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
1.1 Solution Platener . . . 1
1.2 Problem we try to solve . . . 1
1.3 A very brief: What have we achieved. . . . 1
2 USERINTERACTION 3
2.1 Drag n Drop Main Interaction . . . 3
2.2 Customize View . . . 3
2.2.1 Material Parameters . . . 3
2.2.2 Device Parameters (Calibration) . . . 3
2.3 Debug View . . . 3
2.3.1 Pipeline Visualizations . . . 3
2.3.2 Algorithm/ Method Selection . . . 3
2.3.3 Testables (Operating Mode) . . . 3
2.3.4 Console Debug Ouput . . . 3
2.4 CLI . . . 3
2.4.1 Usage and Results . . . 3
2.4.2 Reports . . . 3
3 ARCHITECTURE 5
3.1 Architecture Overview . . . 5
3.1.1 Package Responsibilities . . . 5
3.1.2 Decoupling the Framework into Packages . . . 6
3.2 Convertify Package and its Plugin Architecture . . . 7
3.2.1 Convertify Purpose - Application Helpers . . . 7
3.2.2 Render Lifecycle . . . 8
3.2.3 A Plugin is... . . . 8
3.2.4 Control Flow and Plugin Communication . . . 9
3.3 Plugins . . . 11
3.3.1 Plugin Overview . . . 11
3.3.2 Platener Pipeline . . . 13
3.3.3 Node Visualizer . . . 13
3.3.4 Isolated Testing . . . 13
3.3.5 Scorer and Solution Selection . . . 14
3.4 Client Package . . . 14
3.4.1 Overview - Custom Frontend Code . . . 14
3.4.2 React Templates . . . 14
3.4.3 Redux Data-driven Control Flow . . . 14
3.5 Server Package . . . 14
3.5.1 Overview - Custom Server Code . . . 14
3.5.2 Model Cache . . . 14
3.5.3 Test Pipeline . . . 14
4 PROCESSING PIPELINE / SVEN OR DIMITRI 15
4.1 Einzelne Pipeline steps kurz erklärt . . . 15

4.2 Begriffe erklärt: EdgeLoop, Shape, Plate etc. 15
4.3 model loading and storage 15
5 APPROXIMATION / DIMITRI 17
5.1 Mesh Simplification 17
5.1.1 Vertex Welding 19
5.1.2 Simplification Pipelinestep 21
5.1.3 Reusage for redundant information elimination 26
5.1.4 Alternative Methoden 27
5.1.5 Warum diese Methode gewählt? 27
5.2 Point on Line Removal 27
5.3 vllt: Remove Contained Plates - in Lukas Kapitel 27
5.4 Prism Classifier - in Classifier Kapitel 27
5.5 Hole Detection - in Plates Kapitel 27
6 PLATES 29
6.1 Overview of approaches for finding plates 29
6.2 Prerequisites for finding plates 29
6.2.1 Coplanar Faces 29
6.2.2 Shape Finder / Deus 29
6.2.3 Hole Detection / Dimitri 29
6.3 Finding inherent plates 29
6.4 Extruding plates 30
6.5 Removing contained plates / Dimitri 31
6.6 Stacking plates 31
7 ADJACENCY PLATEGRAPH / KLARA 33
7.1 Analysing spatial arrangement 34
7.1.1 Finding intersections 34
7.1.2 Determine angles between plates 35
7.1.3 Truncating intersection lines 36
7.2 Alternative Solutions 37
7.2.1 Dustin: how he did it 37
7.2.2 Dustins wrong angles 37
7.2.3 Down sides 38
7.2.4 Whats better now? 38
7.3 How we got there 38
7.3.1 Floating Point inaccuracy 38
7.3.2 Bruteforce finding lines 38
7.3.3 different line intersection algorithms 38
7.4 Future work 38
8 JOINT COMPUTATION 39
8.1 Joint computation 39
8.1.1 Volume based clipping 39
8.1.2 Female Joint Computation 39
8.1.3 Male Joint Computation 39
8.1.4 Different Fingerjoint types 39
8.1.5 adjusting fingerjoints length when plates are angled 40

8.1.6 Alternative solutions . . . 40
8.2 Future work . . . 40
9 CURVES/DEUS 41
9.1 Cutting curved shapes . . . 41
9.2 General approach . . . 41
9.3 Setting the joint type . . . 41
9.4 Building the bent plates . . . 42
9.5 Alternative solutions . . . 42
10 ASSEMBLY 43
10.1 somebody will have to do the following sections shortly 43
10.2 What we currently have (not good solution) . . . 43
10.2.1 Plate-method . . . 43
10.2.2 Stacked-method . . . 43
10.3 What might be better, but is not implemented . . . 43
10.3.1 Idea 1: images showing if plate is horizontal or vertical etc . . . 43
10.3.2 Idea 2: large number in the middle of the plate 43
11 BENCHMARK 45
11.1 aasdf . . . 45
12 CLASSIFIERS 47
12.1 Classifiying idea . . . 47
12.2 RANSAC . . . 47
12.3 Primitives . . . 47
12.3.1 cylinder . . . 47
12.3.2 Plane . . . 47
12.3.3 Prism - Dimitri . . . 47
13 FUTURE WORK 49
13.1 Ultimate Goal . . . 49
13.2 Classifier . . . 49
14 CONCLUSION 51
14.1 aasdf . . . 51
14.2 User testing . . . 51
14.3 Maker Faire . . . 51

BIBLIOGRAPHY 53

LIST OF FIGURES

Figure 1 Main Packages of the Platener Architecture . . 6
Figure 2 Main Packages of the former Brickify Architecture 7
Figure 3 Dispatcher manages inter-application communication during the lifecycle of the app. 10
Figure 4 An empty scene showing the coordinate system. 12
Figure 5 Stanford Bunny with 8662 faces 18
Figure 6 Simplified Stanford Bunny with 616 faces (welding distance: 10mm) 18
Figure 7 Vertex Welding 19
Figure 8 Vertex Welding 20
Figure 9 Original Makerbot letters 22
Figure 10 Simplified Makerbot with welding distance 3mm - letters completely removed 23
Figure 11 Simplified Makerbot with welding distance 0.8mm - artifacts 23
Figure 12 Beveled cube with 1, 2, 10 subdivisions and their resulting cube without beveled edges . . 24
Figure 13 Extruded details 24
Figure 14 Pushed in details 25
Figure 15 Vertex Welding UI Element 25
Figure 16 (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. 36
Figure 17 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. 36

1

INTRODUCTION

1.1 SOLUTION PLATENER

1.2 PROBLEM WE TRY TO SOLVE

1.3 A VERY BRIEF: WHAT HAVE WE ACHIEVED.

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 Ouput*

2.4 CLI

2.4.1 *Usage and Results*

2.4.2 *Reports*

3 ARCHITECTURE

3.1 ARCHITECTURE OVERVIEW

Platener is designed to enable web-based manipulation and rendering of 3D-Models. Figure 1 shows the major packages *Convertify*, *Client*, *Server* and *Plugins*. The arrows indicate dependencies between the packages. The architecture emphasizes a uni-directional data and event flow. Similar to mobile application design, lifecycle events and the concept of delegation[1] establish clear communication among packages.

3.1.1 *Package Responsibilities*

Each package takes over a set of distinct responsibilities ensuring decoupled components. Thus a flexible, maintainable system is created.

Convertify provides generic tools which support plugins in manipulating 3D-Models. This includes utilities for vector analysis as well as rendering routines and scene management. The application lifecycle can be initiated and observed via a *Bundle*. A Bundle represents an instance of the application's computation unit. The Client and Server packages run a Bundle.

Plugins provide an exchangeable set of features which are used by the Client and Server package. A Plugin interacts with the Scene and its 3D-Models via lifecycle events. E.g. a concrete conversion strategy provided by Platener is implemented by a single Plugin. Plugin features can be enabled for Client, Server or both.

The *Client* package gives the look and feel of the application. This package contains frontend components. The developer can choose any template engine[2] which serves the application's purpose. So the Client wires up the user-interface and the computation logic.

The *Server* package is the headless[3] counterpart to the Client package. A Command Line Interface enables the user to run the applica-

1 https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/Delegation.html

2 http://www.sitepoint.com/overview-javascript-templating-engines/

3 A headless web-application runs without the graphical user interface of browsers.

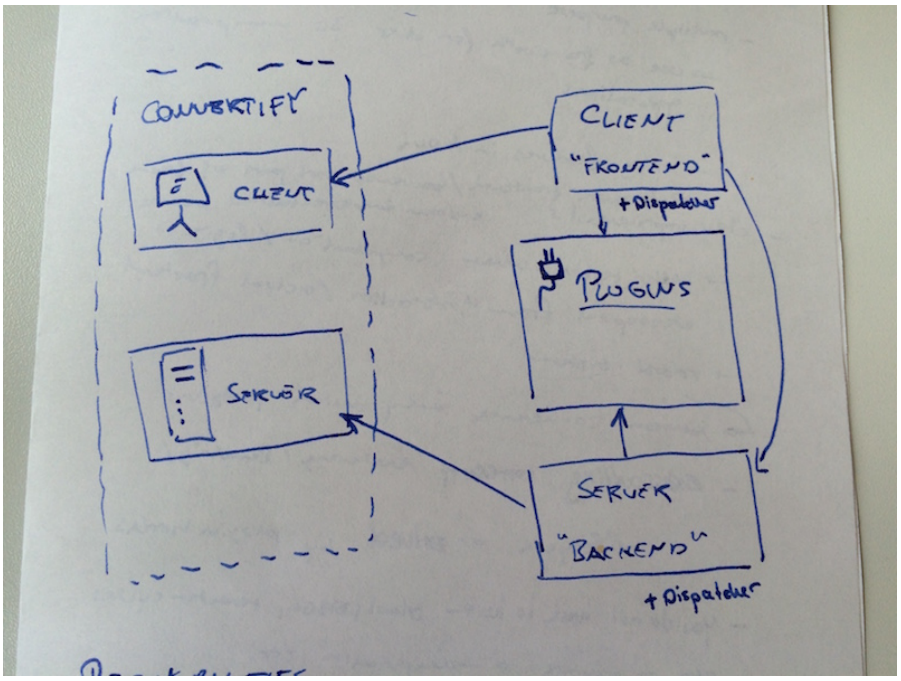


Figure 1: Main Packages of the Platener Architecture

tion without a browser. The Server also satisfies requests from the Client, such as caching and loading models in a RESTful interaction[4].

3.1.2 *Decoupling the Framework into Packages*

Decoupling the software into packages builds a robust system. Computation logic and UX components are kept apart, which allows isolated testing.

The Convertify package is meant to be used as a white-box framework for building 3D manipulation applications. Platener is such an application, supporting model imports, rendering, altering and export of geometries. Introducing a web framework makes sense, because WebGL gets more and more stable as of the year 2016 TODO: reference!. Thus graphics software can be brought to huge audience providing cross-platform web services.

Plugins are self-contained units which mainly interact with the WebGL scene and its scene graph. Whole feature-sets can be switched on or off at once. They are loosely coupled but have high cohesion at the same time. This prevents spaghetti code and supports maintainability of each component. Furthermore, we use software hooks to integrate plugins with rendering of models and access to geometry data. This event-based approach enables developers to write a 3D

4 http://www.drdobbs.com/web-development/restful-web-services-a-tutorial/240169069

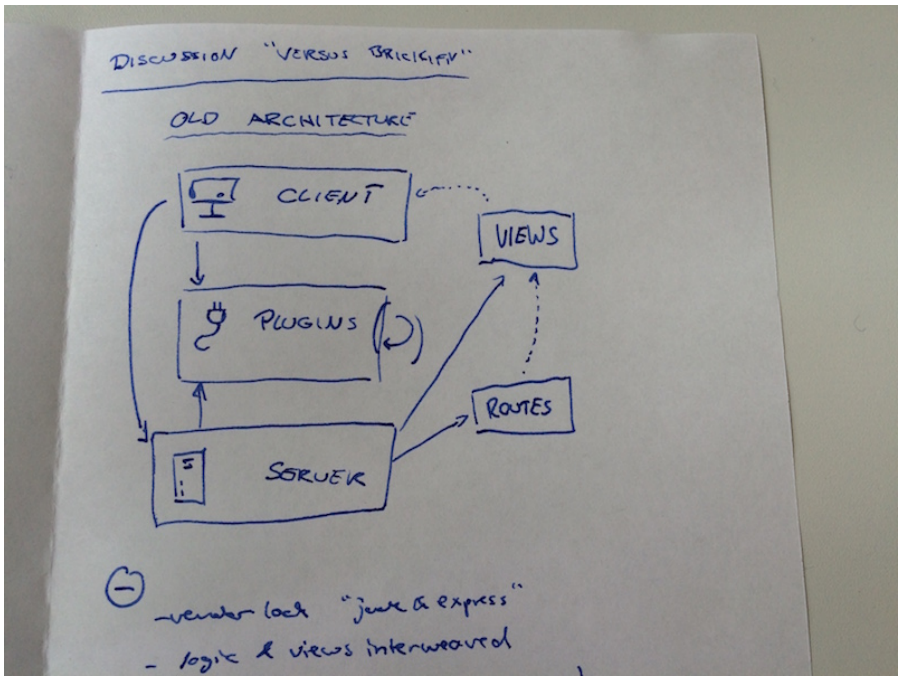


Figure 2: Main Packages of the former Brickify Architecture

manipulation tool without having detailed domain knowledge about WebGL and web-services. This allows developers from the Computer Graphics domain to concentrate on the graphics problem, rather than the web environment.

The Convertify package is not present in the Brickify architecture, see Figure ??. This means Brickify provides a mix of UX components, computation logic and interfacing code in the Client and Server packages. Plugin communication is hard to observe and only implicitly ordered. We introduce the new package to escape the vendor lock of previously used UX libraries and to create a thoroughly testable code base.

3.2 CONVERTIFY PACKAGE AND ITS PLUGIN ARCHITECTURE

Platener uses the Convertify package as entry points to the 3D-Modelprocessing. The package provides features for browser- and headless-environments.

It encapsulates basic logic for 3D-Modelmanipulation and model import. Additionally, we provide utilities to simplify working with *three.jsx* or rendering objects into scene.

Convertify handles the scenegraph of the application. *Nodes* are injected into the scene ref brickify and lifecycle events are emitted, e.g. on load, on draw or on touch interactions with the *Node*.

3.2.1 *Render Lifecycle*

WHAT?

- App Lifecycle: Model Loading
- Plugin Init
- Scene Refreshes
- Scene Graph Interactions
- refer to Brickify

WHY?

- Gain control for computation units (plugins)
- interact with the system

HOW?

- plugin hooks (refer to pluginHooks.yaml)
- called on each plugin
- allow plugin to handle the event (e.g do some manipulation to the scene)

3.2.2 *A Plugin is...*

WHAT?

- pluggable set of feature
- or self-contained unit
- can be activated for server/ client
- composing multiple plugins –> extensible, maintainable method of building complex applications

WHY?

- Convertify as a framework
- do not mix logic/ ui
- decomposable logic (no spaghetti code)

HOW?

- define with package.json (could conceptionally be another npm package)
- add to PluginMap
- define hooks

- space of freedom is huge...

see chapter ??, a list of plugins see chapters ?? for more detail maybe figure about lifecycle and hooks

3.2.3 *Control Flow and Plugin Communication*

As Platener is composed of 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. We propose a Dispatcher component, behaving similar to the mediator pattern[5].

3.2.3.1 *Dispatcher*

WHAT?

The Dispatcher is the entry point for any client or server code. The Dispatcher loads and initializes a set of configured plugins. It organizes the communication between plugins and client, plugins and server and plugins and plugins in a single place.

reference figure

events Convertify fires lifecycle events on which the plugins can react. The mediator knows an explicit execution order for each plugin when a lifecycle event is fired.

protocols

- communication between packages (no lifecycle)
- use protocols (explicit definition of what will happen)
- protocol embodies a set of feature for the application

WHY?

- one guy who pulls all the strings (one source of truth)
- similar designs in known application frameworks (android, ios)
- javascript, non-type language –> protocols ensure correct interfaces

HOW?

- refer to Brickify, PluginLoader
- explain plugins.yaml shortly
- implements plugin hooks itself (show code snippets, hooks property)

5 https://sourcemaking.com/design_patterns/mediator

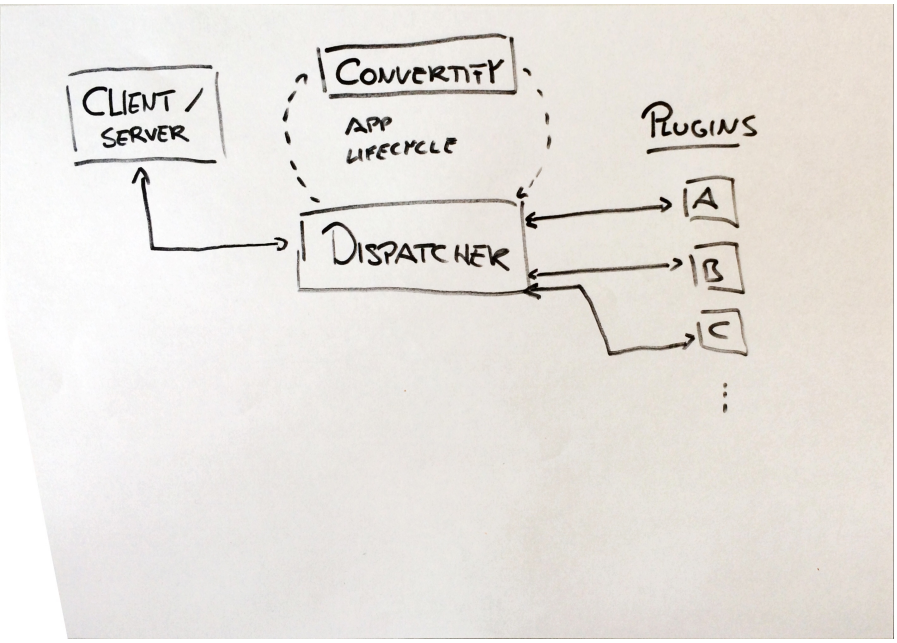


Figure 3: Dispatcher manages inter-application communication during the lifecycle of the app.

- implements protocols for plugins (show code snippets)
- explain how plugins push state (control-flow information) back to the dispatcher

3.2.3.2 *Control Flow vs Brickify only plugin hooks*

- wiring up plugins
- explicitly calling plugin hooks
- highly customizable when adding in new plugins
- how is control flow achieved? –>
- ++ pros
- loading sequence control (impossible to repair)
- fine grained scheduling of communication
- explicit execution behavior
- ... (look at notes on desk)
-
- – cons
- boilerplate code
- custom integration for each plugin
- ++ pros

- no further config needed
-
- – cons
- implicit order
- in different use cases, different plugins have to interact first (order is not fixed)
- frontend often has to work on data which is not available before another plugin loaded

now plugins can...

- push application state back to the dispatcher
- communication with each other by specifying a protocol which has to be implemented on the mediator
- mediator then can oversee the communication

3.3 PLUGINS

3.3.1 *Plugin Overview*

The plugins composed into Platener provides its computation logic and WebgGL scene rendering. We will give a brief introduction of each plugin in the following paragraphs.

3.3.1.1 *Coordinate System*

This plugin provides orientation enhancements for the WebGL scene. Rendering xyz-axes and a an axis-aligned grid, users can grasp alignment and dimensions of 3D-Models. Figure ?? shows the coordinate system in the WebGL view.

3.3.1.2 *Platener Pipeline*

The Platener Pipeline plugin is the main computation unit. The plugin defines multiple Fabrication Methods. A Fabrication Method is a conversion approach of a 3D-Model. Multiple components, which can manipulate the input model, are chained after another to produce 2D- or 3D-output. For example construction plans for the original input model as SVG-files.

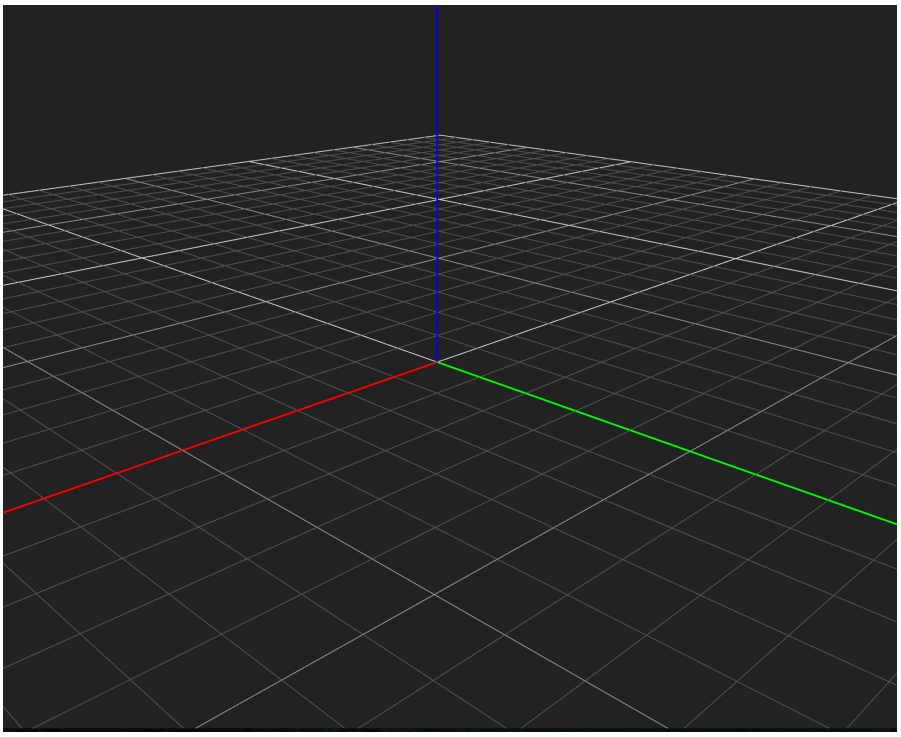


Figure 4: An empty scene showing the coordinate system.

3.3.1.3 *Node Visualizer*

We visualize the results of the Platener Pipeline plugin in the WebGL view. The Node Visualizer plugin renders the results of each component of each Fabrication Method respectively.

3.3.1.4 *Scoring*

When multiple Fabrication Methods are run, we want to choose the best fitted conversion as output. Thus each Fabrication Method is scored by a method-specific scoring algorithm. This plugin provides these scoring algorithms.

3.3.1.5 *Solution Selection*

This plugin utilizes the Platener Pipeline plugin and the Scoring plugin to run and evaluate all Fabrication Methods. It outputs the result of the Fabrication Method with the best score and notifies the Dispatcher.

3.3.1.6 *Isolated Testing*

While we worked at different stages of a linearly executed Fabrication Method in parallel, we needed a mechanism to test each component of the Fabrication Method before its preceeding or succeeding components were finished. The Isolated Testing plugin provides an isolated environment, which allows to execute a single component of a Fabrication Method with pre-defined input.

3.3.2 *Platener Pipeline*

1. Overview/ Purpose/ Plugin Structure
2. - conversion strategies and logic
3. Pipeline Steps

   What are pipeline steps? Where to be found in the code? How are they used in Platener Pipeline?
4. Fabrication Methods
   a) Plate Method
   b) Stacked Plates Method
   c) Classifier Method
5. Immutable Pipeline State
   a) Overview/ Purpose
   b) Immutability
   c) Implementation Details

3.3.3 *Node Visualizer*

1. Overview - Visual Debugging
2. Visualizer and Visualizations
3. Interwoven with Platener Pipeline
   a) Role of Immutability
   b) Extracting Intermediate Data of the Pipeline

3.3.4 *Isolated Testing*

1. Static Input and Expected Output

2. Testables

3.3.5 *Scorer and Solution Selection*

Selecting the best estimated solution by default.

3.4 CLIENT PACKAGE

3.4.1 *Overview - Custom Frontend Code*

3.4.2 *React Templates*

3.4.3 *Redux Data-driven Control Flow*

1. Dispatcher and Redux ‘dispatch‘

3.5 SERVER PACKAGE

3.5.1 *Overview - Custom Server Code*

3.5.2 *Model Cache*

3.5.3 *Test Pipeline*

1. Headless Conversion of Objects
2. Reports
3. Benchmarks

4

PROCESSING PIPELINE / SVEN OR DIMITRI

4.1 EINZELNE PIPELINE STEPS KURZ ERKLÄRT

4.2 BEGRIFFE ERKLÄRT: EDGELOOP, SHAPE, PLATE ETC.

4.3 MODEL LOADING AND STORAGE

5

APPROXIMATION / DIMITRI

TODO: Kurzer Zusammenfassender Absatz über das Kapitel

5.1 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 5 is simplified with an unusually high welding distance of 10mm in Figure 6. 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.

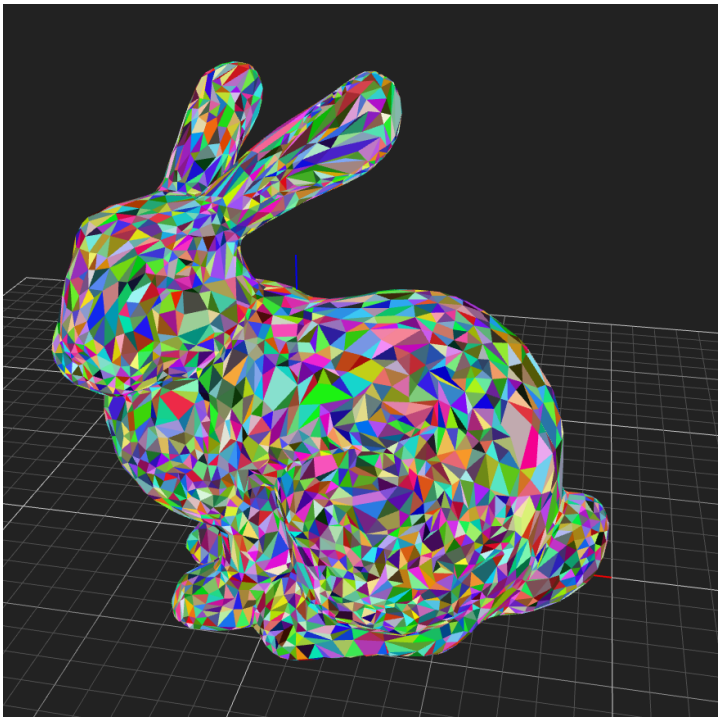


Figure 5: Stanford Bunny with 8662 faces

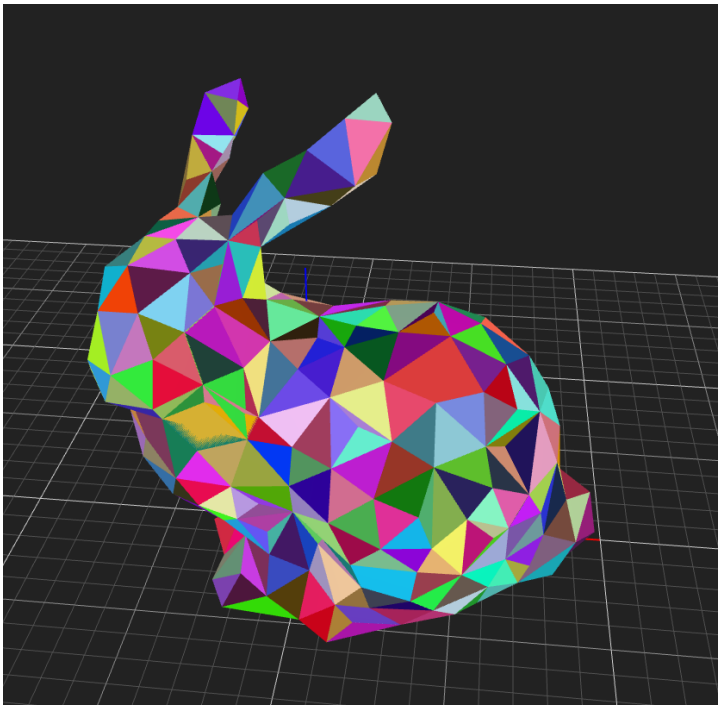


Figure 6: Simplified Stanford Bunny with 616 faces (welding distance: 10mm)

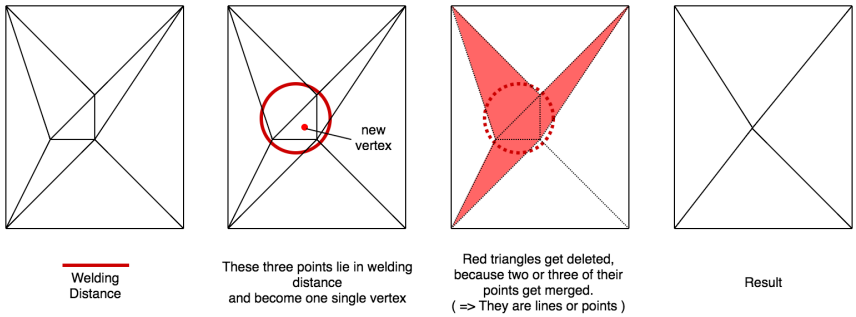


Figure 7: Vertex Welding

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.

5.1.1 *Vertex Welding*

Vertex Welding merges vertices and works as shown in Figure 8:

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 8 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.

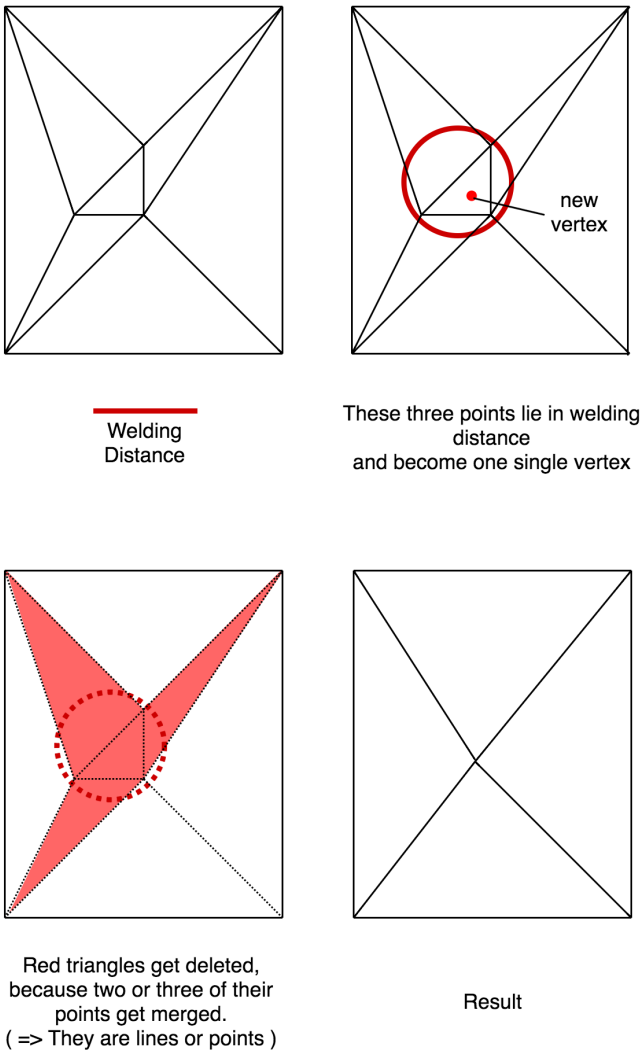


Figure 8: Vertex Welding

5.1.1.1 *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 vertices given by the *VertexWelding*. 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:

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.

TODO: Usage

5.1.1.2 *Implementation*

TODO: Implementation

5.1.2 *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)

Figure 9: Original Makerbot letters

5.1.2.1 *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 10. With a welding distance of 3mm the result is a smooth surface while the original model shown in Figure 9 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 11.

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...

5.1.2.2 *Unwanted Details*

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

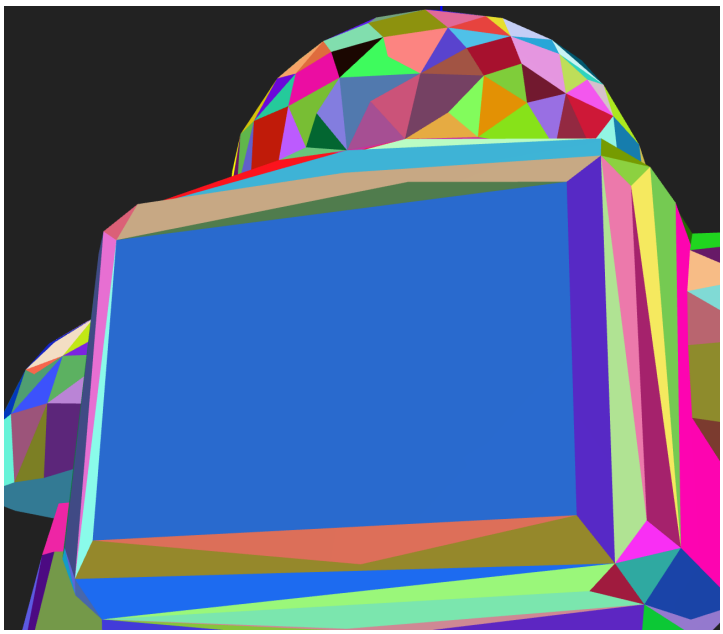


Figure 10: Simplified Makerbot with welding distance 3mm - letters completely removed

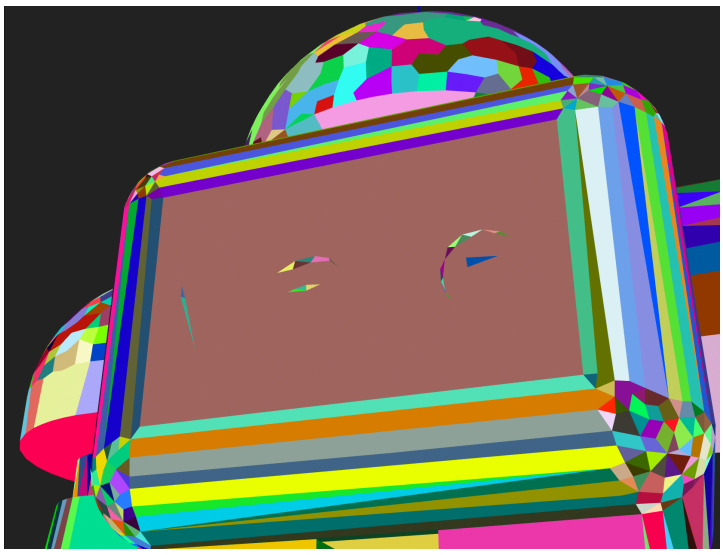


Figure 11: Simplified Makerbot with welding distance 0.8mm - artifacts

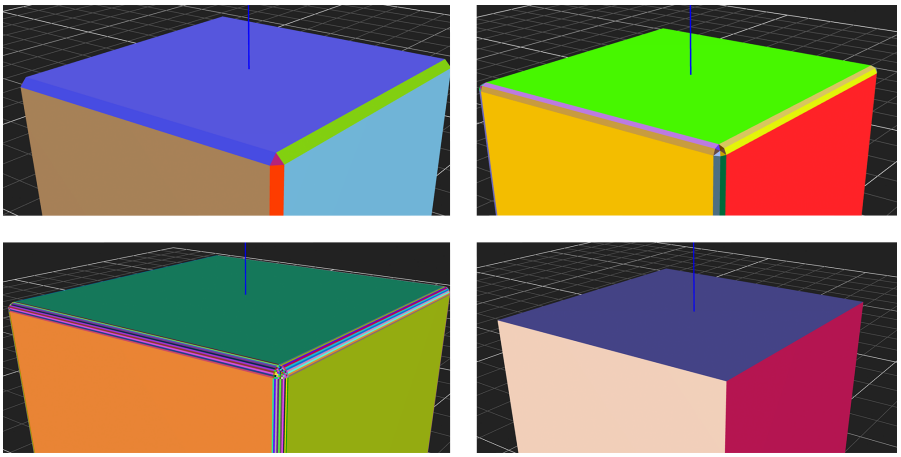


Figure 12: Beveled cube with 1, 2, 10 subdivisions and their resulting cube without beveled edges

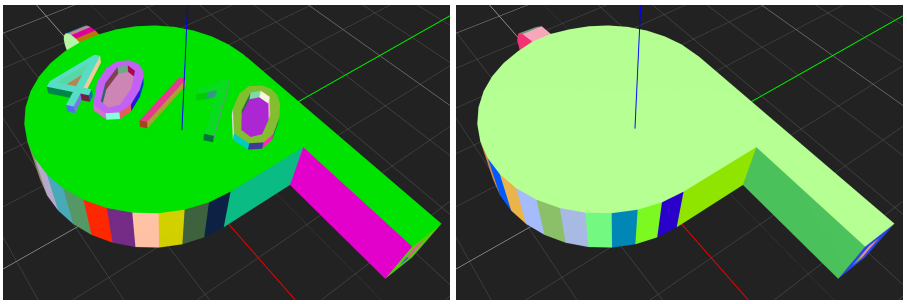


Figure 13: Extruded details

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

Figure 12 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 13 there is text on top of the actual surface and in Figure 14 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?

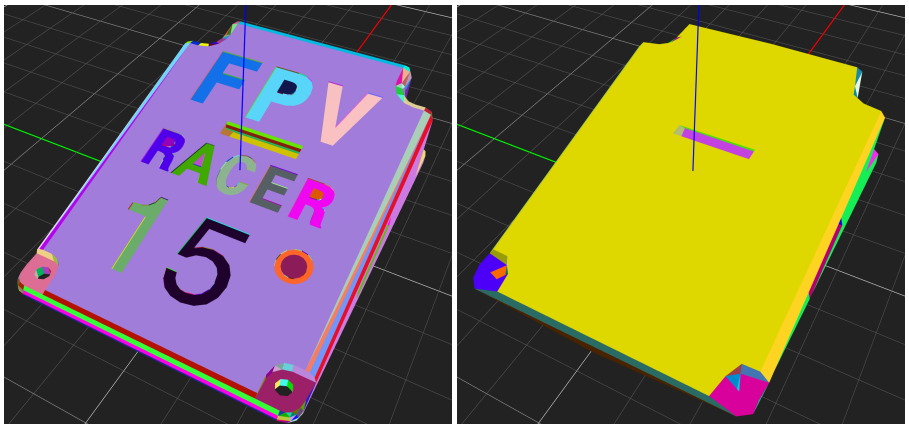


Figure 14: Pushed in details

Figure 15: Vertex Welding UI Element

5.1.2.3 *Processing Time Improvement*

TODO: processing time improvement

TODO: Zahlen Runtime Vergleich: mit und ohne Simplification

5.1.2.4 *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

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 15. 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 com-

mon 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.

5.1.3 *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

1 https://www.blender.org

5.1.4 *Alternative Methoden*

5.1.5 *Warum diese Methode gewählt?*

5.2 POINT ON LINE REMOVAL

5.3 VLLT: REMOVE CONTAINED PLATES - IN LUKAS KAPITEL

5.4 PRISM CLASSIFIER - IN CLASSIFIER KAPITEL

5.5 HOLE DETECTION - IN PLATES KAPITEL

6

PLATES

6.1 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 actually modeled in the mesh with both a top and a bottom side. The second approach, extruded plates, uses the mesh surface to extrude plates into the object. 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.

6.2 PREREQUISITES FOR FINDING PLATES

coplanar / shapesfinder / holedetection

6.2.1 *Coplanar Faces*

fvdfsjkldf

6.2.2 *Shape Finder / Deus*

öösdfjfsdkf

6.2.3 *Hole Detection / Dimitri*

sdfjkljklsdfsdfjkl

6.3 FINDING INHERENT PLATES

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

Listing XYZ: Plate candidate pseudo code

```
candidates = []
  for shape1, index1 in shapes
    for shape2, index2 in shapes when index1 < index2
      if normals parallel and surfaces facing apart
        if distance between shape1, shape2 in plate thicknesses
          candidates.push { shape1, shape2 }
return candidates
```

While the testing for normal parallelism is done with built-in vector functions, the check if the surfaces 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. The distance between the surfaces is calculated by creating a three.js plane from one of the shapes and computing the plane-to-point distance towards one of the other shape's points.

After finding these plate candidates, the shapes 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. 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 rotation matrix of the base shape. With the resulting shapes, plates are created.

Listing XYZ: Plate creation

```
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
```

This step uses the jsclipper library for intersecting the shapes. 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.

6.4 EXTRUDING PLATES

blabla

6.5 REMOVING CONTAINED PLATES / DIMITRI

6.6 STACKING PLATES

blablabla

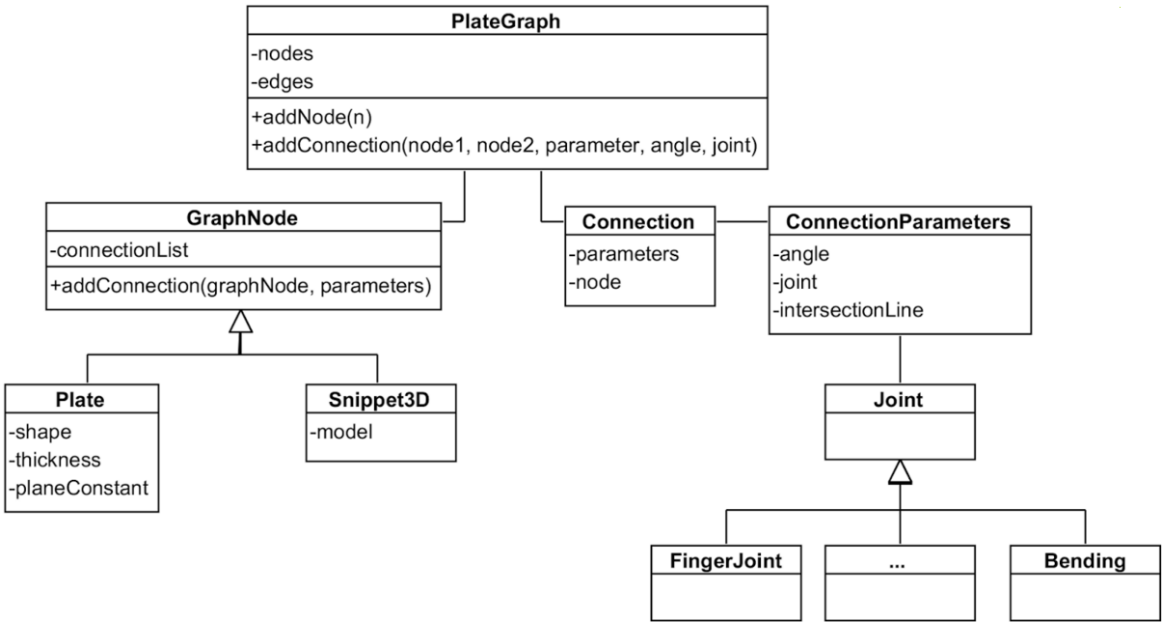
7

ADJACENCY PLATEGRAPH / KLARA

On the basis of the graph we can create connectors for plates in a later step (8). 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 6. Two or more plates are adjacent to another when at least one side touches or overlaps with another plate. In addition, the angles in between the plates are measured.

The graph always holds a list of all found plates and their neighbors. A plate is just one possible object to be listed. Other graph nodes can be left over 3D snippets which will have to be handled when all plates have been correctly connected.
For iterating over the graph we traverse all edges of the graph which also hold the important neighborhood parameters such as angle and the line at which nodes intersect.



7.1 ANALYSING SPATIAL ARRANGEMENT

As a prerequisite the step 6 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 8 we truncate the inner intersection lines which would otherwise overlap with adjacent plate intersections.

7.1.1 *Finding intersections*

When two planes intersect there is an intersection line. Since we work with plates which equal 2 parallel planes we expect to find up to 4 intersection lines.
Firstly, we retrieve the direction vector(*dir*) of any intersection line between two plates by calculating the cross vector of both normals.
In order to retrieve all possible intersection lines of two plates we 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
- and the other way around

First, a possible position vector has to be found which lies on both planes.

Plane constants: d_1, d_2
Normals: n_1, n_2

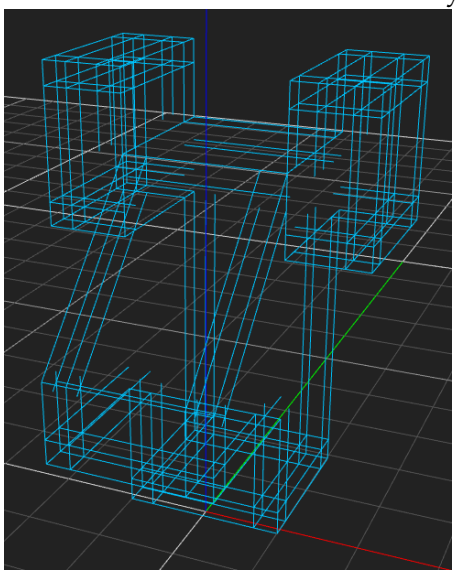
$$p = \frac{d_1 * n_2^2 - d_2 * (n_1 * n_2)}{n_1^2 * n_2^2 - (n_1 * n_2)^2} * n_1 + \frac{d_2 * n_1^2 - d_1 * (n_1 * n_2)}{n_1^2 * n_2^2 - (n_1 * n_2)^2} * n_2$$

On the basis of a position vector an intersection line can be computed.

$$line = p * x + dir$$

Now that all lines are found we have to test if the lines actually 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.
In order to find the boundaries of the lines we calculate the intersec-

tions of the lines with all boundary edges of the plates.



7.1.2 *Determine angles between plates*

In order to generate nicely fitting joints in the following step and for grouping plates the angles are necessary.
Firstly, we determine the angle between the according planes.
plane normals: u, v
angle between planes: θ

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

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 dependent on the direction in which the normals are pointing.

In order to find out in which cases the angle has to be adjusted we check for two properties.
Firstly, there is the question which of the sides of the plates intersect. A plate is defined by a 2D-shape which is called main side. The other side which exists due to a specified thickness of the plate is called parallel side.
Additionally, we 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

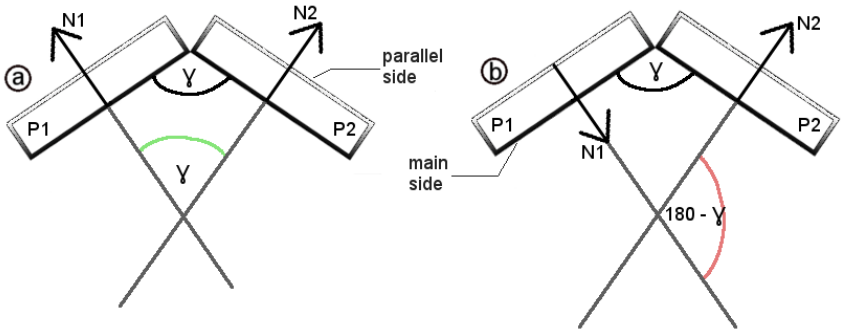


Figure 16: (a) The angle between the plates' normals correspond with the angle γ between the plates. (b) The angle between the plates' normals is in this case an adjacent angle to the requested angle γ.

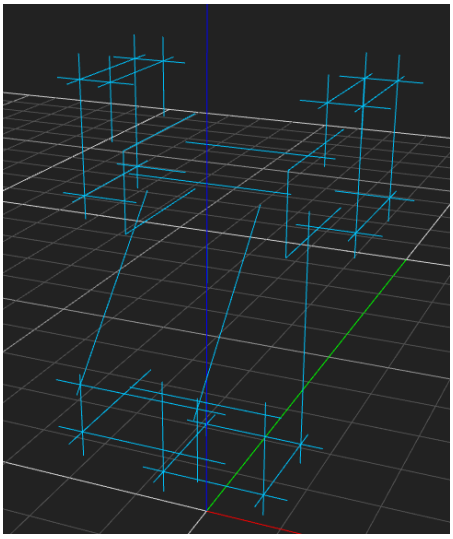


Figure 17: The inner boundaries of the plates overlap. In the next step when joints will be created they are only supposed to be on the inner part of the line. Therefore the lines have to be truncated.

- Their directions are both positive OR both negative

7.1.3 *Truncating intersection lines*

Now that all 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 that 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.

Algorithm 1: Truncate 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 }
```

7.2 ALTERNATIVE SOLUTIONS

7.2.1 *Dustin: how he did it*

7.2.2 *Dustins wrong angles*

he didnt actually specify how he calculated the angles...

7.2.3 *Down sides*

7.2.4 *Whats better now?*

7.3 HOW WE GOT THERE

7.3.1 *Floating Point inaccuracy*

7.3.2 *Bruteforce finding lines*

7.3.3 *different line intersection algorithms*

7.4 FUTURE WORK

- find out if model is assemblable in the end

8

JOINT COMPUTATION

8.1 JOINT COMPUTATION

8.1.1 *Volume based clipping*

PREREQUISITE: PLATEGRAPH INTERSECTION LINES

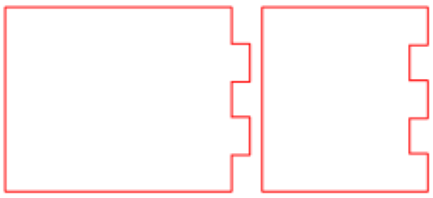
8.1.2 *Female Joint Computation*

8.1.3 *Male Joint Computation*

8.1.4 *Different Fingerjoint types*

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

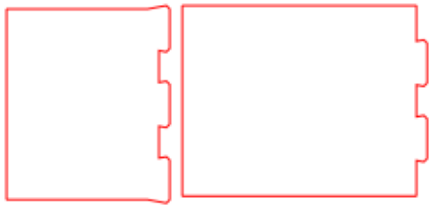
8.1.4.1 *Fingerjoint template*



HOW THESE JOINTS LOOK LIKE

HOW THESE JOINTS WORK, AND FOR WHAT MATERIAL

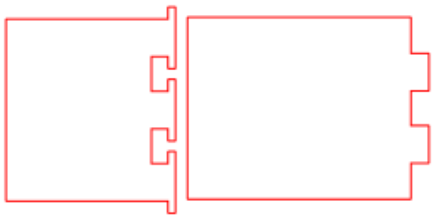
8.1.4.2 *JimJoint template*



HOW THESE JOINTS LOOK LIKE

HOW THESE JOINTS WORK, AND FOR WHAT MATERIAL paper or other easily bendable/eindrueckbar material

8.1.4.3 *Schwalbe template*



HOW THESE JOINTS LOOK LIKE

HOW THESE JOINTS WORK, AND FOR WHAT MATERIAL

8.1.5 *adjusting fingerjoints length when plates are angled*

DEUS

8.1.6 *Alternative solutions*

8.2 FUTURE WORK

9

CURVES/DEUS

9.1 CUTTING CURVED SHAPES

Approximating curved shapes with parts created by the laser cutter is a special challenge because only 2d shapes can be cut. Nevertheless, there are several approaches to make the cut material bendable, for example, paper can already be bent, 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). Practically this problem is not important for us because the 3d models we use are triangle meshes so only flat surfaces are used to represent the surface wich makes it always developable.

9.2 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. Furthermore, it is separated into two important steps. The first one annotates each connection between plates if this should be a bending joint or not and the second creates a flatted shape from the plates that are connected with bends.

9.3 SETTING THE JOINT TYPE

In this step, we try to find out wich connections between two plates could be a bending joint so that the resulting shape of the connected plates is flattable without overlaps.

To do so we start with one plate and check for all the connections it has:

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

If so, the connected plate is added to this plate and they form a bent plate. This is repeated for all the connections of the bent plate until thy are all set to be a bending or a finger joint. If after this all plates are not assigned to a bent plate the process is repeated for one of these until no one is left.

To check if a plate could be added to an existing bent plate we merge all the finger joint shapes of the outer connections of the bent plate and those of the probably added plate to the corresponding shape except for the ones that are from the connection between the bent plate and the new one. 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

If one of the conditions is not fulfilled the connection can't be a bend and is annotated as finger joint.

9.4 BUILDING THE BENT PLATES

9.5 ALTERNATIVE SOLUTIONS

10

ASSEMBLY

10.1 somebody will have to do the following sections shortly

10.2 what we currently have (not good solution)

10.2.1 *Plate-method*

just write numbers on lines -> equal numbers belong together

10.2.2 *Stacked-method*

number plates

10.3 what might be better, but is not implemented

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

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

11

BENCHMARK

11.1 AASDF

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

12

CLASSIFIERS

12.1 CLASSIFIYING IDEA

12.2 RANSAC

12.3 PRIMITIVES

12.3.1 *cylinder*

12.3.2 *Plane*

12.3.3 *Prism - Dimitri*

13

FUTURE WORK

13.1 ULTIMATE GOAL

13.2 CLASSIFIER

how this is gonna be soooooooo cool

14

CONCLUSION

14.1 AASDF

14.2 USER TESTING

14.3 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