

TIM OESTERREICH

BUILDING LARGE DYNAMIC TRUSS STRUCTURES



# BUILDING LARGE DYNAMIC TRUSS STRUCTURES

TIM OESTERREICH



Using TrussFormer  
July 2018 – version 4.5



*Ohana* means family.  
Family means nobody gets left behind, or forgotten.  
— Lilo & Stitch

Dedicated to the loving memory of Rudolf Miede.

1939 – 2005



## ABSTRACT

---

Short summary of the contents in English...a great guide by Kent Beck how to write good abstracts can be found here:

<https://plg.uwaterloo.ca/~migod/research/beckOOPSLA.html>

## ZUSAMMENFASSUNG

---

Kurze Zusammenfassung des Inhaltes in deutscher Sprache...



*We have seen that computer programming is an art,  
because it applies accumulated knowledge to the world,  
because it requires skill and ingenuity, and especially  
because it produces objects of beauty.*

— Donald E. Knuth [1]

## ACKNOWLEDGMENTS

---

Put your acknowledgments here.

Many thanks to everybody who already sent me a postcard!

Regarding the typography and other help, many thanks go to Marco Kuhlmann, Philipp Lehman, Lothar Schlesier, Jim Young, Lorenzo Pantieri and Enrico Gregorio<sup>1</sup>, Jörg Sommer, Joachim Köstler, Daniel Gottschlag, Denis Aydin, Paride Legovini, Steffen Prochnow, Nicolas Repp, Hinrich Harms, Roland Winkler, Jörg Weber, Henri Menke, Claus Lahiri, Clemens Niederberger, Stefano Bragaglia, Jörn Hees, Scott Lowe, Dave Howcroft, José M. Alcaide, and the whole LATEX-community for support, ideas and some great software.

*Regarding LyX:* The LyX port was intially done by Nicholas Mariette in March 2009 and continued by Ivo Pletikosić in 2011. Thank you very much for your work and for the contributions to the original style.

---

<sup>1</sup> Members of GuIT (Gruppo Italiano Utilizzatori di TEX e LATEX)



## CONTENTS

---

List of Figures	xiii
Listings	xiv
<b>1 INTRODUCTION</b>	<b>1</b>
1.1 TrussFab	1
1.2 TrussFormer	1
1.3 TrussControl	2
<b>2 RELATED WORK</b>	<b>3</b>
2.1 Large-scale Personal Fabrication	3
2.2 Construction Kits	3
2.3 Prototyping with Ready-Made Objects	3
2.4 Building with Variable Geometry Trusses	3
2.5 Software Pipeline for Animatronics	3
2.6 SketchUp	3
<b>3 WALKTHROUGH</b>	<b>5</b>
3.1 Designing Static Structures	5
3.2 Adding Movement to the Structures	6
3.2.1 Force Analysis	7
3.3 Controlling the Structure	9
3.3.1 PID Control	10
3.4 Building the Final Object	10
3.4.1 OpenSCAD	10
3.4.2 Printing the Parts	10
3.4.3 Assembling the Structure	11
<b>4 HARDWARE</b>	<b>13</b>
4.1 Building Parts	13
4.1.1 Links	13
4.1.2 Hubs	13
4.1.3 Hinge Chains	14
4.1.4 Stability	15
4.2 Controls	16
4.2.1 Electric vs. Pneumatic Actuators	17
4.2.2 Open Loop vs. Closed Loop	17
<b>5 IMPLEMENTATION</b>	<b>19</b>
5.1 Architecture	19
5.1.1 Designer	20
5.1.2 SketchupObjects	22
5.2 Physics Simulation	24
5.3 TrussFab Designer	26
5.3.1 User Interface	26
5.3.2 Structure Creation	28
5.3.3 Modifying the Structure	28
5.3.4 Relaxation Algorithm	28

5.4	Export	29
5.4.1	Hinge Placement	29
5.4.2	Minimization Logic	29
5.4.3	OpenSCAD Export	29
5.5	TrussFormer Physics Engine	29
5.5.1	Automatic Actuator Placement (if it works soon-ish)	29
5.6	Force Control	29
5.6.1	PID	29
6	CONCLUSION	31
	BIBLIOGRAPHY	33

## LIST OF FIGURES

---

Figure 3.1	The T-Rex built in TrussFab	5
Figure 3.2	An exported leg asset...	6
Figure 3.3	... which can be used to quickly build the spider	6
Figure 3.4	An exported leg asset...	7
Figure 3.5	... which can be used to quickly build the spider	7
Figure 3.6	The animation pane showing two actuators group, the red one contains two actuators	8
Figure 3.7	A sensor measures the force on the central actuator, and speed and acceleration on a node at the “nose” of the dino	9
Figure 3.8	adsf	9
Figure 3.9	Overview of the OpenSCAD editor	11
Figure 3.10	A hinge	11
Figure 4.1	The connector is inserted into the bottle opening ...	14
Figure 4.2	... and secured with a cuff	14
Figure 4.3	(a) spherical joint mechanism connecting 5 edges. (b) rendering of TrussFabs’ hinge chain design connecting 6 edges.	15
Figure 4.4	(a) We measured the forces on the bottom front edge of the T-Rex (b) using a digital force gauge. (c) The measured forces agree with the simulated forces.	16
Figure 4.5	Hardware setup for controlling the T-Rex, with Arduino, electric pressure control valves, and compressor.	17
Figure 5.1	TrussFab Architecture	20
Figure 5.2	Class Diagram showing the high-level Graph Structure of the TrussFab Designer	21
Figure 5.3	Class Diagram showing the UI components of the graph structure	23
Figure 5.4	Visualization of forces acting on edges. Blue - tension force, red - compression force, white - little or no force	26

## LISTINGS

---

Listing 5.1	Merging of two Nodes	<a href="#">22</a>
Listing 5.2	Simulation nextFrame method	<a href="#">25</a>
Listing 5.3	excerpt from UI callbacks	<a href="#">27</a>
Listing 5.4	The relaxation algorithm	<a href="#">29</a>

## ACRONYMS

---

## INTRODUCTION

---

Personal fabrication devices, such as 3D printers, are already widely used for rapid prototyping and allow non-expert users to create interactive machines, tools and art. As consumer-grade 3D printers are usually desktop-sized, the size of these objects is, however, fairly limited. TrussFormer aims to enable users to create large-scale dynamic objects using desktop-sized 3D printers. Scale can be achieved by creating multiple small-sized objects and connecting them to each other. If all parts of a large object would be 3D printed, this process would take a long time and special large-size 3D printers would be needed. Our solution to this problem is to take ready-made objects, like empty plastic bottles, and only print the connectors that keep them together. To aid users in this process, we developed a software simulation that can create objects which are capable of handling the substantial forces large object intrinsically have. We achieve this by providing stable primitives which can be attached together. These primitives resemble truss structures - beam-based constructions creating closed triangle surfaces, which are intrinsically sturdy and material-efficient.

In order to build the simulated objects, we provide export-functionalities. Our software also provides tools to evaluate the magnitude of force acting on the links.

- TODO:
- node-link-structure
- export
- force

### 1.1 TRUSSFAB

- create big structures
- create them quickly and cheaply
- explain concept of nodes and edges

### 1.2 TRUSSFORMER

- make structures move
- observe forces during movement
- create animation
- define hinges

### 1.3 TRUSSCONTROL

- closed-loop movement control
- automatic conversion of simulation animation to arduino code

# 2

## RELATED WORK

---

2.1 LARGE-SCALE PERSONAL FABRICATION

2.2 CONSTRUCTION KITS

2.3 PROTOTYPING WITH READY-MADE OBJECTS

2.4 BUILDING WITH VARIABLE GEOMETRY TRUSSES

- Steward Platform
- Walking Octa

2.5 SOFTWARE PIPELINE FOR ANIMATRONICS

2.6 SKETCHUP



# 3

## WALKTHROUGH

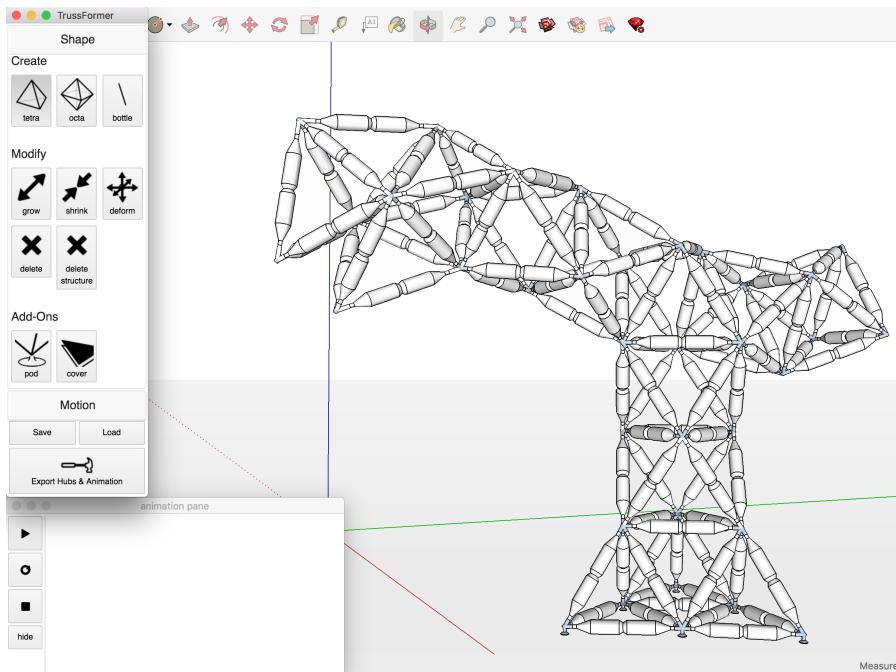


Figure 3.1: The T-Rex built in TrussFab

This chapter will present the functionalities of the system by showing the process of creating the T-Rex shown in figure . This will include all steps from creating the static structure, over introducing animation up to fabricating the final object.

Don't show animation pane...

Add image

### 3.1 DESIGNING STATIC STRUCTURES

Users can use predefined and structurally stable primitives to create their objects. These primitives are tetrahedra and octahedra. The structure can be formed as desired by using the grow and shrink tool. These tools elongate or shorten edges, deforming the structure dynamically in such a way that the form stays in tact as much as possible. The deform tool does a similar job, but rather than working on edges, this tool can move nodes.

The created Objects can also be saved to a JSON file. This way the user can either save his work for working on it later or create new primitives that can be attached to another object.

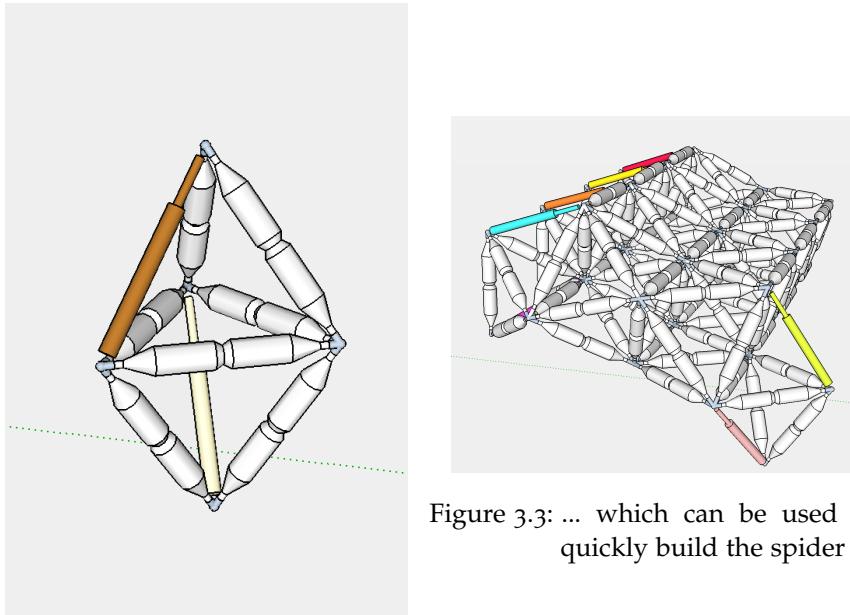


Figure 3.3: ... which can be used to quickly build the spider

Figure 3.2: An exported leg asset...

### 3.2 ADDING MOVEMENT TO THE STRUCTURES

Caption!

Movement is added to the structure by placing special physics links. These links act like linear actuators - struts that can extend and retract in a straight line. There are multiple methods for placing these actuators, ranging from a fully-automated way to manual placement. The automated way works by demonstrating a desired movement. The user selects the *Demonstrate Movement Tool*, clicks a node that should experience a certain movement and drags a line to the desired end position. TrussFab will then search for an actuator that brings the node closest to the desired position and turns the resulting edge into an actuator. The tool runs through all edges of the structure, replacing one after another with an actuator and simulates the resulting movement in the background. The actuator that solves the problem the best will be created.

Another way to add movement is to use predefined dynamic assets. It can be difficult for a user to fully grasp how an actuator will move the whole object. That's why we encapsulated often-used atomic sub-assemblies into quick-access tools. These assets connect to the rest of the structure through a dedicated triangle surface. Because the motion is localized in this asset, the result in the bigger structure is easier understandable.

The manual actuator placement requires the most knowledge about the resulting motion. It is, however, the most flexible way to create movement. The user can choose the actuator tool to turn every edge into or connect two nodes with an actuator. This can be done by clicking on the desired edge or the nodes. Transforming an existing edge

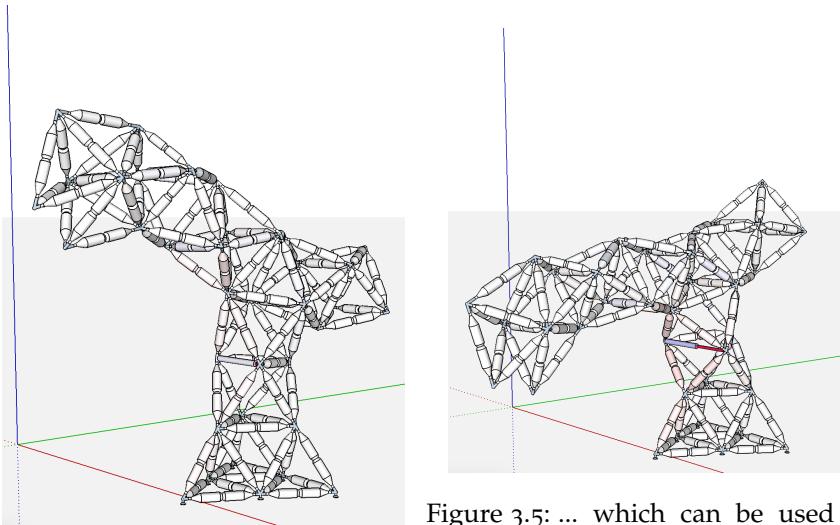


Figure 3.4: An exported leg asset...

Figure 3.5: ... which can be used to quickly build the spider

is usually the desired use case, as the introduction of more edges (by connecting two previously unconnected nodes) tends to make the truss more stable and can prevent motion altogether. Turning an edge into an actuator essentially removes this edge and adds a degree of freedom.

The user can actuate these links in two different ways. The *animation window* provides a slider for manual movement, as well as an animation pane which allows placing keyframes and playback modes: single and looping. All actuators have an actuators group assigned to them. Per default, an actuator is placed in its own, empty group. Using the actuator tool, already used for placing actuators, this group can be changed, which is indicated by the actuator changing its color. Control elements of the animation window always work on the whole group. The color of the animation line indicates which group will be actuated by either the slider or the keyframes. To add a keyframe to the animation pane, first the slider has to be moved to the desired position. The object will start to move according to the sliders position - if the slider is at the top, the actuator will be fully extended and vice versa. That way, the user can visualize the extend of the movement. If the user is satisfied, the indicator line can be moved to the desired position in the timeline and the diamond button is pressed to add this point to the animation.

### 3.2.1 Force Analysis

If the structure fulfills the desired motion, the user will want to check if the forces created during executing it will not exceed the breaking force of the object. A lot of factors play a role in the formation of the forces. These range from weight forces over lever forces to inertial

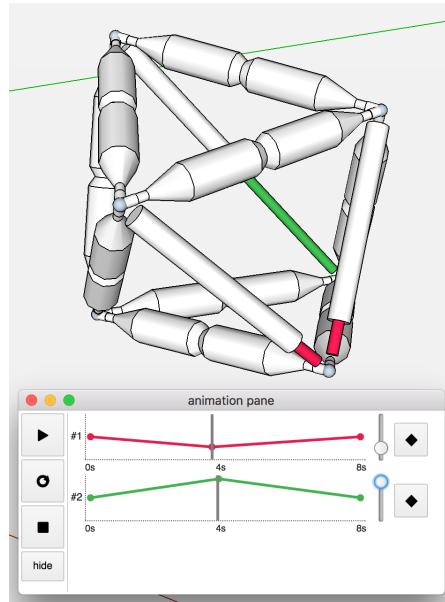


Figure 3.6: The animation pane showing two actuators group, the red one contains two actuators

forces. TrussFab aids the user to detect weak points and force peaks during a motion in different ways.

TrussFab's tools provide the possibility to constantly monitor the forces that occur during interactive movement of the structure. In simulation mode, all edges will be colored red or blue in increasing intensity the higher the force on them is. Red indicates compression force, while blue means tension force. A completely white color indicates a force of 0 N. These tension forces are automatically calculated by the built-in physics engine.

The weight of each node, which plays a big role in the force distribution, is calculated based on the number of edges connected to it and whether it is a hinge or a hub. If the user decides that certain nodes will have more load, these can be fitted with additional weight using the *Add Weight Tool*.

On top of the coloring of edges, the user can also use a sensor tool. This tool can be used on a single edge and observes this one more closely. The force data on an edge with a sensor will be recorded and visualized over time in a chart. This tool also works on nodes. Rather than recording the force, if the sensor tool is used on nodes, speed and acceleration data will be visualized. The result can be seen in figure 3.7.

Using these tools, the user can detect unwanted movements, like wobbling during change of poses, overly stressful actuations and even foresee breaking points.

If the movement of an animation exceeds the breaking force of the simulation, TrussFab has means of helping the user to find a movement curve that puts less stress on the structure. If TrussFab detects

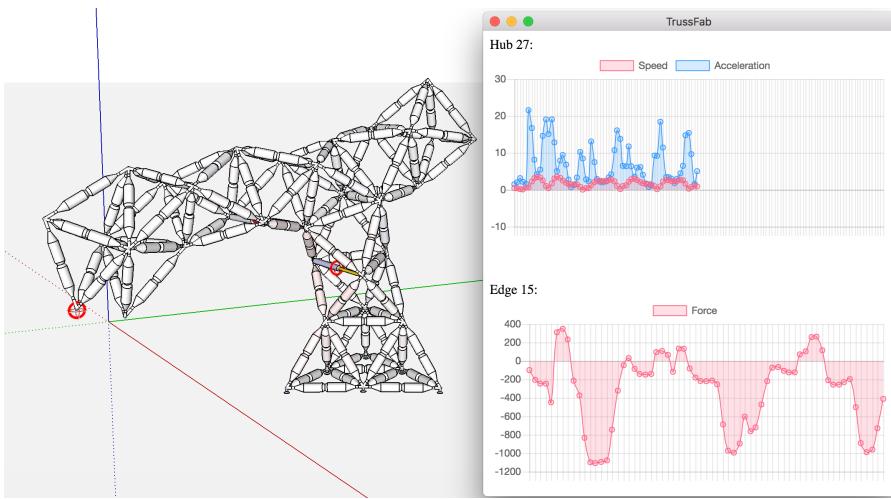


Figure 3.7: A sensor measures the force on the central actuator, and speed and acceleration on a node at the “nose” of the dino

that the structure broke, a popup window will appear asking the user to fix the animation. Two possibilities are available: fixing the animation by a) reducing the speed or b) reducing the motion. If option a) is chosen, TrussFab will elongate the animation sequence for all piston groups. This results in a slower movement and less force on the structure. Option b) will keep the length of the animation, but move the keyframes closer to the center line. The amplitude of the motion will be decreased this way.

Both versions will reduce the acceleration of the structure. The force formula  $F = m * a$  shows, that the force increases proportionally with the acceleration, as the mass is constant in the structure. Reducing acceleration also reduces the force.

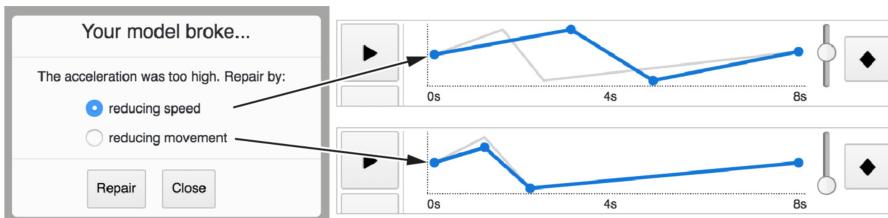


Figure 3.8: adsf

### 3.3 CONTROLLING THE STRUCTURE

Pneumatic actuators themselves do not have any knowledge about their position and extent. Their actuation underlies so-called open-loop control, meaning that the actuator can not react to changing external influences. - closed-loop control -> more sophisticated and complex movements possible

### 3.3.1 PID Control

- short intro: how does PID work?
- how do we use it?
- i.e. position control of actuators
- forward reference to section 4 (setup of length measurement)

## 3.4 BUILDING THE FINAL OBJECT

After the object was sufficiently tested in the editor, it is time to print the connectors and assemble the final object. To do this, TrussFab will calculate which connections will need to move and which can be static in the printed object. It also takes into account constraints, such as the minimum distance from a bottle neck to the center of a hinge, which might otherwise restrict movement. This process will be explained in detail in Section 5.4.1. This information is used to create OpenSCAD files - a modeling language which we use to modify templates of hubs and hinges.

### 3.4.1 OpenSCAD

At first, our abstract description of the object has to be converted into a physical representation. In order to achieve this, we used a modeling language called *OpenSCAD*. The *Export Hubs and Hinges* button will automatically morph the structure into a statically sound object, i.e. it will elongate and shorten edges so, that the ideal amount of movement is possible.

This needs to be more detailed for sure!!

The resulting arrangement of nodes and edges will be transferred to OpenSCAD. OpenSCAD enables us to create 3D structures programmatically. We use it to create 3-dimensional primitives, such as spheres, cubes or cylinders, and to apply set operations, like *difference* or *union* on them.

OpenSCAD provides an editor which can be used to prototype a model. This editor, including some example operations can be seen in Figure 3.9. We used this editor to create the template functions used for creating the final hinge and hub models. This will be explained in more detail in Section 5.4.3.

### 3.4.2 Printing the Parts

Each OpenSCAD file represents a single part in the structure. These files can easily be converted to *.stl* files, which are typically used for 3D printing. These files have to be imported into any 3D printing software, arranged efficiently and sent to a 3D printer. Printing of one hinge part using an UltiMaker 3 printer takes about 20 minutes.

evaluate!?

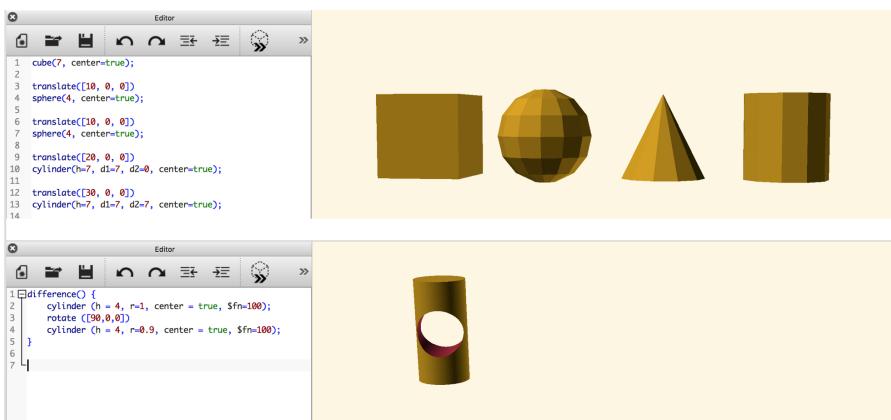


Figure 3.9: Overview of the OpenSCAD editor

### 3.4.3 Assembling the Structure

The resulting hubs and hinges contain an ID system for easy assembly. Each part of a node has the node ID printed on. That way it is easy to find out which hinge-parts belong together. Additionally, each edge elongation contains the id of the connected edge. A compound elongation, which is the usual case for a hinge, is therefore assembled by finding two parts with the same node and edge ID. For static hubs, this concept is similar, but of course these do not have to be assembled.

Verlängerung einer Edge, also quasi die Elongation. FIND A BETTER NAME!

Two connectors with different node IDs but the same edge IDs will be connected by a link.



Figure 3.10: A hinge

This will be a better image.



# 4

## HARDWARE

---

This chapter will talk about challenges and solutions we faced while finding connections for large-scale dynamic objects. It will explain the materials and methods used in transforming the TrussFormer-designed models into actual objects.

Furthermore, this chapter will explain the techniques and hardware used to get the structure to move.

### 4.1 BUILDING PARTS

We can differentiate between three essential building parts for our truss structures. *Links* are the connecting and shaping parts. We used PET bottles for these parts, because they are readily available, cheap and sturdy.

We have two different ways of connecting links. For dynamic connections, i.e. connections that allow movement of the structure, we use a hinge chain system. Rigid connections use single-part objects, we call *hubs*.

Links, and hubs or hinge chains are connected by purpose-designed *cuffs*, which fit over the bottles' thread and a connector part on the hub.

#### 4.1.1 *Links*

We opted to use 1l (big) and 0.5l (small) reusable PET bottles because of their stability and abundant availability. Two bottles are connected on their bottom side by a wood screw, which is inserted using a special long-necked screwdriver. The resulting link-lengths are:

1. 60 cm - two big bottles
2. 53 cm - one big and one small bottle
3. 46 cm - two small bottles

#### 4.1.2 *Hubs*

We call the connecting parts *Hubs*. These hubs were designed to withstand loads in the range of a human weight. Earlier tests showed, that the bottle links we use can withstand a compression force of around 85 kg.

Hubs are solid one-piece objects that can connect two or more links.



Figure 4.1: The connector is inserted into the bottle opening ...



Figure 4.2: ... and secured with a cuff

They therefore consist of two or more regions that can hold the neck of a bottle. We call these regions *connectors*. These connectors consist of a flat part, slightly wider (30 mm) than the bottle openings outer diameter (28 mm) and an extrusion in the middle, the size of its inner diameter (20 mm). The extrusion can simply be plugged in to the bottle opening, giving a snug fit and good lateral stability.

To prevent the bottles from slipping out, a cuff is clipped over the connector and the bottle opening. The cuff is designed as a circular section, slightly larger than a semicircle. Its upper and lower end have different sizes: the smaller end fits over a rim on the hub connector, the larger one over an extrusion on the bottle neck. Due to its flexibility, the cuff acts like a spring, making it easy to clip over the bottle and connector, but giving enough stability once clipped on.

#### 4.1.3 Hinge Chains

We are using truss structures specifically because of their structural stability. In order to introduce movement to these constructions, we came across some challenges. Multiple edges have to be able to pivot around a common hub. This kind of spherical can potentially achieved using ball joints. However, these typically only allow for a connection of two edges without obstructing each other.

We solve this issue by using a *spherical joint mechanism*. As can be seen in Figure 4.3, these chains of hinges can connect multiple edges, which can all rotate around the same center. This is possible, because the axes of rotation do not occupy the rotational center itself, creating room for movement.

We started out using an open hinge chain, meaning that the chain had special end parts which were not connected. This gave us a lot of freedom of movement, however it turned out that this design was not strong enough for our needs and frequently broke during testing builds of the T-Rex. This required us to rethink our design and we

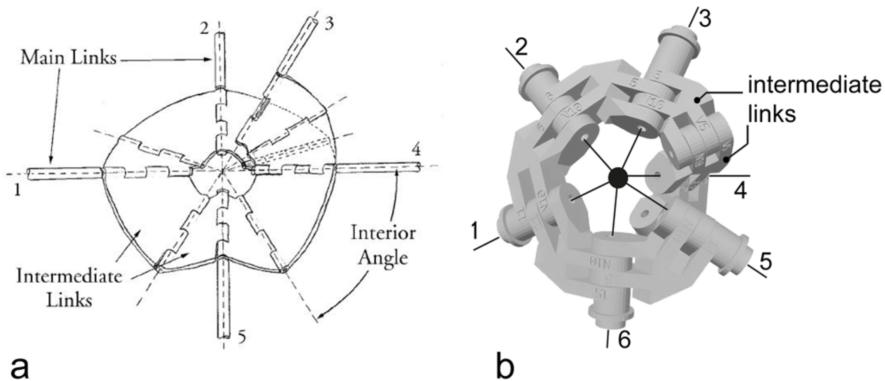


Figure 4.3: (a) spherical joint mechanism connecting 5 edges. (b) rendering of TrussFabs' hinge chain design connecting 6 edges.

came up with a closed hinge chain.

The degrees of freedom (DoF), our hinge system needs to have, underly constraints, allowing us to limit the possible movement of the hinges. The original spherical joint mechanism shown in Figure 4.3a connected all edges using two hinges, allowing for 2 DoF. This is not necessary if we are dealing with truss structures. An example of TrussFabs's hinge design can be seen in Figure 4.3b. Only edge 4 is connected to its neighboring edges using two intermediate hinges. All other edges only require rotation (resulting in 1 DoF).

#### 4.1.4 Stability

The hinge parts and connecting cuffs are printed using consumer-grade desktop FDM<sup>1</sup> 3D printers. The filament consisted of PLA<sup>2</sup> plastic and the parts had a 15% infill with a wall thickness of 3 mm. We chose PLA filament instead of ABS<sup>3</sup> primarily because of its easier usability. ABS plastic, while being stronger, needs to be printed on a heated surface, which many hobbyist printers do not have. As we designed our tool for 3D-printing enthusiasts and not professionals, we wanted to target owners of consumer-grade printers. Additionally, PLA consists of organic materials (mainly cornstarch and sugarcane), which makes it biodegradable, as opposed to ABS plastic, which is oil-based.

We aimed to make our software simulation as accurate to the real-world object as possible. To calibrate the simulation values, we therefore measured the forces of our T-Rex example.

We used a digital force sensor, with a capacity of 5000N and an error rate of 0.5%, on the front bottom edge of the T-Rex (Figure 4.4 a-b). It was placed between two small bottles, giving the edge the same

<sup>1</sup> Fused Deposition Modeling

<sup>2</sup> Polylactic acid

<sup>3</sup> Acrylonitrile butadiene styrene

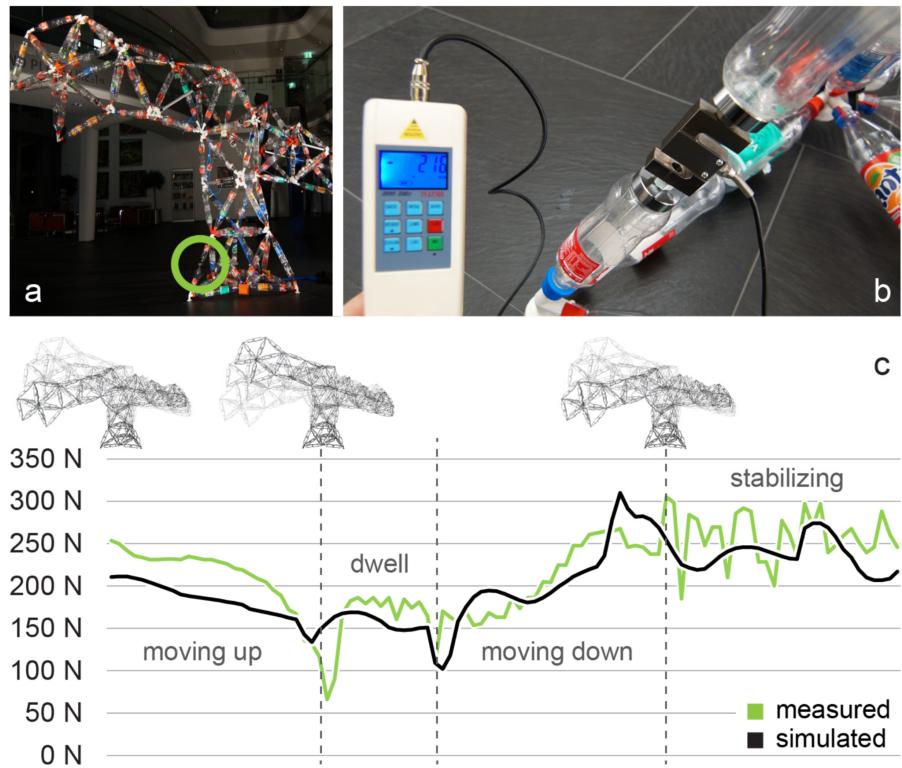


Figure 4.4: (a) We measured the forces on the bottom front edge of the T-Rex (b) using a digital force gauge. (c) The measured forces agree with the simulated forces.

length as it would have had with two big bottles. We chose this position, because it bears the largest force due to the long lever force the neck of the T-Rex produces. Our test case consisted of moving the head up from its lowest to its highest and then back down to its lowest position again. The same movement, with the same speed, was programmed in the simulation. Figure 4.4c shows, that both, the simulated and the measured force, are in agreement.

#### 4.2 CONTROLS

We control the pneumatic actuators using a MIDI control interface. The slider inputs are sent to an Arduino which translates them to control signals for digitally controllable pressure valves. The actuators have two connections for air pipes: one for extending the actuator and one for retracting it. The signal from the slider is interpreted as a mixture between these two inputs, with the slider at the top meaning that all the air is sent to the extending input and the slider at the bottom completely retracting it. This way we achieved manual open-loop position control.

Figure 4.5 shows the main parts of our hardware setup. The electrically controllable pressure regulators receive signals from the control

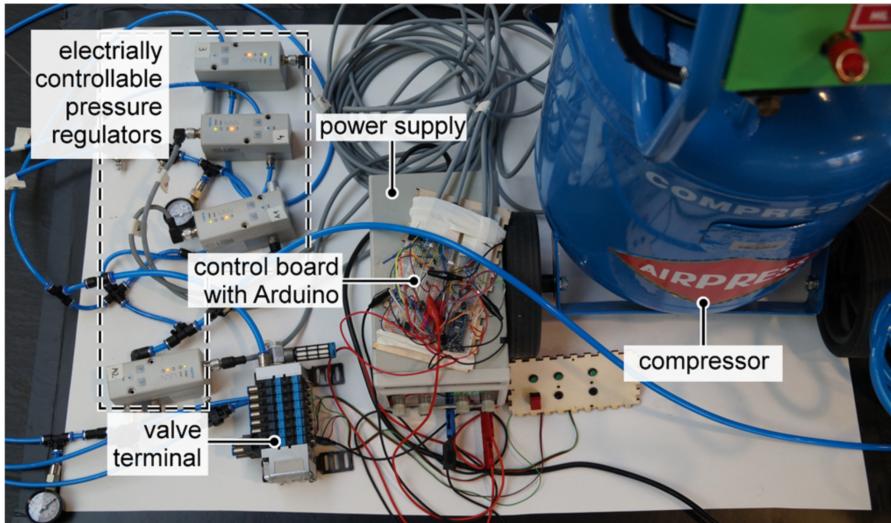


Figure 4.5: Hardware setup for controlling the T-Rex, with Arduino, electric pressure control valves, and compressor.

board and limit the air pressure according to the needed level of extension of a given actuator. One pressure regulator is connected to exactly one actuator. The *Airpress HL 360* compressor delivers the regulators with up to 8 bars of pressure.

The limited pressure is tunneled to a valve terminal. This terminal opens and closes airflow to the actuators. If it opens a valve, the regulated pressure will enter its air circuit and change the extension of the actuator. If it closes it, the pressure will stay constant and the actuator keeps its position.

#### 4.2.1 Electric vs. Pneumatic Actuators

Early TrussFab prototypes used electric actuators, instead of the pneumatic ones we used for the T-Rex. Electric actuators can produce a fairly large force and are easier to control, as they can usually provide their current extension. However, at around 0.03 m/s they extend very slowly, compared to our pneumatic actuators, which can reach up to 20 m/s. As our structures are fairly lightweight, the force of the actuators is not a big concern, while the higher speed of pneumatic actuators enables us to create more dynamic structures.

#### 4.2.2 Open Loop vs. Closed Loop

The MIDI controller is a well working possibility to control the actuators. It is, however, still a closed-loop control, meaning it can not react to external influences, like weight shifts or additional pressure acting on edges. This is not a big problem for our T-Rex example, as it usually moves mostly undisturbed and stand-alone. If we want to

check if this is right

create an interactive object, this will not be sufficient.

Pneumatic actuators on their own tend to not have the possibility to measure their own rate of extension. We therefore created an easy to build distance sensor, using a string on a spindle. One side of the string is attached to the piston part of the actuator, while the spindle is attached to the static body. While the actuator extends, the spindle will spin, releasing more string. We attached a rotary encoder to the spindle, that can count the turns of the spindle. Using the diameter of the reel, we can calculate how much string was released during the extension, which correlates with the extension of the piston itself.

Using this approach, we can have closed-loop control of the actuators, making it possible to constantly monitor its the extension and applying force accordingly.

We created a rudimentary motion-platform using closed-loop PID control (s.a. Section 5.6.1). The motion platform had three modes. It started fully extended, applying a little bit of force to keep the actuator upright, but little enough that a human sitting on it pushed it back. If the actuator reached an extension of 10 cm, the control loop applied a holding force, making it possible to sit on the platform without it moving. If the human applied more force, pushing the piston further down, the third mode activated, making the motion platform oscillate up and down.

# 5

## IMPLEMENTATION

---

Our editor is based on an existing plugin for the 3D modeling software SketchUp. It implemented placing of truss primitives, modifying these structures and asynchronously simulating forces acting on the objects. We did a rewrite of this plugin, aiming to achieve a more maintainable and readable code-base and enabling us to use newer versions of SketchUp, as the old version was only compatible to SketchUp's 2016 version. The main goal of our new plugin was to create dynamic truss structures. This required us to introduce a real-time physics engine, special physics components and animation functionalities. Our plugin supports the user in the creation of large-scale dynamic objects by visualizing the forces acting on the object, while the user can move the object interactively. It can also create animations that play automatically, warning the user if the forces during the animation exceed the force limits of the structure and helping them in finding a better approach. Furthermore, it aids the user in creating the most efficient movement layout in the complex truss structure.

The user interface is implemented in JavaScript, using the node.js framework for advanced features. The connection to SketchUp and most functionalities are written in Ruby. The physics engine is based on a C++ engine, using a Ruby wrapper to embed it in our code base. After explaining the architecture, this chapter will demonstrate the tools and functionalities in greater detail and explain the underlying components.

### 5.1 ARCHITECTURE

The software can be divided into four components. The most user-facing one is the TrussFab Designer. It contains the user interface and the construction functionalities. The other components can be seen as extensions to the designer. The *Force Analysis* calculates tension force on the created structure. The tensions forces are calculated using an adapted version of the *MSPhysics*<sup>1</sup> physics engine, which is a Ruby wrapper around the C++ physics engine *NewtonDynamics*<sup>2</sup>.

This physics engine is also used by another component, the *Hinge Placement Algorithm*. It uses the physics features to detect changing angles between Edges, indicating the need for a hinge at a Hub.

The export function using OpenSCAD will be explained in section 5.4.3. The structure diagram in figure 5.1 shows an overview of the

---

<sup>1</sup> <https://extensions.sketchup.com/en/content/msphysics>

<sup>2</sup> <http://newtondynamics.com/forum/newton.php>

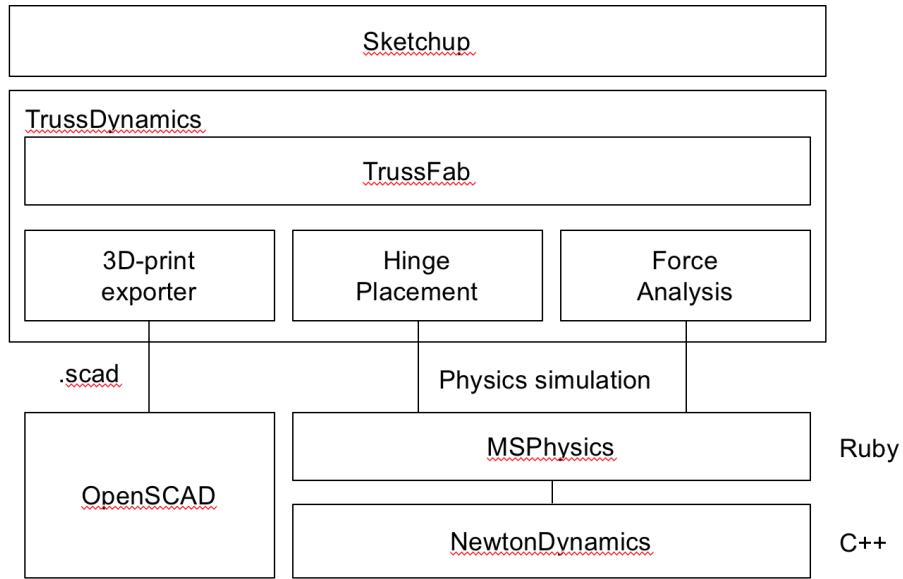


Figure 5.1: TrussFab Architecture

components. Details for each component will be explained later in this chapter.

### 5.1.1 Designer

All components are stored in a graph structure. The building parts are *Edges*, *Nodes* and *Triangles*. They all inherit *GraphObject*. The purpose of these objects is providing user-facing functionalities and storing lower-level components. An overview of the graph structure can be seen in [5.2](#).

explain what a Singleton is

The *Graph* is implemented as a *Singleton* that stores and provides access to all *GraphObjects*, creates new ones and provides convenience functions for user interactions, such as finding the node closest to the mouse cursor. As this class is a singleton, every module of the software has access to the objects.

Clarify. Either explain what that means (Hubs, Links, Surfaces) or at least have a forward reference

Each of these objects has access to its underlying logic-bearing component, called *SketchupObject*. The access to this functionality is, however, not implemented in this superclass, but in each subclass, having the specific name as an accessor. This design decision was made to improve code readability, and decrease coding errors caused by accessing the wrong *SketchupObject*.

Show before and after code snippet

The responsibility of the *GraphObject* class is primarily unifying the way the appearance in SketchUp of the underlying object can be changed as much as possible. This includes highlighting a specific object if the mouse hovers over it, resetting the object to its default state and creating and deleting it. More complex methods need to be implemented in the respective subclass.

*Nodes* are the connecting components of the structure. *Edges*, as well

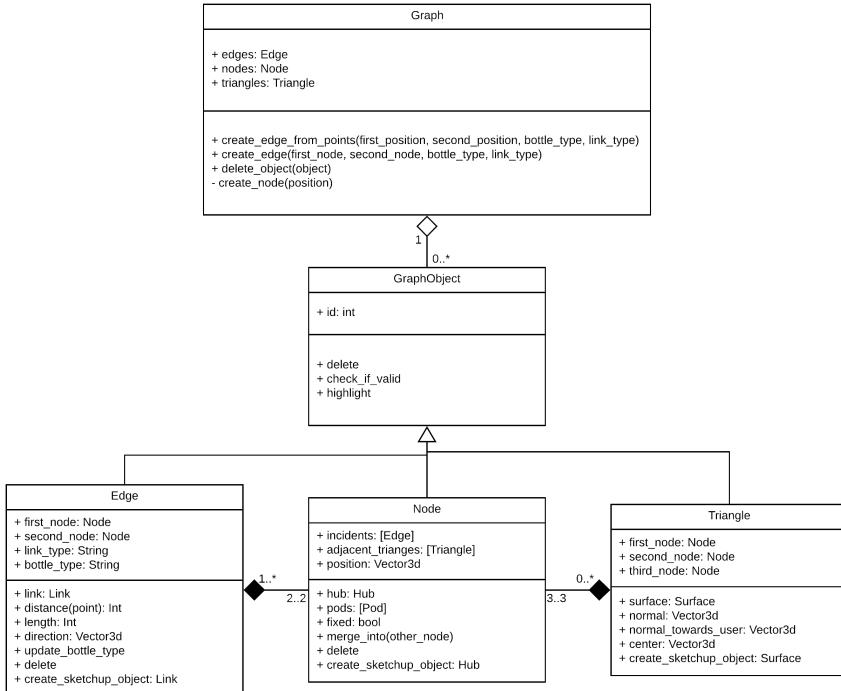


Figure 5.2: Class Diagram showing the high-level Graph Structure of the TrussFab Designer

as *Triangles* are created based on *Nodes*. Apart from storing adjacent *Edges* and *Triangles*, a *Node* can specify their positions in the SketchUp world. The *Nodes*' adjacent objects constantly check if their position has changed and update their SketchUp representation accordingly. If the structure is deformed in such a way that a *Node* will be at the same position as another one, the *Node* object can automatically merge into the other *Node*. The *Node* will iterate over all its adjacent *Edges* and tell each one, apart from the *Edges* that run from the other *Node* to the *Edge* the *Node* is hinging around (i.e. the *Edge* that is opposite the *Node*), to exchange itself with the *Node* it wants to merge into. These *Edges* are removed from its own adjacent *Edges* and added to the collection of the new *Node*. The same happens for all adjacent *Triangles*. As a last step, the *Node* deletes itself and all remaining adjacent *Edges* and *Triangles* (which will be the *Edges* and *Triangles* that got merged). The object will then be adapted according to the new positions using the *Relaxation algorithm*, described in section 5.3.4.

```

1   def merge_into(other_node)
2     merged_incidents = []
3     @incidents.each do |edge|
4       edge_opposite_node = edge.opposite(self)
5       next if other_node.edge_to?(edge_opposite_node)
6       edge.exchange_node(self, other_node)
7       other_node.add_incident(edge)
8       merged_incidents << edge
9     end
10    @incidents -= merged_incidents
11
12    merged_adjacent_triangles = []
13    @adjacent_triangles.each do |triangle|
14      new_triangle = triangle.nodes - [self] + [other_node]
15      next unless Graph.instance.find_triangle(new_triangle).nil?
16      triangle.exchange_node(self, other_node)
17      other_node.add_adjacent_triangle(triangle)
18      merged_adjacent_triangles << triangle
19    end
20    @adjacent_triangles -= merged_adjacent_triangles
21
22    delete
23  end

```

Listing 5.1: Merging of two Nodes

Another component that is tightly coupled to Nodes are *Pods*. A Pod acts as a stand for the object and tells TrussFab that this Node should not change its position.

The *Edges* are the most visual components of TrussFab. They are visualized by bottles of different lengths, if they are static links, or as two cylinders forming an actuator, if they can have variable lengths. The Edges handle creating the correct model and changing it if the user decides to place a different kind of Edge. Edges play a big role in the simulation. The last high-level component in TrussFab is the *Triangle*. A Triangle is primarily used as a convenient access to multiple Nodes or Edges. Most tools that work on Nodes, such as the *Add Weight Tool*, can also be applied to Triangles, adding weight to all three connected Nodes. The Triangle also provides functions for telling the *MouseInput* in where a certain face is directed.

[add image](#)[IMPROVE!](#)

### 5.1.2 SketchupObjects

As mentioned before, each *GraphObject* contains a lower-level *SketchupObject*. These objects are responsible for more complex, lower-level tasks, such as physics calculations, rendering and communication to the simulation engine.

Each *SketchupObject* has a *Sketchup::Entity*, which is a class provided by SketchUp that is capable of handling the representation in SketchUp itself. This includes changing the color of the model, hiding and trans-

forming. On creation, each SketchupObject is also persisted in the entity.

find out why exactly I did that

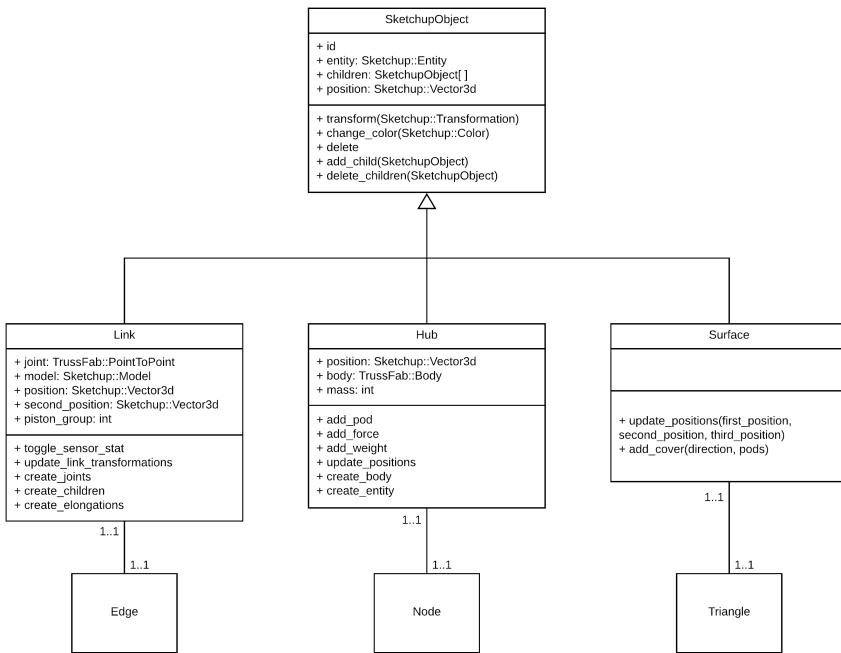


Figure 5.3: Class Diagram showing the UI components of the graph structure

#### 5.1.2.1 Hubs

*Hubs* are the underlying structures used by Nodes. For ease of calculation and increased performance of the *Simulation* (s.a. section 5.2), only the Hubs of a structure have physical properties. Hubs therefore have to store information about the object, such as the weight. The weight is calculated based on the number of bottle links and actuators connected to this node. For our system, we measured these values empirically by taking the weight of a screw and half the weight of a bottle link or an actuator per connection and adding it to the average weight of an empty printed hub. The *add weight* and *add force* tools will additionally increase this value, while the Hub also displays the indicators for these tools.

These values, together with a few other variables, then form the basis of our simulated structure.

#### 5.1.2.2 Links

*Links* define the connection between two hubs. They are tightly coupled to the physics engine and contain the *Joints*, which are objects used by the engine itself. Available joints are:

- TrussFab::PointToPoint - A static connection between two points

- TrussFab::PointToPointActuator - A variable-length connection between two points
- TrussFab::PointToPointGasSpring - A variable-length connection between two points with a distance-based force factor
- TrussFab::GenericPointToPoint - A variable-length connection between two points with a custom force factor

The Link is therefore responsible for defining the distance between two Hubs. Because of the nature of truss structures, this can have impact on the whole object and create the variable geometry truss. Joints will be discussed in more detail in the simulation section [5.2](#).

#### 5.1.2.3 Surface

The *Surface* is primarily used to visualize what face of the truss the user is currently selecting, by changing the color between three bottles. It can also hold a cover, which has mainly optical purposes, i.e. a user can cover up a surface with a sheet of wood if they want to have this surface closed up after building.

## 5.2 PHYSICS SIMULATION

TrussFabs' force analysis used to be based on *Finite Element Analysis*, calculated asynchronously on a remote server. This provided fairly accurate results and did not require a powerful computer to run. However, TrussFabs' responsibilities evolved during the course of its life and we decided to implement a real-time physics engine inside of our plug-in.

We decided to use the SketchUp plug-in *MSPhysics* by Anton Synytzia<sup>3</sup>. *MSPhysics* is capable of calculating real-time physics on SketchUp elements and creates a customizable physics world in the modeling software. This *MSPhysics* world has parameters, like gravity, update timestep, and solver model which we can adapt to maximize accuracy and speed of the simulation.

The simulation uses the animation feature of SketchUp. A ruby class can act as a Sketchup::Animation when it implements the *nextFrame* method, which must return true until the animation ends. This method is called every time SketchUp receives the signal that a new frame should be rendered. We do that by calling *view.show\_frame* (s.a. listing [5.2](#)), which will trigger SketchUp to start rendering the next frame based on the simulation updates that happened earlier. We call this function as the first step in our *nextFrame* method, because this way, SketchUp can start rendering, while our simulation does the next physics update.

---

<sup>3</sup> <https://github.com/AntonSynytzia/>

```

1 def nextFrame(view)
2   view.show_frame
3   return @running unless @running && !@paused
4
5   update_world
6   update_hub_addons
7   update_entities
8
9   if (@frame % 5).zero?
10     send_sensor_data_to_dialog
11   end
12
13   @frame += 1
14   update_status_text
15
16   @running
17 end

```

Listing 5.2: Simulation nextFrame method

During this update, the physics engine calculates new forces on each physics component of the built object. For that, first all static forces are applied to the object. These are forces added by the *Add Force Tool* or static forces calculated by the *GenericPointToPoint* joints. Using these values and all other intrinsic parameters included in the physics objects, we call the entry point to our MSPhysics plug-in. The *@world.advance* function calculates the change of forces and positions from one timestep to another. In our physics world, one timestep correlates to 1/60s, to achieve realistically timed results assuming that SketchUp itself runs with 60 frames per second.

After each world update, the tensions on each link are recorded for visualizing them later. This has to be done, because there could potentially be multiple world updates per render step and we do not want to miss crucial forces.

These steps in *update\_world* are done for a specified number of times. In regular animation mode, one world update is calculated per frame, however some tools use the simulation in the background for static checks or other calculations. These tools do not need to display the in-between steps, so they can calculate multiple world updates back-to-back.

With the knowledge of the new physics calculations, we can send information about the stress level of each joint to SketchUp. We color the links depending on the tension force on them. A blue color means negative tension force, i.e. pulling force, while red color means positive force, i.e. compressing force. The higher the force, the deeper the color gets.

Once the physics portion of the rendering step is done, our own graph structure has to be updated. For that, each edge and node updates their own positions and transformations based on their internal physics objects.

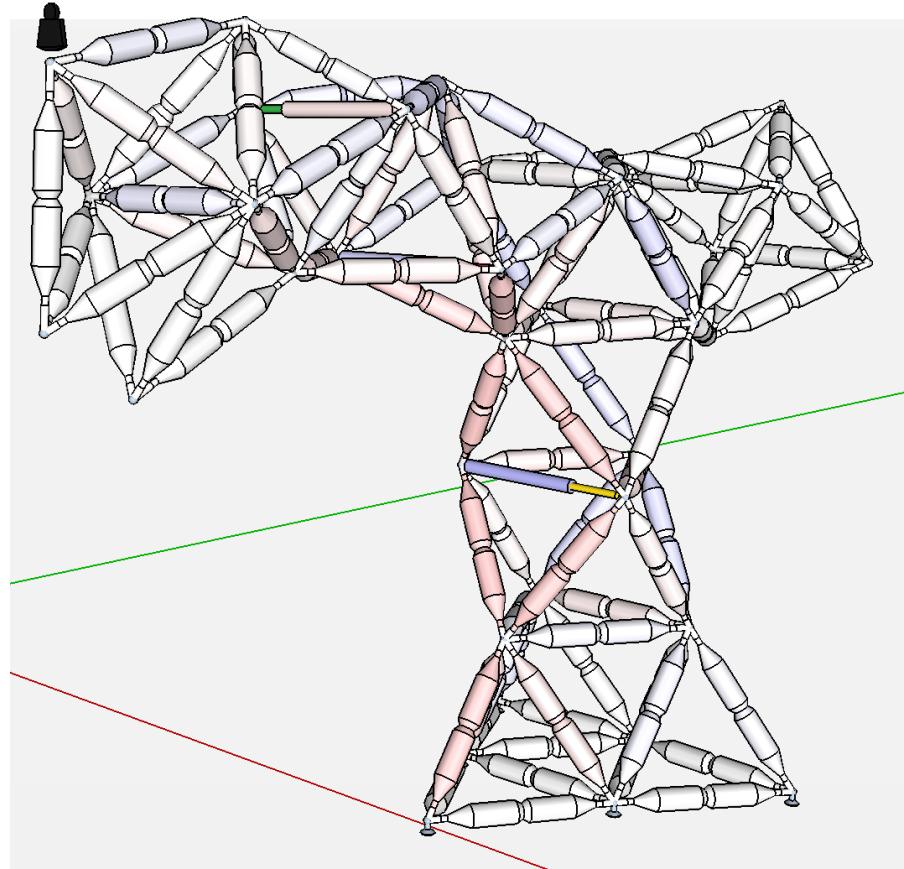


Figure 5.4: Visualization of forces acting on edges. Blue - tension force, red - compression force, white - little or no force

As a last step, sensor data is send to the respective UIs in order to draw a chart depicting physical data.

### 5.3 TRUSSFAB DESIGNER

The TrussFab Designer provides static sketching functionalities. It can create and display different predefined models, has knowledge about the connections of different components and can modify the resulting objects structure.

#### 5.3.1 User Interface

The user interface (UI) is written in JavaScript. It uses SketchUp's built-in *HtmlDialog* class, which lets us interact with HTML dialog boxes using the Ruby API<sup>4</sup>. The *HtmlDialog* is a modified version of Googles' Chrome browser and supports modern HTML5 code, as

---

<sup>4</sup> Application Programming Interface

well as state-of-the art JavaScript functionalities and extensions. Our user interface has four distinct modules:

- the sidebar
- the animation pane
- force charts
- context menus

Each of these modules consists of a number of HTML and JavaScript files, which implement the design and functionality of the UI elements, and one Ruby file. The Ruby file is a proxy that communicates between the JavaScript side and the rest of the system. It subscribes to JavaScript callbacks, to react to UI interactions and can directly execute JavaScript code to pass data from ruby to the UI elements.

```

1 Class AnimationPane
2   def add_piston(id)
3     @dialog.execute_script("addPiston/#{id}")
4   end
5
6   def stop_simulation
7     @dialog.execute_script('resetUI();')
8   end
9
10  def register_callbacks
11    @dialog.add_action_callback('start_simulation') do |_ctx|
12      if @simulation_tool.simulation.nil? || @simulation_tool.simulation.stopped?
13        start_simulation_setup_scripts
14      end
15    end
16
17    @dialog.add_action_callback('stop_simulation') do |_ctx|
18      unless @simulation_tool.simulation.nil? ||
19        stop_simulation
20        Sketchup.active_model.select_tool(nil)
21      end
22    end
23
24    ...
25  end
26 end
27

```

Listing 5.3: excerpt from UI callbacks

As can be seen in listing 5.3, these proxy classes have two ways of communicating between Ruby and JavaScript. The `execute_script` function on the `HtmlDialog` can call arbitrary code on the UI element at any time. It is possible to pass ruby primitives, such as strings, integers or arrays, through this function call. This makes it possible to a) have complex interactions with the user interface and b) keep state on the ruby side and focus on visualization in JavaScript.

The other way works equally asynchronously. JavaScript can send signals to SketchUp. The SketchUp side can register to those callbacks and execute ruby code.

### 5.3.2 Structure Creation

Terminology:

1. Edge:
  - a) Connects two nodes
  - b) Can be:
    - i. Bottle Link
    - ii. Actuator
    - iii. PID link

### 5.3.3 Modifying the Structure

#### 5.3.4 Relaxation Algorithm

The relaxation algorithm is used to distribute the change of an edge over the whole structure. This way, we can adapt edges without changing the overall appearance of the object.

If we want to elongate an edge, we give this edge a new *optimal length*. This length and the edge itself is stored internally in the relaxation class. With this information, the actual relaxation algorithm is triggered. For a set number of iterations, 20'000 in our case, the algorithm will pick an edge out of the stored set randomly and checks if its current length differs from its optimal length, if any is set. If no optimal length is set or the deviation is sufficiently little, the next edge will be investigated. For the first iteration, the only edge in the set will be the one we want to elongate or shrink. All edges connected to the same nodes as this edge will also be added to the set and the node positions of the edge will be adapted to achieve the targeted length, damped by a factor to receive a more natural end result over more iterations. The direction vector of the edge will stay the same. That means, that other edges will change their lengths as well during this step.

During the next iteration of the loop, another edge in the set, now containing the incident edges as well, will undergo this process; checking if it needs to change its length, adding its incidents to the set and bringing the nodes closer to the optimal length. At some point, all edges will be sufficiently close to their targeted length or the maximum number of iterations will be reached.

When this step is reached, we have to update the sketchup representations according to the data we collected in the internal storage of the relaxation, in order to visualize the result to the user.

```

1 def relax
2     # Abort if there is nothing to do
3     return if @edges.empty?
4     number_connected_edges = connected_edges.length
5     @max_iterations.times do
6         # pick a random edge
7         edge = @edges.to_a.sample
8         # only adapt edge if we still have stuff to do
9         deviation = deviation_to_optimal_length(edge)
10        next if deviation.abs < CONVERGENCE_DEVIATION
11        # add neighbors if we still have edges left to add
12        add_edges(edge.incidents) unless @edges.length ==
13        number_connected_edges
14        adapt_edge(edge, deviation * @damping_factor)
15    end
16    move_nodes_to_new_position
17    self
end

```

Listing 5.4: The relaxation algorithm

## 5.4 EXPORT

### 5.4.1 Hinge Placement

### 5.4.2 Minimization Logic

- elongates and shortens edges so that maximum movement is possible with minimum material use
- uses iterative relaxation algorithm, will be explained in [5.3.4](#)

### 5.4.3 OpenSCAD Export

## 5.5 TRUSSFORMER PHYSICS ENGINE

### 5.5.1 Automatic Actuator Placement (if it works soon-ish)

## 5.6 FORCE CONTROL

### 5.6.1 PID



# 6

## CONCLUSION

---



## BIBLIOGRAPHY

---

- [1] Donald E. Knuth. "Computer Programming as an Art." In: *Communications of the ACM* 17.12 (1974), pp. 667–673.



## 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 2018*

---

Tim Oesterreich