Pipeline* is composed from *PipelineSteps*

PLATE METHOD This *FabricationMethod* does hull and surface reconstructions of the input model. We produce a set of plates which are connected via finger joints to approximate the original 3D model. Therefore, we first detect the outlines of all surfaces in the model. Then, we search for existing plates based on that surfaces. For example two parallel surfaces form an *inherent* plate, when they are about three to five millimeters apart. After we find all *inherent* plates, we construct plates from the remaining surfaces by *extruding* the planar shapes. Then, we connect the plates with finger joints where they touch each other. Figure **??** depicts the important *PipelineSteps* of the *Plate Method* and their visualizations.

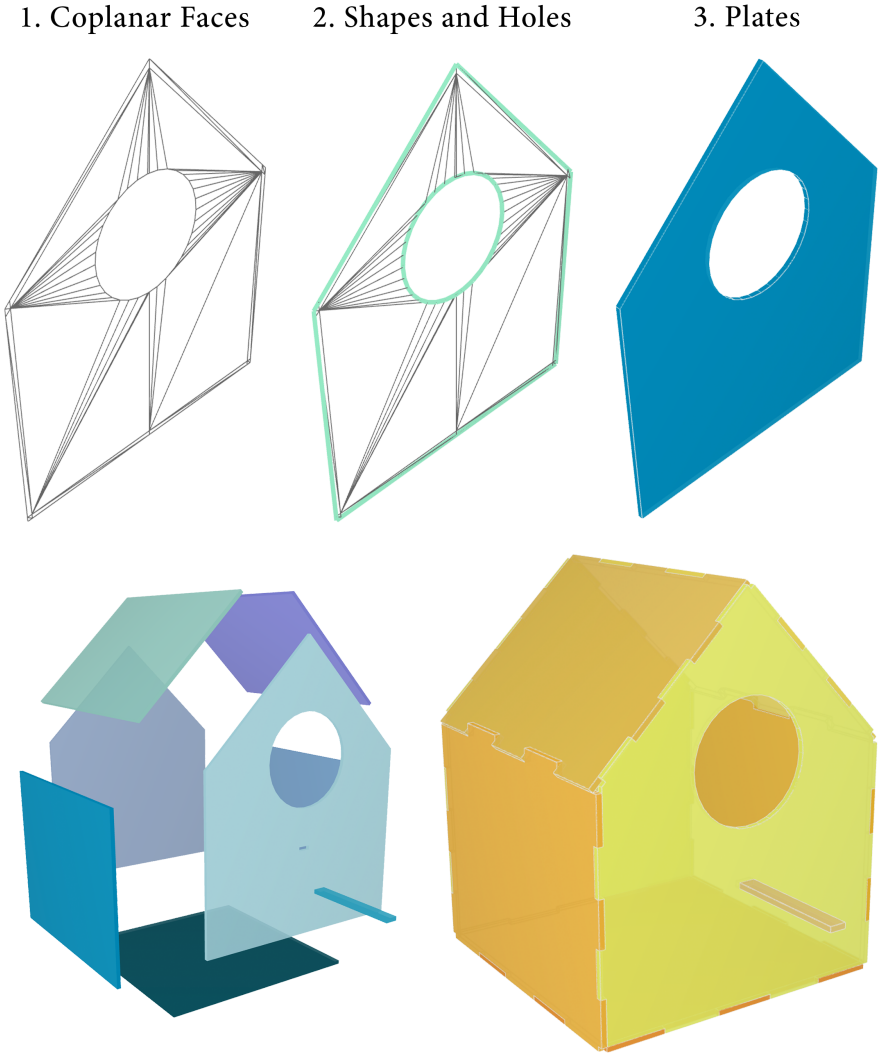


Figure 30: An overview of computation steps that are applied to the model, so that it can be built from plates.

ref to chapters which explain the algorithms to that method in detail

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 **??**. This preserves the look and feel of the model. We slice the model into equally thick layers which form the plates. The plates are then connected via shafts to simplify the assembly process.

ref to chapters which explain the algorithms to that method in detail

Figure 31: 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 **??** 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. In Chapter **??** ref the chapter! we give details about the *Classifier Method*.

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 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.

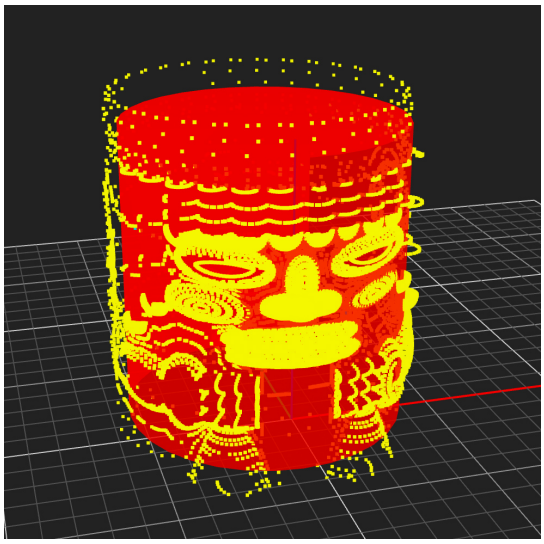


Figure 32: A classified cylinder in a model with textures. The yellow points, show the outline of the actual model.

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 **??** illustrates how snapshots are obtained between the *PipelineSteps*.

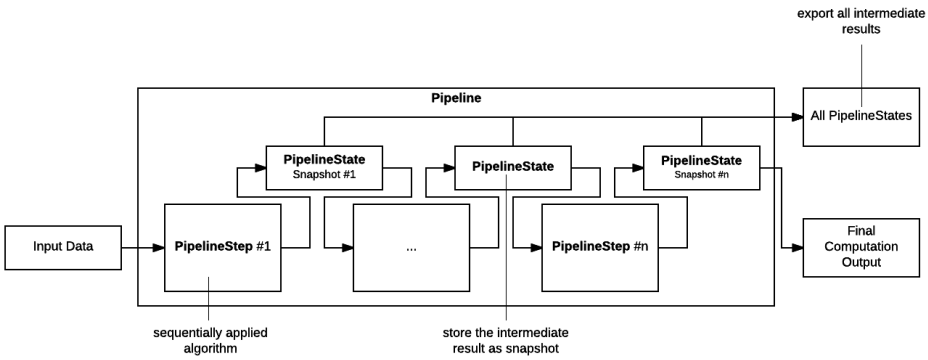
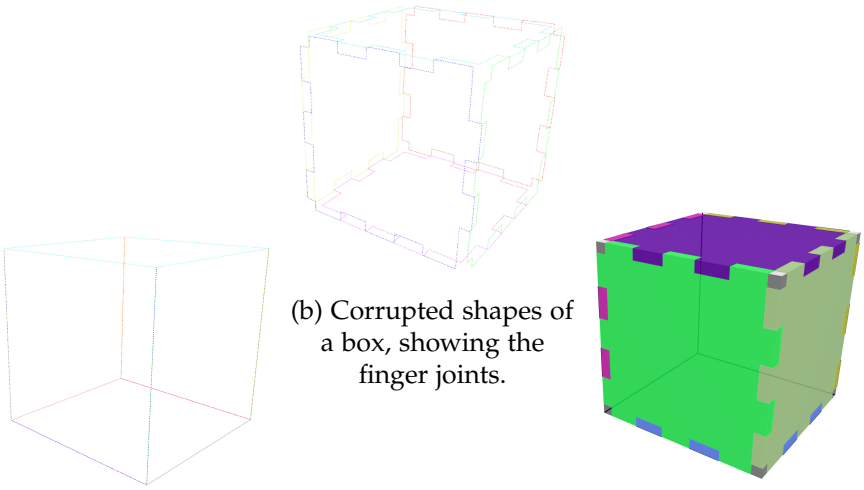


Figure 33: The pipeline preserves deep copies of intermediate computation results.

The encapsulated data in the state can be hierarchical and complex. We provide a fully cloneable data structure. That means, any nested data structure in the *PipelineState* implements a cloneable interface and supports copying as well. Thus, the clone of an instance of *PipelineState* is a deep copy.

A deep copy of a data structure is a clone 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 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 **??** shows how the surface reconstruction step would be effected by the creation of finger joints, when *PipelineSteps* are not deep clones. Figure **??** 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 **??**. When the edges data structure is not copied before adding the finger joints, the edge visualization will also show the finger joints (Figure **??**).



(a) The shapes of a box.

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

(c) A box with finger joints.

Figure 34: Modifications to shallow copies of the data corrupt the visualizations.

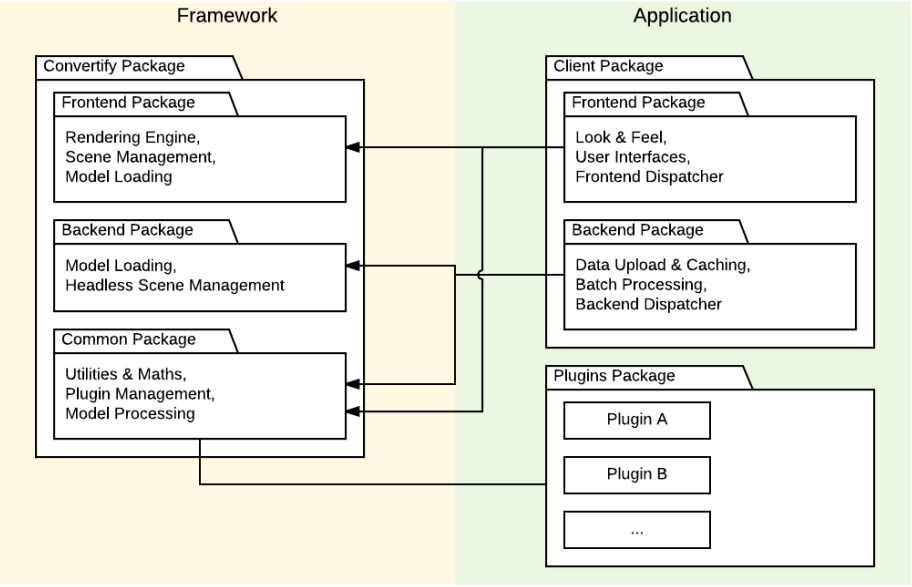


Figure 35: Main Packages of the Platener Architecture

4.3.3 *Code Packages of platener*

platener is divided into the code packages convertify, plugins and client. Figure **??** shows the detailed package diagram. 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 **??**.

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 **??**.

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 **??**. The backend application is a command 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 **??**. 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. Also, we implement *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 [**?**]. 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* [**?**]. 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 [**?**]. 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 **??** 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 again, when the Component's data changes. This causes React to "throw away the entire UI and re-render it from scratch"[11] [**?**].

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

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

```
# 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.

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" [**?**]. 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 [**?**]. Listing **??** 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 **??** 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.

When the state is changed by a reducer all React Components that use data from the changed portion of the state are 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 [**?**].

```
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.

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 **??** 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 dictionary. We receive detailed summaries and logs for each conversion. A walkthrough of our CLI is given in Section **??**.

A command line interface is an application solely consisting of textual input and output. A CLI is typically more flexible and powerful

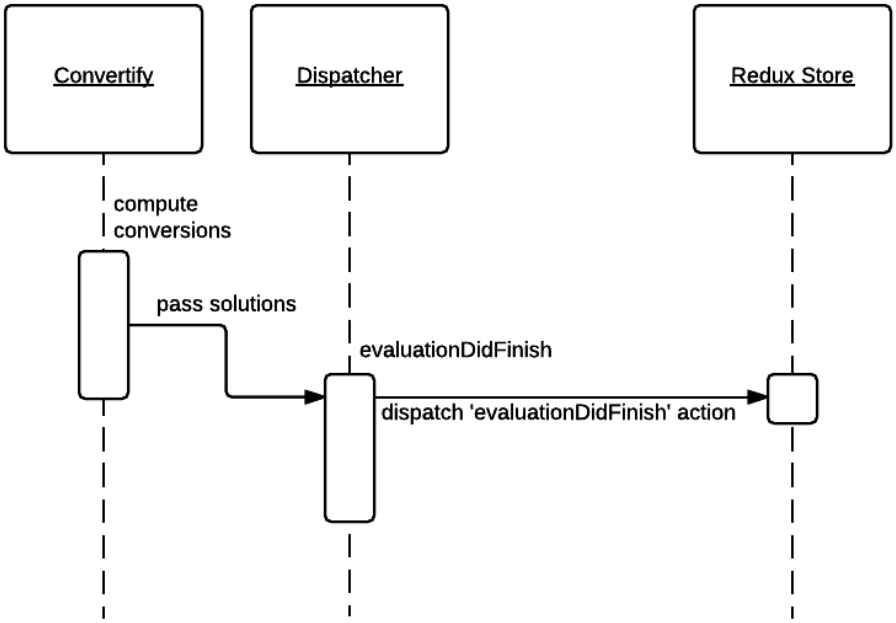


Figure 36: The *Dispatcher* connects convertify and the Redux store.

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 [**?**].

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. Therefore, 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 **??** shows the variety of available reports.

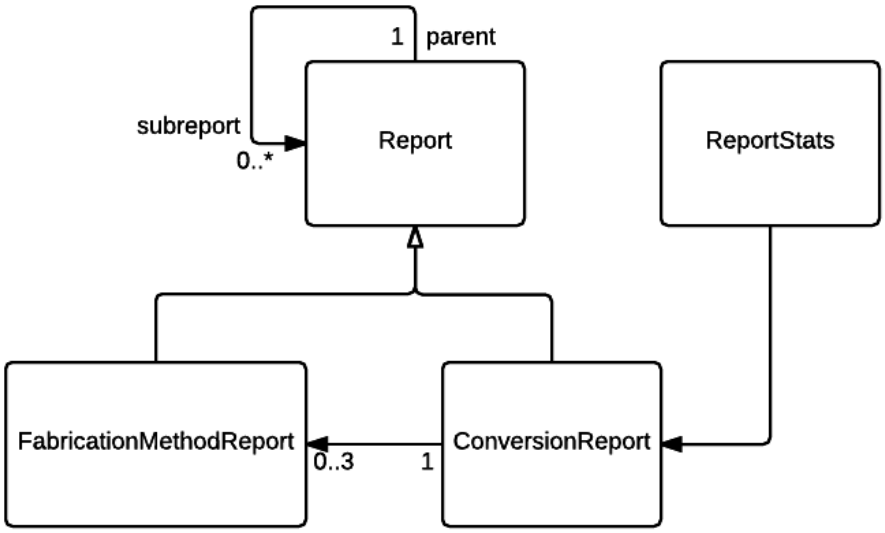


Figure 37: The CLI logs reports for each conversion.

5

PROCESSING PIPELINE / SVEN OR DIMITRI

5.1 EINZELNE PIPELINE STEPS KURZ ERKLÄRT

5.2 BEGRIFFE ERKLÄRT: EDGELOOP, SHAPE, PLATE ETC.

5.3 MODEL LOADING AND STORAGE

6

APPROXIMATION / DIMITRI

TODO: Kurzer Zusammenfassender Absatz über das Kapitel

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 keeping the differences as low as possible.

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. Therefore 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 **??** is simplified with an unusually high welding distance of 10mm in Figure **??**. 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)

Our mesh simplification tackles three issues:

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 that are not functional but created by 3D-modeling software because of aesthetic reasons. The vertex welding approach gives us the needed shape abstraction to extract tiny details and minor curvatures we do not want to reproduce.

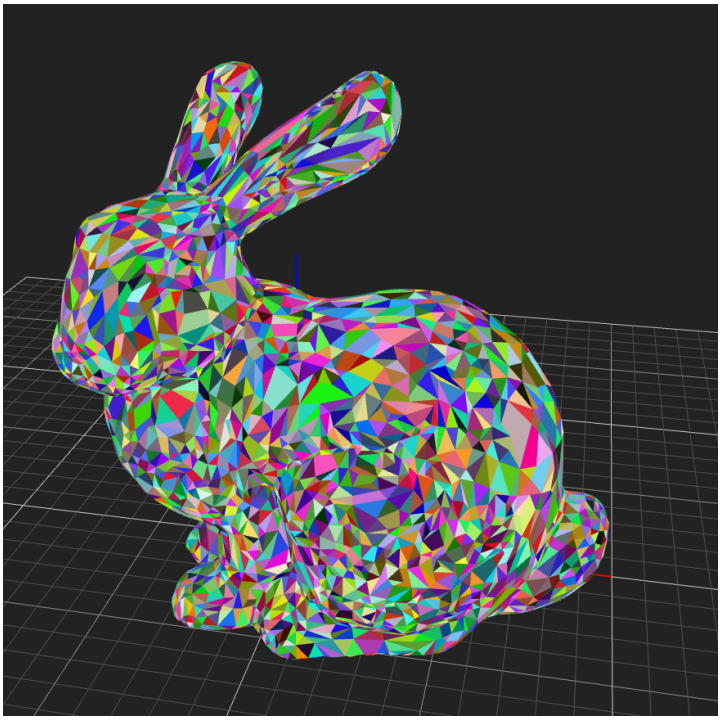


Figure 38: Stanford Bunny with 8662 faces

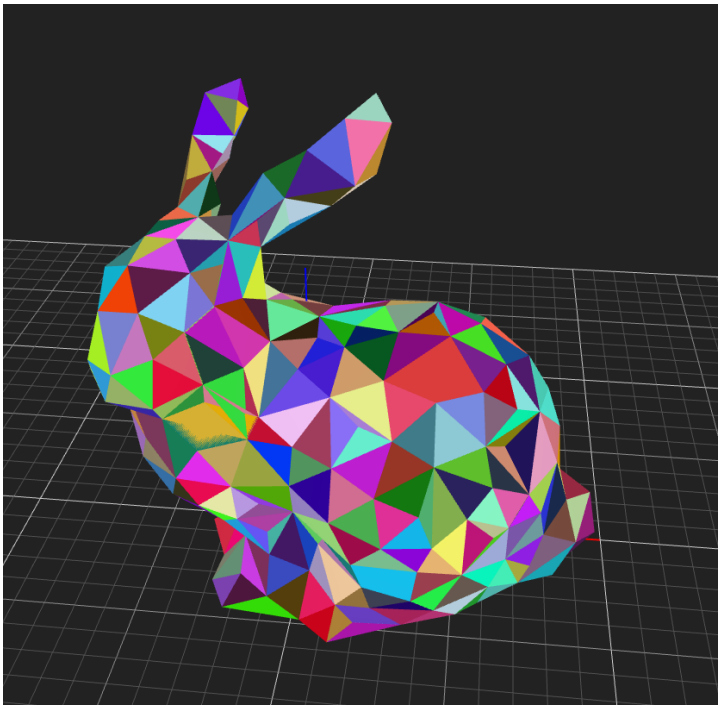


Figure 39: Simplified Stanford Bunny with 616 faces (welding distance: 10mm)

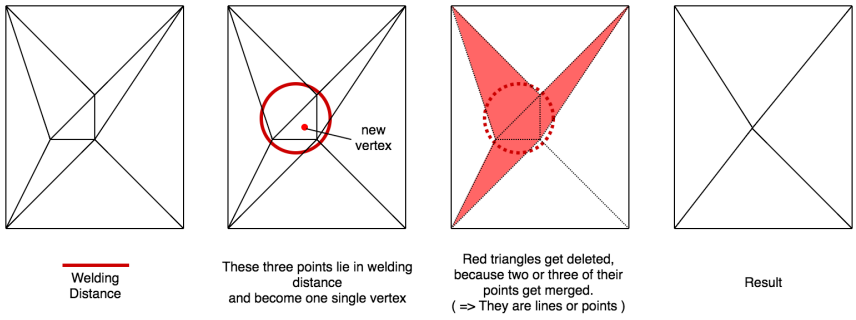


Figure 40: Vertex Welding

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.

6.1.1 *Vertex Welding*

Vertex Welding merges vertices and works as shown in Figure **??**:

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

For each cluster of vertices that get 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 in the same merge cluster (A and B) 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 is a single point (VVV). In these cases the triangle gets deleted. The initial area of these deleted triangles is now covered by its 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 **??** 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.

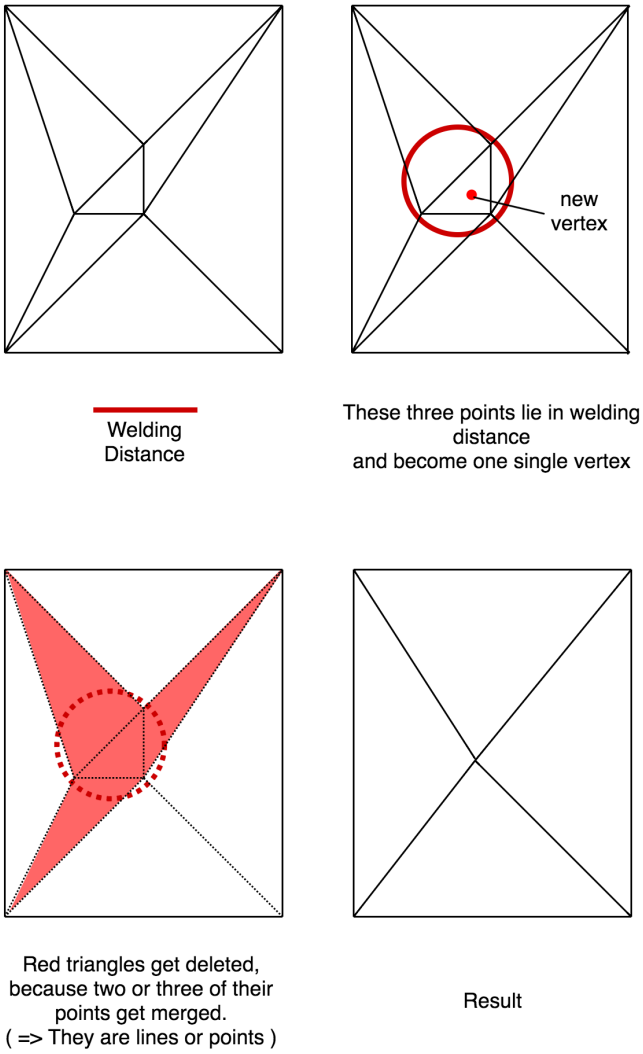


Figure 41: Vertex Welding

6.1.1.1 *Usage*

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 (for instance while using very small welding distances).

VertexWelding provides 5 public functions:

ist 'public' das passende Wort? - javascript ist alles 'public'

TODO: KLein-Großschreibung

PREPROCESSMODEL
: preprocesses the given meshlib model.

PREPROCESSVERTICES
: preprocesses the given array of vertices. A Vertex may be any object containing x, y and z values.

PREPROCESSVERTEX
: preprocesses the given vertex which may be any object containing x, y and z values.

GETCORRESPONDINGVERTEX
: returns a new vertex based on the given one. This function is called for replacing each point of your object.

REPLACEVERTICESANDDELETENONTRIANGULARFACES
: handles the welding process for the given meshlib model: It replaces all vertices and deletes triangles that are not needed any more.

So if you want to weld a set of vertices, you can call *preprocessVertices*, iterate over all points and replace them with the vertex from *getCorrespondingVertex* or just replace them without preprocessing.

6.1.1.2 *Implementation*

TODO: reread and rewrite start

VertexWelding builds a list of weighted vertices during preprocessing. Each new vertex is either merged with an existing one or added to the list. A *WeightedVertex* saves the number of represented points to merge new vertices correspondingly as shown in Figure **??**. Then 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.

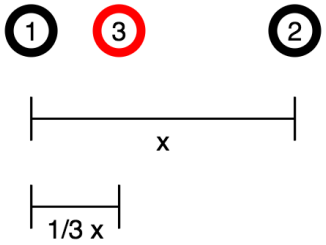


Figure 42: Resulting point (point 3) of welding a WeightedVertex with weight=2 (point 1) and point 2

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.

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* is instanced with weight 1. When a point is added, weight gets increased and the new coordinates are computed:

$$V_{weighted} = V_{weighted} + \frac{V_{new} - V_{weighted}}{w}$$

with
$V_{weighted}$:= *WeightedVertex*
V_{new} := the new vertex which is added
w := the weight of the *WeightedVertex*

TODO: reread und rewrite end

VERTEX REPLACEMENT The actual vertex replacement is done manually for each point with *getCorrespondingVertex*. 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 three points get replaced and a face get 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 StackedPlatesMethod (richtiger Name einzusetzen)

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 **??**. With a welding distance of 3mm the result is a smooth surface while the original model shown in Figure **??** 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 you try to outline the text with a smaller welding distance there will always be some kind of artifacts: missing or degenerated letters like in Figure **??**.

In order to be able to represent such texts with stacked plates *Simplification* is not run in StackedPlatesMethod (richtiger Name einzusetzen)

letzten Satz weglassen? steht ja schon vor dieser section...

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.

Obviously it is much easier to reproduce two connected plates without a beveled edge in InherentPlates (richtiger Name einzusetzen).

Figure 43: Original Makerbot letters

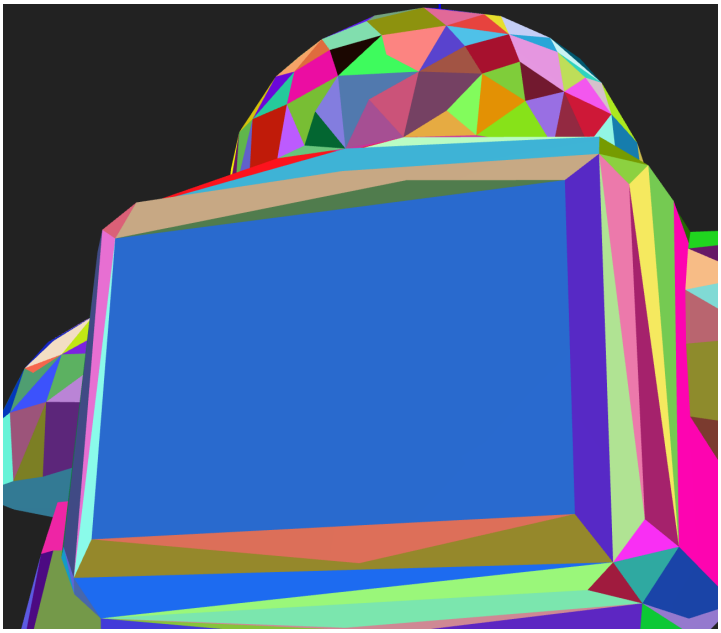


Figure 44: Simplified Makerbot with welding distance 3mm - letters completely removed

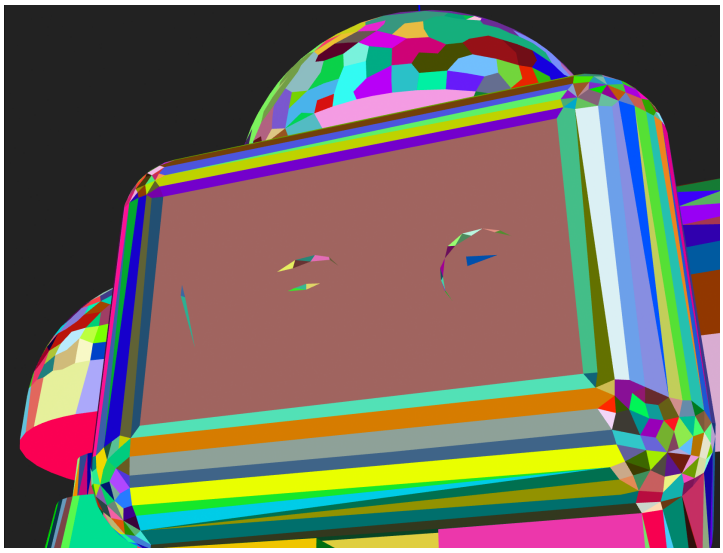


Figure 45: Simplified Makerbot with welding distance 0.8mm - artifacts

Since these curvatures are just for aesthetic reasons we revert the rounding without any loss of functionality.

Figure **??** 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 **??** there is text on top of the actual surface and in Figure **??** there is text pushed into the surface while both text is removed in the resulting object.

statt an "extruded details" lieber an makerbot von vorher erklären?

6.1.2.3 *Processing Time Improvement*

TODO: processing time improvement

TODO: Zahlen Runtime Vergleich: mit und ohne Simplification

6.1.2.4 *Process*

The pipeline step clones the model due to our immutable state approach so the original model can be accessed at any point in time.

immutable state approach sollte in Kap. Architecture erklärt sein

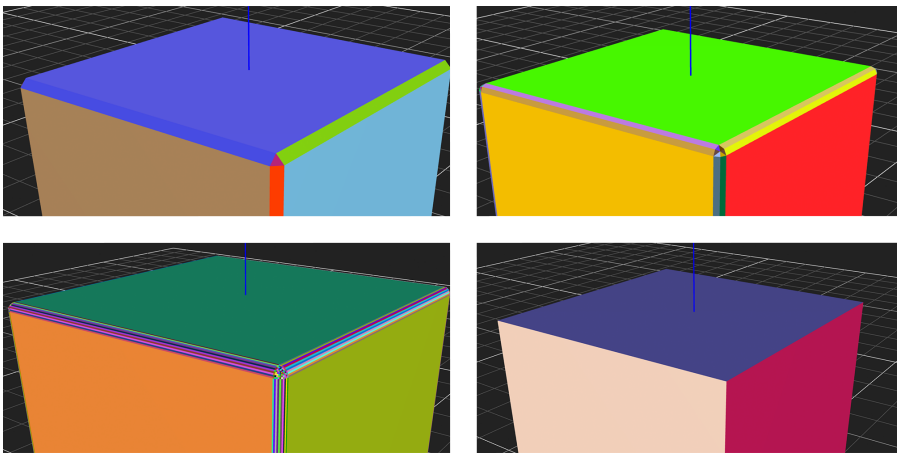


Figure 46: Beveled cube with 1, 2, 10 subdivisions and their resulting cube without beveled edges

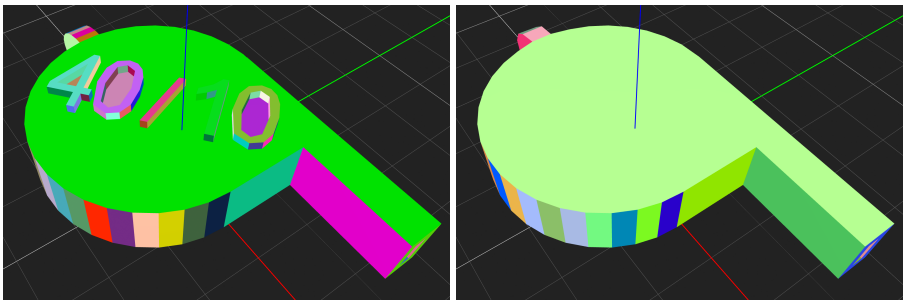


Figure 47: Extruded details

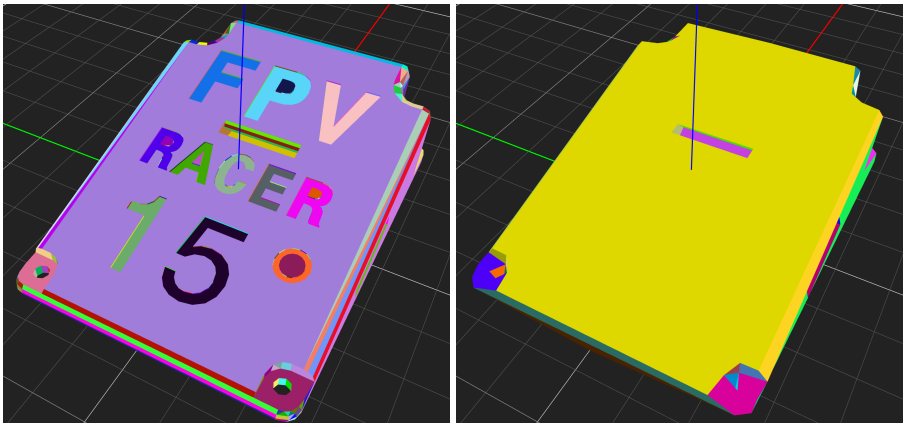


Figure 48: Pushed in details

Figure 49: Vertex Welding UI Element

WELDING DISTANCE TODO: Kompletter Abschnitt "Welding Distance" neu schreiben, da leicht obsolet

TODO: calculation of welding distance

The user can modify the welding distance in the UI element shown in Figure **??**. This is necessary to support extreme 3D models and adapt to their specialties. For instance both tiny models that span only a few millimeters and huges ones where a stronger simplification would be beneficial.

TODO: change ui text

The value gets updated in the global config file which will be used in the next recomputation. The default is 2mm which works perfect for most 3D models and removes all beveled edges created by common editors like Blender[1]. Due to the nature of beveled edges all methods only modify small areas around corners which results in dense vertices that get merged by our vertex welding.

TODO: no config??

TODO: Zahlen - 2mm perfekt für x von y modellen

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.

1 https://www.blender.org

6.1.3 *Reusage for redundant information elimination*

The algorithm can be reused in later processing steps whenever we want to ensure that a vertex is not split into multiple ones that lie close together.

For instance, this may occur in CurvedShapes (richtige Bezeichnung einzusetzen) where *VertexWelder* is used to eliminate these point cluster. In general whenever a new form or object is created (as well in 3D as in 2D) welding can be applied to ensure unambiguous points. Otherwise these vertices may be scattered due to rounding differences during rotation or other transformations: If you have one vertex in two different objects and the objects get transformed in different ways the resulting vertex of each object may slightly differ because of floating inaccuracies.

TODO: letzten Satz besser schreiben

TODO: Überschrift nächste subsection

6.1.4 *passende Überschrift zu: Alternative Methoden und warum diese Methode gewählt, vllt sogar: future work*

TODO: reread und rewrite this subsection. Der Text muss irgendwie 'wissenschaftlicher' werden: Quellen?? -> hab ich nicht wirklich welche...

Different approaches for welding are sorting into an octree and hashing.

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

This has the same time complexity of $O(n^2)$ as the current implementation. However it 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 bucket system does not bring any advantages.

HASHING Hashing is usually done if only identical points are merged instead of vertices that lie close to each other. It is similar to the Octree approach and sorts vertices into a hash list.

BINARY SPACE PARTITION TODO: binary space partition

6.2 POINT ON LINE REMOVAL

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 figure einzusetzen: Point 2 lies on the line between point 1 and 3 and gets deleted.

TODO: figure

Shapes may represent only a line or a point instead of an 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 by the global config since minor deviations from the line are not relevant.

A simplified pseudo code version of the actual removal algorithm is shown in listing **??**: *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 7: 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 figure einzusetzen: Point 1 gets deleted in case point 2 lies within threshold distance to the line between point 0 and 1. But the actual point that should be deleted is obviously point 2 since it lies directly on the line between point 1 and 3.

Therefore it is needed to check if a point lies in between the points spanning the line and not outside. In figure figure einzusetzen point 1 lies outside of the line between point 0 and 2. With this additional check the resulting triangle consist of the points 0, 1 and 3 as obviously desired.

TODO: figure

6.3 REMOVE CONTAINED PLATES - IN LUKAS KAPITEL

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* [**?**] 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 with 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 only necessary in a minimal fraction. Therefore the comparison is fast enough and the amount of comparisons must not be reduced with a different iteration algorithm.

6.3.1 *Plate completely inside another*

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.

6.3.2 *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.

6.4 HOLE DETECTION - IN PLATES KAPITEL

7

CLASSIFIERS

7.1 PRIMITIVES

7.1.1 *Prism - Dimitri*

Diese subsection wird im Classifier-Kapitel eingefügt

TODO: Herangehensweise erklären, etc.

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 (usually faces or shapes).

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 hat Sven flood fill erklärt?.

7.1.1.1 *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 surounding 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 whose (whose ist falsch, was muss da hin?) calculation is explained later in this text.

$$\vec{n} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

TODO: buckets sind minimal größer als threshold...

All normals lie on the surface of the unit ball (Einheitskugel, wie Einheitskreis, richtiger Begriff?) because of their normalisation. Considering opposite normals as equal (planes with $(1,0,0)$ and $(-1,0,0)$

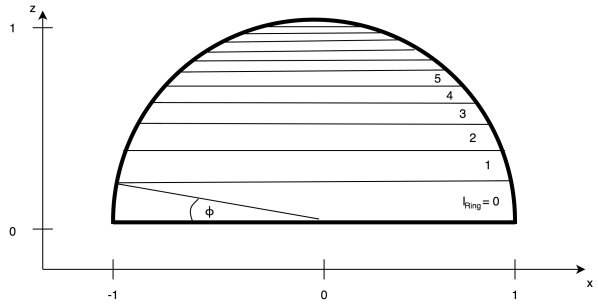


Figure 50: rings$_F$*romSide*

normals are parallel), all normals are transformed into a hemisphere as shown in Listing **??**: A normal is negated in case its z value is negative.

```
_normalInHemisphere: (normal) ->
    if normal.z < 0
        return normal.clone().negate()
    else
        return normal
```

Listing 8: 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 **??** and Figure **??**. 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 demonstraded in Figure **??**: the angle between $\vec{n}$ and $(x, y, 0)$:

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

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

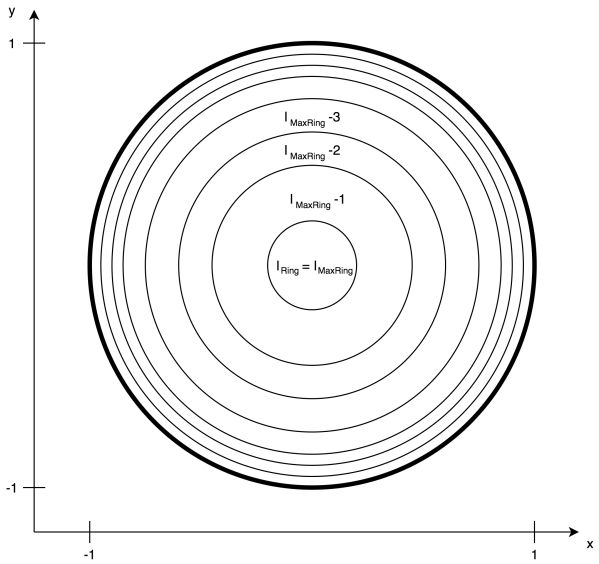


Figure 51: rings$_{F}$*romTop*

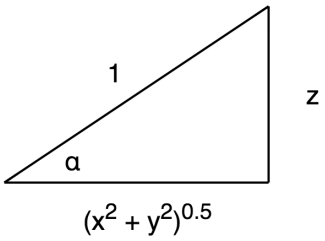


Figure 52: ringAngle

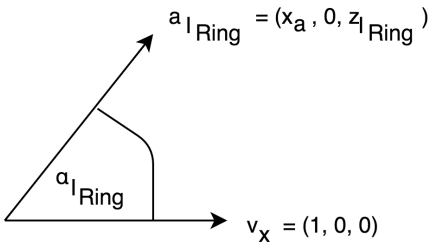


Figure 53: middleZ

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

The maximum angle is 90° (normal $= (0, 0, 1)$). 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 **??** 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 $z_{I_{Ring}}$.

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

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

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

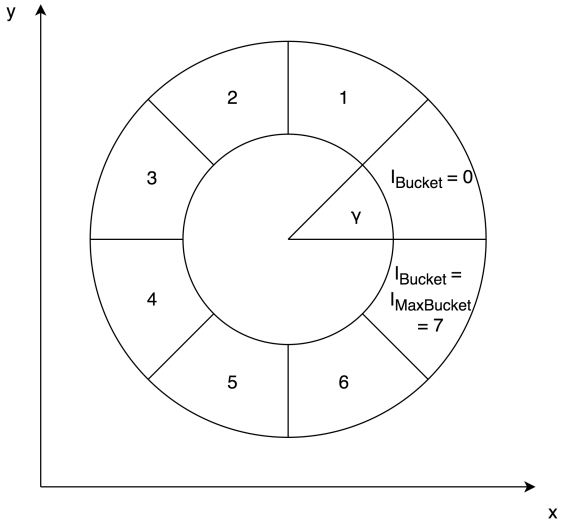


Figure 54: buckets

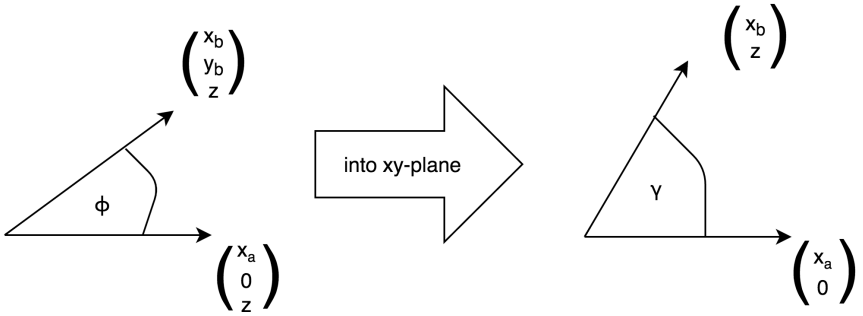


Figure 55: projection

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

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

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

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

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

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 $\vec{b}'_{I_{Ring}}$:

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

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:

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

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°:

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

7.1.1.2 *Implementation*

TODO: Klassendiagramm

The *PrismClassifier* works on *shapes* from the *ClassifierGraph* and should also take account of *planes* in future development. Therefore it depends on the *PlaneClassifier*. (sind dependencies vorher erklärt? sollte so sein) It returns a set of *prisms* and uses *PrismShapeNormals-Buckets* to check shapes for parallelism.

```
1  run: ->
2      return new Promise( (resolve) =>
3          buckets = new PrismShapeNormalsBuckets(_bucketFactor
   ↪  * _angleThreshold)
4
5          @_putIntoBuckets @shapes, buckets
6
7          prisms = @_detectPrisms @shapes, buckets
8
9          log.debug("PrismClassifier found " + prisms.length +
   ↪  " prisms")
10
11         resolve(prisms)
12     )
```

Listing 9: run method of the PrismClassifier

PRISMCLASSIFIER Listing **??** shows the implementation of the *Prism-Classifier* run method:

At first the normal buckets are initialised with a slightly greater threshold than the actual angle as described in section *Bucket System*. These values are stored in the global config with *_angleThreshold* = 0.15 (= 8.6°, previously labeled ϕ) and *_bucketFactor* = 1.1 so shapes with an angle of 8.6° will be considered parallel.

Since $\gamma_1 = 0.153886$ with $\phi = 0.15$ the *_bucketFactor* should be at least 1.026. We choose a value of 1.1 to support later changes of the threshold as described in section **??** Future Work without having to change the *_bucketFactor*. With this value all thresholds $\leq 0.832 = 48°$ are covered.

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.

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

PRISMSHAPENORMALSBUCKETS *PrismShapeNormalsBuckets* is based on the specification in section **??** 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 ball got transformed into a hemisphere. Therefore the buckets 180° around have to be taken into account for parallelism checks. Let's 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 **??** 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 handlet yet.

7.1.1.3 *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.

8

PLATES

8.1 OVERVIEW OF APPROACHES FOR FINDING PLATES

There are multiple approaches for finding plates contained in a 3D-mesh. The first, called inherent plates, requires the plates to be modeled in the mesh with both a top and a bottom side (see Figure **??**). The second approach, extruded plates, uses the mesh surface to extrude plates into the object (Figure **??**). While this method works on more meshes then the inherent approach, it can produce doubled plates if they are modeled in the mesh. The third approach is to stack plates, creating a filled approximation of the mesh (Figure **??**).

8.2 PREREQUISITES FOR FINDING PLATES

Before finding plates, the model's faces have to be grouped into planar surfaces (Figure **??**). This is done by checking the angle between adjacent faces. Afterwards, the resulting shapes' edge loops are generated and the contour is differentiated from the holes.

8.2.1 *Coplanar Faces*

The algorithm for finding coplanar faces requires the models to be stored as a face-vertex mesh. This is calculated by Meshlib, the li-

REPLACE WITH BETTER IMAGE

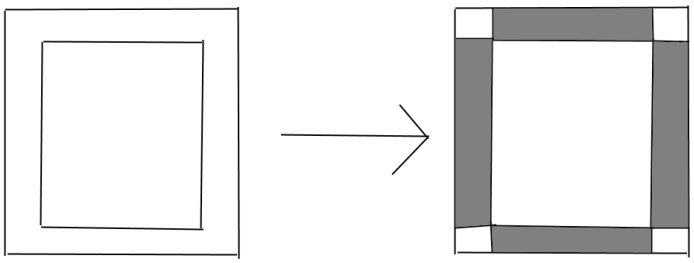


Figure 56: Finding inherent plates.

REPLACE WITH BETTER IMAGE

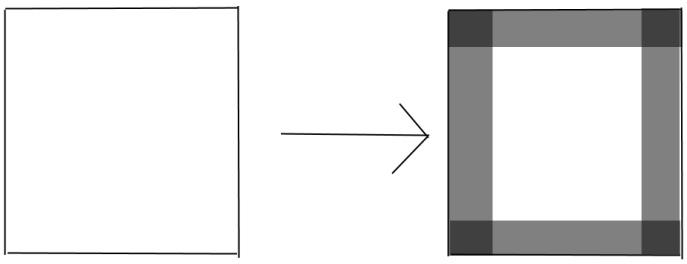


Figure 57: Finding extruded plates.

REPLACE WITH BETTER IMAGE

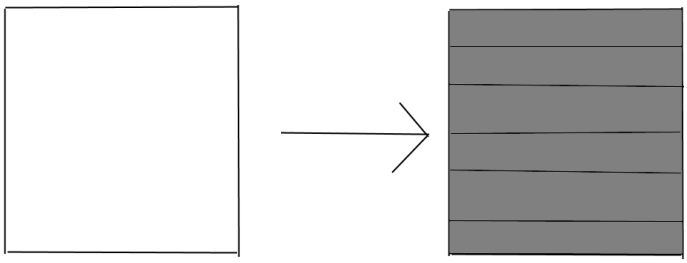


Figure 58: Finding stacked plates.

REPLACE WITH BETTER IMAGE

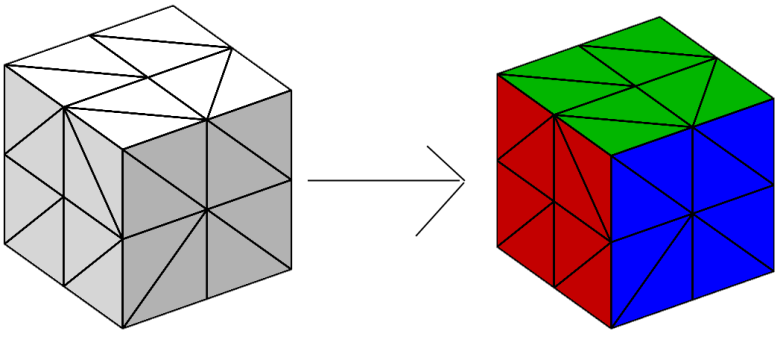


Figure 59: Finding inherent surfaces.

brary used for importing models. In a face-vertex mesh, faces only store their vertices' indices, with the vertices being stored in a different lookup table. This allows for easier adjacency checks - only the vertex indices have to be compared. Additionally, the face normals are stored in an array, in the same order as in the face list.

Now, two more lookup tables are added: One contains all edges (stored as a sorted pair of vertex indices) belonging to each face, while the second one allows looking up the faces adjacent to an edge. Both lists can be created in one single pass (see Listing **??**).

```
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 10: Simplified lookup table generation.

Afterwards, the faces are grouped. 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 (Listing **??**).

There are multiple important variables being set here: First, we have the new `faceGroup`, containing only the current face. Next, we have the index of the group, used for creating a `faceToFaceGroup` lookup table. The `outerEdgeGroup` contains all edges surrounding the group, which allows the creation of a shape containing the face group. The group's normal vector is initialized with the current face's normal vector. Lastly, the current face is marked as visited in order to avoid checking it multiple times.

After the edges have been pushed in the queue, we start processing it (see Listing **??**).

```
for faceIndex in [0...faceCount] when not faceVisited[faceIndex]
  faceGroup = [faceIndex]
  faceGroupIndex = faceGroups.length
  outerEdgeGroup = []
  groupNormal = @faceVertexMesh.getFaceNormal(faceIndex)
  @faceToFaceGroup[faceIndex] = faceGroupIndex
  faceVisited[faceIndex] = true
  # connect the adjacent faces
  edgeQueue = []
  adjacentEdges = @faceVertexMesh.getEdgesFromFace(faceIndex)
  for edge in adjacentEdges
    edgeQueue.push({ edge, faceIndex })
  @traverseAdjacentFaces(...)
```

Listing 11: Iteration over faces with creation of new face groups.

[list tco code? or just mention?] list tco code? or just mention?

This is implemented as a *tail recursion*, the helper function `tco` is based on an existing implementation of tail calls in CoffeeScript[1]. It runs `traverseAdjacentFaces` as long as `recur` is called each pass. 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 (Listing **??**). 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. After checking the faces' coplanarity, we signal that there may be more elements in the queue by calling `recur`.

Listing **??** shows what happens when testing for coplanarity. `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. 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.

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.

1 TCO in CoffeeScript, https://gist.github.com/adrusi/1905351

```
traverseAdjacentFaces: tco (...) ->
  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
    # [...]
  if not continued and edge[0] isnt edge[1]
    # the edge is an outer edge for this coplanar group
    outerEdgeGroup.push(edge)
  @recur(...)
```

Listing 12: Function repeated for each edge in queue.

8.2.2 *Shape Finder / Deus*

DO STUFF HERE

8.2.3 *Hole Detection / Dimitri*

PASTE HERE

8.3 FINDING INHERENT PLATES

In order to find inherent plates in a mesh, the first step is to find all coplanar shapes which are parallel and check if the distance between them fits one of the given plate thicknesses (Listing **??**).

8.3.1 *Checking parallelism*

While the testing for normal parallelism is done with built-in vector functions, the check if the shapes are facing apart uses a vertex of each of the surfaces and calculates the angle of the resulting vector to one of the normals. If this angle is smaller than 90°, the surfaces are indeed facing apart.

```
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 13: Check for coplanar faces.

```
candidates = []
for shape1, index1 in shapes
  for shape2, index2 in shapes when index1 < index2
    if @normalsParallelAndSurfacesFacingApart shape1,
 ↪  shape2
      thickness = @checkPlateThickness shape1, shape2
        if thickness?
          candidates.push { shape1, shape2, thickness }
return candidates
```

Listing 14: Finding plate candidates.

8.3.2 *Calculating best plate thickness*

The calculation of the optimal plate thickness is shown in Listing **??**. 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.

8.3.3 *Creating plates*

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 in Listing **??**. Now, the intersection of both shapes is calculated. This is done by using the already calculated mapping of vertices into the x-y-plane. The resulting intersection is transformed back into 3D-space using the

```
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.

rotation matrix of the base shape. With the resulting shapes, plates are created.

```
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.

The intersection between the shapes is done using the Javascript Clipperlibrary. After parsing them into the library's polygon class, they can be easily clipped, resulting in a list of intersections which can be parsed back into shapes. 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.

8.4 EXTRUDING PLATES

The extrusion of plates is a simpler approach, which is shown in Listing **??**. The selected plate thickness has to be inverted, due to the plate being extruded against the face's normal direction. The plane constant of the plates base plane is the same as the z-value of the shapes x-y-representation. After checking the shape's area, the new plate is created.

```
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.

8.5 REMOVING CONTAINED PLATES / DIMITRI

PASTE HERE

8.6 STACKING PLATES

As an alternative approach, plates can be stacked. This method does not directly use the models surfaces. However, they can be used for optimization. The main function for stacking is shown in Listing **??**. First, the model is rotated in order to optimize the stacking direction. Afterwards, the clipping planes are calculated. After clipping the models faces with these planes, the resulting edges are merged into edge loops, which are used for creating shapes and, as wall 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.

8.6.1 *Rotating the model*

In order to find a good rotation, the model's biggest surface is aligned to the x-y-plane. 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 **??**). This matrix can be used for transforming all vertices so that the chosen surface is parallel to the x-y-plane.

```
findRotation: (shapes) ->
  maxArea = 0
  rotationMatrix = new THREE.Matrix3()
  shapes.forEach((shape) ->
    area = shape.area || shape.getContour().computeArea()
    if area > maxArea
      maxArea = area
      rotationMatrix = shape.rotationMatrix
  )
  return rotationMatrix
```

Listing 19: Finding an optimal rotation.

8.6.2 *Finding the clipping planes*

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.jsplane 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-to-be, which is an approximation which works for most applications. The plane creation is shown in Listing **??**.

```
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.

8.6.3 *Clipping the model's faces*

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 **??**). I order to quickly find this plane, binary search is used, as shown in Listing **??**. Starting at the median plane (`@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 **??**). 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 doesn't 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 **??**). 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.

```coffee
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.

```coffee
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 clipping plane 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 seperately (see Listing **??**).

8.6.4 *Creating shapes*

Using the intersections stored in each plane, shapes can be created. These shapes represent the cross-sections of the model. Since this requires merging the intersection edges, the shapes finder can be reused for this step. Additionally, the Javascript Clipperlibrary is used to create 2D polygons, which are used in the next step.

8.6.5 *Adding shafts*

In order to assemble the stacked plates, we connect them with shafts (Figure **??**). Our approach for this: if two shapes which are located in adjacent planes intersect, they have to be connected by at least one

```
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.

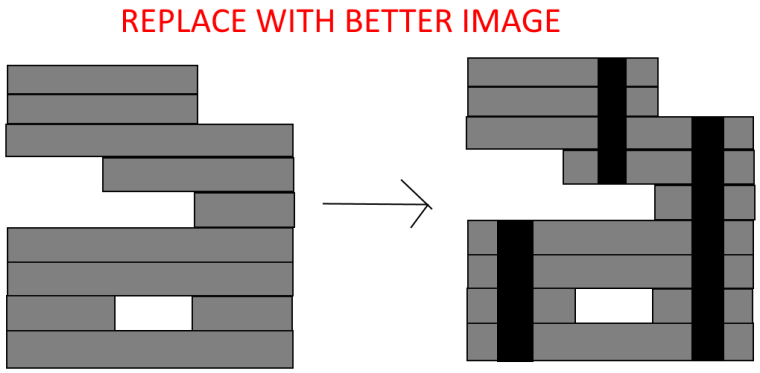


Figure 60: Connecting plates with shafts.

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.

8.6.5.1 *The shaft object*

A shaft has these properties:

- polygons - a list of the polygons which the shaft connects
- intersection - the intersection of all the polygon which the shaft connects

- finished - true if the shaft is complete and won't be extended to connect more polygons
- lastlevel - the layer of the last added polygon
- shaftData - contains the shaft's cross section's x and y dimensions

It contains functions which allow adding new polygons to the shaft, and building the shaft's cross section and contour.

8.6.5.2 *First pass*

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 **??**.

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 **??** 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 **??**). Additionally, the shaft is overlapped "backwards", up to three layers. This improves stability in the assembled model.

8.6.5.3 *Fixing unconnected plates*

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 **??**.

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, han-

dling for this polygon is completed. Otherwise, we create a new shaft at this polygon and try to expand it in both directions.

8.6.5.4 *Cleaning up shafts*

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 **??**.

8.6.6 *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 < 1
        log.error "FOUND SHAFT WITH ZERO POLYGONS"
      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.

9

ADJACENCY PLATEGRAPH / KLARA

On the basis of the plategraph we can create connectors for plates in the later step joint creation. Depending on angles and neighborhood relationships an adequate connector type can be chosen.

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 to be analysed are found in the previous step Plates. Two or more plates are adjacent to another when at least one side touches or overlaps with the other plate. In addition, the angles in between the plates are calculated.

The graph holds a list of all found plates and their neighbors.
For graph iteration we traverse all edges of the graph which also hold the neighborhood parameters such as angle and the line at which nodes intersect.

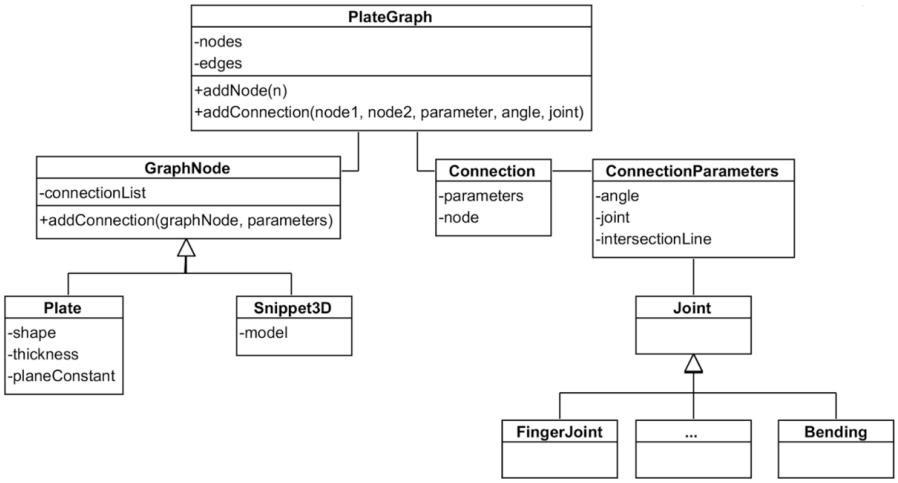


Figure 61: Class diagram of the plategraph structure. Nodes and Connections can be added to the graph when a new intersection has been found.

TODO: Plategraph beinhaltet nodes, edges ... und was machen der graph, was sind nodes und edges, dadurch kann auf diese weisen...

das ... erreicht werden...

9.1 DETECTING SPATIAL ARRANGEMENT

As a prerequisite the step Plates needs to find inherent or create extruded plates first. Afterwards the plane-plane intersections of all combinations of both sides of the plates are computed which results in up to four intersection lines. In preparation for the joint generation step **??** we truncate the inner intersection lines which would otherwise overlap with adjacent plate intersections.

9.1.1 *Finding intersections*

When two planes intersect there is an intersection line. Since we work with plates which equal two 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.
TODO: include citation
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 has to be found which lies on both planes.

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

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

TODO: cite this

On the basis of a position vector p and the direction $\vec{dir}$ an intersection line can be computed: $line = p * x + \vec{dir}$
Now that all lines are found we have to test if the lines actually go through both plates and that the plates touch.

TODO: Why should it not go through both plates?
In addition, this step retrieves the exact start and end points of the line segment that defines the intersection of both plates.

TODO: When and how?
In order to find the boundaries of the lines we calculate the intersections of the lines with all boundary edges of the plates.

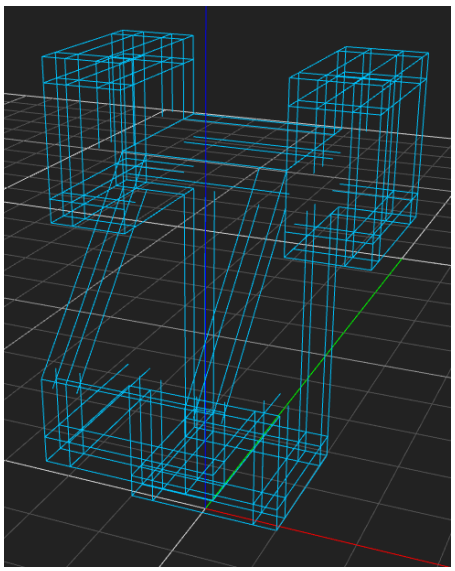


Figure 62: All intersections that were found.

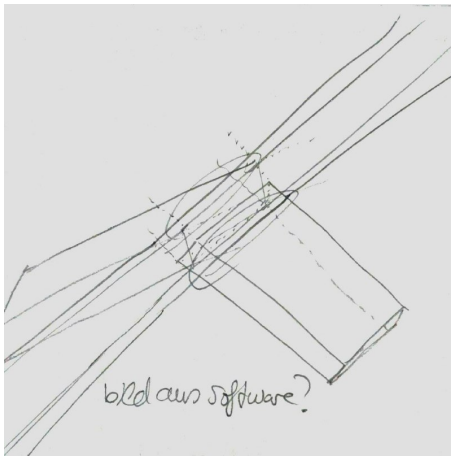


Figure 63: Single plate pair with all of its intersection lines. The marked part is the segment that we continue to work with.

9.2 PREPARING PLATES FOR CONNECTORS

9.2.1 *Volume based clipping*

Before joints can be added to the plates we have to prepare them. We cut the shapes back so that the joints do not overlap with the other plate.
As seen before, up to four intersectionlines have been calculated per intersection. Two of them lie both 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. We use those two rectangles to remove the parts of the plates where we will place fingerjoints.

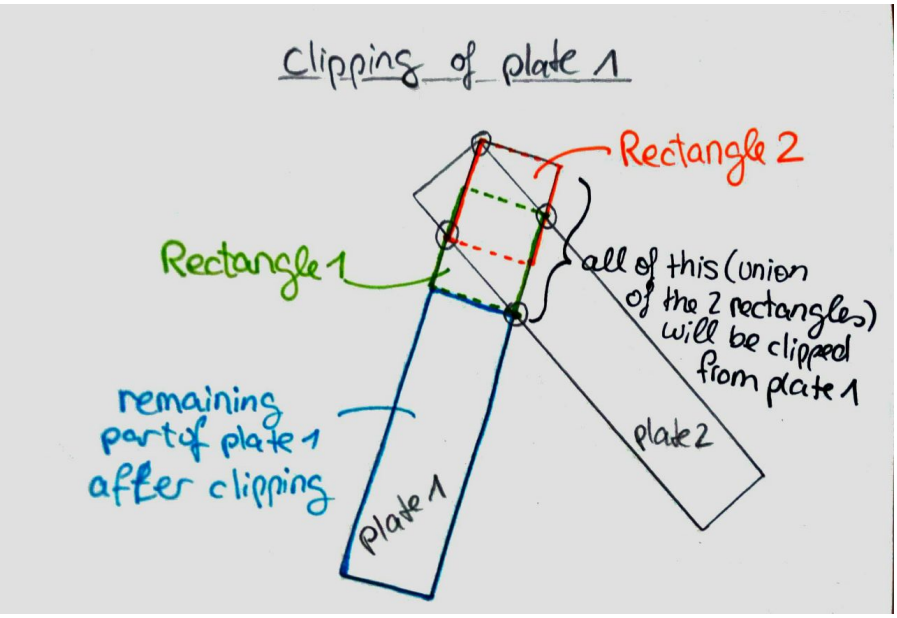


Figure 64: Side view of a single plate pair. The circles mark where there are intersection lines. The rectangles show what is clipped away from plate 1.

There are cases when not all four lines are actually a plate intersection but only a plane intersection.

In this case the infinitifly 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.

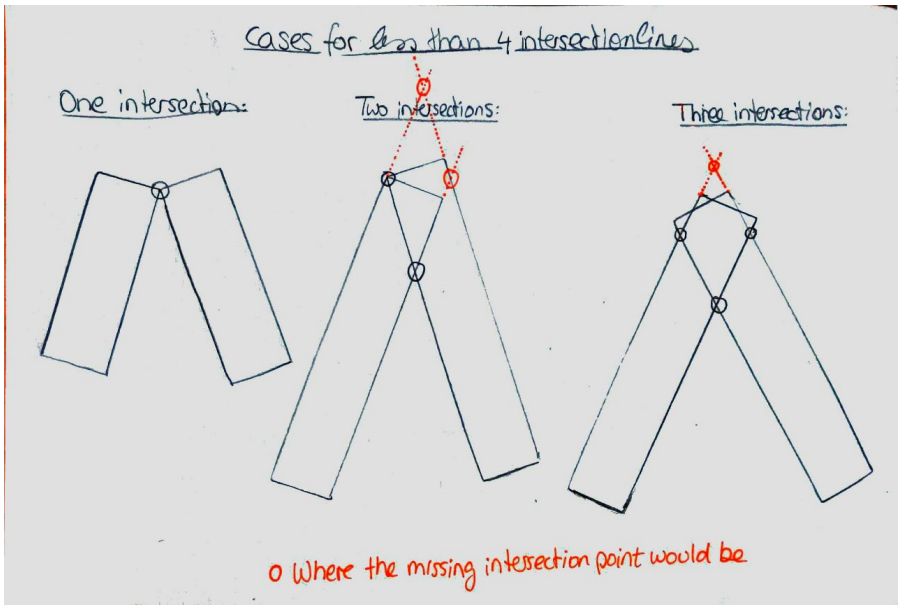


Figure 65: There can also be less than 4 actual intersections. In this case we have to calculate the projected intersectionlines as well which are only a correct plane-plane-intersection but not actually where the plates overlap

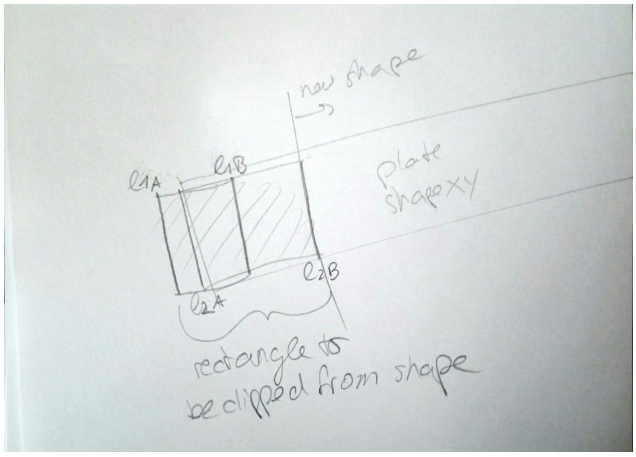


Figure 66: The shape of the plate is rotated so it lies on the xy-plane. The rectangles derived from the intersection line segments define what has to be clipped away from the shape.

9.3 ANALYZING SPATIAL ARRANGEMENT

9.3.1 *Angle calculation*

In order to generate fitting joints in the upcoming step and for grouping plates to curves (see Chapter Curves) the angles are necessary. First, we determine the angle between the according planes.

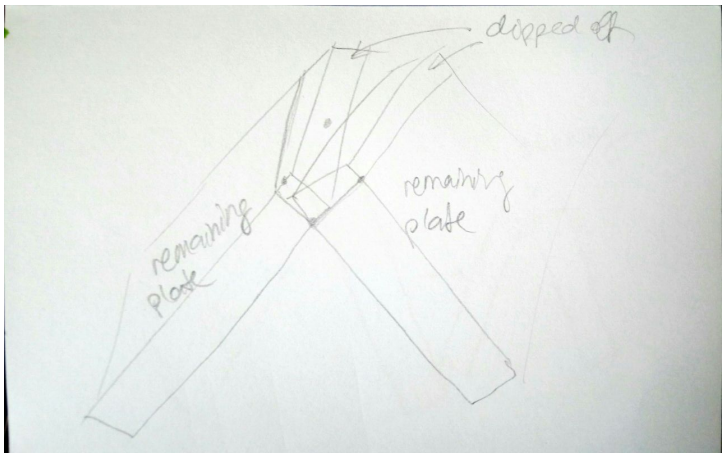


Figure 67: When the shapes have been cut in 2D they are rotated back to form a 3-dimensional plate again. This figure depicts the volume which was cut off by the algorithm leaving the remaining plates p_1 and p_2

plane normals: $\vec{u}, \vec{v}$
angle between planes: θ

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

But we are not talking about infinitely large planes instead we want the angle which is enclosed by the finitely large plates. Therefore we need to adjust the angle in some cases.

TODO: rewrite previous 2 sentences!!
Thanks to the previous clipping step we can find out when the angle has to be adjusted.
We observed that plates with an acute angle still touch when cut back and that plates with an obtuse angle do not touch after cutting them back.

Finally, we test if the plates still touch and adjust the angle accordingly.

9.3.2 *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. Where the distance is the smallest they are the two lines which determine where to place the joints.

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

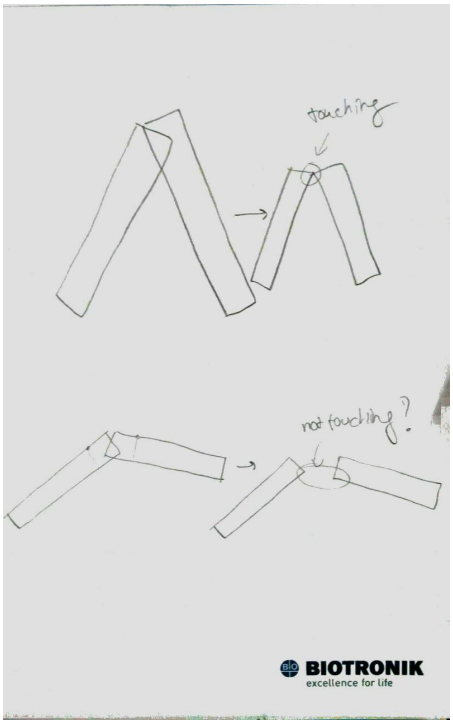


Figure 68: After the clipping process the plate pair might be not touching anymore. This means that the angle between the plates has to be larger than 90°.

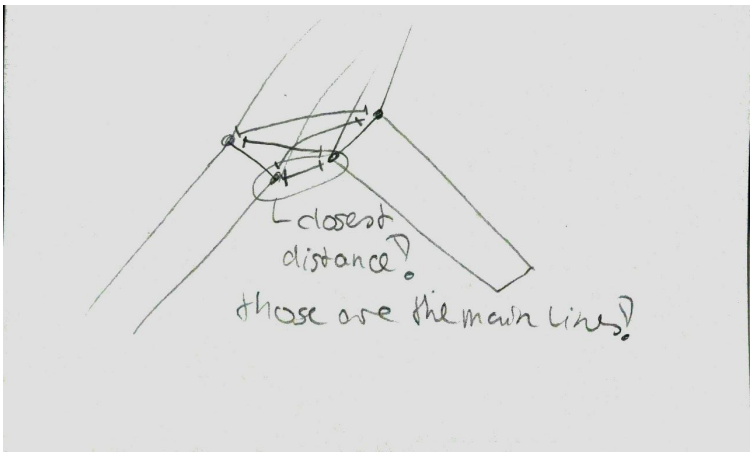


Figure 69: The distances between the new edges after the clipping process reveals which of them are the inner lines. Those lines will be used for adding joints in the next step.

9.3.3 *Truncating intersection lines*

Now that all main lines are known we need to shorten the lines so that no other lines overlap with it. If this step is missing then the following step for creating joints will run in to problems because the joints overlap each other.
If we now have a look at only the inner intersections of plates in a

model we can identify the overlaps.

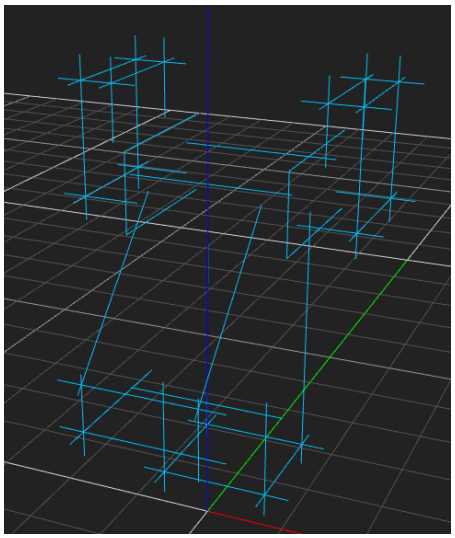


Figure 70: The inner boundaries of the plates overlap. In the next step when joints will be created they are only supposed to be on the inner part of the line. Therefore the lines have to be truncated.

Algorithm 1: Truncating lines

```
Data: lines
Result: truncated lines
initialize variables here
for line_pair in lines do
    line1 = line_pair[0]
    line2 = line_pair[1]
    point = lineLineIntersection(line1, line2)
    if isPointOnLine(point, line1) and isPointOnLine(point, line2)
     then
        // 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()
         then
            if startSegmentLine2.distance() >
             endSegmentLine2.distance() then
                return { endSegmentLine1
                endSegmentLine2 }
            else
                return { endSegmentLine1
                startSegmentLine2 }
        else
            return { startSegmentLine1
            startSegmentLine2 }
```

Finally, we found all plate adjacencies and necessary information on the connected parts. Also, we adjusted the plates. Therefore the next step explained in Chapter Joint Creation can start adding joints.

9.4 ALTERNATIVE APPROACHES

The solutions presented in this chapter may not be the only way to go. Therefore, we want to give an overview of other algorithms we tried or are aware of and why we replaced them.

9.4.1 *Possible data structures for intersection tracking*

TODO: cite Beyer

Dustin Beyer 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.

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.

9.4.2 *Approaches for finding intersections*

TODO: cite
In the master thesis on Platener 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 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 for finding intersections.

9.4.3 *Representation of the joint location*

As seen before in the section Main Lines we calculate one line per intersection and plate which determines where we will place the joints. We chose this approach because it is a straight forward way. 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.
TODO: Include image that shoes hinge twice. One where the axis is marked and one where our two main lines are marked!
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. But 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.

9.4.4 *Finding main lines by checking if they lie on a corner of the shape*

As already mentioned 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.
TODO: Image of the four lines and showing which one should always be the main line
This means all lines which intersected with any points of the shapes of the plates were ignored. The other line became main line.
Our assumption which we adopted from the Platener thesis
TODO: cite
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.

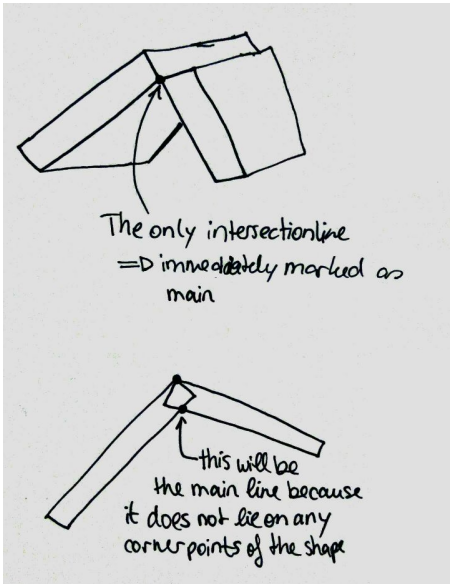


Figure 71: Our assumption was that we only need to define one main line per plate pair. If there was just one intersection at all it was obvious. And when there were more than one intersection we thought the inner intersection should be the main line.

But the problem is that plates can intersect in many different ways which do not all verify our first assumption.

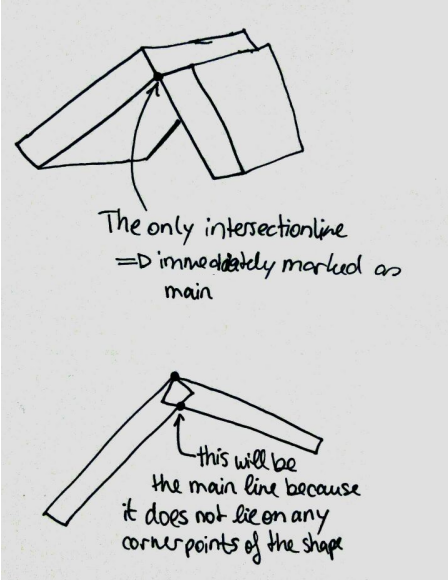


Figure 72: In this case the angle between the plates is obtuse. If we just took the inner intersection here without first clipping of the plates there would be no space for joints.

TODO: replace this with image of two planes where this doesnt work
This is why we replaced this algorithm with the one from the previous section Main Lines.

9.4.5 *Adjusting angles based on which sides of the 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:
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.

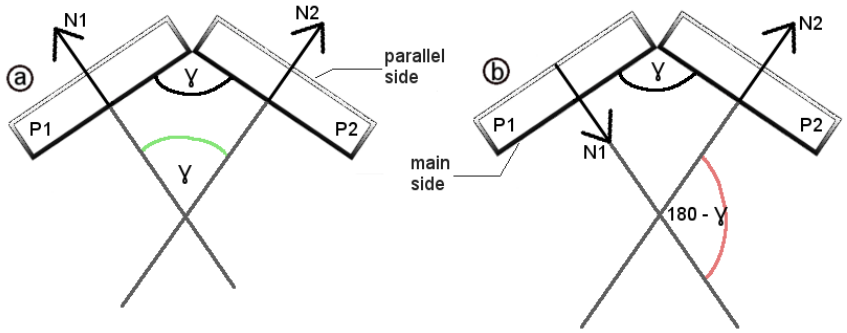


Figure 73: (a) The angle between the plates' normals correspond with the angle γ between the plates. (b) The angle between the plates' normals is in this case an adjacent angle to the requested angle γ.

Additionally, we look at the direction of the normals. The direction is positive when it is directed from the main to the parallel side and negative otherwise.
For an angle to be in need to be adjusted the following conditions need to be satisfied.

- The two plates touch with the same type of side

  AND

- Their directions are both positive OR both negative

But as mentioned before all of these calculations are based on the assumption that the previous main line calculation only yielded one line.

TODO: Include image that shows when this is not the case, with captions explaining why our current algorithm fails here
In this case the previous algorithm fails which is why we developed a new approach for finding the correct angle as already explained in the section Angle Calculation.

9.5 FUTURE WORK

- To be not forgotten: A line can be split into pieces!:
  Plates do not only touch another along one line.
- find out if model is assemblable in the end
  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 actually 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 put up before plate b but plate b also needs to be

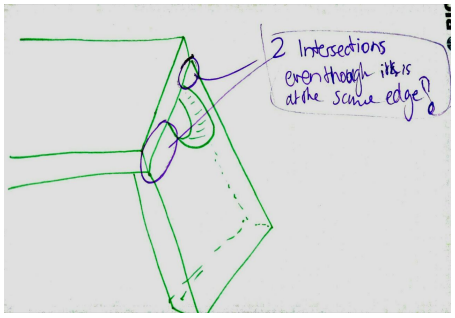


Figure 74: Plate 2 touches Plate 1 twice. This should result in two edges and intersections which should be handled seperately.

in the model before plate a. This should not happen or at least be mentioned to the user.

- rejoining broken up plates (t-connection)
  This is what Dustin also suggested and already implemented. He suggested to group two plates together again when:

  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

  It can happen that during finding inherent plates some plates are broken by another. This should result in a t-junction. For this the plategraph has to recognize those broken-up plates and reunion them.
  For this the following steps are necessary:

  1. Iterate over all nodes and check if it 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 intersections needs to be checked for parallelness and overlaps between p1 and p3 and p2 and p3.
  5. When determined that p1 and p2 are actually one plate which is parted by p3 then they need to be unioned and marked for a t-junction to be handled in the next step.

  (not sure if this is all it takes maybe a plate is broken up more than once, meaning not only p1 and p2 might be coplanar)

10

JOINT COMPUTATION / KLARA

10.1 JOINT COMPUTATION

TODO: buy green and orange (or at least two different colors) of acrylic for demo objects and pictures in the paper

A lasercutted 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 lasercutter. This is why we provide three types of joints which are added to the determined plates.

TODO: im Gesamtsystem einordnen, was war davor, was kommt danach?

10.2 BUILDING JOINTS

TODO: Make this section easier to read!

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.

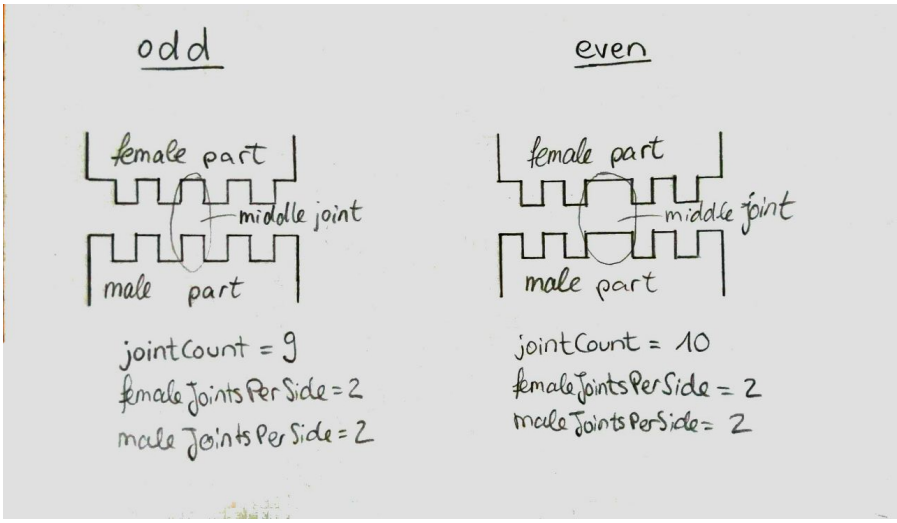


Figure 75: Distribution of joints depending on the number of joints. An even number results in a thick inner joint.

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 inbetween the joints for a female one to fit in. 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.

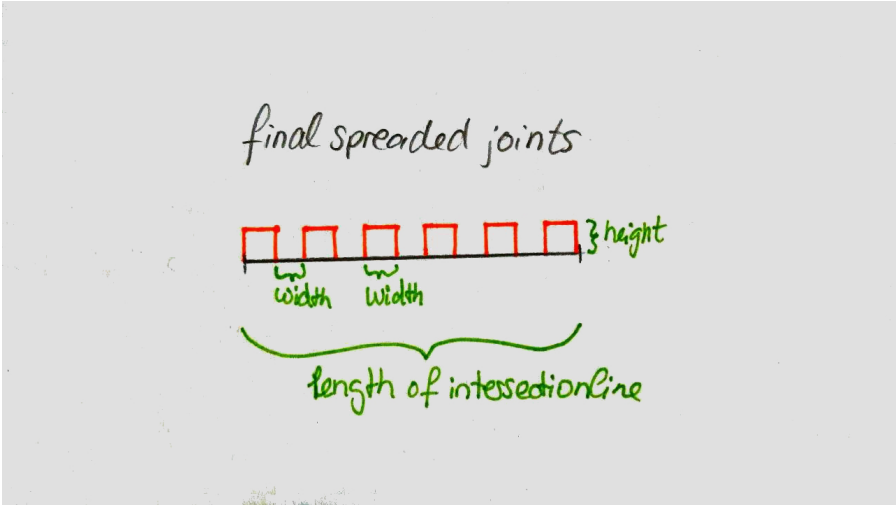


Figure 76: The joints are spreaded evenly along the line leaving space of a joint with in between for the other plates joints to fit in.

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

When creating *fingerjoints* 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.

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.

10.3 DIFFERENT FINGERJOINT TYPES

TODO: Bilder noch so anpassen, dass nur females/males zu sehen und in echt, wie sie zusammen stecken

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.

10.3.0.1 *Fingerjoint template*

Fingerjoints 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 fingerjoints do not fit into each other at all or fall apart and have to be glued. When using fingerjoints 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.
TODO: Maybe show this in a picture? Boat vs only two plates, where the angle cannot be determined without the other plates
Regarding the material fingerjoints 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 fingerjoints hold up due to the friction of one plate to the other. The fewer material the less friction.
We usually use fingerjoints for acrylic and wood.

10.3.0.2 *JimJoint template*

These joints 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. Jim McCann already used it for foam. We now use it especially for paper because this material would be too thin for creating enough friction for fingerjoints to work.

10.3.0.3 *Dovetail template*

TODO: Image of wood dovetails where this joint is derived from.

Typically a dovetail joint is used in woodwork. 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 Alternative Solution for information on how to lasercut the female part of a dovetail.
Instead, we developed a joint type very similar to the original dovetail. This can be easily cut with a lasercutter. The males and females are alot like typical fingerjoints. But the females have an additional part which will fixate the other plate from one direction.
TODO: insert image
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.

10.3.1 *adjusting fingerjoints 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. The length of the long diagonal d is the key to finding the appropriate length for the joints.

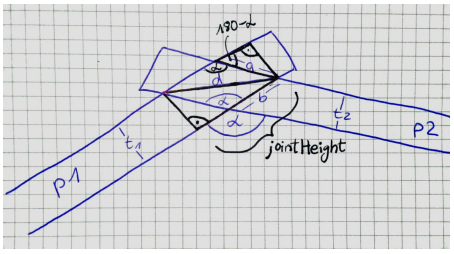


Figure 80: The parallelogram spanned by the overlap of the plates allows us to find the appropriate height for a joint.

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

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

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

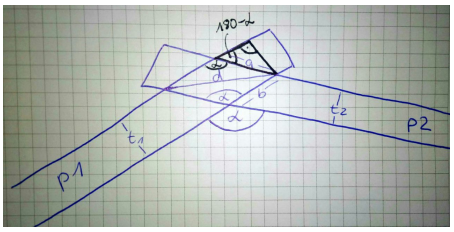


Figure 81: The diagonal d can be calculated by the sides of the parallelogram.

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

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

Finally, the triangle which helps us retrieve the height for the finger-joints h can be constructed.

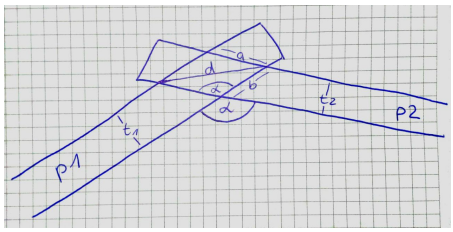


Figure 82: The diagonal d is the hypotenuse of the right triangle with the wanted height for the joints.

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

TODO: Abschließender Satz?

10.4 ALTERNATIVE SOLUTIONS / RELATED WORK

10.4.1 *more joint type*

10.4.1.1 *snap fits*

Snap fits we already tried. Hard to implement because enough space needs to be given when cutting the spring.
TODO: show image of our cube with snap fits!

10.4.1.2 *pettis joints*

other joint type: pettis joint (fingerjoint with screws)

10.4.1.3 *t-joint*

slot joinery technique when overlapping plates had to be rejoined

10.4.1.4 *real dovetails with the lasecutter*

In order to achieve different depths of cutting you can set the laser-cutter to use lower power. This is usually used for engraving which is a method to not cut through the material.

TODO: Name Lasercut like a boss here

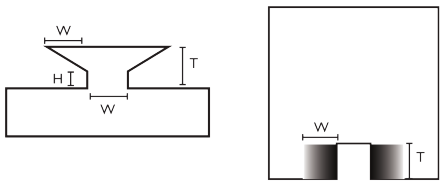


Figure 83: Instructions on how to build dovetails with the lasercutter.

By coloring a part in the svg-file in a gradient from white to black this part will be cut in an angle. The more white the less power.

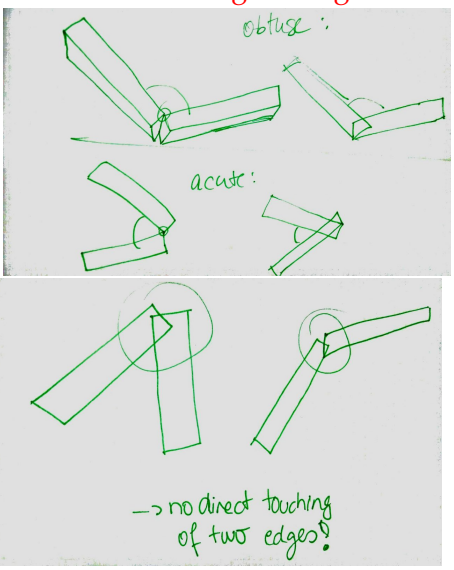
10.4.2 *Wrong fingerjoint height adjustment*

Firstly, the angle can be projected onto values between 0° and 90°. Then the cosine function returns a number between 0 and 1 for the angle. The thickness of the plate is divided by this number to achieve a height that fits the angle and the thickness of the plates.

10.4.3 *Dustins useless angle stuff*

He computed different angle types. They are quite crappy because he assumed that plates would only meet at edges. But plates can lie in another

TODO: Make images to figures and add caption



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.

10.5 FUTURE WORK

- Dustin also suggested T-joints
- joints should not stand out on the side.
  Either their width has to be adjusted to be not only the width of when the joint starts building away from the plate. but also include how far away it is growing to the sides in its full height. (Beware that the joints still have to be placed according to its inner with otherwise they are too lose (except for fingerjoints))
- when creating fingerjoints do not calc femaleJointsPerSide (unneccessary since using the stencilBox)

11

CURVES/DEUS

11.1 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 already be bent because of the material properties, acrylic if it is heated and wood with the help of living hinges.

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 are based on triangles, which are themselves 2D, our meshes are always developable.

11.2 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 so one bent plate could be cut as one piece. A Bent Plate consists of a set of plates. It has the method generateShape which develops the surface of the connected shapes. An other method it implements is generateBendLines.

11.2.1 *Function generateShape*

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

If there are more then one plate in the bent plate the generation works the following way: The shapes from all plates of the bent plate are transformed using the bend matrix (see section **??**). Then they are converted into polygons of 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. The resulting polygon is converted back to a shape. (see Listing **??**)

11.2.2 *Function generateBendLines*

In the current implementation this method generates a dashed line at the connection lines between two plates in the shape of the bent plate. This helps to know where which part must be bend and also weakens the material to make it more easy to bend.

11.3 GENERAL APPROACH

In our implementation, we use bends as joint type as an alternative to, for example, finger joints. Therefore, it is based like the finger joint generation 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.

11.4 PREREQUISITES CREATING BENT PLATES

Because we use bends as a connection type the bent plate creation is based on the plate graph. It uses the plates and its connections. From the plates it needs the shape, the normal and the rotation matrix to lay it into the xy-plane. From the connection the implementation needs the angle and the intersection line. Another step that have to be done before is the finger joint generation. The shapes of those have to be stored in the corresponding connection.

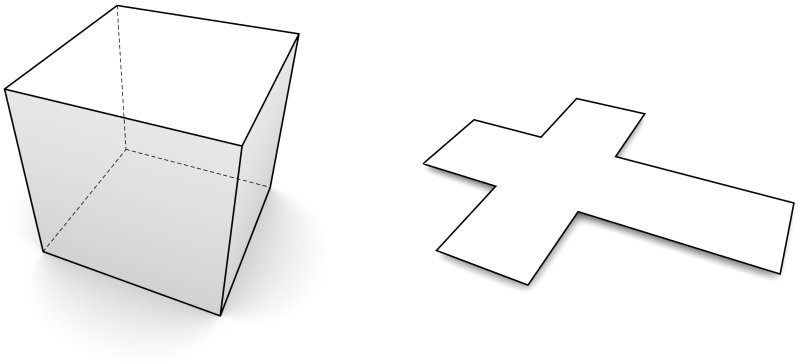
11.5 SETTING THE JOINT TYPE

In this step, we try to find out which connections between two plates could be a bending joint so that the resulting shape of the connected plates is flattable without overlaps. An example for a model where its surface is not devlopable without overlaps is shown in Figure **??**.

To do so, we start with one plate and check for all of its connections:

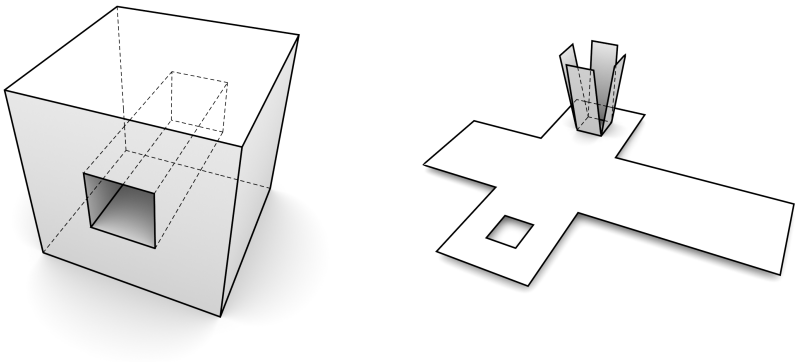
- Is this connection not set already?
- Is the connection angle close enough to 180° which means bending the material this far is possible? (What near enough means depends on the used material)
- Is it possible to add the shape of the connected plate without overlapping the already existing shape?

If the answer is yes to all, the connected plate is added to this plate and they form a bent plate. If the connection type not set but the



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

(b) One possible developed surface of a cube.



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

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

Figure 84: A scene graph showing a box, composed of plates.

angle does not match or there are overlaps the type is set to finger joint.

This is repeated for all the connections of the bent plate until they are all set to be a bending or a finger joint. If after this some plates are not assigned to a bent plate the process is repeated for one of these until none is left.

To check 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 shape and all of its finger joint shapes except for the connection to the new plate. The second is the union of the shape of the new plate and its finger joint shapes except for the connection to the bent plate.

Then 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 and the connection annotated as a bending joint.

11.6 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 that are not added to a bent plate, at least a one plate bent plate, it is repeated with one left plate until no plate is left.

11.6.1 *Traverse along bend connection*

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

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

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

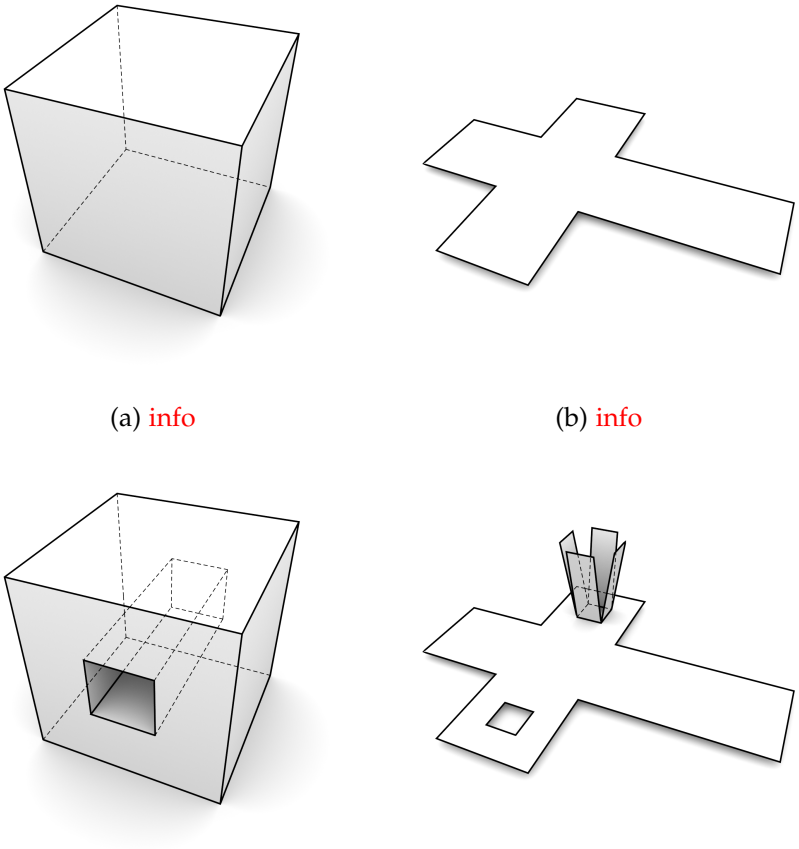
Additionally, the bend axis is determined building the cross product of the two normals of the connected plates. This way the axis does not only lie at the right place but also points in the right direction (which it would not if only the connection line between the two plates is used).

This axis has to be transformed to lay in the xy-plane at the edge of the previous plate. To do so our implementation uses the start point of the intersection line and end point created by adding the axis vector to the start point. These two points are transformed by the be transformation matrix of the previous plate. From the transformed points the final axis is determined by subtracting the end point form the start point.

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

Because the rotation matrix only rotates around the point of origin, two translation matrices are created. First, one for translating the plate to the origin to perform the rotation there. Second, one to translate the plate back to the connection edge.

To get the final transformation matrix the transformation matrix of the previous plate is multiplied by the first translation matrix, the rotation matrix and finally the second translation matrix.



(a) info (b) info

(c) info (d) info

Figure 85: how the bend matrix is created

11.7 ALTERNATIVE SOLUTIONS

11.7.1 *Check for best solution*

The current implementation does not always find the best solution. While traversing the graph it is possible that there multiple different paths which result in unequally well sets of bent plates. One path could be cut by another via an overlap but the better solution would be if the second path along the graph would cut the first. Because the currently only one attempt to traverse the graph is made the second possibility would not be found. To avoid this a possible solution is the following: If one plate that should be added to a bent plate but overlaps with one other plate of the bent plate to also check if removing the older plate (and the only via this plate connected plates) and add the new plate results in a better solution. This will increase the run time and will only result in better solutions for models where the bendable parts of the 3D model form a highly connected graph.

11.7.2 *Curved joints*

In the current implementation the finger joints are generated just based on the plates but not on the bent plates. This means if multiple small plates which each have to small edges to place finger joints on them, the bent plate generated from them would not get Finger joints too.

To avoid this, a new method must be found to generate finger joints at curved edges, because the edge of the plate that connects to the bent plate will be curved. Additionally the finger joints has to be regenerated for every attempt to add a new plate to a bent plate to check for overlaps because adding plates will change the generated joints.

11.7.3 *More bend line types*

The current implementation generates independent of the used material a dashed line as bend line. But This is good for paper to fold it there. For acrylic it would be possible to generate a not dashed line that will be engraved to just give a bending hint without weaken the material to much. For wood, on the other side, living hinges should be generated to make the wood bendable at all.

```
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 30: Creating the transformation matrix for a plate as part of a bent plate.

```
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 31: Creating the transformation matrix for a plate as part of a bent plate.

12

ASSEMBLY

12.1 SOMEBODY WILL HAVE TO DO THE FOLLOWING SECTIONS SHORTLY

Assembling the laser cut models 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 **??** 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 **??**.

12.2 WHAT WE CURRENTLY HAVE (NOT GOOD SOLUTION)

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

12.2.1 *Plate method*

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

REPLACE WITH BETTER IMAGE

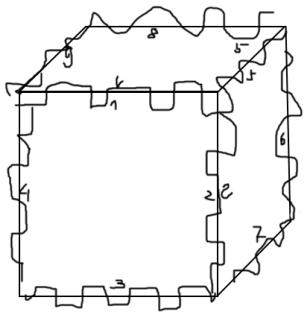


Figure 86: Assembly instructions for plates connected by finger joints.

REPLACE WITH BETTER IMAGE

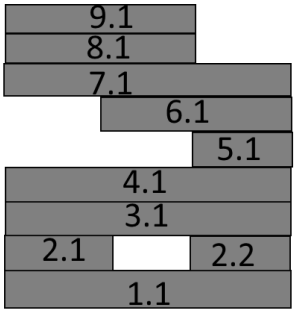


Figure 87: Assembly intructions for stacked plates.

12.2.2 *Stacked-method*

12.3 WHAT MIGHT BE BETTER, BUT IS NOT IMPLEMENTED

12.3.1 *Idea 1: images showing if plate is horizontal or vertical etc*

12.3.2 *Idea 2: large number in the middle of the plate*

13

BENCHMARK

13.1 AASDF

asdfasdf 3 Standard modelle mit unterschiedlichen eigenschaften (curves etc) -> images + 500 modells compare -> testpipeline

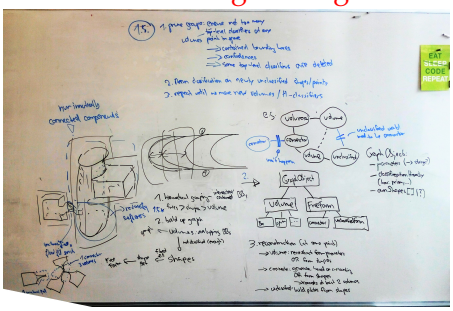
14

CLASSIFIERS / KLARA

14.1 CLASSIFIYING IDEA

In the previous chapters we covered the approach of solely finding plates within 3D-models. This approach needs the model to be very 'boxy'. Therefore sometimes the algorithm fails because of too many round edges or other noise.
This is why we tried another approach which is also being used by CGAL. We look for all kinds of primitive shapes in addition to plates. This allows us to split the model into separate objects which can then be independently converted to lasercutable 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.
TODO: include image
Currently, our software system does not include the findings of 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 up a theoretical structure to make the most of the findings.

TODO: Make images to figures and add caption



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 uses the previously found planes. As soon as all classifiers have finished their ....
TODO: What happens next... : gefundene sachen gruppieren, einzeln dann Bauweisen dafür bestimmen, die 'benoten' die mit der besten note nehmen

TODO: From what paper did we take this? and CGAl etc.

WHAT IS IT USUALLY USED FOR? - POINTCLOUDS... WHICH POINTS DO WE INTEND TO TAKE FOR THE ALGORITHM? 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. In our case 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 all 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 definitely not be used in every case with any 3D-model. Instead it should be used as an additional way to split the model.

14.2 RANSAC - RANDOM SAMPLE CONSENSUS

The RANSAC-approach firstly chooses a defined number of random points. These points are a minimal set from the point data and its number is defined by the shape which is being classified.
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 a best model is saved or overwritten after several candidate attempts.

14.3 PRIMITIVES

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

14.3.1 *Plane*

The minimal set for a plane are 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.

After the detection of a best model it may be necessary to refit the candidate to all its inliers. We use the Least Squares method [
TODO: cite here
]. We are aware that this method can only compute planes where its z-values are dependent on the x- and y-values which is not the case the the plane is perpendicular to the x-y-plane. Therefore we ignore planes to which this applies. 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.

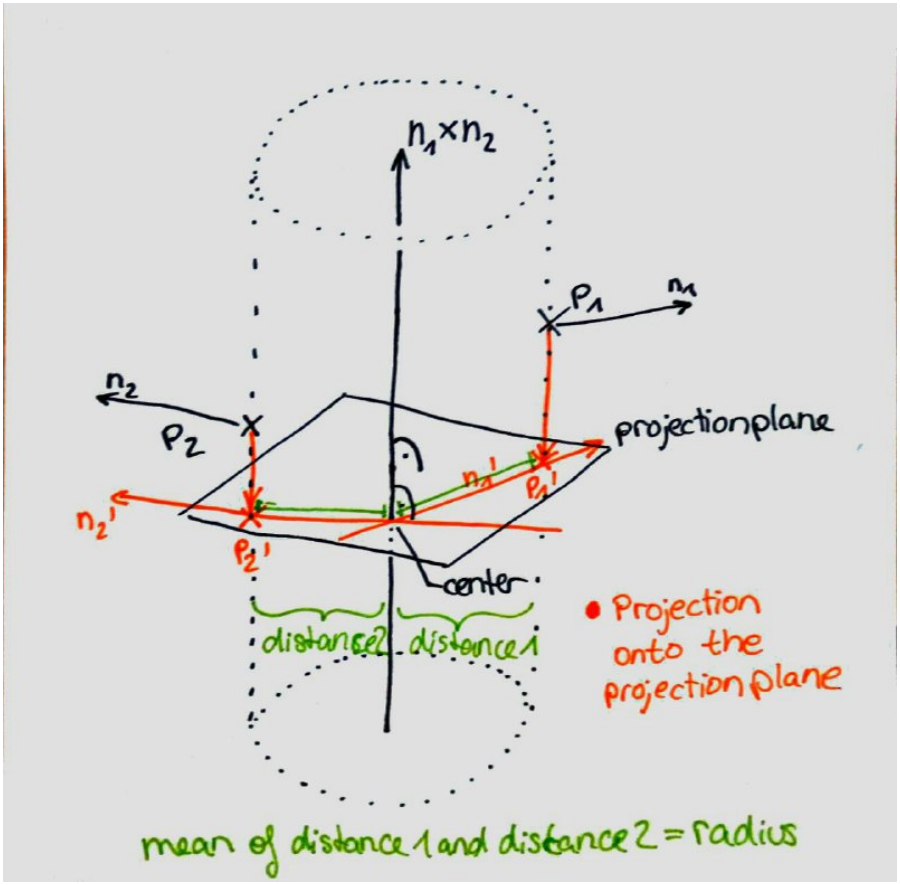
$$\begin{bmatrix} \sum_{i=1}^{m} x_i^2 & \sum_{i=1}^{m} x_i y_i & \sum_{i=1}^{m} x_i \\ \sum_{i=1}^{m} x_i y_i & \sum_{i=1}^{m} y_i^2 & \sum_{i=1}^{m} y_i \\ \sum_{i=1}^{m} x_i & \sum_{i=1}^{m} y_i & \sum_{i=1}^{m} 1 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{m}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

The least squares solution is the following plane equation: $z = Ax + By + C$.
See http://www.geometrictools.com/Documentation/LeastSquaresFitting.pdf for a detailed explanation.

14.3.2 *Cylinder*

The minimal set for a cylinder are two points $\{p_1, p_2\}$. Those points are assumed to lie on the shell of the cylinder. The axis can be calculated by calculating the crossproduct 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 intersectionpoint 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.

TODO: Make images to figures and add caption



After a valid candidate has been formed we have to check the plausibility. For this we look for three indication. Firstly, the randomly chosen points p1 and p2 should not be the same, 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.

14.3.3 *Prism - Dimitri*

14.3.4 *Other primitives*

14.4 PROBLEMS WITH NON-POINTCLOUD INPUT

In our usecase we do not have a pointcloud as input. Instead we operate on the much fewer points in a polygon mesh.
This has large influences on the threshold values that are used. The values taken from CGAL almost never achieved the desired results. We tried finding 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.

15

FUTURE WORK

15.1 ULTIMATE GOAL

15.2 CLASSIFIER

how this is gonna be soooooooo cool

16 CONCLUSION

16.1 AASDF

16.2 USER TESTING

16.3 MAKER FAIRE

asdfasdf

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 2016

Daniel-Amadeus J.
Gloeckner, Sven
Mischkewitz, Dimitri
Schmidt, Klara Seitz, Lukas
Wagner