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 and the creation of large-scale objects is mostly a privilege of industry.

TrussFormer aims to enable users to create large-scale dynamic objects using desktop-sized 3D printers. We achieve scale by creating multiple small-sized objects and combining them. However, even this technique of breaking down objects into printer-sized parts does not scale on its own, because large models consume material and time proportional to their size. 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 of creating large objects, we developed a software simulation that can create objects which are capable of handling the substantial forces such objects produce. Forces grow cubed with the size of the object, so designing large objects requires substantial engineering skills. TrussFormer provides 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.

Our work is based on *TrussFab* (TODO: Reference TrussFab paper).

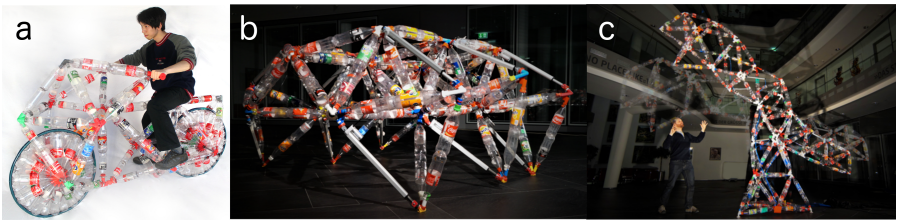

Figure 1.1: TrussFormer enables users to create structurally stable truss structures, such as (a) a motor bike, (b) a walking spider or (c) an animated T-Rex.

TrussFab provides possibilities for creating large-scale static truss structures and supports a wide range of applications, like furniture or art installation. However, systems like TrussFab are limited to static structures. This paper will explain how we extended this area of architecture by providing possibilities to create large-scale *kinematic* truss structures.

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 a truss 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.

1.1 TRUSSFORMER'S UNDERLYING STRUCTURE ACHIEVES STABILITY

To achieve stability in big structures, TrussFormer incorporates two main ideas. We employ bottles as links in their structurally most sturdy way, i.e. from bottom to bottle neck. As a complement, these links are connected to sturdy "closed frame structures", also known as trusses.
As demonstrated in Figure 1.2 (a/b), freestanding bottles tend to break easily. Truss structures consist of triangles, which create a structure that prevents deformation, reducing the force on individual bottles. Occurring lateral forces are turned into compression and tension forces along the length of the links. This makes bottles a great choice for trusses. They buckle easily when loaded from the side, but are very strong along their main axis. (c) The resulting structure, such as a tetrahedron, is strong enough to bear the weight of a human. (d) Multiple of these *primitives* can be combined to create truss honeycomb structures. We decided to use tetrahedra and octahedra as primitives, as they are space filling and make construction simpler than, for example, icosahedra.

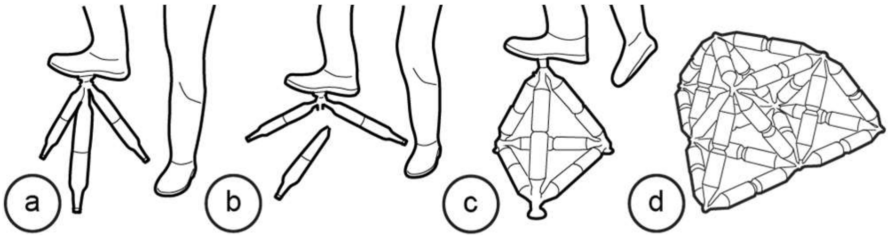

Figure 1.2: (a) Large objects involve large levers, causing bottle links (b) to break under load. (c) TrussFormer instead affords structures based on closed triangles, here forming a tetrahedron. Such structures are particularly sturdy. (d) TrussFormer extends this concept to tetrahedron-octahedron trusses of arbitrary size.

1.2 ADDING MOVEMENT USING VARIABLE GEOMETRY TRUSSES

Apart from creating structurally stable structures, TrussFormer helps users to bring those objects into movement. It does this by incorporating linear actuators into rigid truss structures in a way that they move "organically", i.e., hinge around multiple points at the same

time. These structures are known as *variable geometry trusses* (TODO: Reference). Figure 1.2 illustrates this on the smallest elementary truss, (a) the rigid tetrahedron. (b) We swap one of the edges with a linear actuator, (c) resulting in a variable geometry truss. The only required change for this is to introduce hubs that enable rotation at the nodes. We call these hinging hubs. This simple approach to create variable

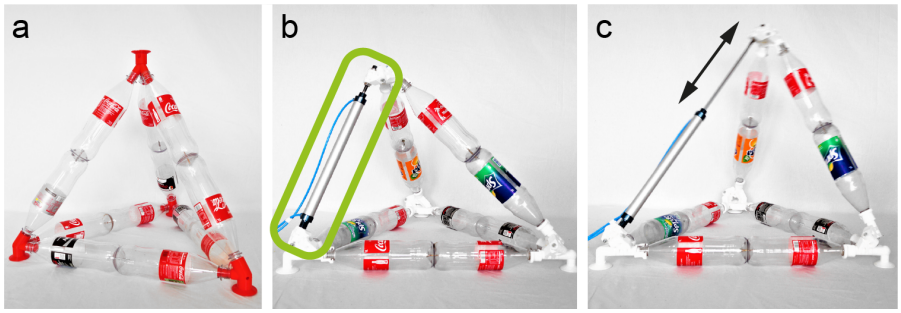

Figure 1.3: (a) The static tetrahedron (b-c) is converted into a deformable structure by swapping one edge with a linear actuator. The only required change is to introduce connectors that enable rotation.

geometry truss mechanisms scales well to arbitrarily larger structures. Our T- Rex model from Figure 1.1 (c) contains five linear actuators and thus offers five degrees of freedom (DoF). As shown in Figure 1.4, it can (a) lift or lower its neck (1 DoF), (b) turn its head left and right (1 DoF), (c) sweep its tail (2 DoF), and (d) open its mouth (1 DoF).

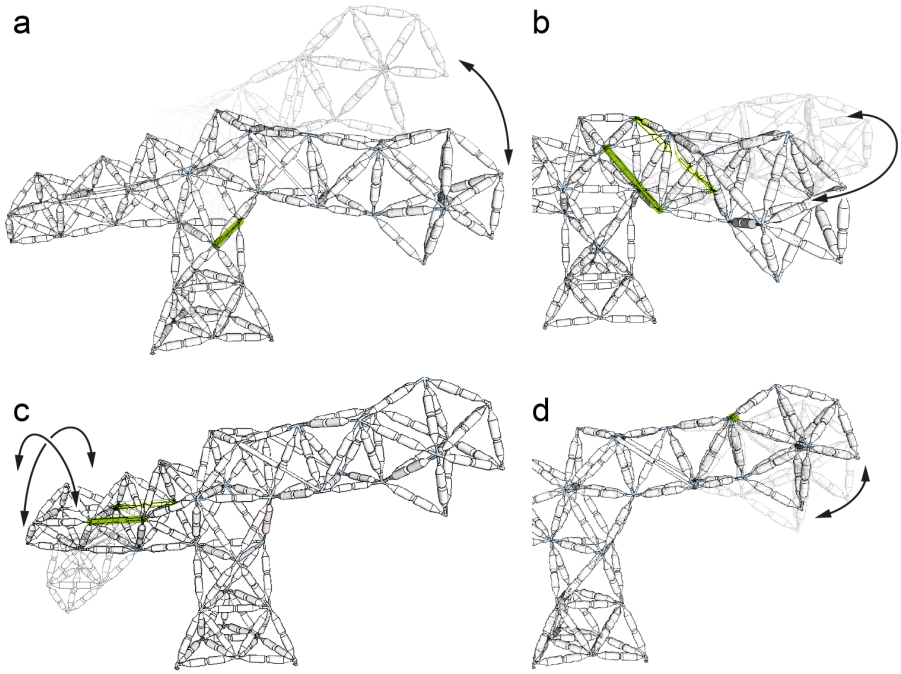

Figure 1.4: The T-Rex offers 5 degrees of freedom.

1.3 TRUSSFORMER EDITOR

The TrussFormer editor is the essential part of our system. It is a plugin for the 3D modeling software SketchUp and enables users to create truss structures by placing truss primitives. Apart from designing these truss objects, the editor also allows users to test static and dynamic forces by displaying them during an interactive simulation, create animations and export the design. Using the export functionality, users can recreate the designed object in the real world by 3D printing the generated STL files and using the exported animation sequence with an Arduino controller.

2 RELATED WORK

2.1 LARGE-SCALE PERSONAL FABRICATION

Large-scale personal fabrication has been a topic in research for a long time. Especially in the metal industry and architecture, scaled-up machinery, such as concrete 3D printers, are used for additive manufacturing. The Denmark-based company *3D Printhuset*[1] offers a product called *building-on-demand*. Using a 6m x 6m x 3m concrete printer, the company printed a small office building within 50 hours. A variety of other companies also offer concrete 3D printers, some being able to build a house in just a day.

Instead of making the printer bigger, other solutions work by breaking up the object into smaller parts and print them on desktop machines.

Figure 2.1: Apis Cor's concrete 3D printer can create a house in 24 hours.

3D printers use a fixed frame the print head moves on, limiting the size of the printed object to the size of the frame. To work around this limitation, researchers proposed techniques to place the print head on a mobile platform or make it fly by hanging it from a drone.

Yet another approach is to use human assisted devices, such as Protopiper.

1 https://3dprinthuset.dk/

2.2 CONSTRUCTION KITS

Construction kits are used for fast prototyping and fabrication. Prefabricated elements can be combined generically in various ways, depending on the desired outcome. The ZomeTool by Henrik and Kobbelt (TODO: Reference) is a mathematical modeling kit that can accurately approximate complex shapes, such as the Stanford Bunny. Skouras et al. (TODO: Reference) created an interactive editor to computationally combine interlocking elements and approximate any shape.

Figure 2.2: The Stanford Bunny modeled with the ZomeTool construction kit.

2.3 PROTOTYPING WITH READY-MADE OBJECTS

MixFab [33], and Encore [3] allow users to integrate existing objects into their design. For creating objects enclosing electronic components Ashbrook et al. [2] developed an augmented fabrication system. Devendorf and Ryokai [6] proposed a human-assisted fabrication system that helps users incorporate everyday objects into 3D print. Beady [10] approximates models from beads and offers an editor for refining them.
Gellért assembled wooden boards combined with 3D printed connectors in a node-link structure [8]. Combining carbon tubes with 3D printed metal has been proposed for creating functional cars [40]. Skilled individuals have stacked or tied plastic bottles in order to make art pieces, furniture, rafts or houses [39]. Yamada et al. [36] proposed a system for arranging ready-made objects into 3D shapes using 3D-printed connectors.

2.4 BUILDING WITH VARIABLE GEOMETRY TRUSSES

Fumihiro Inoue (TODO: Reference (book Robotics and Automation in Construction, chapter 16)) proposes the use of variable geometry trusses (VGTs) to introduce flexible movement into otherwise static

structures. First used in space technology (TODO: reference to Natori & Miura, 1994), variable geometry trusses are widely used today.
One use case is the Stewart platform. A Stewart platform consists of six actuators. A pair of two attach to a base plate, and cross over to attach to a top plate as a pair with their respective other neighbor actuator. This setup allows motion with 6 degrees of freedom (DoF), while keeping the stability of a truss. It is used in situations where high inertial forces can occur, such as motion platforms.
VGTs are also widely used in robotics. Tetrobot (TODO: Reference) is built by chaining the tetrahedron edges with linear actuators, which unite at a vertex in a spherical joint. The design and mechanics behind this type of spherical joints have been extensively analyzed [30, 32]. Tetrobot was designed to enable robots to reconfigure into different usages by reusing the same basic primitives. Researchers and engineers have explored variations of this VGT design in different contexts, for example: space applications (TODO: Reference), reconfigurable robotic manipulators (TODO: Reference), and shape morphing trusses (TODO: Reference).
Other researchers introduced design variations in this basic cell, allowing the resulting structure to afford new qualities. For instance, the Spiral Zipper (TODO: Reference) is an extendable edge, based on extending a cylinder that allows for extreme expansion ratios (e.g., 14:1). Similarly, Pneumatic Reel Actuator (TODO: Reference) is based on a mechanism that extrudes and retracts a plastic (tape-like) tubing, to act as an actuator. The mechanism is designed to be lightweight and low-cost ($4 USD) while being limited in its robustness.
TrussFormer takes inspiration from VGTs and builds on the conceptual design of Tetrobot. To this work, TrussFormer contributes a spherical joint design that is automatically generated based on the designed truss geometry.

2.5 SOFTWARE PIPELINE FOR ANIMATRONICS

Algorithmic tools can help users create moving mechanisms. For example, kinematic synthesis of mechanisms [34], or generation of personalized walking toys from a library of predefined template mechanisms [2]. These can be embedded in design support systems, for example, generating moving toys from motion input [42], or synthetized planar kinematic mechanisms from sketch-based motion input [9].
Several software tools directly help the manual design of linkage-based mechanisms, such as LinkageDesigner [48] and Mechanism Perfboard [15]. These tools sometimes include physics simulation of simple mechanisms (e.g., hinges); examples include Crayon Physics [50] and freeCAD [49].
Researchers in the domain of personal fabrication have started to in-

vestigate the effects of dynamic forces in the resulting models, such as balancing rotating objects [22], interactively designing model airplanes [39], and approximating the elastic behavior of 3D printed materials [7].

TrussFormer extends these approaches by using physical simulation interactively in its editor. This combination is necessary to provide an editor that embodies the domain knowledge needed to produce large scale animated truss structures.

3 WALKTHROUGH

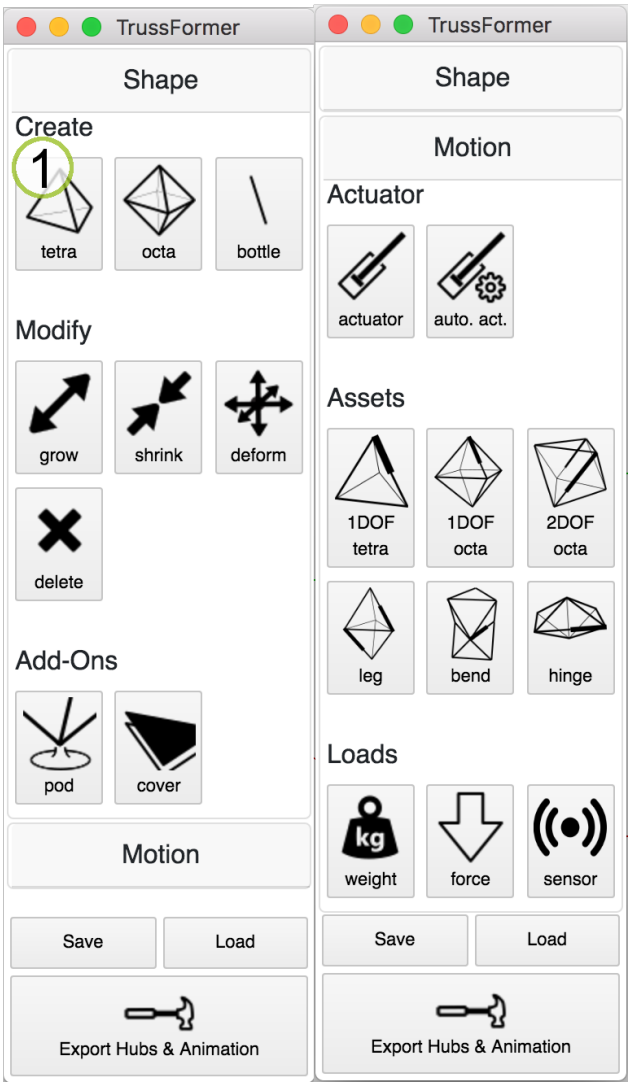

Figure 3.1: TrussFormers' toolbar

This chapter will present the functionalities of the system by showing the process of creating the T-Rex shown in Figure 1.1 (c). This will include all steps from creating the static structure, over introducing animation up to fabricating the final object.

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)). These truss primitives 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*

The primitives can be attached to existing triangle faces, as seen in Figure 3.2 or placed on the ground. If necessary, 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 intact. As our PET bottle system only allows three different sizes of edges (two big bottles, one big and one small bottle, two short bottles), we elongate the connecting parts on the nodes. The grow and shrink tools will change the length of the *elongation* as long as a different bottle configuration is possible. So, 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 and adapt the elongations accordingly. The deform tool (4) does a similar job, but rather than working on edges, this tool can move nodes.
For settling the object on the ground or placing a cover 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))))

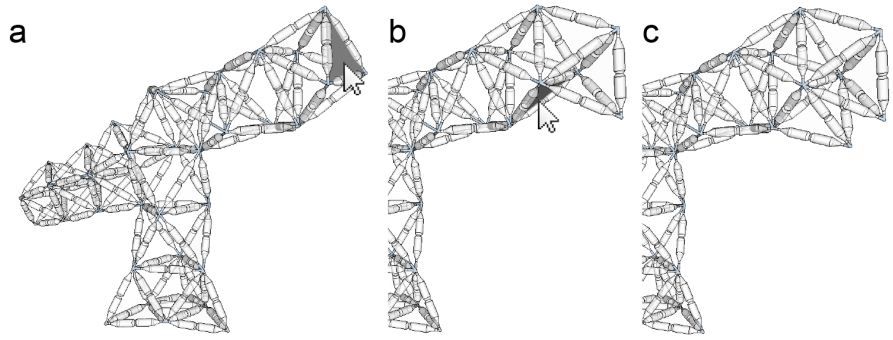

Figure 3.2: Modeling the static shape of the T-Rex. Here, the user creates the jaws of the T-Rex by attaching tetrahedron primitives through the steps (a, b, c).

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 placed.

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 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.

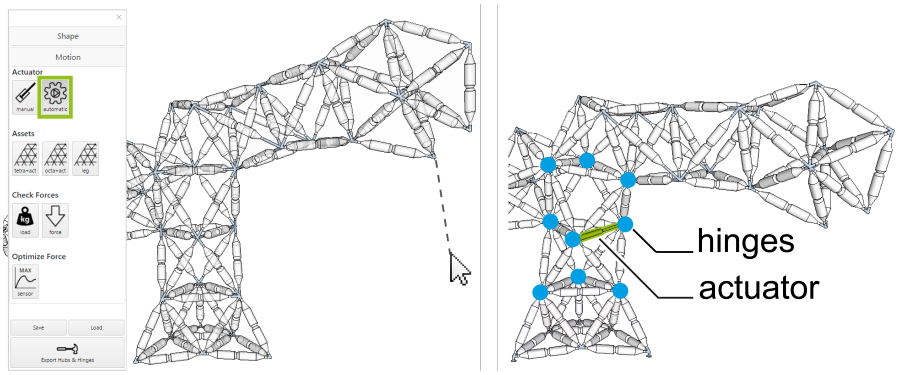

Figure 3.3: (a) The user selects the demonstrate movement tool and pulls the T-Rex head downwards. (b) TrussFormer responds by adding an actuator to the T-Rex body so that it is capable of performing this type of motion. At this point the system also places 9 hinging hubs to enable this motion (marked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

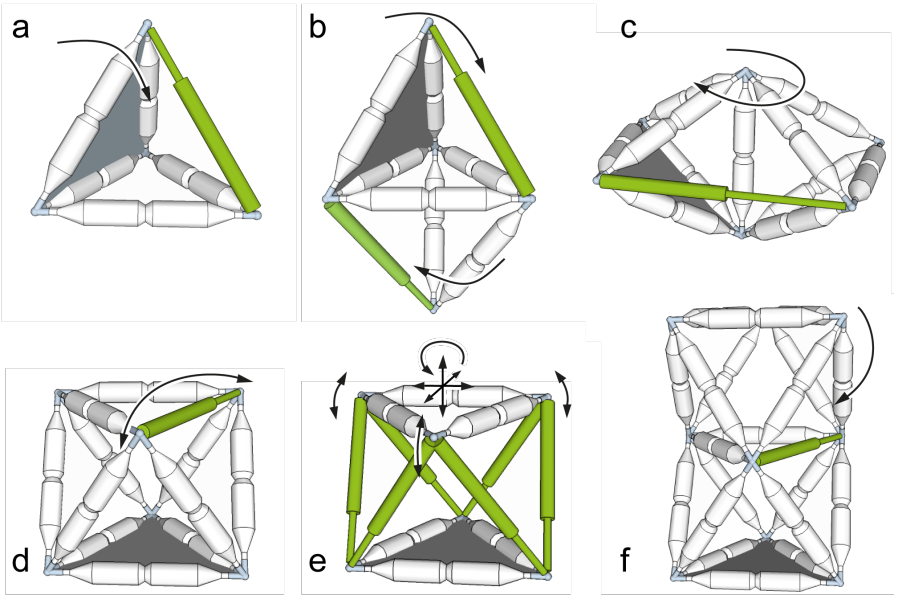

Figure 3.4: A selection of assets: (a) tetrahedron with 1DoF, (b) "robotic leg" asset, (c) hinging tetrahedra, (d) octahedron with 1DoF, (e) Stewart platform (6DoF), and (f) double-octahedron performing "bending" motion.

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.

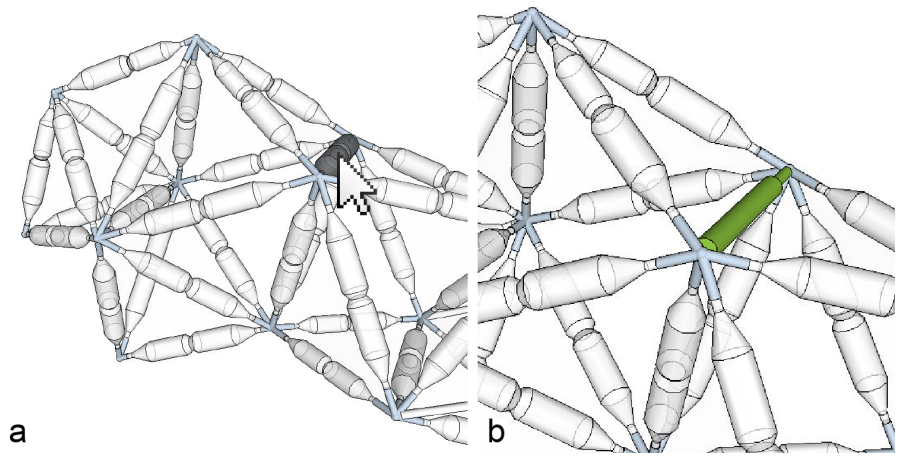

Figure 3.5: The turn edge to actuator tool allows users to turn any edge into an actuator. Here user replaces one edge in the T-Rex's head to make its jaws move.

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, as illustrated in Figure 3.6. 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 for 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 the next section.

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 (11). All actuators have an actuator group

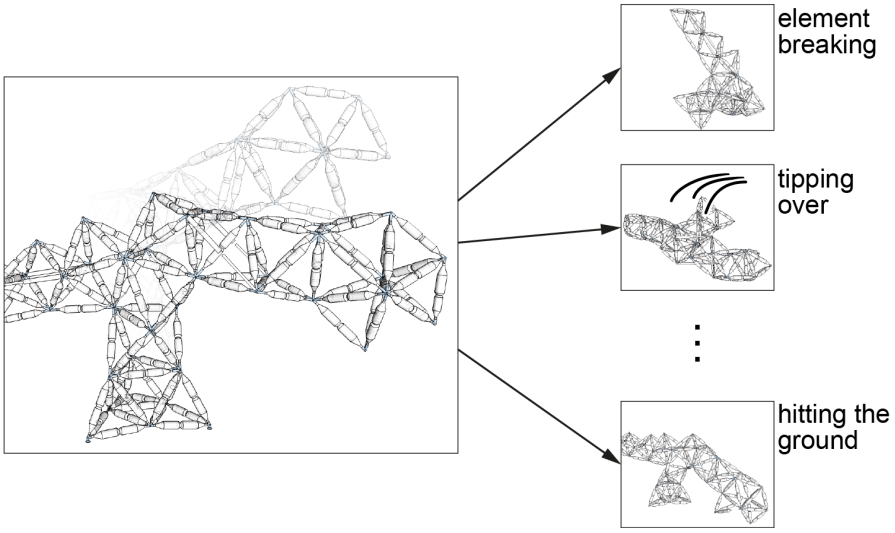

Figure 3.6: In the background, TrussFormer tests each actuator to see if its extension leads to invalid configurations, such as the structure falling over, hitting the ground, or encountering broken structural elements.

assigned to them. Per default, an actuator is placed in 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

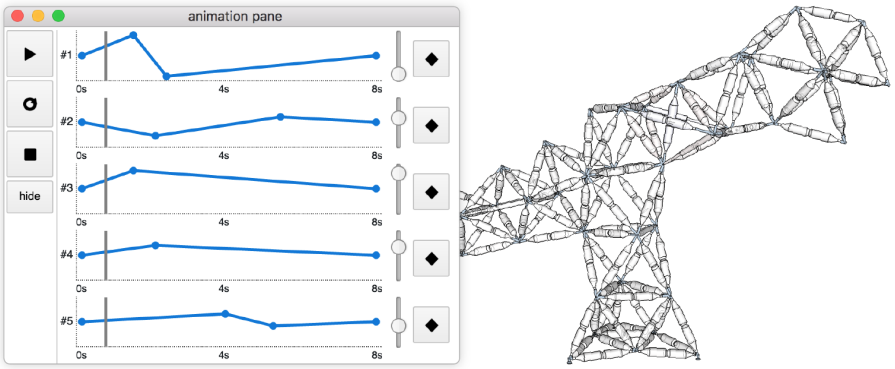

Figure 3.7: Animating the structure. Users sets the desired pose using the sliders in the animation pane and orchestrates the movement by placing key-frames on the timeline.

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 keyframe 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 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 $0N$. 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

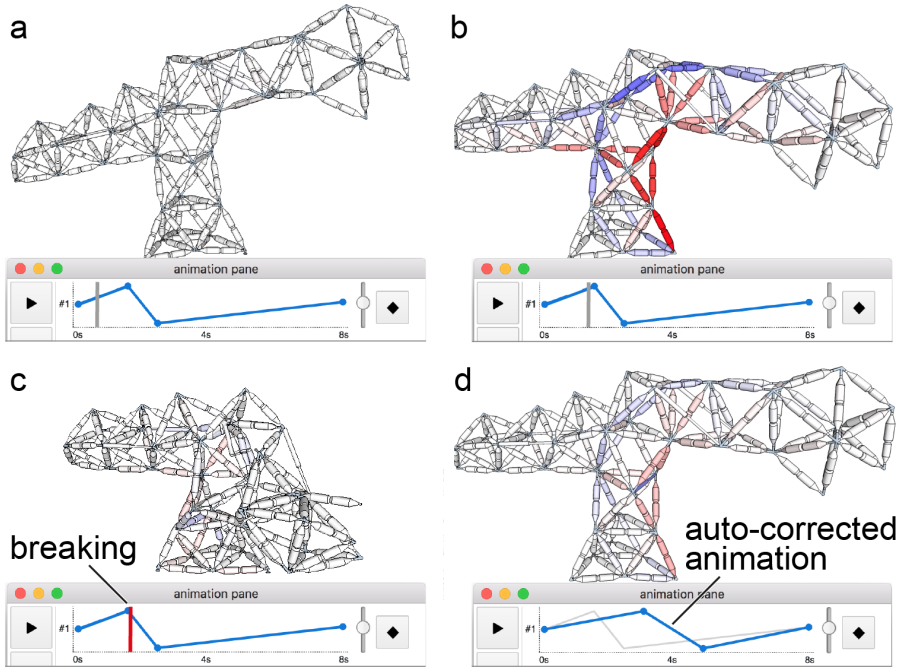

Figure 3.8: TrussFormer verifies that the structure can withstand the inertial forces resulting from the programmed animation. (a-b) The forces in the T-Rex increase with the movement. (c) The structure breaks when the direction of the movement changes rapidly. (d) TrussFormer resolves this by making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

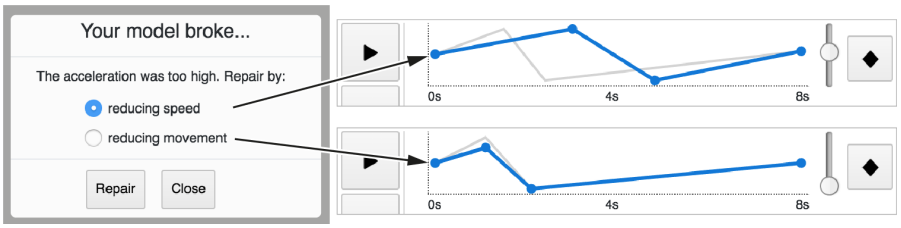

Figure 3.9: If the user-defined animation breaks the model, TrussFormer offers to automatically reduce the speed or the motion range.

can be seen in figure 3.10.

3.4 FABRICATION

After the object was sufficiently tested in the editor, users want to print the connectors and assemble the final object. To do this, TrussFormer will calculate which connections need to move in the printed object and which can be static. 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.

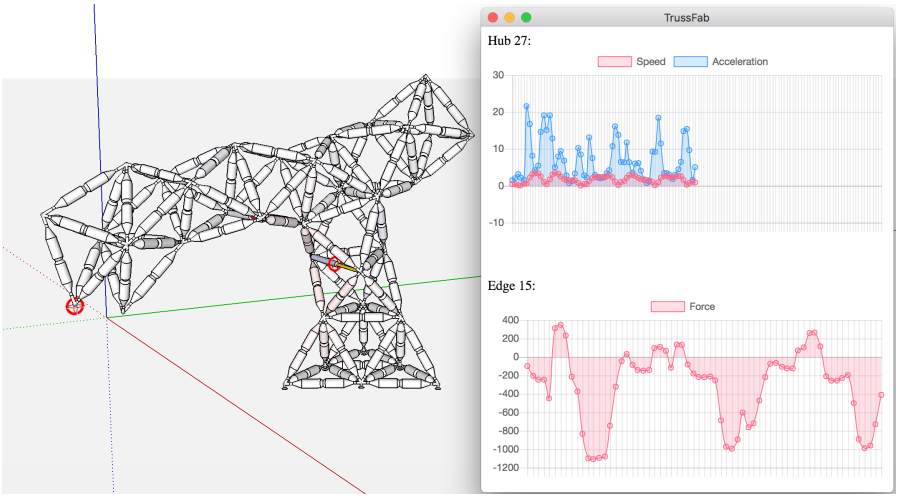

Figure 3.10: A sensor measures the force on the central actuator, and speed and acceleration on a node at the "nose" of the T-Rex

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.

TrussFormers' hinge generation procedure will analyze the structure, ensuring that the final object has the maximum possible movement and creates 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, are created. Actuators that fulfill these requirements 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 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.

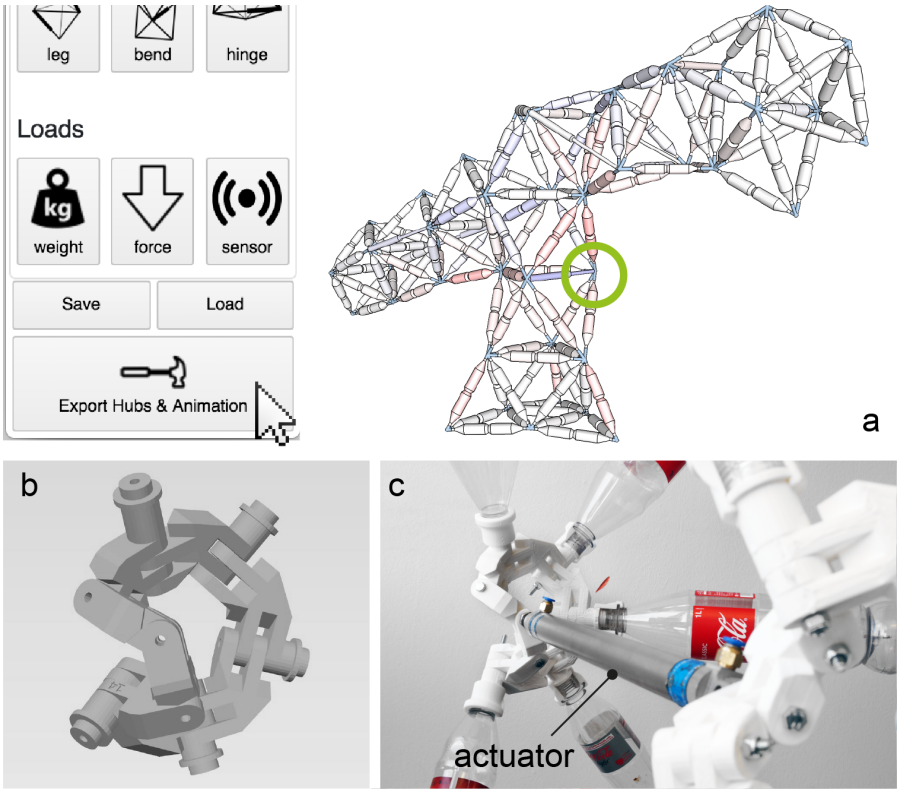

Figure 3.11: (a) To fabricate our T-Rex model, TrussFormer exports: (b) the appropriate 3D printable hinging-hubs for fabricating the structure, (c) and the actuator specifications that inform the users which one to buy. TrussFormer also exports the animation sequence for an Arduino.

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.

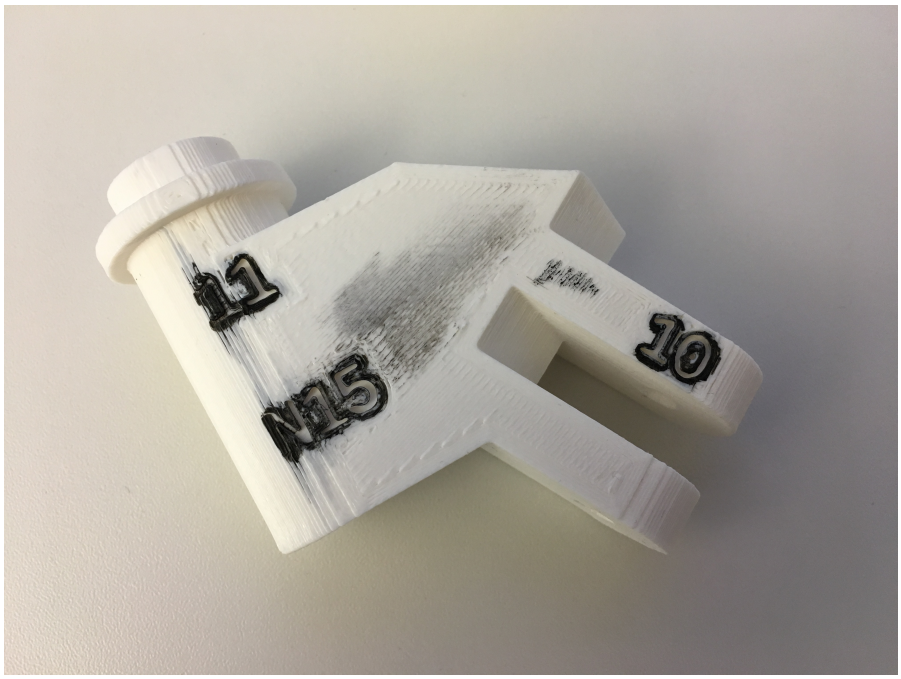

Figure 3.12: A hinge

4

HARDWARE

The system presented in this paper makes use of a variety of mechanical components. This chapter will talk about mechanical challenges and our solutions that allow the construction of large-scale dynamic truss objects using 3D printed parts. It will explain the materials and methods used in creating a real-world object from a TrussFormer-designed model.
Furthermore, this chapter will define the techniques and hardware employed in the structures.

4.1 MECHANICAL COMPONENTS

We can differentiate between three essential building parts for our truss objects. We call the structural components *Links*. In our objects PET bottles are used as links, as they are readily available, cheap and sturdy.
We have two different ways of connecting links. Rigid connections are comprised of *static hubs*. For dynamic connections, i.e. connections that allow movement of the structure, we use *hinging 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, as seen in Figure 4.3.

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. This process is depicted in Figure 4.1. 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 are 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

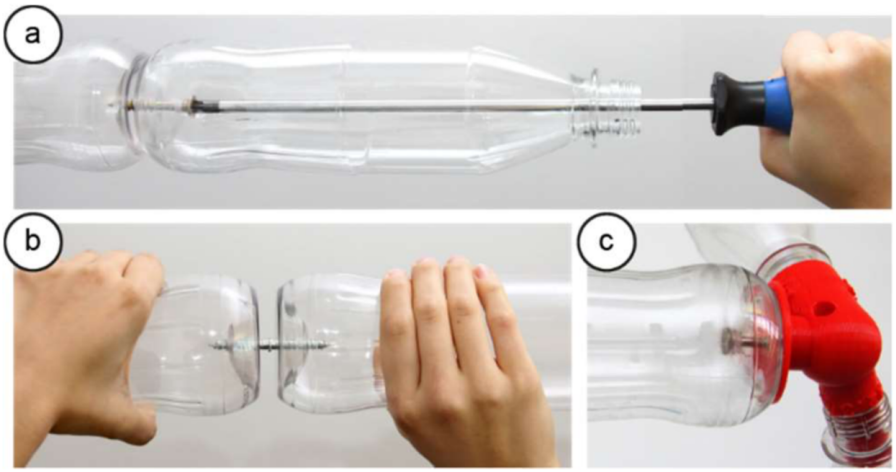

Figure 4.1: (a) Connecting bottles with a wood screw using an extra-long screwdriver and (b) with a double-ended screw. (c) Single-bottle edges require a bottom connector

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

Figure 4.3: ... and secured with a cuff

85 kg.

Hubs are solid one-piece objects that can connect two or more links. They therefore have 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*

In order to introduce movement to truss structures, we came across some challenges. Multiple links 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 links 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 link, 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

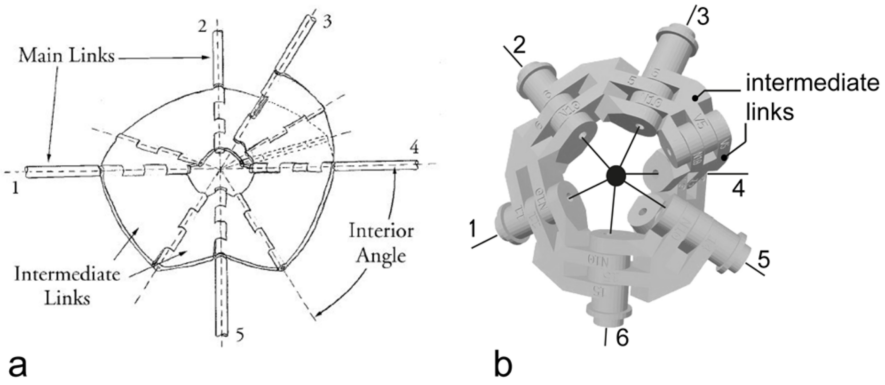

Figure 4.4: (a) spherical joint mechanism connecting 5 links. (b) rendering of TrussFormers' hinge chain design connecting 6 links.

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 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 cornstarch 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

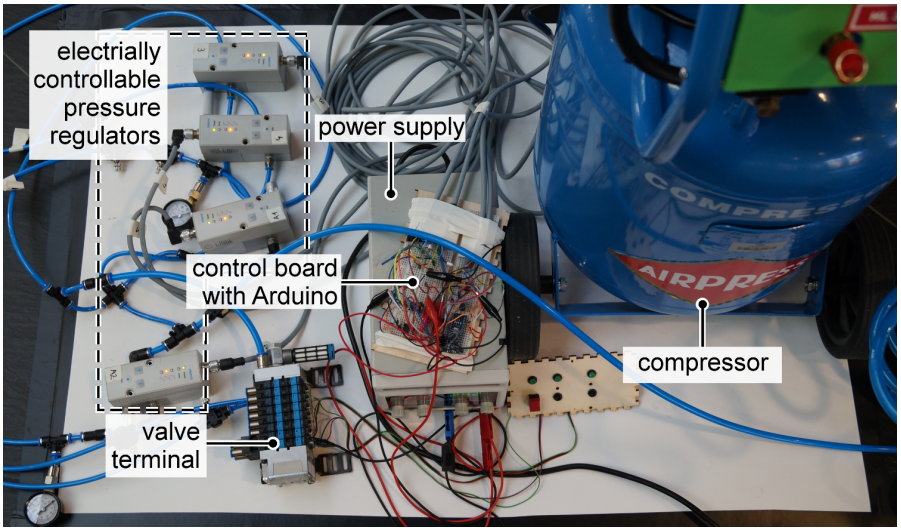

Figure 4.5: Hardware setup for controlling the T-Rex, with Arduino, electric pressure control valves, and compressor.

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 more than 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, as shown in Figure 4.6. 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 on retraction. 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.

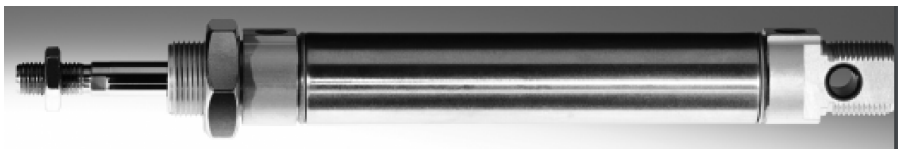

Figure 4.6: A Festo DSNU cylinder with 32 mm cylinder diameter and 30 cm of hub, as used in our T-Rex.

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 an IO link, which can be connected to a control unit. Digital signals are translated to a 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 ratio between these two outputs. This makes positional control of actuators possible.

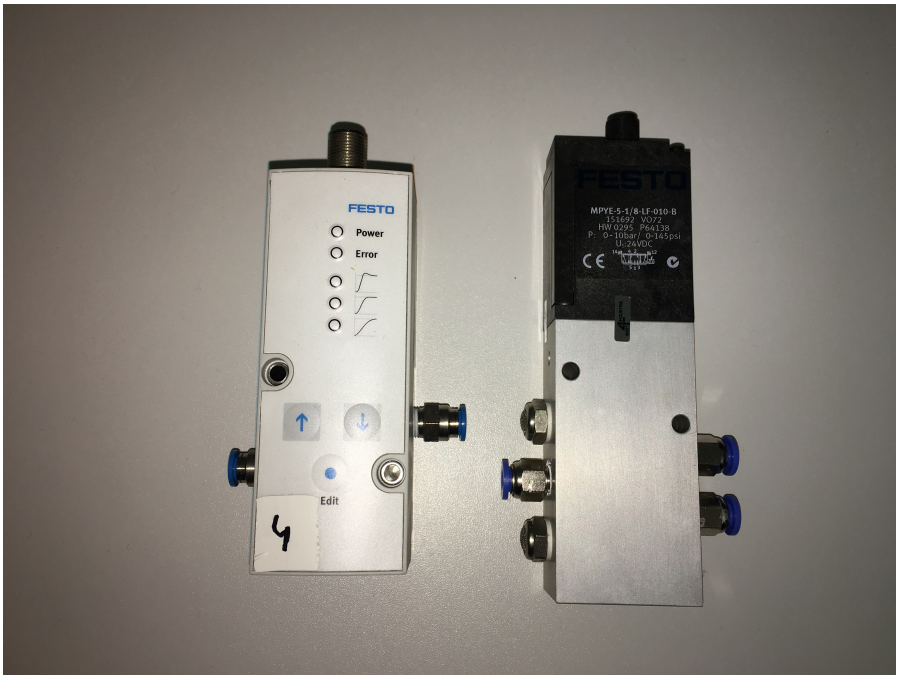

Figure 4.7: TEMPORARY(left) A proportional pressure regulator with one input and one output. (right) A proportional directional control valve, having one input and two outputs. Both can be attached to an IO link on the top.

4.3.2 *Position Sensing*

The MIDI controller is a well working possibility to control actuators. It is, however, 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, counting its rotations. Using the diameter of the reel, we can calculate how much string was released during the extension, which correlates to 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.

4.3.2.1 *PID Controller*

we created a rudimentary motion-platform using closed-loop PID control (s.a. Section 5.3.2.1). The motion platform has three modes. It starts fully extended, applying a little bit of pressure 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 person on the platform applied more force, pushing the piston further down, the third mode activated, which made it 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*.

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 system is composed of an Editor, which can place truss primitives and physics components, and modify created structures, a physics simulation, which can calculate forces and movement interactively, and an export component, which translates the object in the editor into 3D printable files. It is implemented as a plugin for the 3D modeling software *SketchUp*. Its main goal is to support the user in the creation of large-scale dynamic truss objects by providing tools to generate stable structures, add movement to them and visualize occurring forces acting on the object, while the user can move the object interactively. It can also create animations that play automatically, warning users 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 finding 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 the internal program logic is written in Ruby. The physics simulation is based on a C++ physics 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 Editor. It contains the user interface and the construction functionalities. The other components can be seen as extensions to the editor. The *Force Analysis* calculates tension forces on the created structure, 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 *Export Functionality*. It uses the physics features to detect changing angles between edges, indicating the need for a hinging hub. The structure diagram in figure 5.1 shows how the components fit into our system. Details for each component will be explained later in this chapter.

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

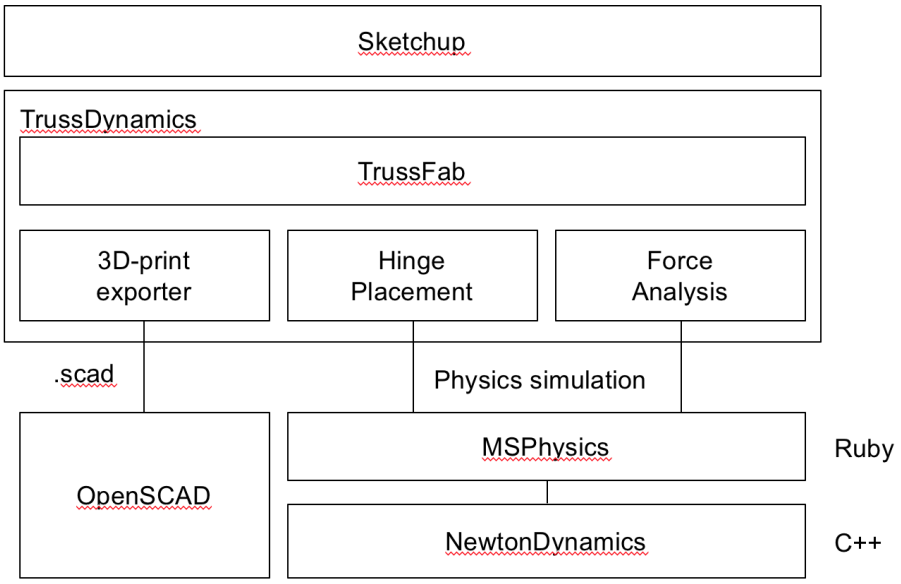

Figure 5.1: TrussFormer Architecture

5.1.1 *Graph*

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.
The *Graph* is implemented as a Singleton, a class which can only be instantiated once in the program, 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.
Each of these objects has access to its underlying logic-bearing component, called *SketchupObject*.
The GraphObject class is responsible for unifying how the appearance of the object in SketchUp is modified. This includes highlighting the object if the mouse hovers over it, resetting the object to its default state, and creating and deleting it. More complex methods are implemented in the respective subclass.

5.1.1.1 *Nodes*

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 its positions in the SketchUp world. The Nodes' adjacent objects constantly check if these positions have changed and update their SketchUp representation accordingly. If the structure is deformed in such a way that a Node will be at

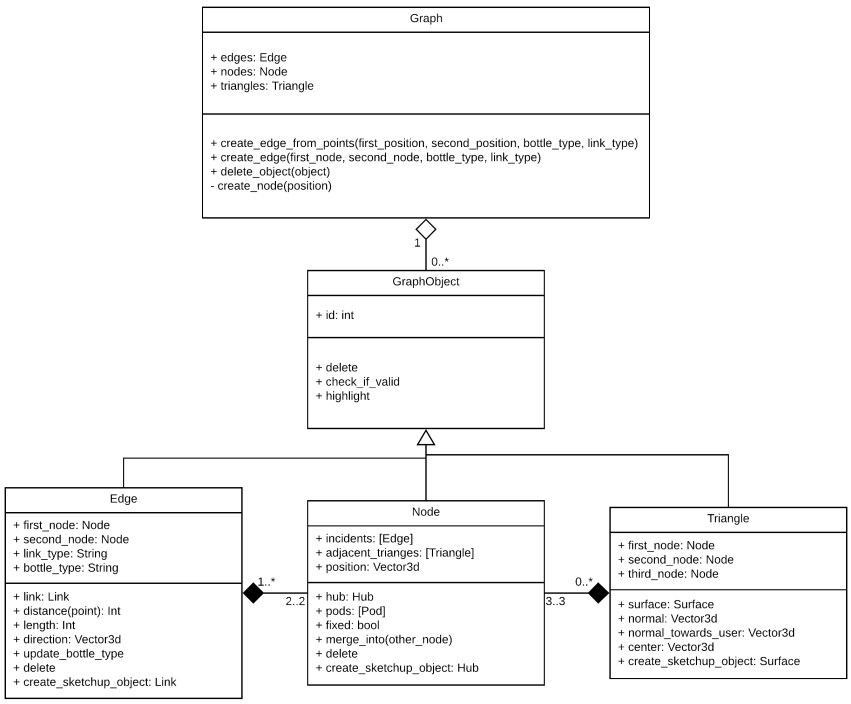

Figure 5.2: Class Diagram showing the high-level Graph Structure of the TrussFormer Designer

the same position as another one, both nodes will merge. To do this, the Node will iterate over all its adjacent Edges and tell each one to exchange itself with the Node it wants to merge into. Excluded are 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). These Edges are removed from the nodes 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 5.2.4.

```
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

A component that is tightly coupled to Nodes is the *Pod*. A Pod acts as a stand for the object and tells freezes the position of the Node it is attached to. This means, that the Node will not move during simulation or by using the relaxation algorithm.
The *Edges* are the structural 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 the creation of the correct model and can change it if the user decides to place a different kind of Edge, e.g. an actuator instead of a bottle link.
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.

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 transforming.

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 5.3), 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

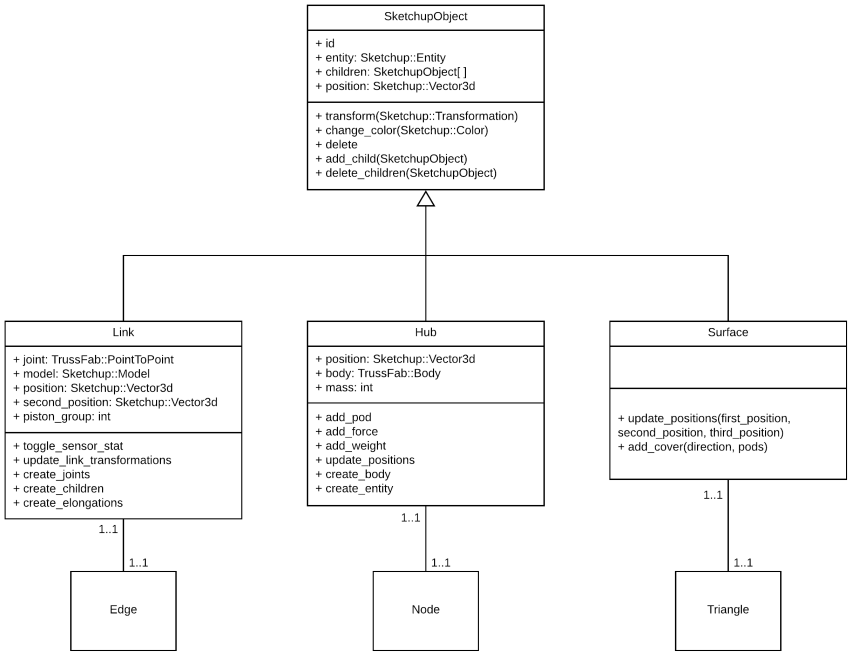

Figure 5.3: Class Diagram showing the UI components of the graph structure

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:

- 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.

5.1.2.3 *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 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 SketchUp's 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

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();')
```

3 Application Programming Interface

```
8   end
9
10  def register_callbacks
11    @dialog.add_action_callback('start_simulation') do |_ctx|
12      if @simulation_tool.simulation.nil? ||
13         @simulation_tool.simulation.stopped?
14        start_simulation_setup_scripts
15      end
16    end
17
18    @dialog.add_action_callback('stop_simulation') do |_ctx|
19      unless @simulation_tool.simulation.nil? ||
20             stop_simulation
21        Sketchup.active_model.select_tool(nil)
22      end
23    end
24
25    ...
26  end
27 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, while focusing 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*

Users interact with the SketchUp environment using tools. These tools are accessed by buttons in TrussFormers sidebar. The majority of them was explained in Chapter 3. This section will explain complex tools in more detail.

5.2.2.1 *Mouse Input Helper*

Most tools require mouse interactions. We wrote a helper class that handles these mouse inputs and provides functions that let us interact with existing TrussFormer geometry. SketchUp itself provides mouse handling functionalities, such as returning the point in the SketchUp world, the mouse is pointing at, essentially converting a 2D screen position into a 3D world position.
Often, users will want to use tools on existing TrussFormer objects. To do that, our MouseInput can snap to different parts of the geometry:

Triangles, Nodes, Edges, or Pods. Based on the SketchUp-provided position, the closest of these snappable objects, that lies within a certain radius, will be selected and its position returned to the tool that uses the MouseInput. If the mouse can not snap to an object, a point on the ground that intersects with the view plane of the camera will be selected.

5.2.2.2 *Stability Check*

This probably has to be a subsection of simulation

The stability check can be used to test the created structure statically in different poses. It is implemented as part of the simulation and intended to be either run during the creation of the structure (especially after placing a new actuator) or by triggering it manually. The stability check places all actuators in the structure in a combination of different poses. Each actuator will be placed in full extension, its middle position and full retraction in a combination with the other actuators in these positions. So, if the structure has five actuators, like our T-Rex, this would result in $3^5 = 243$ different combinations.
Even though the simulation is normally used for dynamic force analysis and does not, on its own, provide the possibility to disable inertial forces, which are unwanted in this case, there is a possibility to work around this issue. By making all joints as stiff as possible, we prevent the structure from causing unwanted movement, because it completely resists inertia.
Actuators do not have the ability to "jump" between two positions, which means the simulation has to extend or retract the pistons over time. As this step should be run in the background and provide timely results, this movement has to occur as fast as possible. If we were to move the actuators at maximum speed, the occurring force would destroy the structure. Our way around this is to make the structure indestructible during stability check. This tool should provide the maximum forces to the user. If the structure would break, the maximal possible force would be the breaking force and even higher forces could not be detected.
For each combination of actuator positions, the actuator parameters are set to the pose checking configuration (i.e. infinite breaking force, very high speed and maximum stiffness). The internal position controller of each actuator is set to the position according to the configuration. After all actuators are set up, the simulation runs for 1 second in the background, without visualizing the forces, in order to move the actuators and settle the structure. Then, the simulation is run again for about 10 world updates, while recording the maximum tensions in a global map. The highest force will be returned and linked to this specific combination. As a last step, a list of piston combinations linked to the resulting forces is returned, so that users can observe which pose causes the highest loads on the structure.

```
1  def check_pose(combination)
```

```
    highest_force = 0

    Graph.instance.edges.each_value do |edge|
      edge.link.joint.stiffness = 0.999
    end
    # move them to the proper position
    combination.each do |id, pos|
      actuator = @pistons[id]
      actuator.joint.breaking_force = 0
      actuator.joint.rate = MAX_SPEED
      actuator.joint.controller = case pos
                                  when -1
                                    actuator.min
                                  when 0
                                    0
                                  when 1
                                    actuator.max
                                  end
    end

    update_world_headless_by(1) # settle down

    # record
    @max_link_tensions.clear
    update_world_headless_by(0.2, true)

    Graph.instance.edges.each_value do |edge|
      force = @max_link_tensions[edge.id]
      highest_force = force if force.abs > highest_force.abs
    end
    highest_force
  end
```

Listing 5.3: The pose check algorithm.

5.2.2.3 *Automatic Actuator Placement*

The *Automatic Actuator Placement* tool is the most supporting method in creating motion in the truss structure. Users can demonstrate the movement they want to achieve by dragging a node to a certain position. The tool will automatically turn the edge into an actuator that can move the selected node closest to the desired position.
The process behind it iteratively turns an edge into an actuator, moves it to its maximum and minimum position using the physics simulation and observes the selected node for position changes. This is done for each edge individually. Each actuator is tested for two seconds in the background. Similar to the pose checking tool, this simulation is "headless", meaning forces are not visualized. Fixed edges, i.e. edges that are connected to pods, will not be tested. After each world update, the position of the observed node is compared to the desired position and if the distance is less than in a previous world update,

this distance is stored. The distance is calculated using the following formula:

$$d(P1, P2) = \sqrt{(P1_x - P2_x)^2 + (P1_y - P2_y)^2 + (P1_z - P2_z)^2} \quad (5.1)$$

The shortest distance will be returned and compared to the result of other edges. If the observed actuator moved the node closest to the desired position, the edge and distance is promoted to the new candidate. The tested edge will be changed back to its original type. After all edges are tested, the latest candidate will be turned into an actuator and the view is updated for the user.
This method works as a naïve approximation, meaning it does not guarantee that the desired position will be reached exactly.

```
1  def test_pistons
2    closest_distance = Float::INFINITY
3    best_piston = nil
4    Graph.instance.edges.each_value do |edge|
5      next if edge.fixed?
6      previous_link_type = edge.link_type
7      create_actuator(edge)
8      simulation = Simulation.new
9      simulation.setup
10     simulation.schedule_piston_for_testing(edge)
11     simulation.start
12     distance = simulation.test_pistons_for(2, @node,
     @desired_position)
13     if distance < closest_distance
14       closest_distance = distance
15       best_piston = edge
16     end
17     simulation.reset
18     reset_actuator_type(edge, previous_link_type)
19   end
20   create_actuator(best_piston)
21 end
```

Listing 5.4: The test_piston method changes each edge into an actuator tests its movement and determines how close it brings a selected node to a desired position.

5.2.2.4 *Correct Forces*

5.2.3 *Load/Save*

In order to save created objects and work on them later, TrussFormer provides a save and load function. The save function stores the current geometry in JSON files. On activation of the tool, a message box appears, asking the user to select a surface, which will be used to attach the object to existing geometry or to place it on the floor on import.
The JSON file is a representation of the graph. It contains attributes

about nodes and edges necessary to recreate the object later.
Important attributes for nodes are:

- the ID
- the position (x, y, z)
- whether or not a pod is attached

Edges require the following information:

- the ID
- the first node's ID
- the second node's ID
- the link type (i.e. bottle link or actuator)
- the bottle type, if applicable (small or large bottle)
- the piston group, if applicable
- the length of the first elongation
- the length of the second elongation

The animation is also stored in the JSON file. If actuators are present, the keyframe positions are stored together with the according piston group.
The load function takes this generated JSON and recreates the graph according to the provided information. The animation pane will be populated with the exported animation.

5.2.4 *Relaxation Algorithm*

The relaxation algorithm is used to propagate the change of an edge over the whole structure. This way, we can adapt edges organically and precisely, minimizing the change in adjacent edges.
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 5.5: The relaxation algorithm

5.2.5 *Animation*

The animation pane is inspired by existing animation software, such as Unity[4]. It is based on so-called keyframes. A keyframe always acts on a certain piston group. It indicates the extension of all actuators in this group at a given time. Keyframes are placed on a timeline, which stretches over a set length of time. By placing keyframes on this timeline, a curve is created, which demonstrates the movement of the actuator group over this time.
Keyframes are positioned by using a slider and placing a vertical line at the desired timeline position. The slider sets the extension of the actuator, which is interactively indicated in the editor. Pressing the *Add Keyframe* button, places the keyframe with the set extension in the timeline.
The created animation can be played in two different modes: single

4 https://unity3d.com/

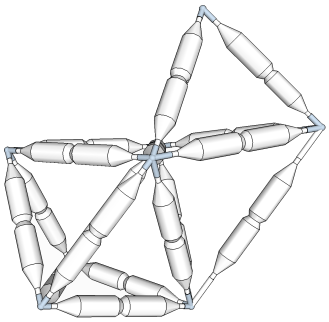

Figure 5.4: The relaxation algorithm applied with only one iteration. The extension of the lower right edge resulted in growing incident edges as well.

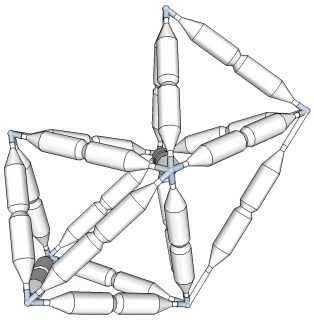

Figure 5.5: The relaxation algorithm applied with up to 20'000 iterations. Growing the lower right edge caused other nodes to translate, but the lengths of incident edges stayed the same.

and looping run. The animation will be played at a constant speed, which is coupled to the system clock. That means that one second of the animation will correlate to one second in real-time. An internal counter keeps track of the current position in the timeline at any given time and triggers a new actuator movement after passing a keyframe. The speed of an actuator is determined by the distance from the current keyframe to the next and the desired change in amplitude. The y position of the keyframe can be seen as a percentage of complete extension. If the keyframe is on the top of the timeline, it has a value of 1 and each actuator in the group will be fully expanded, disregarding its length. Similarly, if the keyframe is on the bottom, its value is 0 and all actuators are fully contracted. The x value of the keyframe is also mapped to [0, 1], with 0 being the beginning of the timeline and 1 the end.

The speed of an actuator is calculated by multiplying the delta of the expansion component of the last and the next keyframe with the range of motion of an actuator and dividing it by the time component of these keyframes, multiplied by the timeline length, modulo the timeline length to account for looping of the animation.

As actuators in a group can have different lengths, they will only receive the Δ_y value, combining it with their possible extension length

and the time component. The simulation will then trigger the movement of the given actuators.

$$\Delta_y = next\_keyframe.y - last\_keyframe.y \tag{5.2}$$

$$\Delta_x = next\_keyframe.x - last\_keyframe.x \tag{5.3}$$

$$v = \frac{\Delta_y * (actuator.max\_length - actuator.min\_length)}{(\Delta_x * timeline\_length) \bmod timeline\_length} \tag{5.4}$$

5.3 PHYSICS SIMULATION

TrussFormer's 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[5]. 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 5.6), 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.

```
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?
```

5 https://github.com/AntonSynytsia/

```
10        send_sensor_data_to_dialog
11      end
12
13      @frame += 1
14      update_status_text
15
16      @running
17  end
```

Listing 5.6: 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 its positions and transformations based on their internal physics objects.
As a last step, sensor data is sent to JavaScript in order to draw a chart depicting physical data.

5.3.1 *Actuators*

Actuators are implemented using the *TrussFab::PointToPointActuator* joint. These joints provide position control. The *controller* of the joint

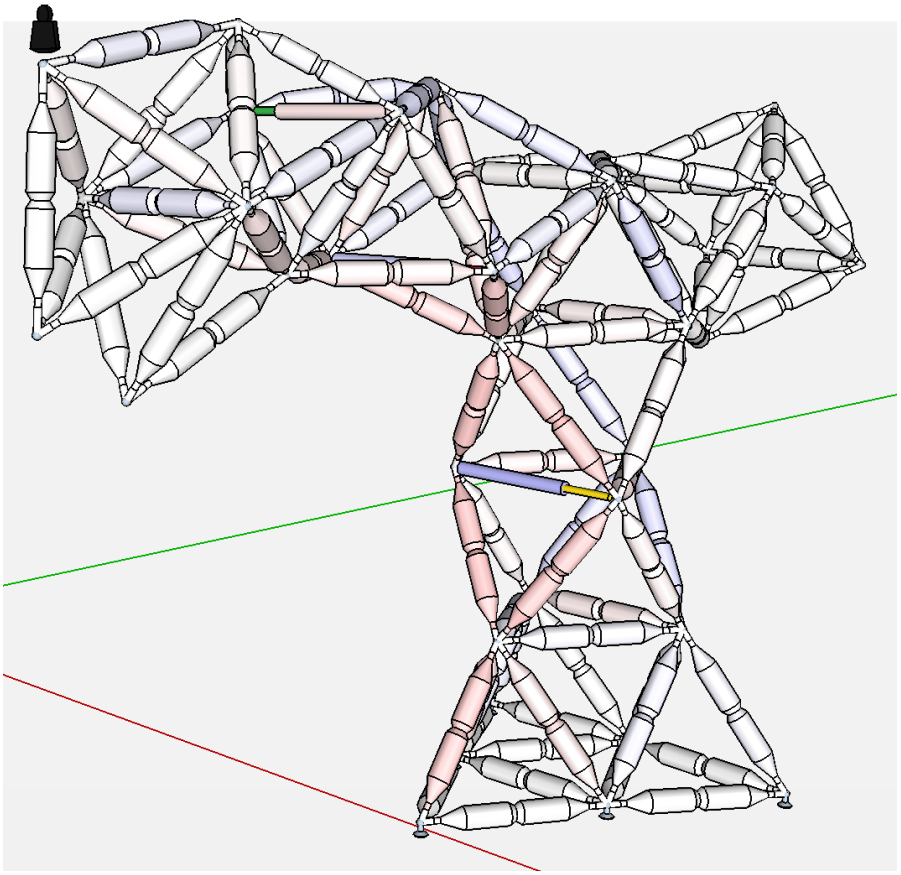

Figure 5.6: TEMPORARY Visualization of forces acting on edges. Blue - tension force, red - compression force, white - little or no force

can be set to a length in m relative to a starting position (per default the middle position). This tells the simulation, that this joint wants to change its length to the provided value. Dependent on the speed of the joint, it will approach this position over multiple world iterations. A setting that can influence the motion of actuator joints is the damping factor, which can be set to a percentage of extension. The damping factor will linearly slow down the actuator starting from the set value, approaching the end with a smoother, less radical motion.

5.3.2 *Force Control*

Instead of position controlled joints, TrussFormer also provides joints with force control. These joints do not have a controller that can be set to a length, but a force function. The force can be set dynamically during the simulation, making force controlled joints a very flexible way to create motion.
Possible uses for these joints are springs. A spring exerts more force, the further it is compressed. Transferring this concept to our cylindri-

cal links, such a spring would produce more force, the smaller the distance between two nodes is, according to the formula $F = -k * x$, with k being a spring constant and x being the difference between the uncompressed length of the spring (e.g. the actuator in its full extension) and its current length.

5.3.2.1 *PID*

Another use case for force-controlled joints is PID controlled actuation. A proportional-integral-derivative (PID) controller is a mechanism for closed-loop control. It uses sensor values from the system, in our case the length of the actuator, and gives feedback to the control unit.
A PID controller will continuously calculate an error value as the difference between a desired value, e.g. the actuator position to be reached, and a current "process variable". The process variable is calculated as the weighted sum of the proportional, integral and derivative control terms. The controller attempts to minimize the error value by adjusting these terms.
The proportional term depicts a value proportional to the current error value. If the error is large, e.g. if the actuator is fully contracted, but should be fully expanded, the proportional value is large as well, telling the control unit to produce high pressure. The proportional term alone will, however, never reach the desired point, because the smaller the error value becomes, the smaller the produced pressure will be. Also it can not account for inertia of an object, as it will only produce a force in the other direction (i.e. a breaking force), if the sensor values overshot the desired position. This will create an oscillating motion around the desired position.
The integral part takes past error values into account, which are integrated over time. It cumulates residual error values, left behind by proportional control. If the error gets smaller, the integral term will stop growing. This will compensate the proportional term approaching zero.
The P and I terms will already achieve a reasonable result. The derivative term will optimize this result further. It can be seen as an estimate of the future, based on the error value's rate of change. The effect of the derivative term is that an extra control factor is included if the error rate increases and a damping factor added if the rate decreases.

5.3.3 *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 rate of 0.5%, on the front bottom edge of the T-Rex (Figure 5.12 a-b).

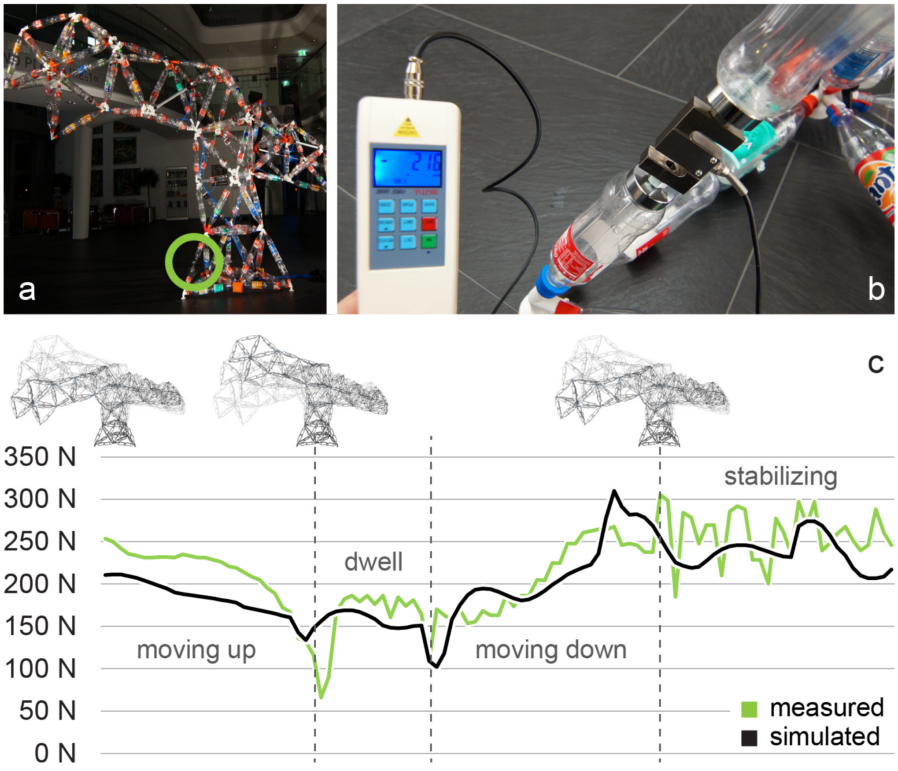

Figure 5.7: (a) We measured the forces on the bottom front edge of the T-Rex (b) using a digital force gauge. (c) The measured forces agree with the simulated forces.

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 5.12c 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. As a last step, the

abstract representation in TrussFormer has to be converted into 3D printable files.

5.4.1 *Hinge Placement Routine*

We use an empirical approach to determine the placement of hinges. As a first step, all adjacent edges will be connected using an intermediate link connection, as illustrated using blue lines in Figure 5.8. This provides 2 DoF between all edges, behaving as if they were connected by ball joints. This result will always work. However, in variable geometry trusses, most of the edges are confined in triangles and larger rigid substructures, therefore not all movements are possible. Placing unnecessary hinges only adds complexity for assembly and reduces mechanical stability.

Using the physics engine, we now move all actuators, one after an-

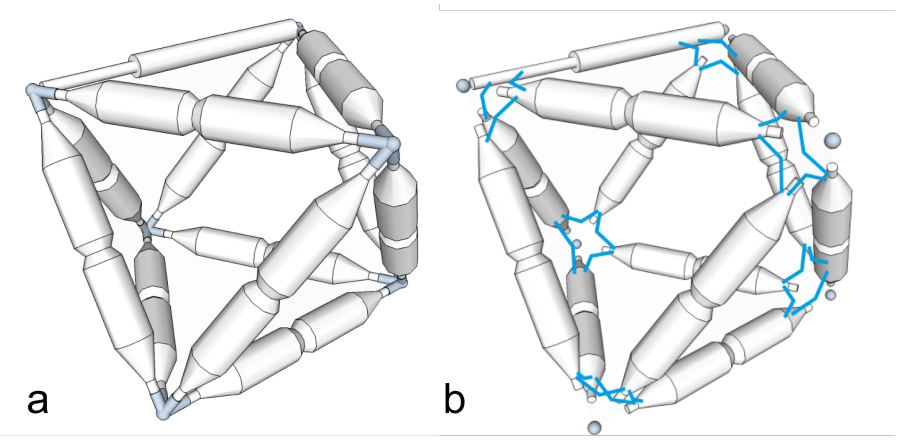

Figure 5.8: (a) Octahedron with an actuator, still with rigid hubs. (b) After the first step, intermediate links are assigned inside all triangles, creating spherical joints.

other, and measure the angular difference between adjacent triangles. If the difference stays below a certain threshold, the common node of these two triangles is considered part of a *rigid substructure*. Figure 5.9a shows the rigid substructures identified in the octahedron, visualized in distinct colors. The triangles containing the actuator are not considered rigid, since their internal angles are changing. Figure 5.9b shows the result of this step on the example of the T-Rex. Here, instead of the triangles, the edges of rigid substructures are colored for clarity.

Now that TrussFormer knows which parts of the structure are rigid, it can remove the unnecessary hinges between the edges which belong to the same rigid substructure and don't move in regards to each other. In Figure 5.10 we show this step on our octahedron example. Between the edges forming rigid substructures, the intermediate

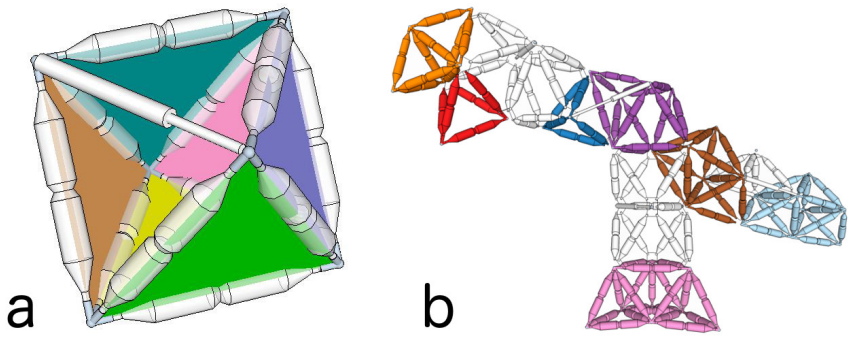

Figure 5.9: TrussFormer identified the rigid substructures (a) in the octahedron from Figure 20, and (b) in the T-Rex.

link connections (former blue lines) are reduced to rigid connections (black lines). Rigid substructures can still rotate in respect to each other. At this stage, the final hinge chain is already found for the octahedron example and the resulting 3D print is shown in Figure 5.10b. At this point, TrussFormer has already assigned an optimized

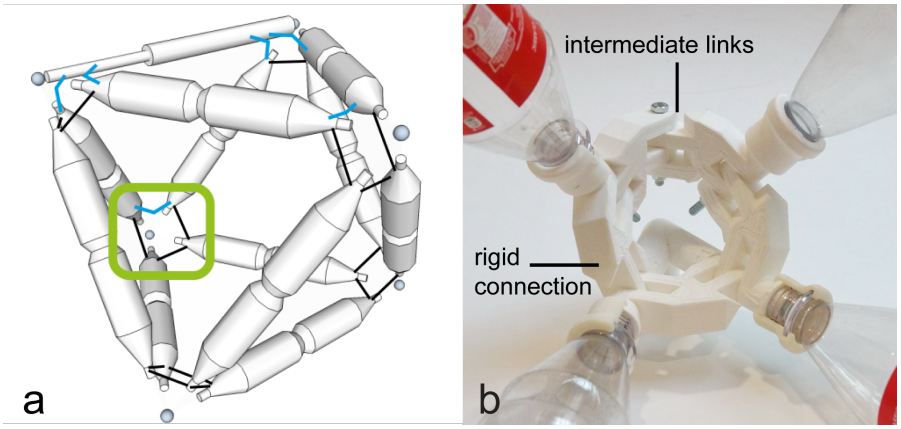

Figure 5.10: (a) The intermediate hinge connections are reduced to rigid connections where rigid substructures are identified (black lines). (b) The fabricated hinging hub of the marked node of the octahedron.

valid hinge configuration, however, not all the connections might be physically possible to assemble. TrussFormer's hinge design has the limitation that it only supports one-on-one hinge connections. Three way connections are not possible, i.e. three parts cannot physically hinge around the same axis. However, after Step 3, there might be hubs violating this condition, e.g. where three hinges are meeting at the same axis. We demonstrate this case in Figure 5.11 on the example of the double-octahedra structure with one actuator, similar to the one found in the body of the T-rex. In Figure 5.11b we highlight the hub where a three-way hinge connection is present after performing the hinge reduction. Fortunately, these connections are redundant in

TrussFormer's kinetic structures, and they can be resolved by eliminating some of the hinges, while still maintaining the hub's structural integrity, i.e., all the edges remain interconnected via a continuous hinge chain. TrussFormer resolves violating connections using a back-

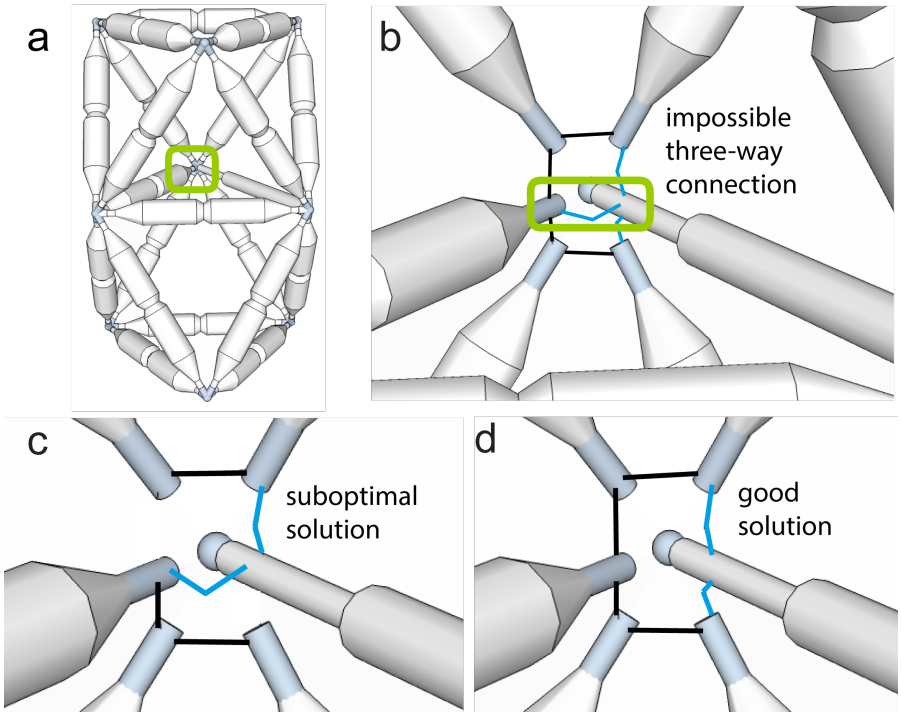

Figure 5.11: (a) Double-octahedral structure, where (b) violating three-way hinge connections appear. (c-d) TrussFormer finds the valid configurations by heuristic elimination and (d) chooses the structurally more stable closed chain.

tracking algorithm that removes connections heuristically. After each removed connection, it checks the validity of the resulting hinge configuration for two constraints: (1) all edges around the node are still interconnected directly or indirectly with each other, and (2) no more than two hinges are connected at each axis. If these constraints are satisfied, a valid hinge configuration was found. The algorithm continues until it finds all valid configurations. If available, TrussFormer will select the configuration with closed hinge chain (Figure 5.11d) over open chain (Figure 5.11c), for stability reasons.

5.4.2 *Generating the 3D Models*

In order to get printable files, our abstract description of the object has to be converted into a physical representation. To achieve this, we use a modeling language called *OpenSCAD*. After finding all hinge chains and static hubs, 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 **??**. We used this editor to create template functions used for creating the final hinge and hub models.
Our hinging and static hubs are fully parameterized. This way we

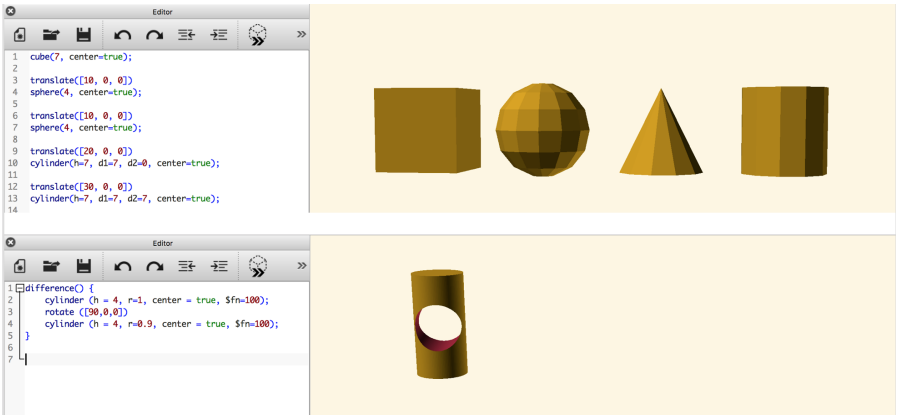

Figure 5.12: The OpenSCAD editor containing some example code.

make sure that all parts will fit together exactly in the final object. Hub configurations, such as the angle, the elongation, the ID, ..., are inserted into a template file and define the shape of the printed hub.

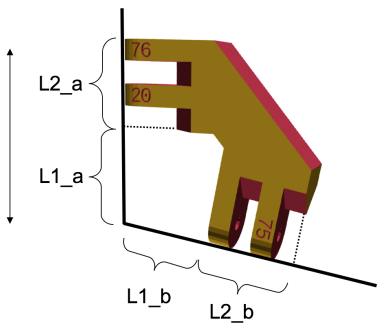

Figure 5.13: TEMPORARY L1 and L2 lenghts of a hinge part

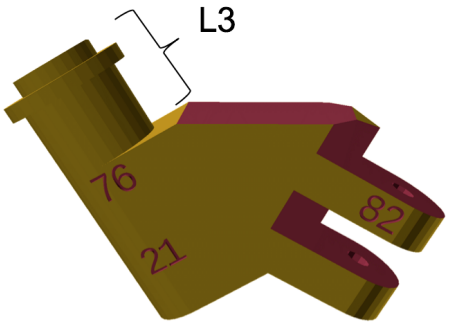

Figure 5.14: L3 length of a connector

6 CONCLUSION

We created a system that enables users to design and fabricate large-scale dynamic truss structures. Our system consists of an editor, which supports creating and simulating stable truss objects with the help of a physics engine. Users can simulate motion by placing linear actuators in the structure and observe occurring forces by interactively moving the structure. Motion patterns can be created by using the keyframe animation feature. This way, the complicated motion distribution in the trusses can be thoroughly tested before building the object.
Our export feature enables users to generate STL files, based on the designed object, which can be 3D printed and assembled. The animation is stored in an Arduino-readable format, so the designed object can be completely recreated in the real world.
For assembly, users connect PET bottles to the 3D printed hubs. An ID system clarifies which of these hubs belong together.

6.1 CLOSED-LOOP POSITIONAL CONTROL

We already made first advances towards integrating PID control into our system. In further iterations of TrussFormer, we want to increase the focus on closed-loop control mechanisms and enable users to design accurate positional actuator control. Our current approach already delivers functional PID control, however the design process is still very manual. Especially finding the correct gain values for each control term can be very challenging to users with little knowledge of PID control. We imagine an automated search process, which heuristically determines reasonable values.
Each control term produces a specific pattern in the motion of the controlled object. If the gain value for the proportional part is too large, the motion will oscillate around the desired position, if it is too small, the approach will be slow. An overly high integral gain value will result in the accumulated error rising very quickly, causing the controller to exert a lot of force if the desired position is far away from the actual position. This can result in an even bigger overshoot and stronger, but fewer oscillations. On the other hand, if the integral is too small, it will provide little extra control to account for residual error in the proportional part. Similarly, the derivative component will fail to control the output, while a very high gain will make it very susceptible to noisy input.

Based on these patterns, a procedure can be implemented that automatically fine-tunes these values.

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