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[1], 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.

1 Members of GuIT (Gruppo Italiano Utilizzatori di TeX e LaTeX)

# CONTENTS

# LIST OF FIGURES

# LISTINGS



# ACRONYMS

# 1 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 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 sturdy by design and material-efficient.
Such objects have been shown to support a wide range of applications, like furniture or art installation. However, these systems are limited to static structures. We want to extend this area of architecture by providing possibilities to create large-scale *kinematic* truss structures. Creating movement in truss structures comes with various challenges. TrussFormer is an end-to-end-system that lets the user create such kinematic structures without needing to have knowledge about the physical properties of moving trusses. It incorporates an editor for virtually creating the desired object, a physics engine that can simulate movement and visualize occurring forces and an export functionality that can convert the created design into 3D printable files.
- TODO:
- node-link-structure?

## 1.1 TRUSSFORMER'S UNDERLYING STRUCTURE ACHIEVES STABILITY

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

## 1.2 ADDING MOVEMENT USING VARIABLE GEOMETRY TRUSSES

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

## 1.3 TRUSSFORMER EDITOR

- 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

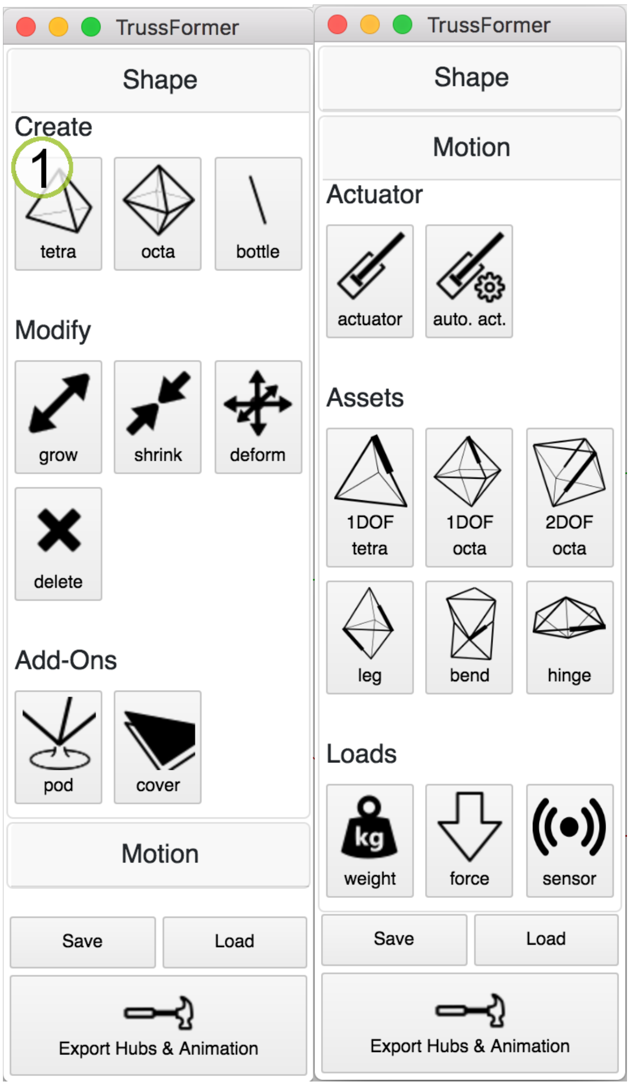


Figure 3.1: TrussFormers' tool pane

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.

Add image

## 3.1 DESIGNING STATIC STRUCTURES

As shown in Figure 3.2, the T-Rex model was first created as a stable static structure. With the help of TrussFormer, the user can make sure that the object to be created is structurally sound before adding motion. The provided tools can create sturdy primitives that are attached to each other, deform the structure, and check forces acting on each part of the object.

### 3.1.1 *Primitives*

Users can use predefined and structurally stable primitives to create their objects. Our system provides tools for creating octahedra and tetrahedra (Figure 3.1 (1)). Octahedra and tetrahedra are used for this task, because they consist entirely of triangular faces, which is an essential property for stable objects and widely used in heavy industry and architecture (e.g. cranes or bridges).

### 3.1.2 *How they are used*

These primitives can be attached to existing triangle faces, as seen in Figure 3.2 or placed on the ground. If the user desires, our system can also create single links between two nodes using the bottle link tool (2).
The form of the structure can be tweaked as desired by using the grow and shrink tool (3). These tools elongate or shorten edges, deforming the structure in such a way that the truss structure stays in tact. As our PET bottle system only allows three different sizes (long-long, long-short, short-short), we use elongations on the bottle necks. The grow and shrink tools will change the length of the elongation as long as a different bottle configuration is possible. If the users wants to shorten a certain edge a lot, the shrink tool will determine, that it should become a short-short bottle link. The deform tool (4) does a similar job, but rather than working on edges, this tool can move nodes.
For placing the object on the ground or placing covers on a triangle face, TrussFormer also provides a pod tool (5). Pods provide a flat surface on nodes that stands proud of the curved bottle bodies, making it easier to have a firm and level stand. Nodes that have a pod will not be changed by the modifying tools.

### 3.1.3 *Check static forces*

Split into checking static forces and dynamic forces? Probably explain the add weight tool here, as well

Using the animation pane, the user can determine static forces acting on the structure. (((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*. (12))))

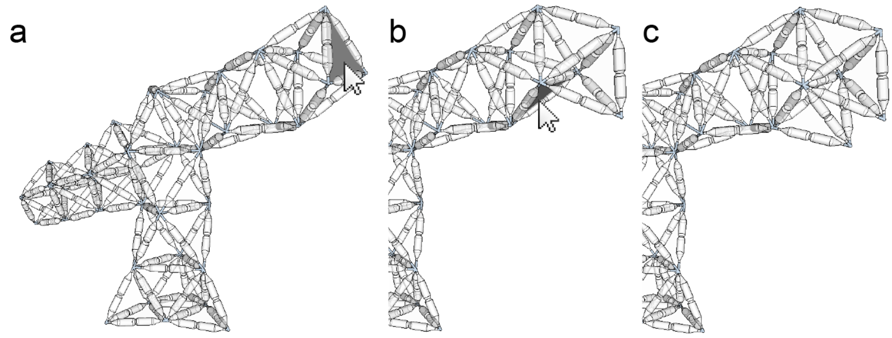


Figure 3.2: Modeling the static shape of the T-Rex. Here, the user creates the jaws of the T-Rex by attaching tetrahedron primitives through the steps (a, b, c).

## 3.2 ADDING MOVEMENT TO THE STRUCTURES

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.

### 3.2.1 *Automatic Actuator Placement*

The automated way works by demonstrating a desired movement. The user selects the *Demonstrate Movement Tool* (6), clicks a node that should experience a certain movement and drags a line to the desired end position. TrussFormer 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.

### 3.2.2 *Placing Primitives*

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. We encapsulated often-used assets into quick-access tools (7). 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. A selection of assets is shown in Figure 3.4
Instead of using the demonstrate movement tool to create the nodding motion of the T-Rex, a "bending" double-octahedron, as seen in Figure 3.4 (f) could have been used in the vertical body part.

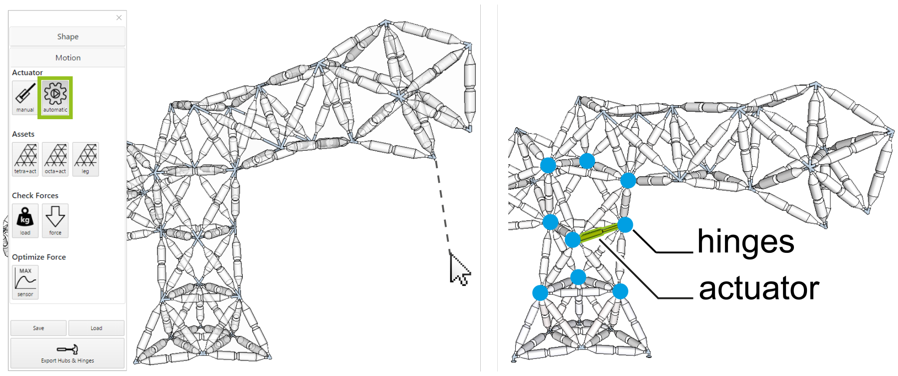


Figure 3.3: (a) The user selects the demonstrate movement tool and pulls the T-Rex head downwards. (b) TrussFormer responds by adding an actuator to the T-Rex body so that it is capable of performing this type of motion. At this point the system also places 9 hinging hubs to enable this motion (marked with blue dots).

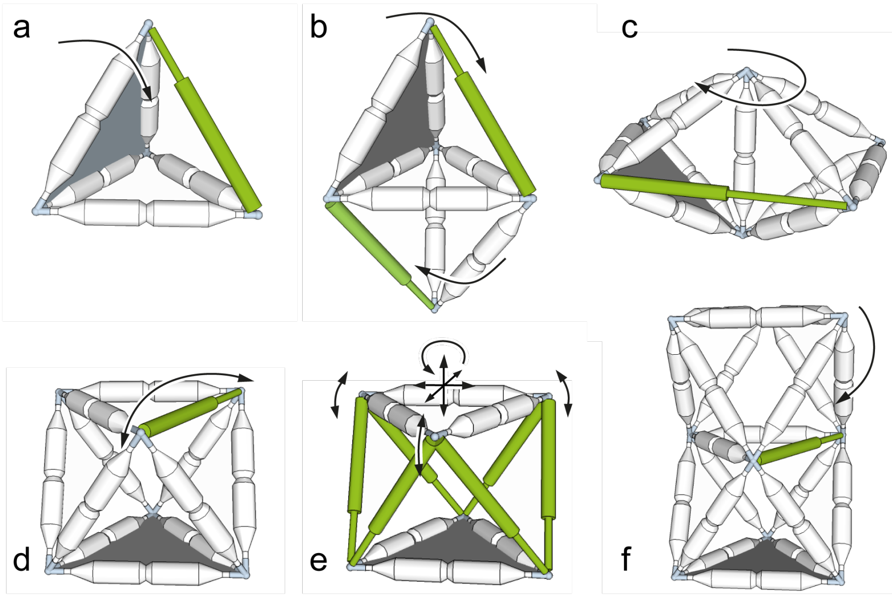


Figure 3.4: A selection of assets: (a) tetrahedron with 1DoF, (b) "robotic leg" asset, (c) hinging tetrahedra, (d) octahedron with 1DoF, (e) Stewart platform (6DoF), and (f) double-octahedron performing "bending" motion.

### 3.2.3 *Manual Actuator Placement*

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* (8) to turn any edge into or connect two nodes with an actuator. This can be done by clicking on the desired edge or selecting two nodes to be connected. Transforming an existing edge 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.

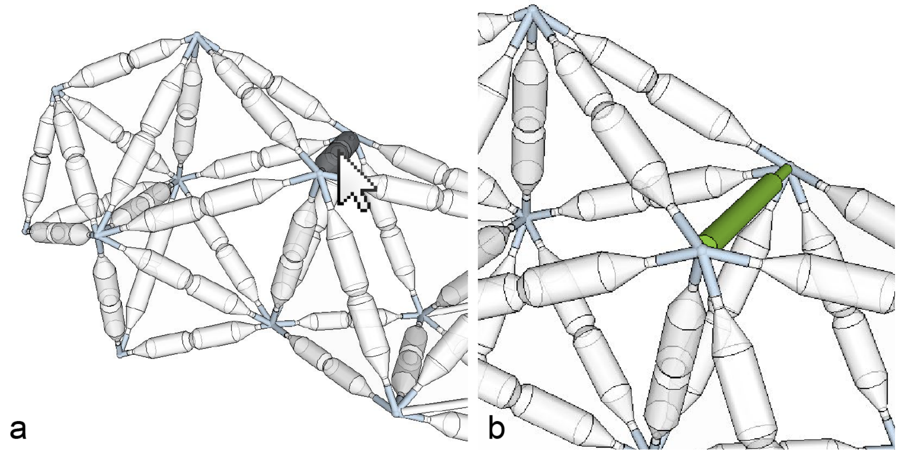


Figure 3.5: The turn edge to actuator tool allows users to turn any edge into an actuator. Here user replaces one edge in the T-Rex's head to make its jaws move.

### 3.2.4 *Stability Check*

During the placement of actuators, TrussFormer verifies that the mechanism is structurally sound. The system finds the safe range of actuation in the background, i.e. how far an actuator can extend without damaging the structure, by simulating the occurring forces in a range of positions. Damage can occur if the structure falls over, hits the ground with too much speed or if too much force is exerted on a structural element, such as an edge. To do this, TrussFormer iteratively extends each actuator and observes the forces. If the force exceeds a preset limit, the last safe actuation range is stored in the actuator. The user will then not be able to over-actuate this actuator.
To completely prevent breaking, this process would have to be done or each combination of actuators. As this does not scale well with an increasing number of actuators in the system, we perform this step for each actuator individually. An additional security check is done while the user tries out the motion manually, as explained in Section 3.3.

## 3.3 ANIMATION

The user can actuate the physics links in two different ways. The *animation window* provides a slider for manual movement (9), as well as an animation pane which allows placing keyframes (10) and provides two playback modes: single and looping (11).All actuators have an actuator group assigned to them. Per default, an actuator is placed in wording

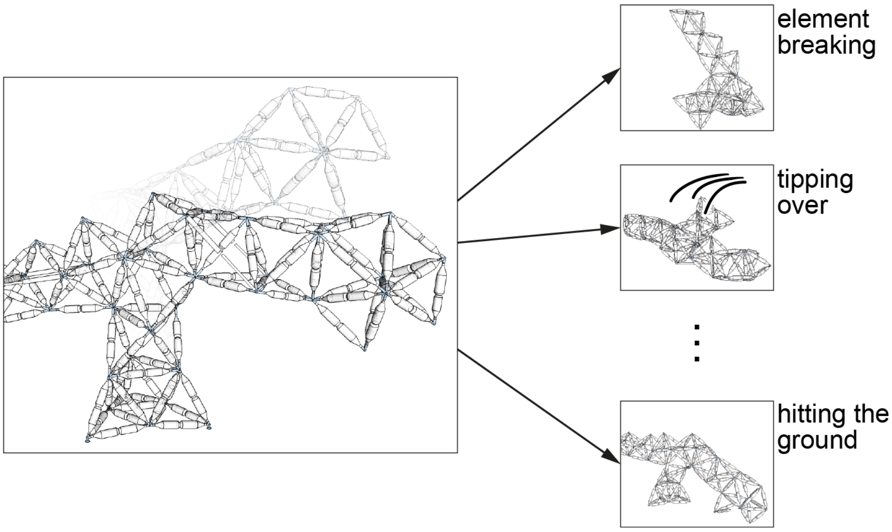


Figure 3.6: In the background, TrussFormer tests each actuator to see if its extension leads to invalid configurations, such as the structure falling over, hitting the ground, or encountering broken structural elements.

its own, empty group. Using the actuator tool (8), already used for placing actuators, this group can be changed. This is indicated by the actuator changing its color.
The 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

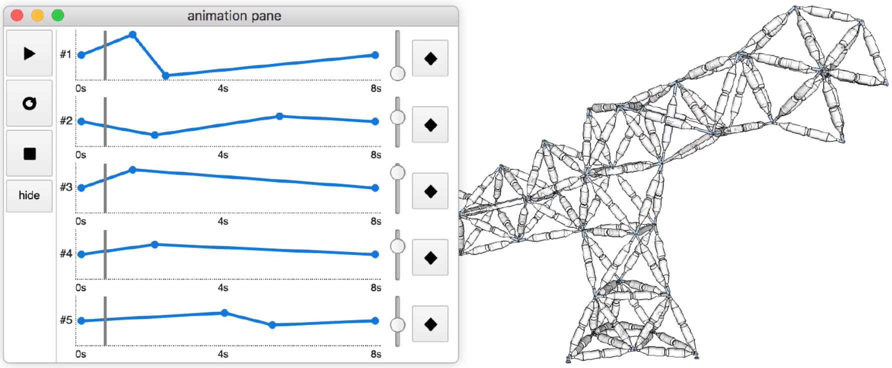


Figure 3.7: Animating the structure. Users sets the desired pose using the sliders in the animation pane and orchestrates the movement by placing key-frames on the timeline.

Redo with actuator groups and numbers

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.

### 3.3.1 *Checking Dynamic Forces*

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 forces. TrussFormer aids the user to detect weak points and force peaks during a motion in different ways.
TrussFormer'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.
As shown in Figure 3.8 (a) the user creates an animation that moves the T-Rex body down and up again. (b) While TrussFormer plays the animation, it calculates the forces. The change from the downwards motion to going up again causes a lot of contraction force on the edges on the front body side. The neck of the T-Rex acts as a lever, which leads to very high inertial forces. This motion eventually exceeds the breaking limit of the structure, (c) causing the construction to fail. The time of breaking is indicated in the animation pane by a red vertical line. This failure is hard to predict by the user, because inertial forces can be multiple times higher than the static forces, checked during the creation of the T-Rex. (d) TrussFormer helps the user to fix the motion. A popup is opened providing two possibilities (Figure 3.9): fixing the animation by reducing the speed or reducing the motion. If the first option is chosen, TrussFormer will elongate the animation sequence for all piston groups. This results in a slower movement and less force on the structure. The second option 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.

On top of the coloring of edges, the user can also use the *sensor tool* (12). This tool can be used on a single edge to observe it more closely. The force data on an edge with a sensor will be recorded and visualized over time in a chart. The sensor tool also works on nodes, visualizing speed and acceleration data, instead of force. The result

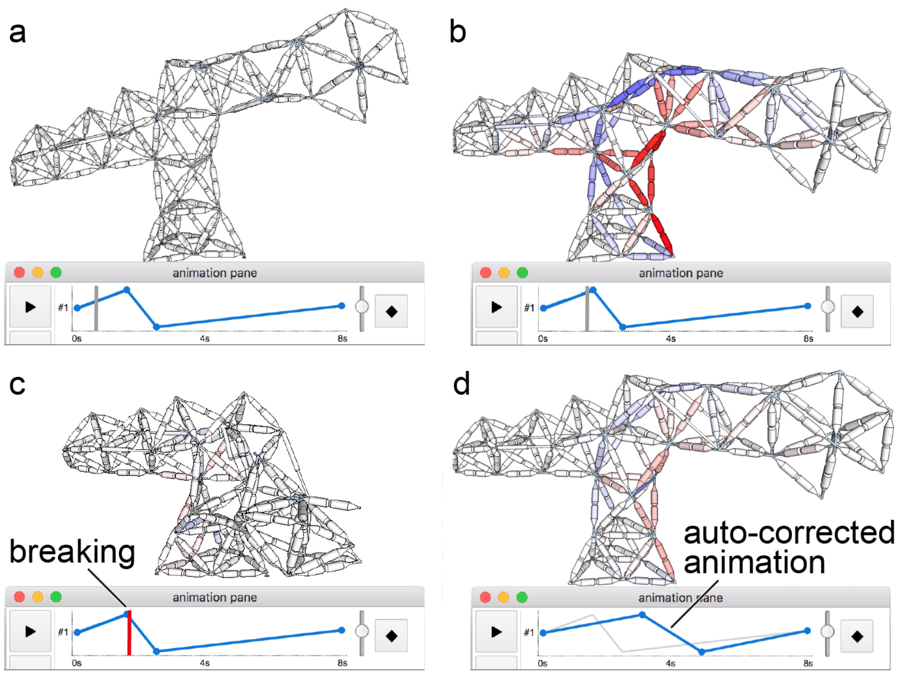


Figure 3.8: TrussFormer verifies that the structure can withstand the inertial forces resulting from the programmed animation. (a-b) The forces in the T-Rex increase with the movement. (c) The structure breaks when the direction of the movement changes rapidly. (d) TrussFormer resolves this by making the movement slower.

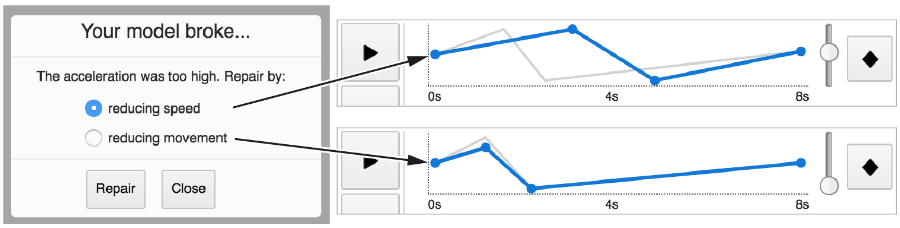


Figure 3.9: If the user-defined animation breaks the model, TrussFormer offers to automatically reduce the speed or the motion range.

can be seen in figure 3.10.

## 3.4 FABRICATION

After the object was sufficiently tested in the editor, it is time to print the connectors and assemble the final object. To do this, TrussFormer 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 cre-

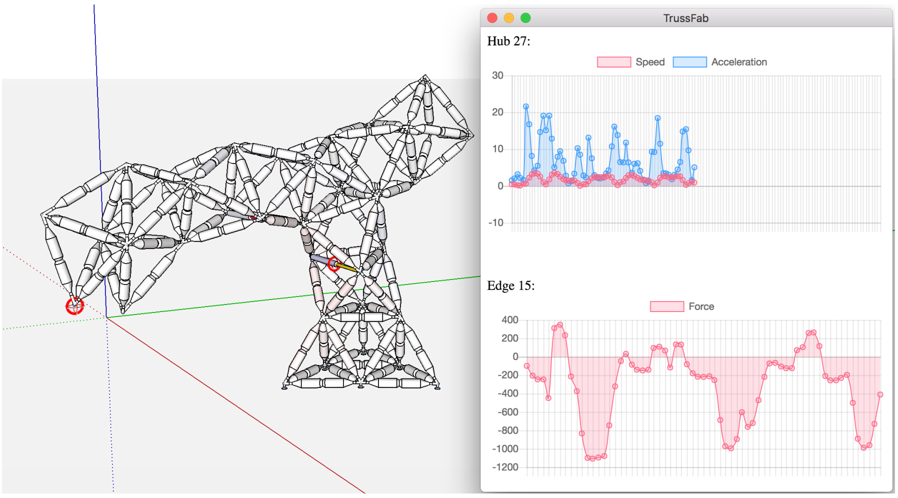


Figure 3.10: A sensor measures the force on the central actuator, and speed and acceleration on a node at the "nose" of the dino

ate OpenSCAD files - a modeling language which we use to modify templates of hubs and hinges.

### 3.4.1 *Export*

When users are satisfied with their design (structure, movement, and animation), they click the *fabricate* button (15). This will trigger TrussFormers' hinge generation procedure and create files which can be used for 3D printing. In order to control the structure the same way as in the animation pane, the animation will also be exported in an Arduino-readable format. Instructions for building will also be provided.

If the fabricate button is pressed, TrussFormers' hinge generation procedure is triggered. It analyzes the structure, ensuring that the final object will have the maximum possible movement and creating 3D printable hinge and hub geometries. The T-Rex consists of 42 3D printed hubs and 135 unique hinge pieces.
Next, the animation pattern is exported to Arduino code. The user can upload this code directly to their microcontroller.
As a last step, specifications containing information about the force, speed and motion range of the actuators necessary for achieving the desired animation pattern. These actuators can be obtained as standardized components. The specifications also include information about the number and size of screws needed for assembly.

### 3.4.2 *Printing the Parts*

Each exported file represents a single part in the structure. These files can easily be converted to *.stl* files, which are typically used for

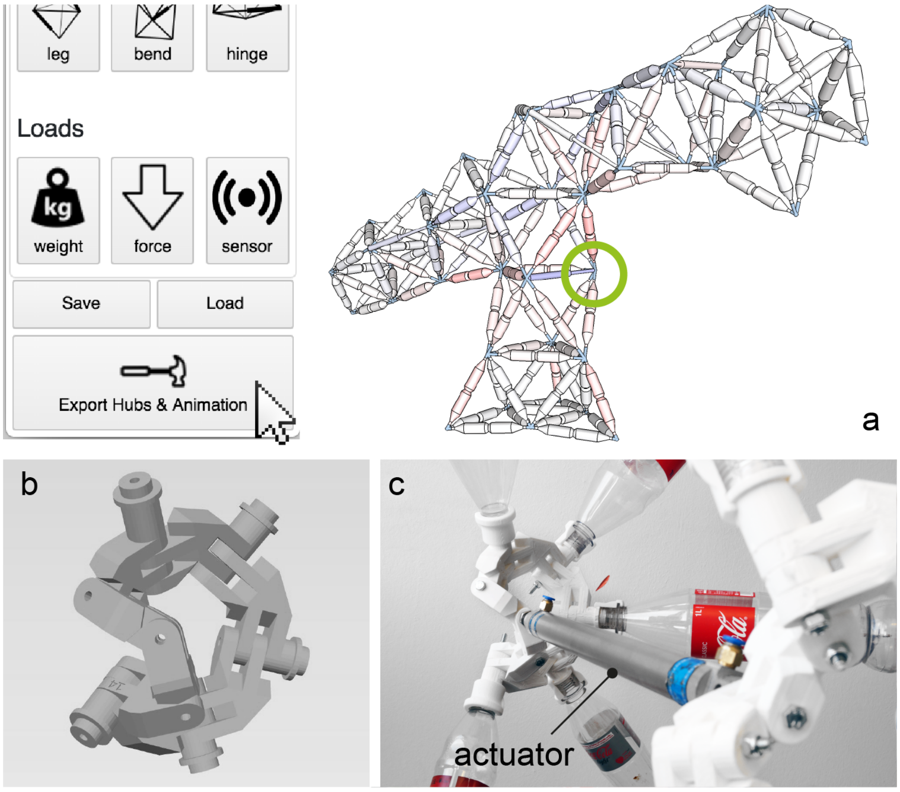


Figure 3.11: (a) To fabricate our T-Rex model, TrussFormer exports: (b) the appropriate 3D printable hinging-hubs for fabricating the structure, (c) and the actuator specifications that inform the users which one to buy. TrussFormer also exports the animation sequence for an Arduino.

3D printing. Using a 3D printing software, the 3D objects can be arranged and sent to a 3D printer. Printing of one hinge part using an UltiMaker 3 printer takes about 40 minutes.

### 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.

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

This will be a better image.

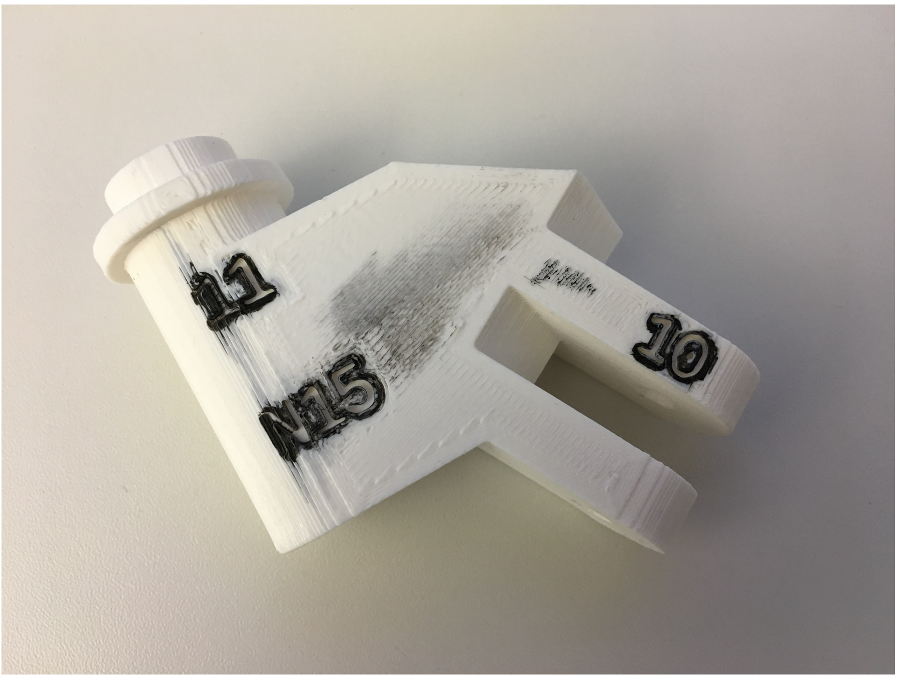


Figure 3.12: A hinge

# 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 MECHANICAL COMPONENTS

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 *static hubs*.
Links, and static or hinging hubs are connected by purpose-designed *cuffs*, which fit over the bottles' thread on one side and a connector part on the hub on the other side.

### 4.1.1 *Edges of the structure*

We opted to use 1l (big) and 0.5l (small) reusable PET bottles because they have thicker walls, providing more stability and are abundantly available. 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 *Rigid Hubs*

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

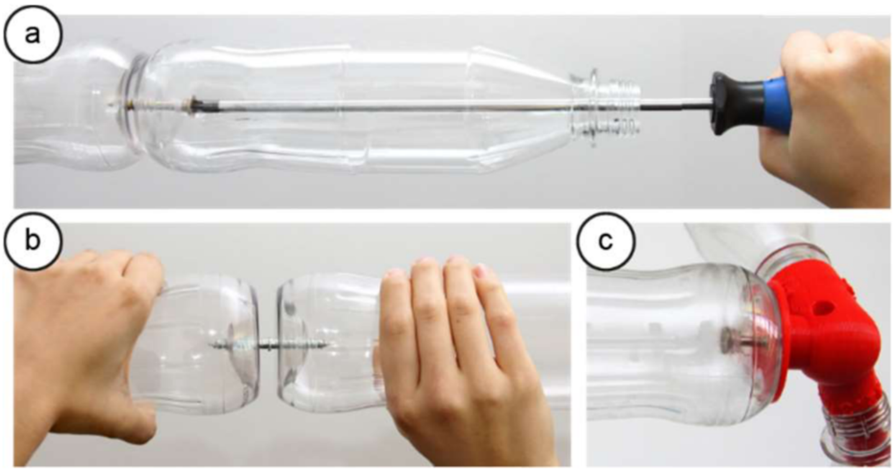


Figure 4.1: (a) Connecting bottles with a wood screw using an extra-long screwdriver and (b) with a double-ended screw. (c) Single-bottle edges require a bottom connector

Figure 4.2: The connector is inserted into the bottle opening ...

Figure 4.3: ... and secured with a cuff

Hubs are solid one-piece objects that can connect two or more links. 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 is plugged into 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, while giving enough stability once clipped on.

### 4.1.3 *Hinging Hubs*

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 motion can potentially be achieved using ball joints, however these joints typically only allow for a connection of two edges without causing obstruction.
We solve this issue by using a *spherical joint mechanism*. As can be seen in Figure 4.4, 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

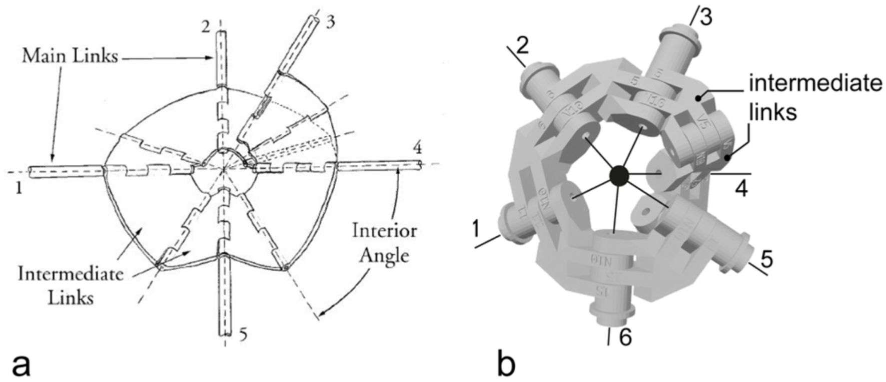


Figure 4.4: (a) spherical joint mechanism connecting 5 edges. (b) rendering of TrussFormers' hinge chain design connecting 6 edges.

had special end parts which were only connected on one side. 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 came up with a closed hinge chain design.
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 4.4a connected all edges using two hinges, allowing for 2 DoF. This is not necessary if we are dealing with truss structures. An example of TrussFormers's hinge design can be seen in Figure 4.4b. Only edge 4 is connected to its neighboring edges using two intermediate hinges. All other edges only require rotation (resulting in 1 DoF).
The hinge chains are arranged automatically in the structure by our TrussFormer system. This heuristic approach will be described in more detail in Section 5.4.1.

#### 4.1.3.1 *3D printing*

The hubs and connecting cuffs are printed using consumer-grade desktop FDM[1] 3D printers. The filament consists of PLA[2] plastic and the printed objects have a 15% infill with a wall thickness of 3 mm. We chose PLA filament instead of ABS[3] primarily because of it is easier to handle. ABS plastic, while being stronger, needs to be printed on a heated surface, which not many mid-range printers have. As we designed the TrussFormer system for 3D-printing enthusiasts and not professionals, we wanted to target owners of consumer-grade printers. Additionally, PLA consists of organic materials (mainly corn-starch and sugarcane), which makes it biodegradable, as opposed to ABS plastic, which is oil-based.
A hub consumes about 50-150g of filament, resulting in a cost of 1-3€.

## 4.2 ACTUATION

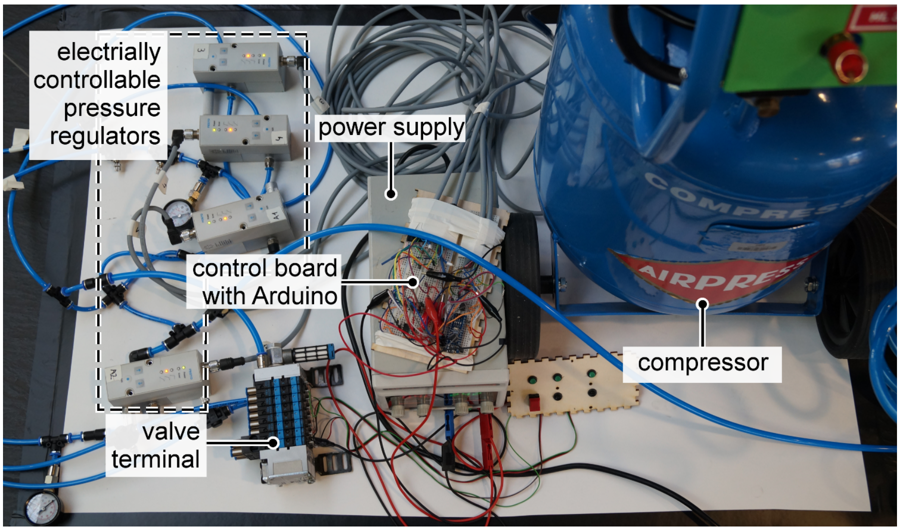


Figure 4.5: Hardware setup for controlling the T-Rex, with Arduino, electric pressure control valves, and compressor.

Figure 4.5 shows the main parts of our hardware setup. The electrically controllable pressure regulators receive signals from the control board and limit the air pressure according to the needed level of extension of a given actuator. One pressure regulator is connected to exactly on 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

1 Fused Deposition Modeling
2 Polylactic acid
3 Acrylonitrile butadiene styrene

actuator. If it closes it, the pressure will stay constant and the actuator keeps its position.

### 4.2.1 *Electric Actuators*

Early TrussFormer 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 2 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 *Pneumatic Actuators*

We use different kinds of pneumatic actuators, depending on the force needed. Our actuators usually have a piston diameter of 20 - 35 mm and can extend from 40 cm to 80 cm. This length works well with our bottle links, which are 60 cm long, meaning that the middle position of the actuator makes it the same length as the links.
A common actuator that could be used is the Festo DSNU round cylinder with a diameter of 32 mm and a hub of 30 cm. This cylinder can operate with up to 10 bars, the following values were measured with 6 bars of pressure and at room temperature. Under these circumstances, this actuator can achieve an extension rate of up to 2.3 m/s (without load) with a force of 480N on the extending stroke and 415N retracting. This force is sufficient for moving the head of our T-Rex up and down, which is the motion requiring the most force in our design. For higher loads, e.g. lifting a human, an actuator with a bigger diameter should be used. A Festo actuator with a diameter of 50 mm is capable of achieving up to 1200 N of force on the extending stroke at 6 bars.

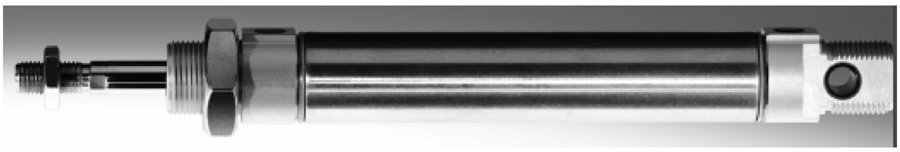

Figure 4.6: A Festo DSNU cylinder with 32 mm cylinder diameter and 30 cm of hub, as used in our T-Rex.

## 4.3 CONTROL SYSTEM

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 send to the extending input and the slider at the bottom completely retracting it.

### 4.3.1 *Valves*

We used two different kinds of valves. *Proportional pressure regulators*, such as the Festo VPPM, are used for open-loop pressure control. Communication to these regulators is achieved with a IO link, which can be connected to a control unit. Digital signals are translated to proportional opening of the valves, providing a constant and reliable pressure. VPPMs have one input and one output valve. This fact differentiates proportional pressure regulators from *proportional directional control valves*, like the Festo MPYE. These valves have one input, but two outputs, making it possible to connect it to both inputs of a pneumatic actuator. Controlling these valves works similar to proportional pressure regulators and also uses a simple IO link. Digital signals send to this type of valve do not, however, only control the pressure of these outputs, but also the ration between these two outputs. This makes positional control of actuators possible.

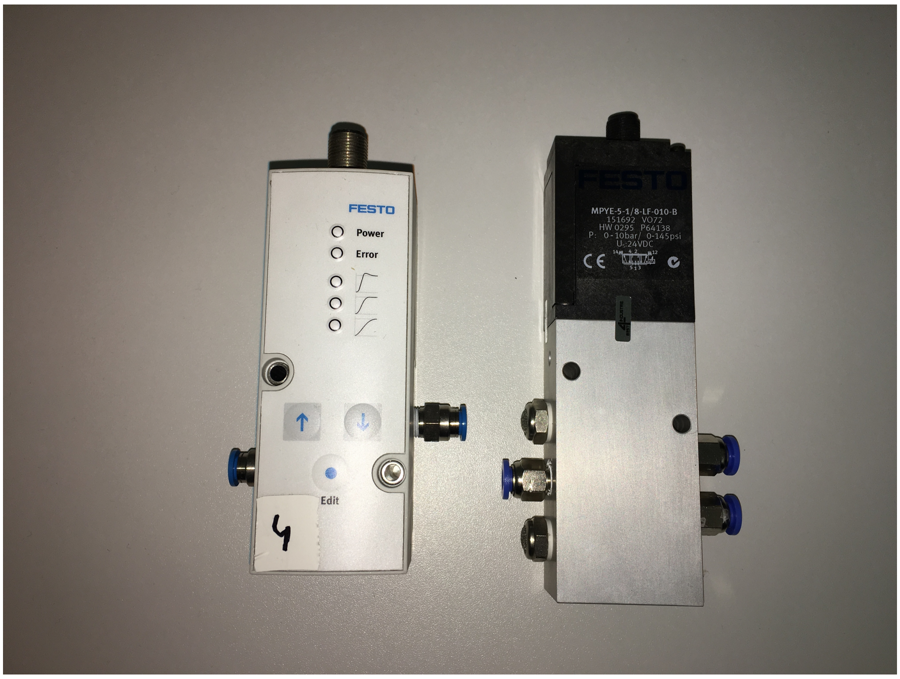


Figure 4.7: TEMPORARY(left) A proportional pressure regulator with one input and one output. (right) A proportional directional control valve, having one input and two outputs. Both can be attached to an IO link on the top.

### 4.3.2 *Position Sensing*

The MIDI controller is a well working possibility to control the actuators. It is, however, still an open-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 undisturbed and without external interaction. If we want to 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 extension and apply force accordingly.
We created a rudimentary motion-platform using closed-loop PID control (s.a. Section 5.3.1.1). 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, which made the motion platform oscillate up and down.

### 4.3.3 *Arduino/Controllino*

Also for our control module we have different approaches for open and closed loop control. Open loop control can be achieved with a simple *Arduino*. It can be attached to a control interface, such as our MIDI controller, and translate the given inputs into signals for the valves.
Another logic controller we used is the *Controllino*. Apart from digital and analog inputs and outputs, it also features relays.

Why do we use it? What can it do actually?

Figure 4.8: TEMPORARY We use a badge holder with a rotary encoder for position sensing. The string of the holder is attached to the actuators body and the extending piston. On extension, the string will roll off the spindle, which we can measure with the encoder and translate into a length.

# 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 SketchUps 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 TrussFormer 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*[1] physics engine, which is a Ruby wrapper around the C++ physics engine *NewtonDynamics*[2]. 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

1 https://extensions.sketchup.com/en/content/msphysics
2 http://newtondynamics.com/forum/newton.php

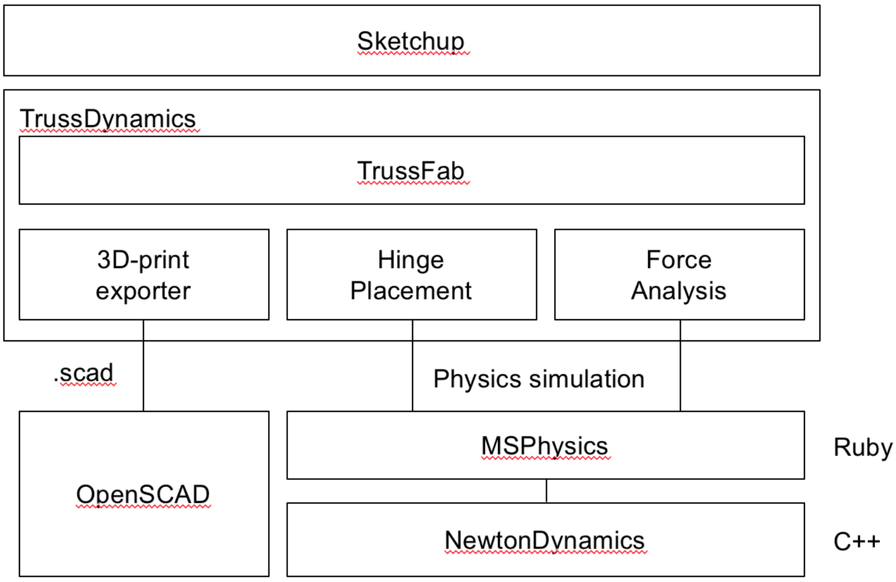


Figure 5.1: TrussFormer Architecture

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

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

Show before and after code snippet

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.

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 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

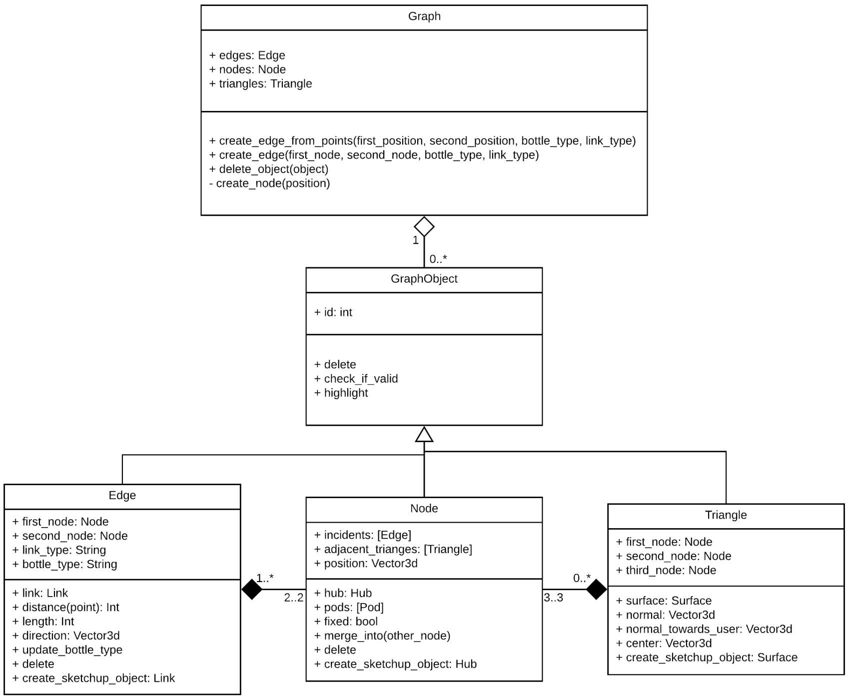


Figure 5.2: Class Diagram showing the high-level Graph Structure of the TrussFormer Designer

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 5.2.3.

```ruby
def merge_into(other_node)
  merged_incidents = []
  @incidents.each do |edge|
    edge_opposite_node = edge.opposite(self)
    next if other_node.edge_to?(edge_opposite_node)
    edge.exchange_node(self, other_node)
    other_node.add_incident(edge)
    merged_incidents << edge
  end
  @incidents -= merged_incidents

  merged_adjacent_triangles = []
  @adjacent_triangles.each do |triangle|
    new_triangle = triangle.nodes - [self] + [other_node]
    next unless Graph.instance.find_triangle(new_triangle).nil?
    triangle.exchange_node(self, other_node)
    other_node.add_adjacent_triangle(triangle)
    merged_adjacent_triangles << triangle
  end
  @adjacent_triangles -= merged_adjacent_triangles

  delete
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 TrussFormer that this Node should not change its position.

add image

The *Edges* are the most visual components of TrussFormer. 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 TrussFormer 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.

IMPROVE!

### 5.1.1 *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

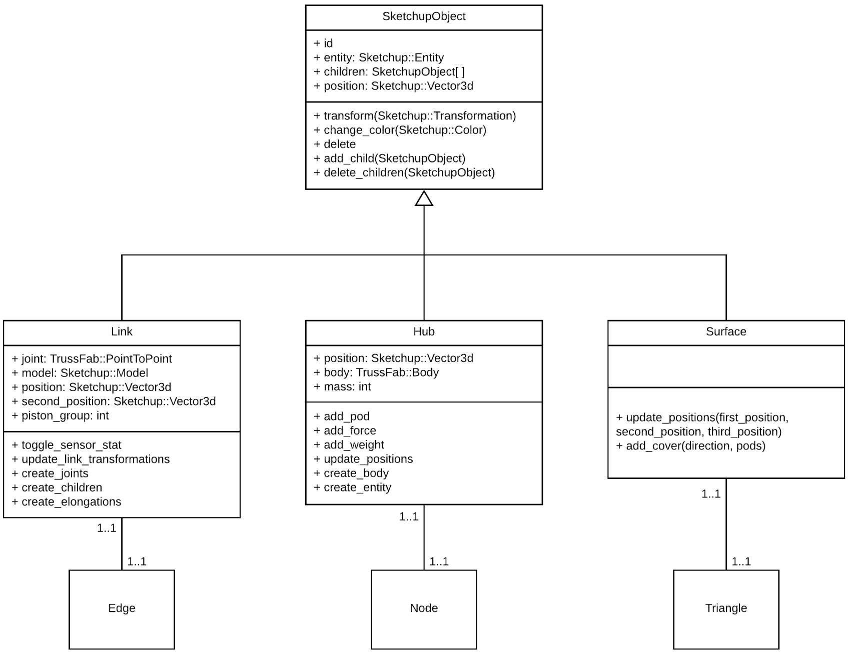


Figure 5.3: Class Diagram showing the UI components of the graph structure

#### 5.1.1.1 *Hubs*

*Hubs* are the underlying structures used by Nodes. For ease of calculation and increased performance of the *Simulation* (s.a. section 5.3), 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.

#### 5.1.1.2 *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:

- TrussFormer::PointToPoint - A static connection between two points
- TrussFormer::PointToPointActuator - A variable-length connection between two points
- TrussFormer::PointToPointGasSpring - A variable-length connection between two points with a distance-based force factor
- TrussFormer::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 5.3.

#### 5.1.1.3 *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 EDITOR

The TrussFormer Editor 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.

### 5.2.1 *User Interface*

The user interface (UI) is written in JavaScript. It uses SketchUps' built-in *HtmlDialog* class, which lets us interact with HTML dialog boxes using the Ruby API[3]. The HtmlDialog is a modified version of Googles' Chrome browser and supports modern HTML5 code, as 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

3 Application Programming Interface

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.

```
Class AnimationPane
  def add_piston(id)
    @dialog.execute_script("addPiston(#{id})")
  end

  def stop_simulation
    @dialog.execute_script('resetUI();')
  end

  def register_callbacks
    @dialog.add_action_callback('start_simulation') do |_ctx|
      if @simulation_tool.simulation.nil? ||
         @simulation_tool.simulation.stopped?
        start_simulation_setup_scripts
      end
    end

    @dialog.add_action_callback('stop_simulation') do |_ctx|
      unless @simulation_tool.simulation.nil? ||
             stop_simulation
        Sketchup.active_model.select_tool(nil)
      end
    end

    ...
  end
end
```

Listing 5.2: excerpt from UI callbacks

As can be seen in listing 5.2, 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.

### 5.2.2 *Tools*

#### 5.2.2.1 *Stability Check*

#### 5.2.2.2 *Automatic Actuator Placement*

#### 5.2.2.3 *Correct Forces*

### 5.2.3 *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 it 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.

```
def relax
    # Abort if there is nothing to do
    return if @edges.empty?
    number_connected_edges = connected_edges.length
    @max_iterations.times do
      # pick a random edge
      edge = @edges.to_a.sample
      # only adapt edge if we still have stuff to do
      deviation = deviation_to_optimal_length(edge)
      next if deviation.abs < CONVERGENCE_DEVIATION
      # add neighbors if we still have edges left to add
      add_edges(edge.incidents) unless @edges.length ==
    number_connected_edges
```

```
      adapt_edge(edge, deviation * @damping_factor)
    end
    move_nodes_to_new_position
    self
  end
```

Listing 5.3: The relaxation algorithm

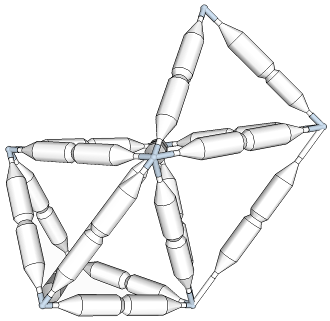

Figure 5.4: The relaxation algorithm applied with only one iteration. The extension of the lower right edge resulted in growing incident edges as well.

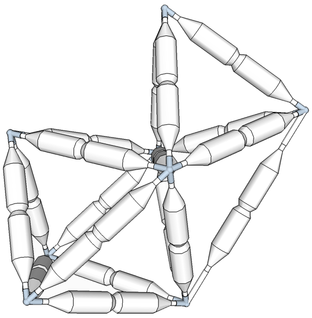

Figure 5.5: The relaxation algorithm applied with up to 20'000 iterations. Growing the lower right edge caused other nodes to translate, but the lengths of incident edges stayed the same.

## 5.3 PHYSICS SIMULATION

TrussFormers' 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, TrussFormers' 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 Synytsia[4]. 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

4 https://github.com/AntonSynytsia/

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 5.4), 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.

```
def nextFrame(view)
    view.show_frame
    return @running unless @running && !@paused

    update_world
    update_hub_addons
    update_entities

    if (@frame % 5).zero?
      send_sensor_data_to_dialog
    end

    @frame += 1
    update_status_text

    @running
end
```

Listing 5.4: 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

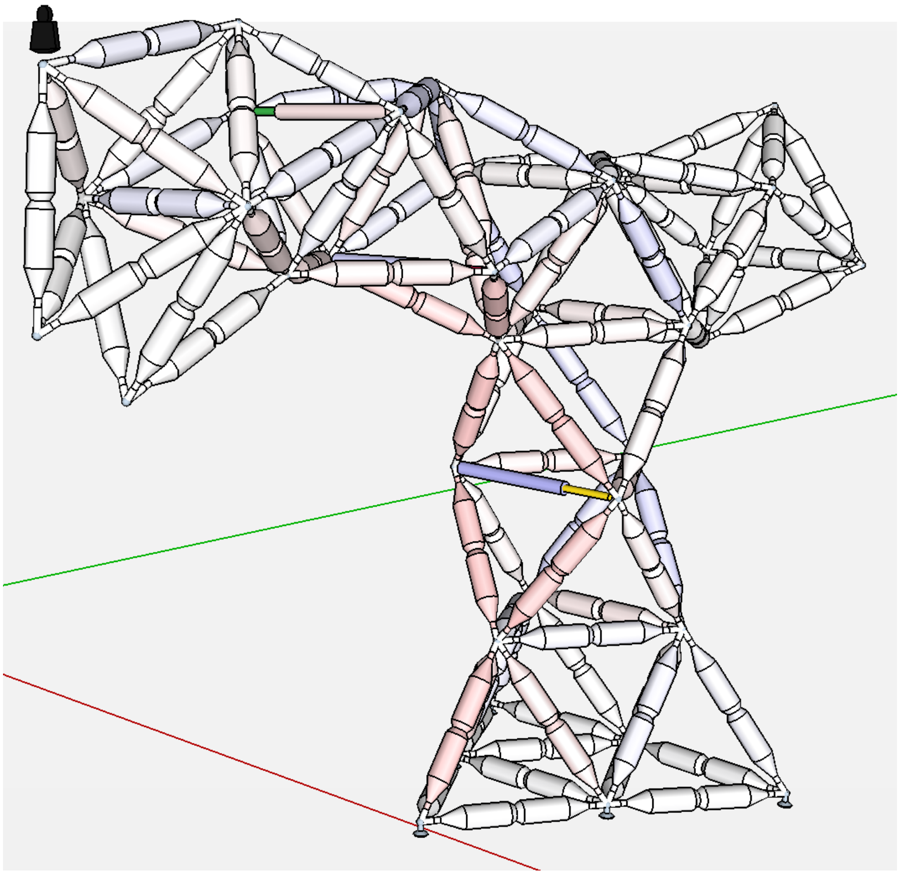

Figure 5.6: Visualization of forces acting on edges. Blue - tension force, red - compression force, white - little or no force

graph structure has to be updated. For that, each edge and node updates their own positions and transformations based on their internal physics objects.

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

### 5.3.1 *Force Control*

#### 5.3.1.1 *PID*

### 5.3.2 *Evaluation*

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

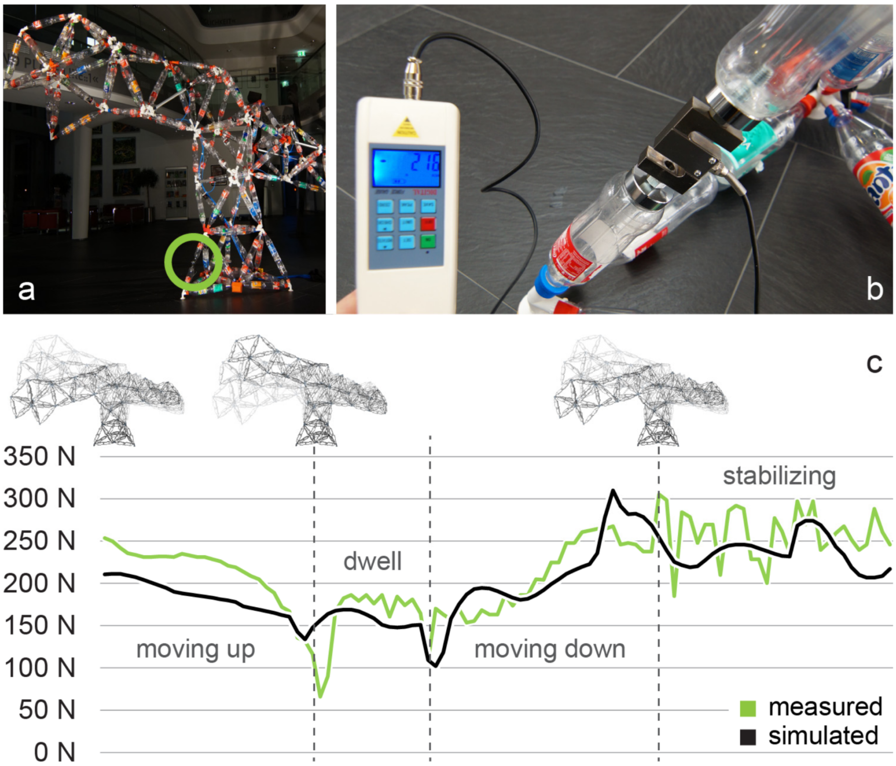


Figure 5.7: (a) We measured the forces on the bottom front edge of the T-Rex (b) using a digital force gauge. (c) The measured forces agree with the simulated forces.

rate of 0.5%, on the front bottom edge of the T-Rex (Figure 5.7 a-b). It was placed between two small bottles, giving the edge the same 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 5.7c shows, that both, the simulated and the measured force, are in agreement.

## 5.4 HINGE SYSTEM

The export of 3D printable files is one of the main contributions of TrussFormer. It enables users to create their designed objects in the real world without the need to understand the complex hinge systems necessary to create the desired movements.
The export functionality consists of multiple steps. First, the system determines where hinges need to be placed and where static hubs are sufficient. After that, the possible motion has to be maximized. The bottles need a minimum distance to the rotation center of each hub in order to not collide with each other, to properly secure them using the cuffs and align them with the hinge movement.

### 5.4.1 *Hinge Placement Routine*

We use an empirical approach to determine the placement of hinges. Using the physics engine, we move all actuators, one after another, and measure the angular difference between adjacent triangles. If the difference exceeds a certain threshold, the common node of these two triangles needs a hinge connecting adjacent edges.
First, all static groups are determined. Static groups are connected structures, in which no rotation is happening, while actuators are moving.

### 5.4.2 *Minimization Logic*

In order to achieve the maximum amount of motion, while keeping the minimum amount of printing time, we needed to find a solution for minimizing hinge and hub sizes.
For hinges, three lengths are of importance. These lengths consist of the distance from the rotation center to the lower side of the hinge connector (l1 distance), the height of the connector itself (l2 distance) and an optional third length being the elongation of this hinge, i.e. the length of the edge connector (l3 distance). TODO: I am actually not sure if we do anything special here. Check this. - elongates and shortens edges so that maximum movement is possible with minimum material use
- uses iterative relaxation algorithm, will be explained in 5.2.3

### 5.4.3 *Generating the 3D Models*

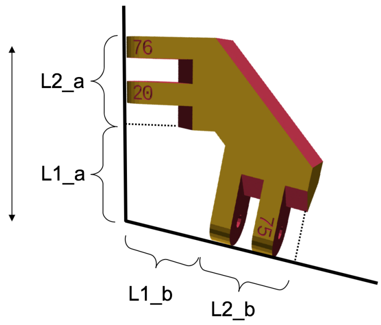


Figure 5.8: L1 and L2 lenghts of a hinge part

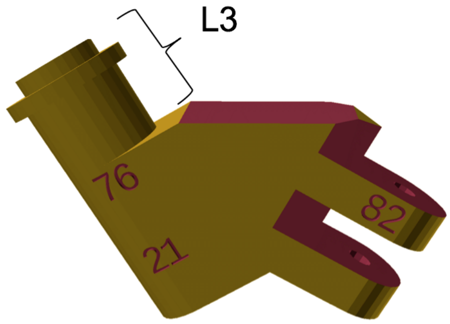


Figure 5.9: L3 length of a connector

# 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