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

LIST OF FIGURES

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 IMPLEMENTATION GOALS AND TOOLCHAIN

3.1 DEPLOYED AS A WEB-SERVICE

3.1.1 *Cross-Platform Browser Application*

- web service
- cross platform
- no installation necessary
- target group: makers, run in labspaces, everywhere
- typical HTML, CSS, Javascript trio

3.1.2 *Headless Version for Integration with Other Software*

- same service without visuals
- batch processing or single
- idea: integrate with other 3d editing applications to benefit from platener conversions
- running on nodejs v 012

3.2 TECHNOLOGIES AND LIBRARIES

- web service -> necessarily javascript
- coffeescript preprocessing, no es6 because previous code base brickify was written in coffeescript -> save effort of rewrites
- webgl -> rendering, now almost all major browser support it prove that
- nodejs for headless version/ server (backend) support (v0.12 because old brickify dependencies)
- threejs -> most acknowledged 3d web library (scene graphs and webgl)
- bluebird -> promises (nice tech in js to handle async behavior, better than natives because of error handling, more features)

- polyfills -> more cross platform support and isomorphic => browsers and nodejs run in different vms
- styles with stylus lib (also taken from brickify)

3.3 BUILD SYSTEM AND DEVELOPMENT SETUP

- gruntfile -> and webpack
- different setups for client, server and cli in prod/ dev mode
- test runner using mocha
- bundle code into minified code and output sourcemaps (nice for in browser debugging)
- smart server (rebuilding on filechanges, hotreloading (not compiling everything again for speedup))
- browser lifereload (code change immediatly displayed in browser)
- react hotreloading -> display changes in ui even without a browser reload (super fast feedback loop)

4

ARCHITECTURE

In this chapter we outline the architecture and composition of the application *Platener*. Therefore, we look at concepts of graphics programming in web-environments in Section **??**. Then, we explore the structure of our software from two points of view. At first, we present *Convertify* in Section 4.2. *Convertify* is the web-framework in which *Platener* is implemented. Secondly, we explain the details of *Platener* in Section 4.3.

With a framework like *Convertify* we can implement generic 3D-Model manipulation web-applications. 3D-Model manipulation means restructuring the model geometry in such a way, that it can be used in a new context - apart from visualization on computer screens. Such manipulations include converting the 3D-Model for physical output devices like 3D-Printers or laser cutters. For example, *Brickify* creates models consisting of 3D-printed parts and *LEGO*^{TM}-assembled[1] parts. Thus, *Brickify* could ideally be built with *Convertify*.

Platener is an application implemented in the framework *Convertify*. Using a framework helps us focusing on the algorithmic challenges of converting arbitrary 3D-Models into physical, laser-cut output.

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 4.1.1 and render loops and scene graphs in Section 4.1.2.

4.1.1 *3D-Model Representation*

This section explains which data structures and disk-formats are used to represent 3D-Models. First, we explain the terminology of meshes. Secondly, we look at the STL-file format

The model geometry is represented by a set of connected polygons, approximating the model surface. These connected polygons form a

1 http://www.lego.com

mesh. Figure **??** shows a mesh. For the sake of simplicity, we use triangles as polygons. A triangle in the mesh is called face. Each face is described by a list of three coordinates in 3D-space. Such a coordinate is called vertex [**?** , p. 3]. An edge is the line between two connected vertices.

We only support geometry which is arranged in two-manifold meshes. Two-manifoldness is a constraint on the mesh, which requires each egde to only touch two neighboring faces. Figure **??** shows a comparison of a manifold and non-manifold part of a mesh. It follows then, that each triangular face has to have three adjacent faces. Thus, the constraint enforces the mesh to be a fully connected graph without holes [**?** , p. 28]. We require two-manifoldness to make sophisticated assumption when designing the conversion algorithms.

The Standard Tessellation Language (STL) format is a common file format for 3D-Model representation in the context of 3D-printing. We support the STL-file format, because our goal is to make models, built for a 3D-Printer, available for the laser cutter. An STL-file consists of a list of faces associated with their face normal. The face normal determines the orientation of the face. We need the face normal, because from the vertices only, we cannot say in which direction the face points. Each face is a list of vertices. A vertex is described by three floating point numbers. Listing **??** shows an STL-file in ASCII encoding. For saving disk-space, the file is mostly stored in a binary format [**?** , p. 8].

Reconstructing the original 3D-Model from an STL-file does not always result in one-to-one solution. The vertices are stored with floating point numbers instead of referring to the same vertex with an index. Thus, 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. With Meshlib we can convert the face-vertex-mesh representation to a three.js geometry. 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 with the concepts of rendering and scene graphs first.

A typical pattern any graphics software cannot live without 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]. The render loop pattern consists of three steps: processing input, updating the 3D-Model representation and rendering [**?**]. Figure 1 shows an exemplary flow.

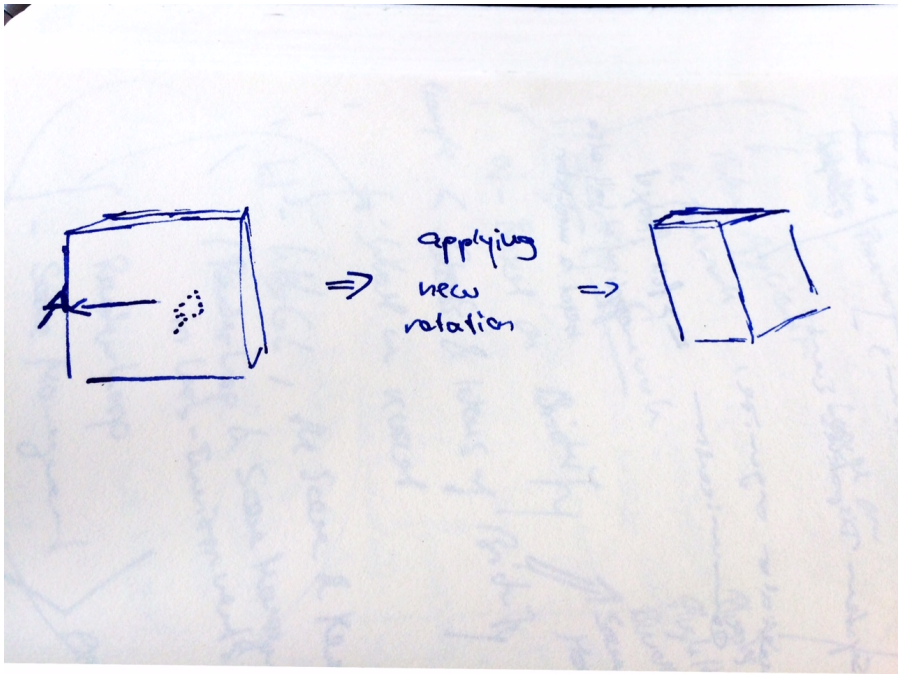
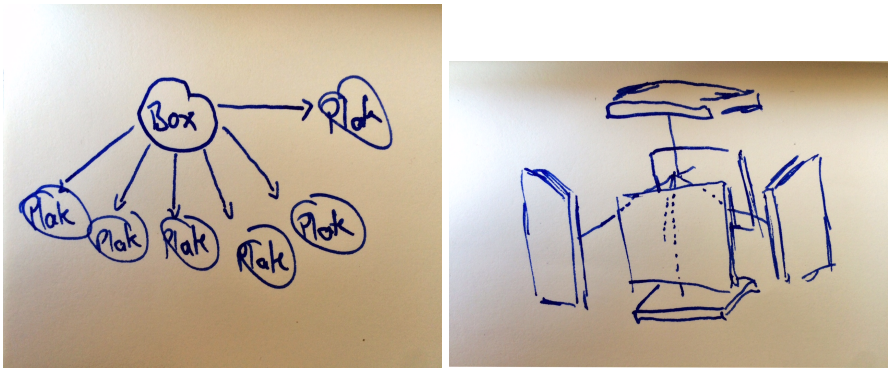


Figure 1: Touch input is processed by the render loop.

When an object is rendered on the screen, it is part of a scene. A scene is the visual space, in which the rendered objects can be seen. Instead of rendering each vertex of the model directly to the scene, we use representations of objects, which are organized in a hierarchical data structure. We refer to this hierarchical structure as scene graph. A simple scene graph is depicted in Figure **??**. The shown graph contains a box as single root node. The sides of the box are children of the root node. With this abstraction we can easily apply transformations to parts of the model only, without necessarily touching each

face or vertex. Every object, that is rendered by *Convertify* is part of a scene graph.



(a) Graph notation of the composed box.

(b) Rendered box, showing decomposition into plates.

Figure 2: A scene graph showing a box, composed of plates.

To enable full-fledged rendering results with sophisticated visual effects in web-browsers, we have to use the Web Graphics Library (WebGL). With the use of WebGL we can process data on the GPU and execute special effects code (shader code) [**?**].
On top of WebGL we use three.js[2]. three.js is a JavaScript library, which wraps the WebGL API and attempts to make the usage of WebGL simpler. It provides a scene graph implementation. three.js data representations are used throughout *Convertify*.

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

4.2 THE FRAMEWORK: CONVERTIFY

In this section we present *Convertify*, a web-framework for working with 3D-Models in three.js. The framework is based on previous work by **?** . We explain the work and concepts of **?** in Section 4.2.1. Then we outline, how we incorporated the previous work into *Convertify* in Section 4.2.2 and Section 4.2.3.

2 http://threejs.org

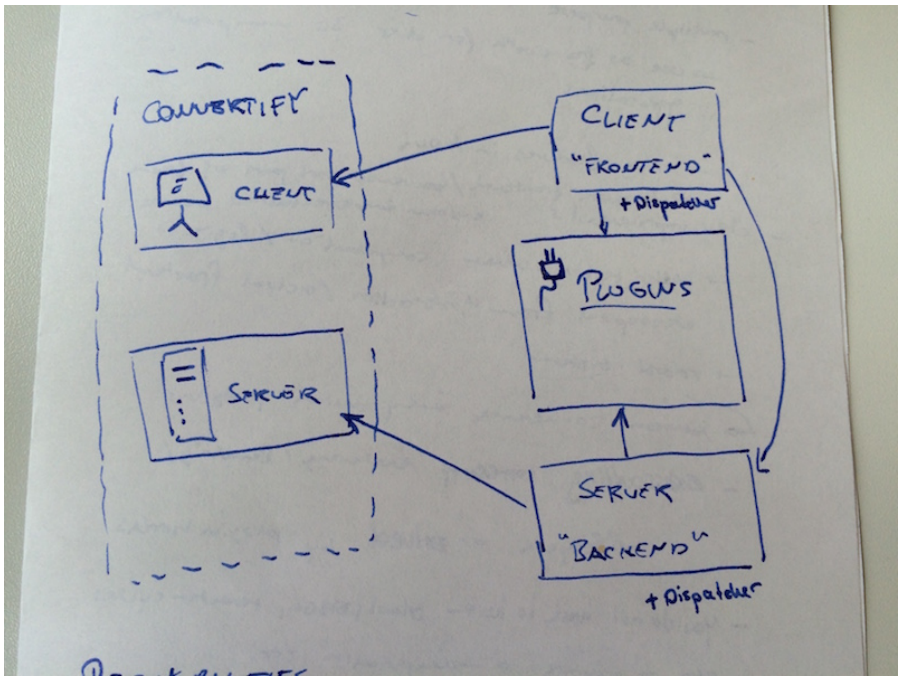


Figure 3: Main Packages of the Platener Architecture

4.2.1 *The Framework is Based on Brickify*

4.2.2 *The Framework Reuses Ideas and Components of Brickify*

4.2.3 *The Framework Provides an Enhanced Plugin System*

4.3 THE APPLICATION: PLATENER

4.4 ARCHITECTURE OVERVIEW

Platener is designed to enable web-based manipulation and rendering of 3D-Models. Figure 5 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[3] establish clear communication among packages.

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

4.4.1 *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[4] which serves the application's purpose. So the Client wires up the user-interface and the computation logic.

The *Server* package is the headless[5] counterpart to the Client package. A Command Line Interface enables the user to run the application without a browser. The Server also satisfies requests from the Client, such as caching and loading models in a RESTful interaction[6].

4.4.2 *Decoupling the Software into Packages*

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

4.4.2.1 *Convertify Is a Framework*

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

4 http://www.sitepoint.com/overview-javascript-templating-engines/
5 A headless web-application runs without the graphical user interface of browsers.
6 http://www.drdobbs.com/web-development/restful-web-services-a-tutorial/240169069

cross-platform web services. For example *Brickify* or *Laser Origami*ref could be implemented with *Convertify*.

4.4.2.2 *Plugins Establish Focus on Graphic Problems*

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 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.

4.4.2.3 *A Comparison with the Brickify Packages*

maybe put statement, rather then description into headline

The Convertify package is not present in the Brickify architecture, see Figure 6. 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 lockexplain vendor lock of previously used UX libraries. Libraries like add refs *jQuery* and *Jade* now pug scatter code throughout the project. By removing UX dependencies from logic components we create a thoroughly testable code base.

4.5 CONVERTIFY PACKAGE AND ITS PLUGIN ARCHITECTURE

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

It encapsulates basic logic for 3D-Model manipulation and model import. Additionally, we provide utilities to simplify working with *three.js* 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*.

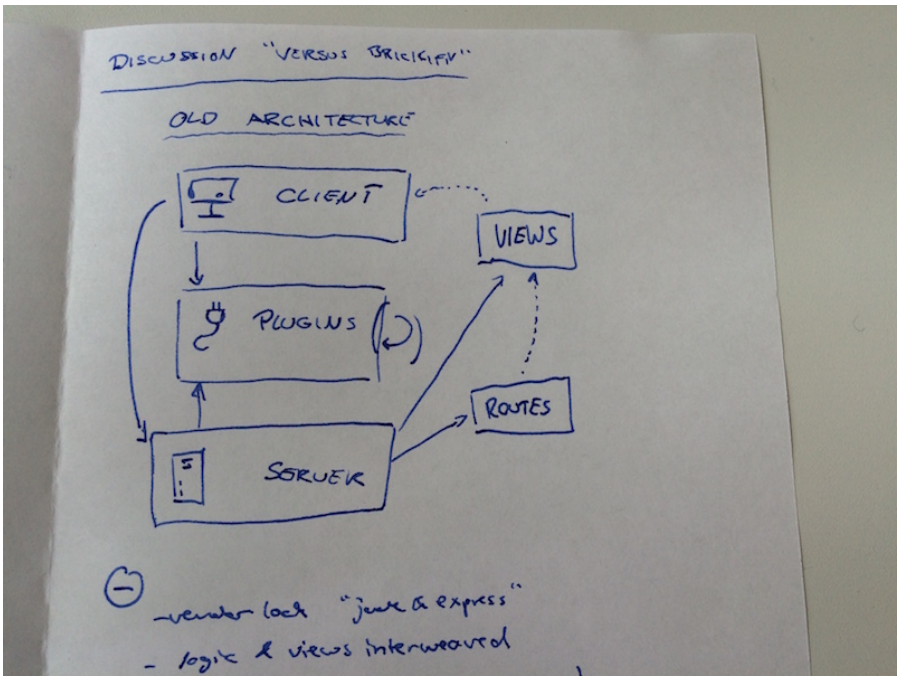


Figure 4: Main Packages of the former Brickify Architecture

4.5.1 *Scene Graph and Scene Management*

this is crap somehow The scene graph is built from *Nodes*. Each *Node* is associated with a single *THREE.Object3D* instance. This instance is rendered by the WebGL renderer. The *Node* is a high-level abstraction of objects in the scene graph. *THREE.Object3D* instances contain fine-grained sub-objects and actual vertices. For each Plugin we provide further *THREE.Object3D* instances, which belong to the *Node*'s *THREE.Object3D*.

Nodes belong to a *Scene*. Multiple *Scenes* are added to a *Project*. The *Project* is the single root node of the scene graph. *Nodes* are added and removed by the *SceneManager*, which is controlled in a *Bundle*.

4.5.2 *Lifecycle Events with Plugin Hooks*

4.5.2.1 *The Internal System Emits Lifecycle Events*

Similar to other applicationswhat applications? *Convertify* dispatches events from the internal system during the application's lifecycle. The internal system refers to framework features, which all applications built with *Convertify* share. Such features are model import, scene-graph manipulation or render updates.

add figure environment here

The lifecycle of *Convertify* is depicted in Figure add figure for platener lifecycle. First we initialize all activated plugins in the *init* event. During the *init3d* event each plugin is passed an empty *THREE.Object3D* instance. We call it the root node. The root node is associated with the *Node* in the scene graph. The plugins append the computed visuals onto the root node. All root nodes are rendered by the *Renderer*, calling the *update3d* callback. look up event name. When a *Node* from the *Scene* is selected by the user, we dispatch the *onNodeSelect* event. When a new *Node* is added to the *Scene*, the *onNodeAdd* hook is triggered. The *onNodeRemove* hook is triggered respectively when a *Node* is removed from the *Scene*. ref brickify

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

The subscribers to lifecycle events are *Plugins*. Each event type exposes a *PluginHook*. Each *Plugin* registers a callback on an arbitrary number of events. These callbacks get called when the event is dispatched by the system. Thus plugins can handle each event and apply their own functionality to the scene or even the model geometry. With this we emphasize compact computation units in plugins, which can still interact freely with the system. The approach is directly adapted from *Brickify* ref chapter in brickify.

4.5.3 *Plugins within Convertify*

A *Plugin* bundles methods and system event callbacks to provide a new set of features to the application, e.g. the *PlatenerPipeline* plugin builds a two-dimensional construction plan from the 3D-Model and exposes it for download. The *Plugin* can be switched on or off on application load for the Client and Server package respectively. Complex applications can be built by composing multiple plugins. This supports the framework character of applications built with Convertify.

4.5.3.1 *Plugins Are Packages of Their Own*

Each *Plugin* resides in the Plugin Package. We define a *package.json* file, which contains metadata of the package, see Listing 1. Conceptionally, a *Plugin* can be a separate *npm* package[7]. This means plugins could be installed like any other typical dependency. For the sake of development speed, we do not use the *npm* setup yet. ref brickify

To register a *Plugin* with *Convertify*, we have to add an entry in the *PluginMap*. The *PluginMap* is a static dictionary, enumerating all

7 Node Package Manager, https://docs.npmjs.com/getting-started/what-is-npm

```
{
  "name": "platener-pipeline",
  "version": "1.0.0",
  "description": "modifies model mesh to enable lasercuttable parts",
  "browser": "./PlatenerPipeline.coffee",
  "main": "./PlatenerPipeline.coffee"
}
```

Listing 1: *package.json* file of *PlatenerPipeline* plugin.

installed plugins and their filepaths. Because *Node.js* as well as *ECMAScript 6* do not support dynamic require statements[8], we have to list each package explicitly. In future systems we could alter the build system to include plugin files automatically before compilation step.

The exposed main file of the package implements a set of *PluginHooks*. These hooks are registered at the *Dispatcher*, which is responsible for dispatching the lifecycle events to the plugins in a predefined order, see chapter ??. Beside *PluginHooks*, a *Plugin* can expose *Protocols* to interact with the system. *Protocols* are described in chapter ??. A plugin implements any internal structure or complexity. The simplest form of a plugin is a single file. But it may provide a whole filetree. A list of plugins used in *Platener* is describe in chapter ??.

4.5.4 *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[9].

4.5.4.1 *The Dispatcher is a Mediator*

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, as shown in Figure 8. The *PluginLoader* instantiates each plugin and exposes all defined *PluginHooks* to the *Dispatcher*. this is all similar to brickify, ref please

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. Figure 7 shows in detail how the *Platener-*

8 A dynamic require statement can include and interpret source files into runtime where file paths are generated during runtime.

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

Pipeline plugin and the *Dispatcher* communicate via events. Codewise, the *Dispatcher* implements each plugin hook and remits the event to the loaded plugins.

```
class ClientDispatcher extends AbstractDispatcher

  #
  # ...
  #

  hooks:
    init: (bundle) ->
      return @callAllPlugins('init', bundle)

    onNodeAdd: (node) ->
      return [
        @callPlugin('solution-selection', 'onNodeAdd', node),
        @callPlugin('node-visualizer', 'onNodeAdd', node)
      ]

    onNodeRemove: (node) ->
      return [
        @callPlugin('node-visualizer', 'onNodeRemove', node)
        @callPlugin('solution-selection', 'onNodeRemove', node)
      ]
}
```

Listing 2: The *ClientDispatcher* implements plugin hooks to remit lifecycle events.

Listing 2 shows, how the *ClientDispatcher* client dispatcher explained?? handles the *init*, *onNodeAdd* and *onNodeRemove* hooks. By remitting events manually, we have fine-grained control about the execution order. This is necessary, because we want to invoke the *PlatenerPipeline* plugin, computing the construction plans, before rendering the results via the *NodeVisualizer* plugin. Also we have to remove the *NodeVisualizer* results before deleting the data, to avoid reference errors. For both *PluginHooks* the plugins are executed in a different ordering.

To allow communication between plugins themselves or plugins and Client, we use *Protocols* because *PluginHooks* are not flexible enough. A *Protocol* defines an interface, which has to be implemented by a

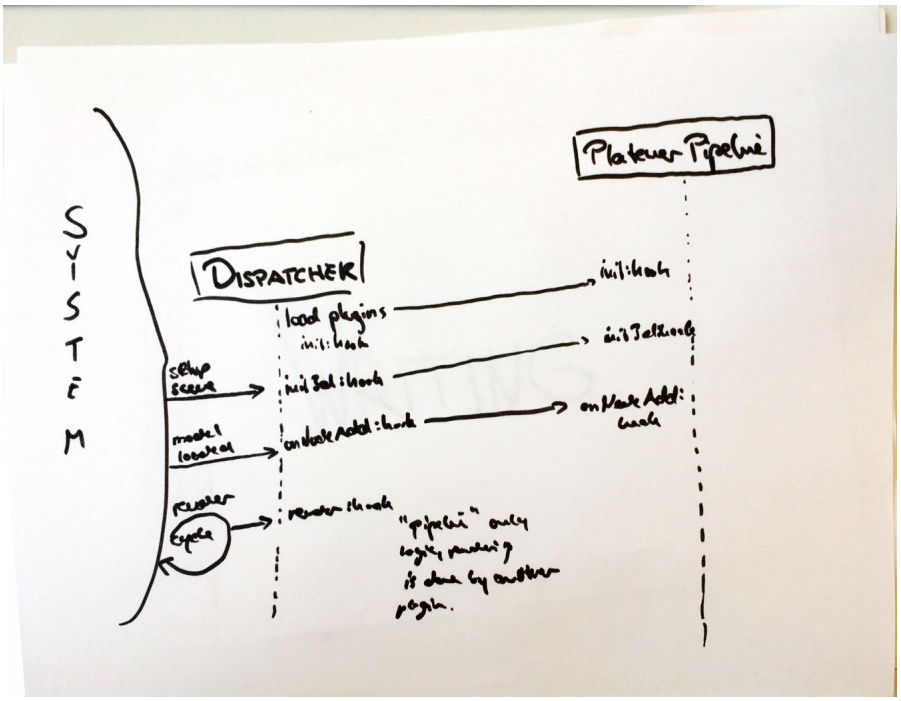


Figure 5: Dispatcher and PlatenerPipeline communicate via lifecycle events.

delegate. Via delegation[10] we define callbacks which are invoked by Plugins on the Dispatcher. The Dispatcher then notifies server, client or other plugins. Furthermore, *Protocols* can add a set of methods to the Dispatcher via meta programming.

Listing 3 shows the definition of the SolutionSelectionDelegate protocol. It requires its implementing class to provide the methods *evaluationDidStart*, *evalutionOfMethodDidFinish*, *evaluationDidFinish*, *evaluationDidFail* and *evaluationDidFailWithSolutions*. Listing 4 shows the *ClientDispatcher* implementing the *evaluationDidFinish* method. By providing the callbacks, the Dispatcher knows about the current state of the conversion and its completion. For example, we render all solutions after their computation. To satisfy the SolutionSelectionDelegate protocol, the Dispatcher also implements the remaining methods.

4.5.4.2 *The Dispatcher is the Single Source of Truth*

When applications grow, it is hard to observe all messaging between components at once. With the *Dispatcher*, we control all communication between the packages at one place. Without implicit ordering of callback executions, we can trace erroneous behavior fast. The design of a singleton controlling the application's lifecycle is known from

10 http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/delegation.html

```
SolutionSelectionDelegate = {
  shouldImplement: [
    # Invoked before a method is evaluated.
    'evaluationDidStart'
    # Invoked after each method finished.
    'evaluationOfMethodDidFinish'
    # Invoked after all methods finished.
    'evaluationDidFinish'
    # Invoked when an error occured in solution selection.
    'evaluationDidFail'
    # Invoked when errors occur in pipeline steps during computation
    'evaluationDidFailWithSolutions'
  ]
}
```

Listing 3: *SolutionSelectionDelegate* protocol definition

```
class Dispatcher extends AbstractDispatcher
  ### Solution Selection ###
  @protocol(SolutionSelectionDelegate)

  #
  # ... more protocol methods ...
  #

  evaluationDidFinish: (solutions) ->
    @getPlugin('node-visualizer').render(solutions)
```

Listing 4: *ClientDispatcher* implements the *SolutionSelectionDelegate* protocol

frameworks in mobile application development, like Android**??** or iOS**??**.

The usage of *Protocols* helps developers keep track of interfaces between components. We enforce this concept, because JavaScriptis a dynamically typed programming language. Statically typed programming languages pay out, when projects grow large. Unexpected type and interface errors happen less often. Because of the web-environment and previous work, we are bound to CoffeeScript. Thus, we emphasize explicit structuring of interfaces via *Protocols*.

4.5.4.3 *Dispatcher and Bundle on Client and Server*

maybe put this up, so we understand earlier what the ClientDispatcher is

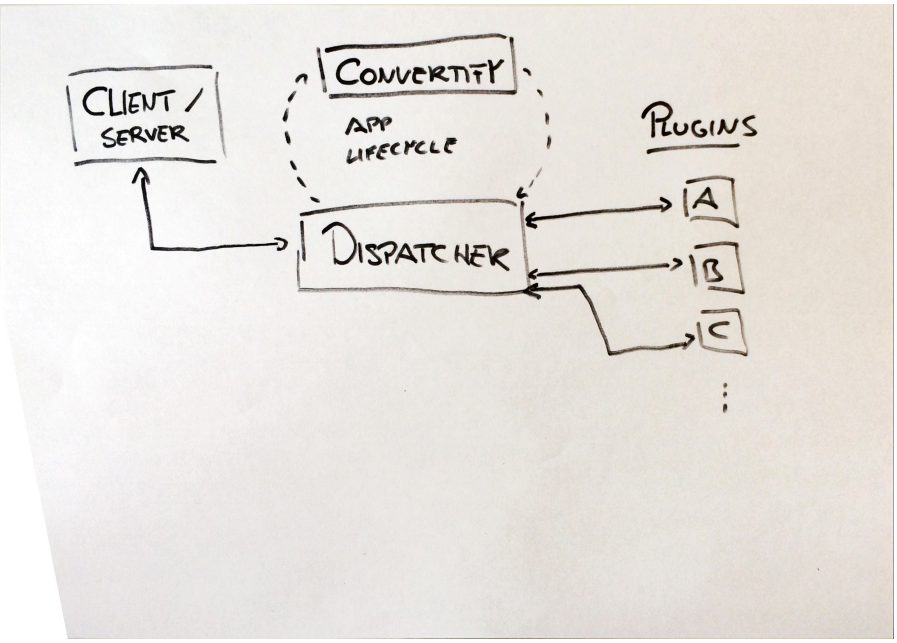


Figure 6: Dispatcher manages inter-application communication during the lifecycle of the app.

The Server and Client package handle lifecycle events differently and use different protocols. That is because the Client package has to handle rendering and user interaction. The Server package merely computes the manipulated models and is used for batch processing of 3D-Models. Therefore, we need two Dispatcher implementations for the client and for the server.

A Bundle is the entry point for any client or server code. As the name indicates, a Bundle bundles all application code into a single instance. It references the specific *ClientDispatcher* or *ServerDispatcher*, which are implemented in the Client or Server package respectively. Thus we can control the system by invoking protocol interfaces. The ClientBundle is exposed to the Client package. The ServerBundle is exposed to the ServerPackage.

4.5.4.4 *Brickify uses PluginHooks only*

Brickify dispatches plugin hooks directly from the internal system to the plugins without using a mediator in between. Compared to *Convertify*, it does not require any boilerplate code to setup the communication. *Convertify* needs a custom integration of each *Plugin* into its *Disptacher*.

On the other hand, *Brickify* could not control the ordering of *PluginHook* invokations. It is implicitly defined by the loading sequence of plugins. Furthermore, the frontend code needs to access plugin data. The data is only available after the plugins are loaded or even specific

hooks were executed. Via a mediator we can notify the frontend about state changes, rather than asyncronically trying to poll the data.

With the introduction of a *Dispatcher*, *Plugins* now can push state back to the system and implement *Protocols* for more control over the communication.

4.6 PLUGINS IN PLATENER

4.6.1 *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.

4.6.1.1 *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 9 shows the coordinate system in the WebGL view. The Coordinate System is taken from *Brickify* as is[11].

4.6.1.2 *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.

4.6.1.3 *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.

add figure of a visualized model in wireframe mode (something fancy)

11 ref original code file

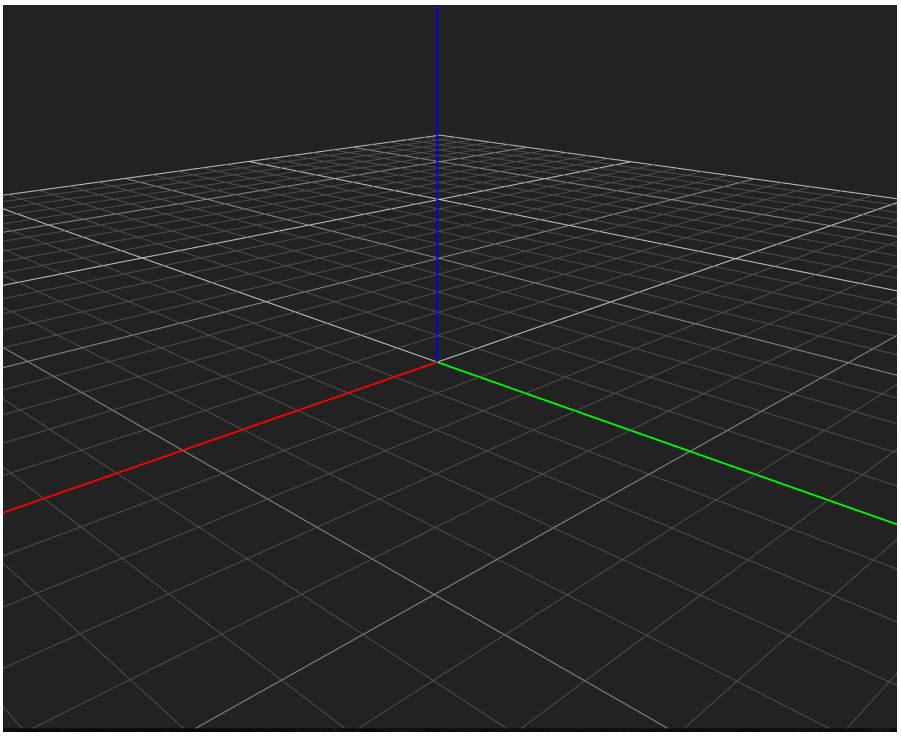


Figure 7: An empty scene showing the coordinate system.

4.6.1.4 *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.

4.6.1.5 *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.

4.6.1.6 *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.

4.6.2 *Platener Pipeline*

1. Overview/ Purpose/ Plugin Structure
2. conversion strategies and logic
    - logic only (ui -> other plugin, frontend)
    - processing broken into steps
    - composable into fabrication methods = execute steps in sequence, use results from previous step, to go on = conversion strategy
    - typical pipelining problem look up typical pipelining problems
    - in general we have to: clean mesh, indexing, find structure (understand the model), convert structures (plates and joints), export
    - different methods may to these things differently (stacked plates vs. inherent plates)
3. Pipeline Steps

    What are pipeline steps?

    - processing unit on model data (or abstraction of it)
    - indepentend from each other, a single task
    - break down problem into sequential sub problems (working with abstractions, generating assumptions -> understand problems better, work in parallel)
    - improve debugging experience (trace errors in decouple computation units, also its hard to see how vertices are from looking at the memory, –> see next bullet)
    - store state of each step separately, to later render a debugging view of the state (like timetravel for manipulations on the model)
    - but keep rendering apart from logic -> NodeVisualizer
    - isolated computing -> isolated testing (nice!)

    How are they used in Platener Pipeline?

    - pipe steps -> build up a pipeline from composition
    - final result will be exported from the browser
    - a pipeline makes up a conversion strategy
    - conversion strategy share certain subproblems -> reusable code

- as steps solve the real algorithmic problems, we dedicated separate chapters to them. look at chapters ??

Where to be found in the code?

- Plugins Package
- organized in subdirs (remember, separate npm package)
- step factories -> configure steps for different strategies, inject dependencies and state, lazy loading

4. Fabrication Methods
   a) Plate Method
      - hull and surface constructions
      - find plates in the model (inherent), build plates from surfaces (extruding)
      - intersect plates (graphing)
      - and join connected plates (finger joints)
      - figure here
      - listing showing how to concat/ pipe steps
      - main parts of algorithm explain in chapters lukas, klara, dimitri, deus
   b) Stacked Plates Method
      - volume approximation
      - stack flat plates to rebuild the object
      - preserve form, look and feel, details
      - loose functional aspects
      - connect via shafts (click together) or use glue
      - figure here
      - main parts of the algorithm explained in chapters lukas
   c) Classifier Method
      - advanced technique, combining intense mesh analysis and construction techniques of known geometries
      - pros: convert noisy models, local conversions: because different local geometries may be better converted with different techniques, e.g. stacked vs. fingerjoints. we can evaluate better (scoring locally also!). global conversions and evaluations are error prone. classifier method could provide a rather robust conversion strategy. WHY CLASSIFIERS?

- work in progress
- we can currently use primitive detection algorithms
- figure of detected cylinder in model
- main parts of the algorithm at the end of paper (classification chapter, ransac stuff)

5. Immutable Pipeline State

   a) Overview / Purpose

      - no unwanted mutations (nice persisting of intermediate results from pipelinesteps)
      - later steps work on data form previous steps
      - code was not written immutable throughout the project -> changing data will result in wrong visualizations of previous steps by the node visualizer, because rendering performs AFTER all computations are done

   b) Immutability

      - explain principle. have more pros and cons.
      - structural data sharing -> unfortunately, not for us yet, but thats the concept :)
      - http://jlongster.com/Using-Immutable-Data-Structures-in-JavaScript
      - http://stackoverflow.com/questions/10034537/persistent-vs-immutable-data-structure
      - enforces certain code style, mutability may be easier to write
      - slow downs maybe (performance)

   c) Implementation Details

      - PipelineState class
      - mutable and immutable state properties
      - providing a clone method for stored properties, because code throughout the project shall not necessarily be written immutable (hard to obey, –> cg dev orientated, c++ is not immutable)
      - immutable properties implement the clonable protocol
      - provide immutable in one spot -> state creation
      - define a schema (all props always available, having defaults) -> reduce faults when accessing data
      - add listing which defines the schema for stacked plates

- accessing undefined values will give us a warning (again, like protocols enforcing interface structure, reducing faulty data access)
- somehow a persistence api (see stackoverflow link)

6. Pipeline Implementation

   a) the pipeline works as follows...
      - Pipeline class
      - pipe interface for Step factories (compare listing above, stacked method)
      - create state factory -> use composition to persist intermediate results (state is later accessed by node visualizer)
      - reduce approach -> pipe last state into next step to produce new state
      - show a listing of pseudo code, explaining how the pipeline works
      - benchmarking (measure time for intermediate steps vs whole processing)
   b) it can be reused (because pipelining is a generic concept)
      - classifier graph (classification method)
      - isolated testing (injectable pipeline)

4.6.3 *Solution Selection*

1. WHAT?
   - Selecting the best estimated solution by default.
   - execute all classification methods
   - use scorer plugin to evaluate each conversion after computation (globally evaluated, local converted geometries could be really bad)
   - provide list of solutions associated with a score
   - solution with max score is provided as default download
2. WHY?
   - strategies not mergeable in current state (would be future work to bring them together, like classifier method approach)
   - so we want to select the method as default, which may has done the job most correctly (scoring)

- also scoring helps to get an idea of the quality of conversion without looking at the result (visually) -> used in headless mode, batch processing we could detect poorly converted objects to have a detailed look of what went wrong (getting development forward, yay!)

3. MEASUREMENTS and HOW SCORING?
   - look at screenshots and describe in short what we thought, would be nice measurements
   - scoring plugin: per measurement per classification method -> score method
   - then adding all measuremnt scores per classification together
   - assign score to the solution
4. HOW SOLUTION SELECTION?
   - combining plugins platener pipeline and scorer
   - promised -> loop overall pipelines of classification methods
   - promised -> then apply scoring
   - when failure, catch and report nicely in evaluationDidFail hook (so fronend can notify users, no hard failure)
   - results pushed back to system via evaluationDidFinish hook

4.6.4 *Node Visualizer*

1. Overview
   - meshlib library for model representation
   - convert meshlib face vertex mesh to three js notation
   - render imported model and its manipulated forms with threejs
   - use data from platener pipeline plugin
   - the dispatcher implements solutionselectiondelegate protocol, which lets the node visualizer render the data after it was computed (see listing ??, above)
2. Visual Debugging
   - Rendering into WebGL only in NodeVisualizer
   - other plugins could also do it
   - but keep visualization of processed model in one place

- for each pipeline step, access its state data in a separate visualization
- step through, like explained in walkthrough chapter

3. Visualizer and Visualizations
   - Visualization class, constructs a drawable (THREE.object3d)
   - injects drawable into root node of NodeVisualizer plugin
   - Renderer will render it
   - Visualizer class, wrapper managing visibilities of visualizations
   - toggable through frontend
   - visualizersets -> group visualization for a fab method (not all fabmethods have all steps, thus dont need all visualizations)
4. Interwoven with Platener Pipeline
   a) Role of Immutability
      - (show example of how shapes have fingerjoints)
      - shapes (find outlines of possible plate geometries)
      - fingerjoints are added to shapes in a later step
      - shapes visualization also shows fingerjoints now (no timetravel - bad debugging (we cannot see if shapes did something wrong))
      - data will not be changed if immutable (yes!!)
   b) Extracting Intermediate Data of the Pipeline
      - we access solutions (state per fabmethod)
      - PipelineVisualization parent class
      - convert state to plain object for fast access
      - listing for PipeVis
      - each visualization extending pipeline viz only has to implement a single method, returning a drawable threejs object

4.6.5 *Isolated Testing*

1. WHY?
   - validation of data/ conversion

- parallel working (make mock data with certain assumptions)
- isolated testing in real environment (test env is not representative, integration tests with manual evaluation (look at the visual results))

2. Static Input
   - build data like assumed for the step we test
   - not that easy: many cross references, after certain steps, complex data...
   - hard to build by hand, we had to use previous steps and serialize the step data
3. Testables
   - 1 step, 1 static input, certain test purpose
   - a single integration test
   - e.g. classify a single shape as plane in classify geometry step
4. Implementation
   - reusing concept of pipeline
   - small changes: allow only a single step, get input from testable
5. Conclusion
   - not perfected/ technically mature
   - mostly we evaluated by dropping in models
   - but wrongly executed previous steps may have corrupted our assumptions
   - thats why working out nice testables would be worth it!

4.7 CLIENT PACKAGE

4.7.1 *Overview - Custom Frontend Code*

- when convertify framework, client is freespace to evolve yaself
- look and feel of the application
- task: connect logic to ui (speak with dispatcher) and build ui
- free choice of frontend framework (we take redux + react), but nothing against jquery or angular or backbone or ...

- e.g. laser origami or brickify would choose a completely different implementation of client -> custom per application
- react by facebook, redux by dan abramov, like flux architecure -> uni directional dataflow, explicit state changes, data driven -> reduce side effects and be efficient in coding and trace down errors easily
- diagram which shows benefits of flux architecture vs no flux arch, look at intro vids for redux from dan abramov

4.7.2 *React Templates*

- show html tree graph and how data communication goes wild when talking with siblings
- ideally dumb components (dont know where data is coming from, just display it)
- stateless, components directory
- WIP: before no redux, so there are some mixed up components left :S
- show short example how a dump component looks like, listing!

4.7.3 *Redux Data-driven Control Flow*

1. Redux 'dispatch' and state
   - one state container
   - functional, no side effects
   - maybe copy or reference redux description (just explain why its awesome)
   - injected into react via context
2. Smart Containers
   - connect to state
   - containers know where data is from (vs dump components)
   - fitler and preprocess raw data, setup interaction events to trigger actions
   - give data and callbacks to a component (they setup the actual ui, but contain no visible elements themselves)
   - listing show how to connect and use component from other listing
3. Manage Async Plugin Hell

- as described before, plugin data is not available on load
- we can use Dispatcher and redux dispatch combined
- protocols -> state change in plugins -> dispatch -> state change in frontend
- no polling or observing of data, fully reactive (system -> client communication)
- as client has access to bundle, we can call interfaces exposed by protocols (client -> system communication)

4.8 SERVER PACKAGE

4.8.1 *Overview - Custom Server Code*

- like client, can have custom implementations
- we have caching and cli (headless version of application)
- requires isomorphic code: execute on client and server equally
- isomorphic means...
- threejs, polyfills, ...

4.8.2 *Model Cache*

- idea: build up repository of models when users interact with it
- taken from brickify: uploading meshlib version of model

4.8.3 *Test Pipeline*

1. WHY do we have a Test Pipeline?
   - robustness tests
   - batch processing
   - headless version for integration with other projects
   - failure detection because of diversity of objects
2. Headless Conversion of Objects
   - run solutionselection plugin also
   - but dispatcher is setup a bit differently
   - no recomputation, no grid, no visualizer

- scene manager will not render anything (unless, WIP we exchange webgl rendering with headlessgl to produce screenshots of each conversion)
- cli tool -> safe results into directories

3. Reports
   - = extended console logs
   - show how conversion was going
   - display failures, status, progress
   - in the end: sum up + give stats
   - WIP: current problems: not all conversions are garbage collected correctly, will run out of memory after some conversions -.- (maybe nobody has to know about that)
4. Benchmarks
   - xxx testmodels
   - we proposed mostly stacked as the best solution
   - too many arbitrary forms hindered plate conversion
   - just shows conversion stats (maybe all models vs box category only)
   - measure times for conversions and evaluate

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 10 is simplified with an unusually high welding distance of 10mm in Figure 11. 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.

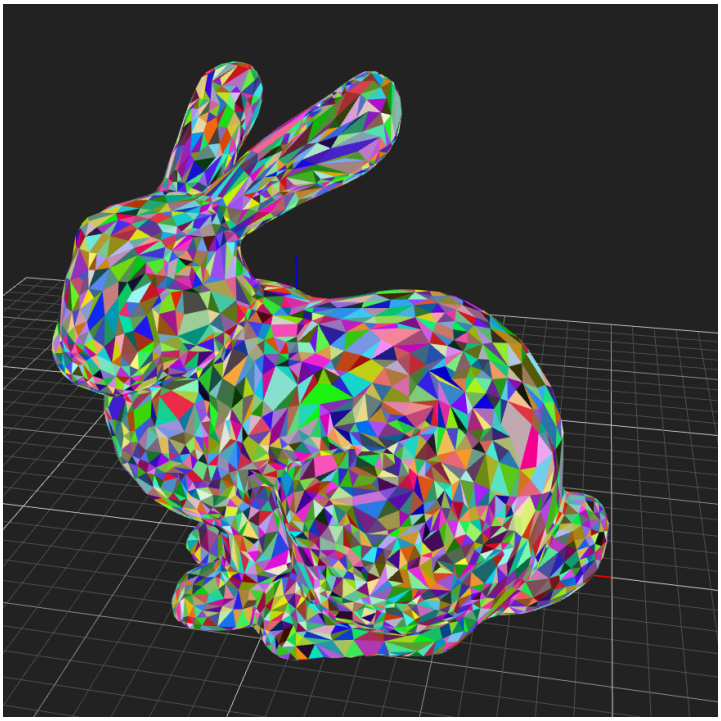


Figure 8: Stanford Bunny with 8662 faces

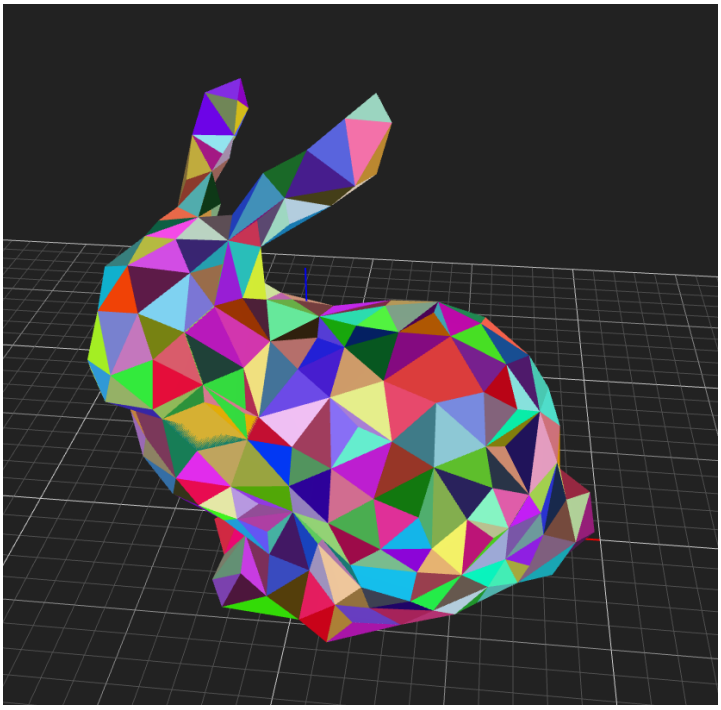


Figure 9: Simplified Stanford Bunny with 616 faces (welding distance: 10mm)

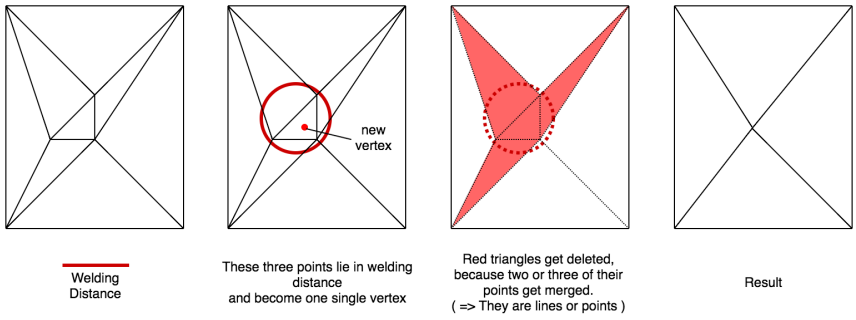


Figure 10: Vertex Welding

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 13:

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 13 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.

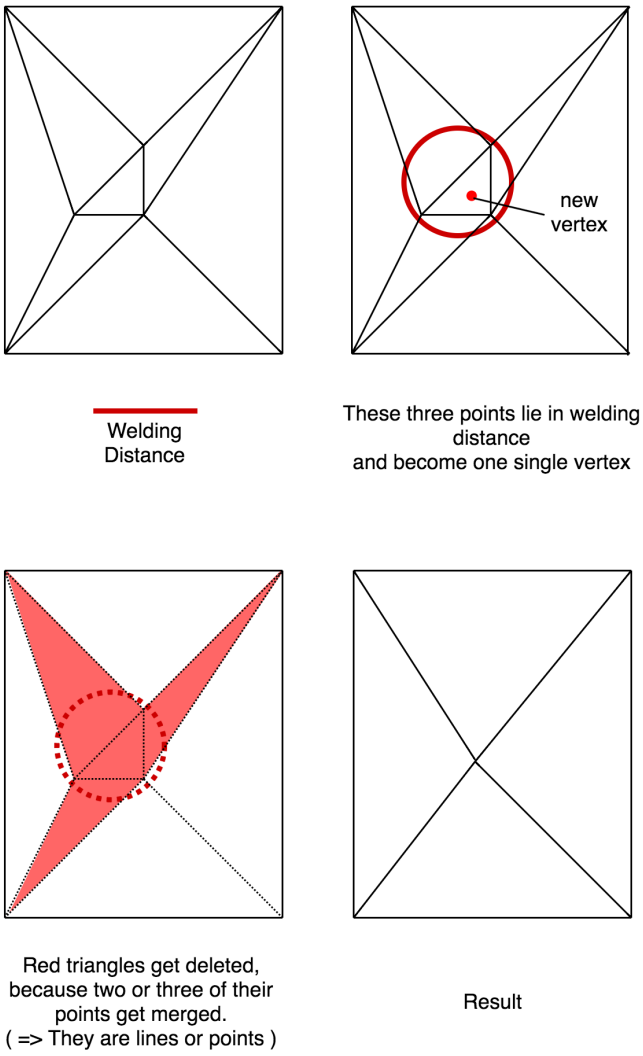


Figure 11: Vertex Welding

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?

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: Implementation

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 14. Then *VertexWelding* 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.

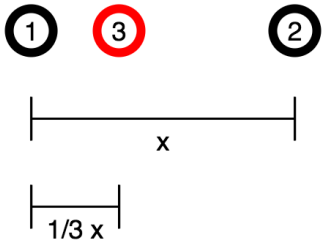


Figure 12: Resulting point (point 3) of welding a WeightedVertex with weight=2 (point 1) and point 2

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*

VERTEX REPLACEMENT The actual vertex replacement is done manually for each point with *getCorrespondingVertex*.

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 represent-

Figure 13: Original Makerbot letters

ing 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 16. With a welding distance of 3mm the result is a smooth surface while the original model shown in Figure 15 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 17.

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...

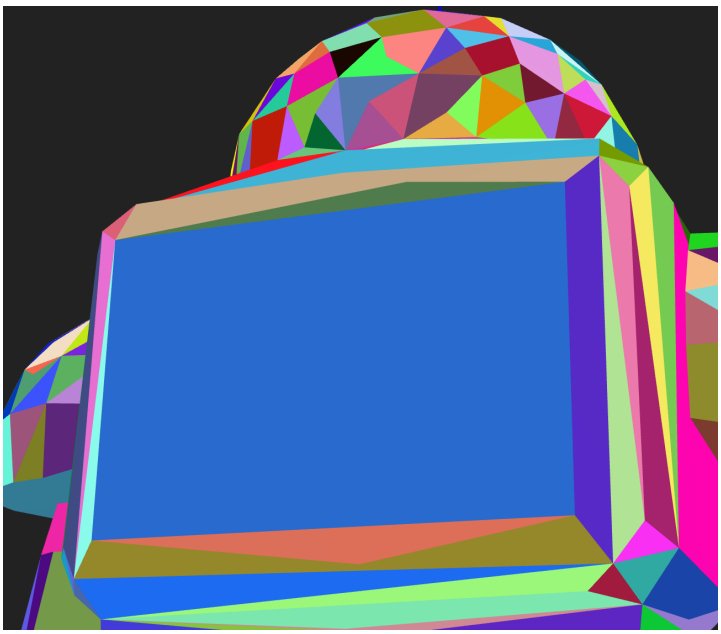


Figure 14: Simplified Makerbot with welding distance 3mm - letters completely removed

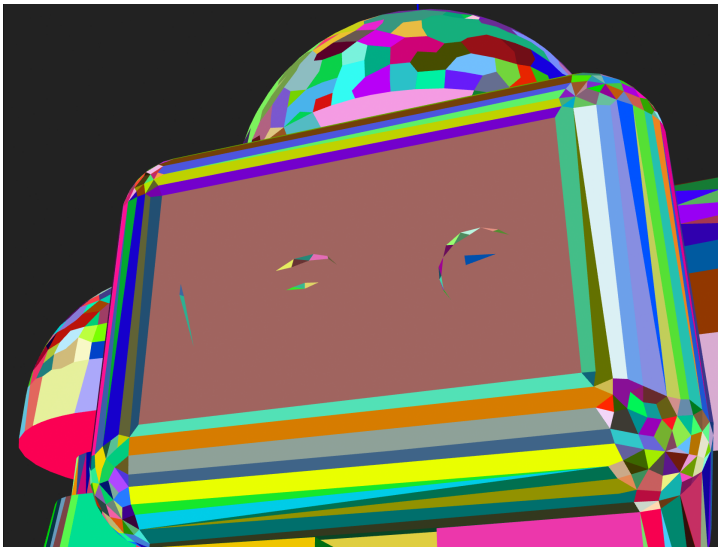


Figure 15: Simplified Makerbot with welding distance 0.8mm - artifacts

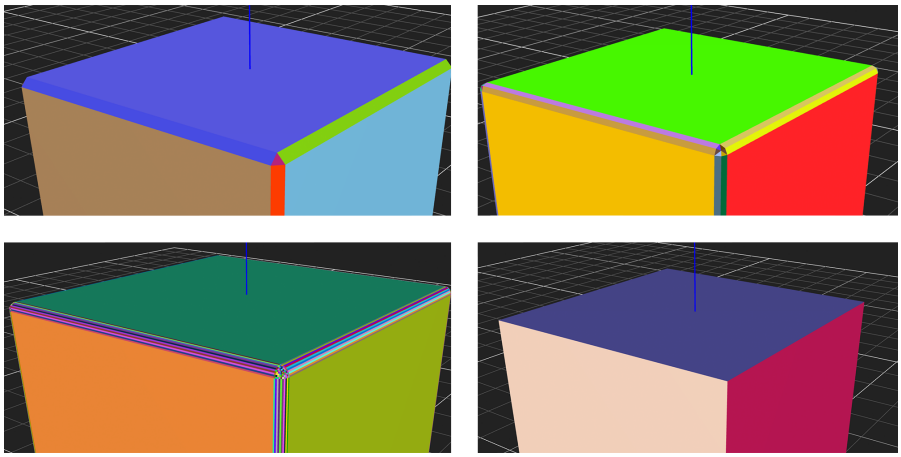


Figure 16: Beveled cube with 1, 2, 10 subdivisions and their resulting cube without beveled edges

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

Figure 18 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 19 there is text on top of the actual surface and in Figure 20 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

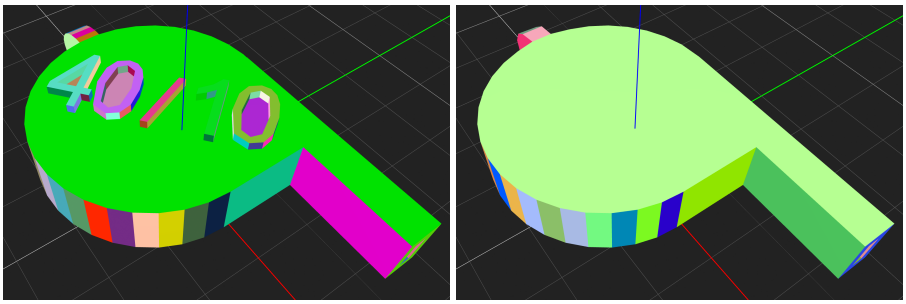


Figure 17: Extruded details

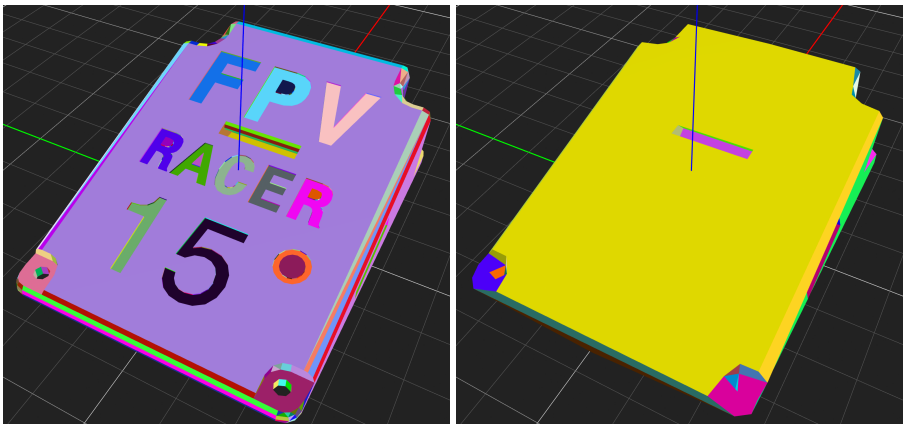


Figure 18: Pushed in details

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

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 21. 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

1 https://www.blender.org

Figure 19: Vertex Welding UI Element

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.

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

6.1.4 *Alternative Methoden*

6.1.5 *Warum diese Methode gewählt?*

6.2 POINT ON LINE REMOVAL

6.3 VLLT: REMOVE CONTAINED PLATES - IN LUKAS KAPITEL

6.4 PRISM CLASSIFIER - IN CLASSIFIER KAPITEL

6.5 HOLE DETECTION - IN PLATES KAPITEL

7

PLATES

7.1 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.

7.2 PREREQUISITES FOR FINDING PLATES

coplanar / shapesfinder / holedetection

7.2.1 *Coplanar Faces*

fvdfsjkldf

7.2.2 *Shape Finder / Deus*

7.2.3 *Hole Detection / Dimitri*

sdfjkljklsdfsdfjkl

7.3 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.

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

```
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
```

Listing 5: Plate candidate pseudo code.

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.

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.

7.4 EXTRUDING PLATES

blabla

7.5 REMOVING CONTAINED PLATES / DIMITRI

dfgdfgdfgdfgdfg

7.6 STACKING PLATES

blablabla

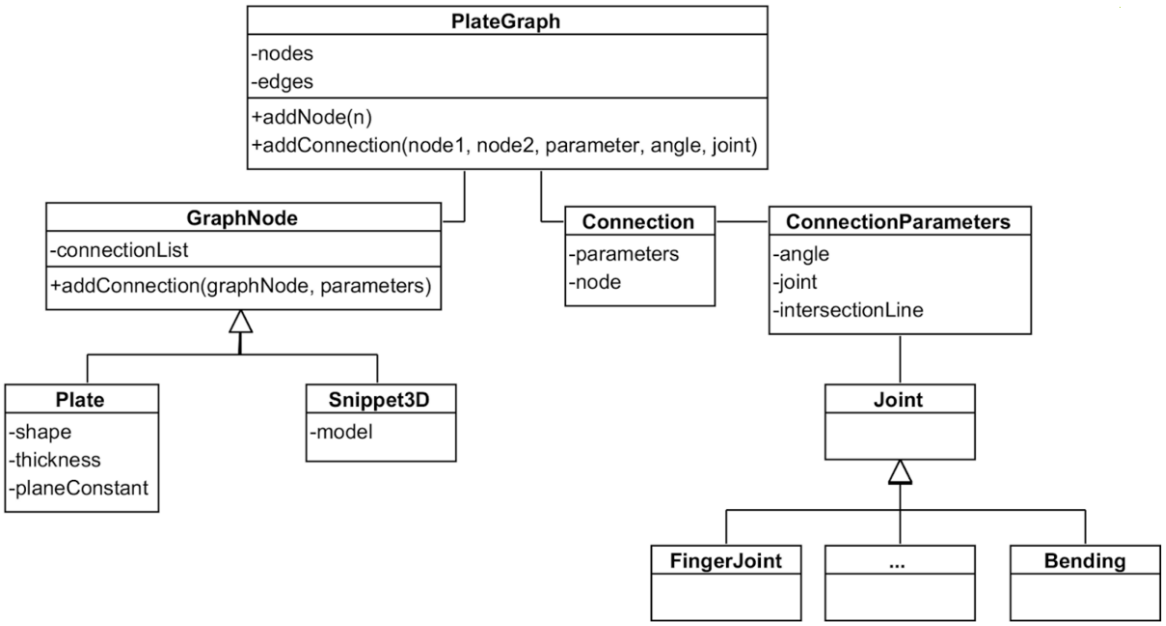
8

ADJACENCY PLATEGRAPH / KLARA

On the basis of the graph we can create connectors for plates in a later step (9). 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 7. 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.



8.1 ANALYSING SPATIAL ARRANGEMENT

As a prerequisite the step 7 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 9 we truncate the inner intersection lines which would otherwise overlap with adjacent plate intersections.

8.1.1 *Finding intersections*

When two planes intersect there is an intersection line. Since we work with plates which 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

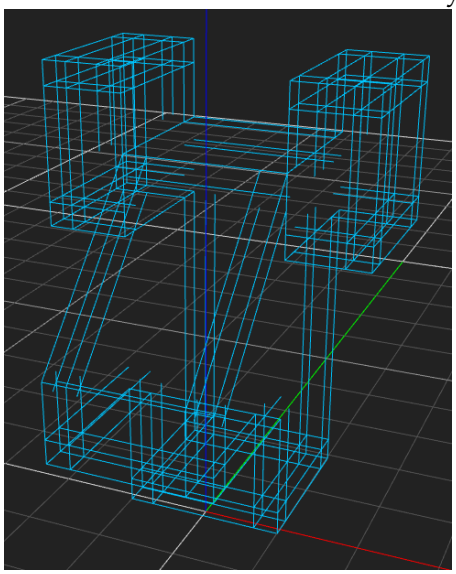
$$p = \frac{d_1 * n_2^2 - d_2 * (n_1 * n_2)}{n_1^2 * n_2^2 - (n_1 * n_2)^2} * n_1 + \frac{d_2 * n_1^2 - d_1 * (n_1 * n_2)}{n_1^2 * n_2^2 - (n_1 * n_2)^2} * n_2$$

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.



8.1.2 *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

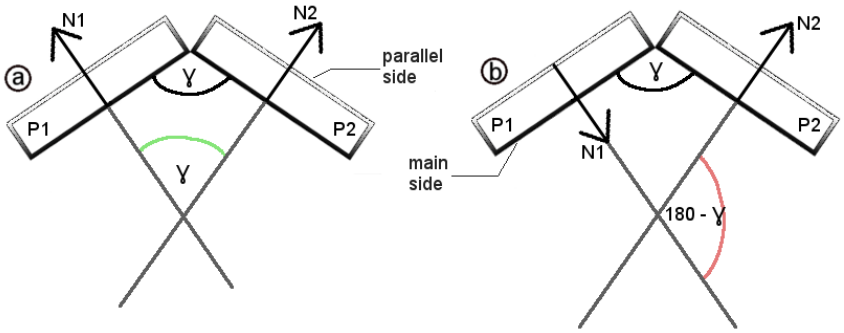


Figure 20: (a) The angle between the plates' normals correspond with the angle γ between the plates. (b) The angle between the plates' normals is in this case an adjacent angle to the requested angle γ.

- Their directions are both positive OR both negative

8.1.3 *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.

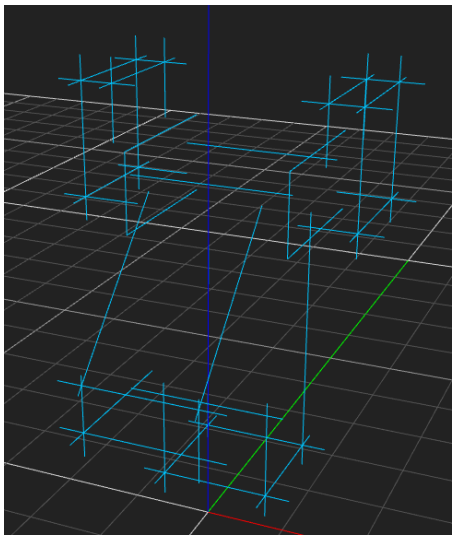


Figure 21: The inner boundaries of the plates overlap. In the next step when joints will be created they are only supposed to be on the inner part of the line. Therefore the lines have to be truncated.

Algorithm 1: 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 }
```

8.2 ALTERNATIVE SOLUTIONS

8.2.1 *Dustin: how he did it*

8.2.2 *Dustins wrong angles*

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

8.2.3 *Down sides*

8.2.4 *Whats better now?*

8.3 HOW WE GOT THERE

8.3.1 *Floating Point inaccuracy*

8.3.2 *Bruteforce finding lines*

8.3.3 *different line intersection algorithms*

8.4 FUTURE WORK

- find out if model is assemblable in the end - rejoining broken up plates (t-connection)

9 JOINT COMPUTATION

9.1 JOINT COMPUTATION

9.1.1 *Volume based clipping*

PREREQUISITE: PLATEGRAPH INTERSECTION LINES

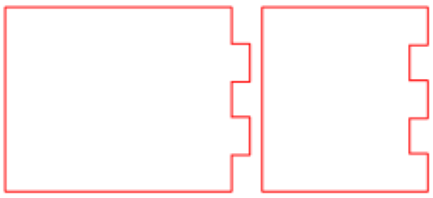
9.1.2 *Female Joint Computation*

9.1.3 *Male Joint Computation*

9.1.4 *Different Fingerjoint types*

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

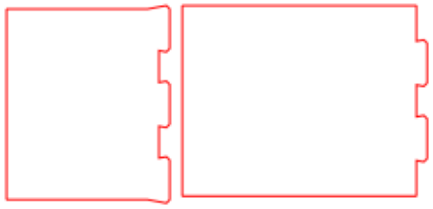
9.1.4.1 *Fingerjoint template*



HOW THESE JOINTS LOOK LIKE

HOW THESE JOINTS WORK, AND FOR WHAT MATERIAL

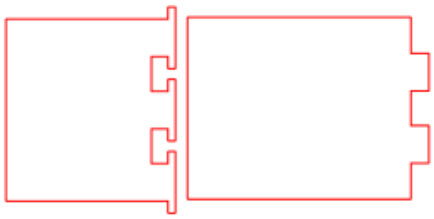
9.1.4.2 *JimJoint template*



HOW THESE JOINTS LOOK LIKE

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

9.1.4.3 *Schwalbe template*



HOW THESE JOINTS LOOK LIKE

HOW THESE JOINTS WORK, AND FOR WHAT MATERIAL

9.1.5 *adjusting fingerjoints length when plates are angled*

DEUS

9.1.6 *Alternative solutions*

9.2 FUTURE WORK

10

CURVES/DEUS

10.1 CUTTING CURVED SHAPES

Approximating curved shapes with parts created by the laser cutter is a special challenge because only 2d shapes can be cut. 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.

10.2 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.

10.3 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.

10.4 BUILDING THE BENT PLATES

10.5 ALTERNATIVE SOLUTIONS

11

ASSEMBLY

11.1 somebody will have to do the following sections shortly

11.2 what we currently have (not good solution)

11.2.1 *Plate-method*

just write numbers on lines -> equal numbers belong together

11.2.2 *Stacked-method*

number plates

11.3 what might be better, but is not implemented

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

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

12

BENCHMARK

12.1 AASDF

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

13

CLASSIFIERS

13.1 CLASSIFIYING IDEA

13.2 RANSAC

13.3 PRIMITIVES

13.3.1 *cylinder*

13.3.2 *Plane*

13.3.3 *Prism - Dimitri*

14

FUTURE WORK

14.1 ULTIMATE GOAL

14.2 CLASSIFIER

how this is gonna be soooooooo cool

15

CONCLUSION

15.1 AASDF

15.2 USER TESTING

15.3 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