

TIM OESTERREICH

BUILDING LARGE DYNAMIC TRUSS STRUCTURES

BUILDING LARGE DYNAMIC TRUSS STRUCTURES

TIM OESTERREICH



Using TrussFormer

April 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 Gregorio¹, 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 L^AT_EX-community for support, ideas and some great software.

Regarding L_YX: The L_YX port was initially 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 T_EX e L^AT_EX)

CONTENTS

1	INTRODUCTION	1
1.1	TrussFab	1
1.2	TrussFormer	1
1.3	TrussControl	2
2	RELATED WORK	3
2.1	Large-scale Personal Fabrication	3
2.2	Construction Kits	3
2.3	Prototyping with Ready-Made Objects	3
2.4	Building with Variable Geometry Trusses	3
2.5	Software Pipeline for Animatronics	3
2.6	SketchUp	3
3	WALKTHROUGH	5
3.1	Designing Static Structures	5
3.2	Adding Movement to the Structures	6
3.2.1	Force Analysis	6
3.3	Controlling the Structure	6
3.3.1	PID Control	7
3.4	Building the Final Object	7
3.4.1	OpenSCAD	7
3.4.2	Printing the Parts	8
3.4.3	Assembling the Structure	8
4	HARDWARE	9
4.1	Building Parts	9
4.1.1	Links	9
4.1.2	Hubs	10
4.1.3	Hinge Chains	10
4.1.4	Cuffs	10
4.2	Controls	10
4.2.1	Electric vs. Pneumatic Actuators	10
4.2.2	Open Loop vs. Closed Loop	10
5	IMPLEMENTATION	11
5.1	Architecture	11
5.1.1	Designer	12
5.1.2	SketchupObjects	14
5.2	Physics Simulation	15
5.3	Minimization Logic	17
5.4	Export	17
5.5	TrussFab Designer	17
5.5.1	User Interface	17
5.5.2	Structure Creation	19
5.5.3	Modifying the Structure	20
5.5.4	Relaxation Algorithm	20

5.5.5	OpenSCAD Export	20
5.6	TrussFormer Physics Engine	20
5.6.1	Automatic Actuator Placement (if it works soon-ish)	20
5.7	Force Control	20
5.7.1	PID	20
6	CONCLUSION	21
	BIBLIOGRAPHY	23

LIST OF FIGURES

Figure 3.1	The T-Rex built in TrussFab	5
Figure 3.2	An exported leg asset...	6
Figure 3.3	... which can be used to quickly build the spider	6
Figure 3.4	An exported leg asset...	7
Figure 3.5	... which can be used to quickly build the spider	7
Figure 5.1	TrussFab Architecture	11
Figure 5.2	Class Diagram showing the high-level Graph Structure of the TrussFab Designer	12
Figure 5.3	Class Diagram showing the UI components of the graph structure	14
Figure 5.4	Visualization of forces acting on edges. Blue - tension force, red - compression force, white - little or no force	18

LIST OF TABLES

LISTINGS

Listing 5.1	Merging of two Nodes	13
Listing 5.2	Simulation nextFrame method	16
Listing 5.3	excerpt from UI callbacks	19

ACRONYMS

INTRODUCTION

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

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

1.1 TRUSSFAB

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

1.2 TRUSSFORMER

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

2 INTRODUCTION

1.3 TRUSSCONTROL

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

RELATED WORK

2.1 LARGE-SCALE PERSONAL FABRICATION

2.2 CONSTRUCTION KITS

2.3 PROTOTYPING WITH READY-MADE OBJECTS

2.4 BUILDING WITH VARIABLE GEOMETRY TRUSSES

- Steward Platform
- Walking Octa

2.5 SOFTWARE PIPELINE FOR ANIMATRONICS

2.6 SKETCHUP

WALKTHROUGH

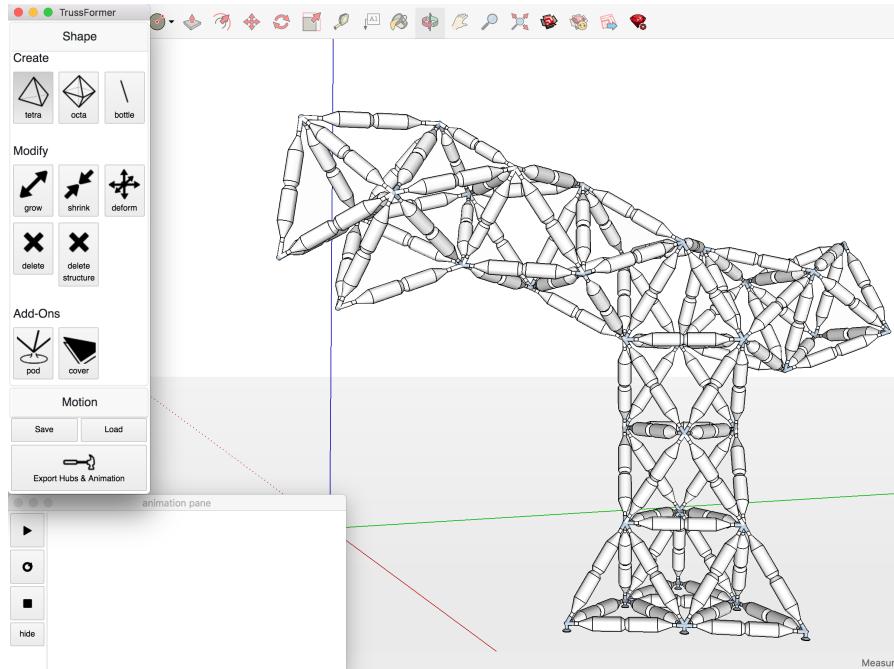


Figure 3.1: The T-Rex built in TrussFab

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

Don't show animation pane...

3.1 DESIGNING STATIC STRUCTURES

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

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

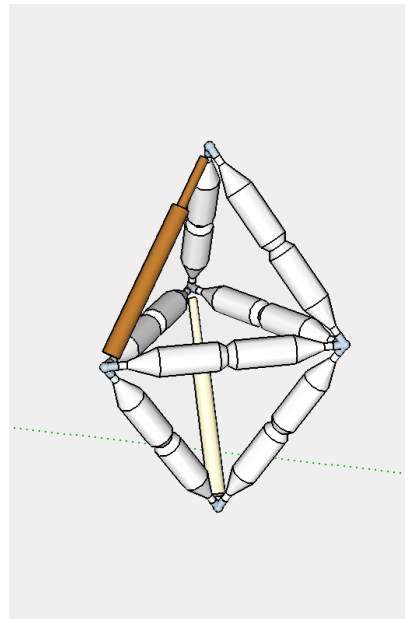


Figure 3.2: An exported leg asset...

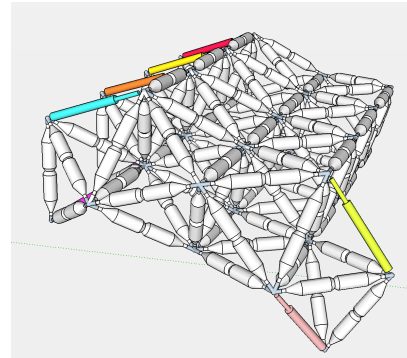


Figure 3.3: ... which can be used to quickly build the spider

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.

- placing actuators
- placing primitives with variable geometry trusses
- demonstrate movement tool

3.2.1 Force Analysis

- check tension force on edges
- check acceleration and speed on nodes
- add loads to object
- check tension while moving
- automatically fix movement when object is exceeding force

3.3 CONTROLLING THE STRUCTURE

- closed-loop control -> more sophisticated and complex movements possible

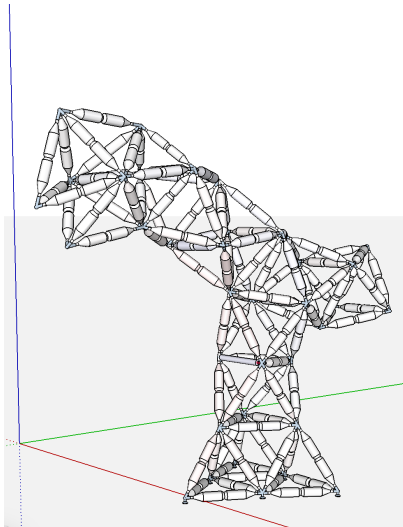


Figure 3.4: An exported leg asset...

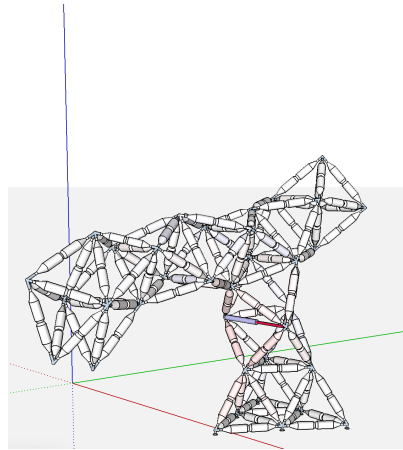


Figure 3.5: ... which can be used to quickly build the spider

3.3.1 PID Control

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

3.4 BUILDING THE FINAL OBJECT

After the object was sufficiently tested in the editor, it is time to print the connectors and assemble the final object.

3.4.1 OpenSCAD

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

The resulting arrangement of nodes and edges will be transferred to OpenSCAD. Using templates, we can create parameterized representations of hubs and hinges, which, when assembled, will exactly represent the object in the editor. This will be explained in more detail in [5.5.5](#).

This needs to be more detailed for sure!!

3.4.2 *Printing the Parts*

put conversion script in here somehow

add some time reference here?

Each OpenSCAD file represents a single part in the structure. These files can easily be converted to *.stl* files, which are typically used for 3D printing. These files have to be imported into any 3D printing software, arranged efficiently and send to a 3D printer.

3.4.3 *Assembling the Structure*

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

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 “extended” edge-line (elongation) contains the id of the connected edge.

A compound elongation, which is the usual case for a hinge, is therefore assembled by finding two parts with the same node and edge ID. For static hubs, this concept is similar, but of course these do not have to be assembled.

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

HARDWARE

- chapter will talk about challenges we faced in finding stable connectors
- material used: PLA (biodegradable, sturdy enough, ...)
- assembled based on ID system
- no special requirements to printer
- we used: UltiMaker3, UltiMaker2 and _____

remember name of other printer

4.1 BUILDING PARTS

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

These links can be connected in two different ways. If the truss primitive is static, i.e. it does not allow deformation, we connect them by hubs. Hubs are single-part connectors for an arbitrary number of edges. They do not allow movement.

make sure people understand what that means

If the structure is intended to allow deformation, we can not use this single-part approach. In this case, movement is created having multiple parts that can hinge around each other. These *hinge chains* are generated according to the number of edges connected to the node and the angle of each edge relative to each other edge. In contrast to hubs, hinge chains have to be assembled manually using nuts and bolts.

Links and hubs or hinge chains, respectively, are connected by specially-printed connecting *cuffs*, which fit over the bottles thread and a fitting counter-part on the connecting end of the node.

4.1.1 Links

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

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

4.1.2 *Hubs*

4.1.3 *Hinge Chains*

- beginning: open hinge chains
- later: closed hinge loops

4.1.4 *Cuffs*

In order to connect links to nodes, we developed a custom coupling system. These cuffs fit exactly over the neck of the bottle and special connecting parts on the hubs. - something about sizes of bottle neck

- size of connecting part
- little dimple for extra stability

4.2 **CONTROLS**

4.2.1 *Electric vs. Pneumatic Actuators*

4.2.2 *Open Loop vs. Closed Loop*

IMPLEMENTATION

We implemented TrussFormer as a plug-in for the 3D modeling software *SketchUp*. It is primarily written in Ruby and JavaScript.

Do we assume TrussFormer is a new product, which uses similar functionality as TrussFab, or do we say it is an improvement?

5.1 ARCHITECTURE

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

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

The export function using OpenSCAD will be explained in section 5.5.5. The structure diagram in figure 5.1 shows an overview of the

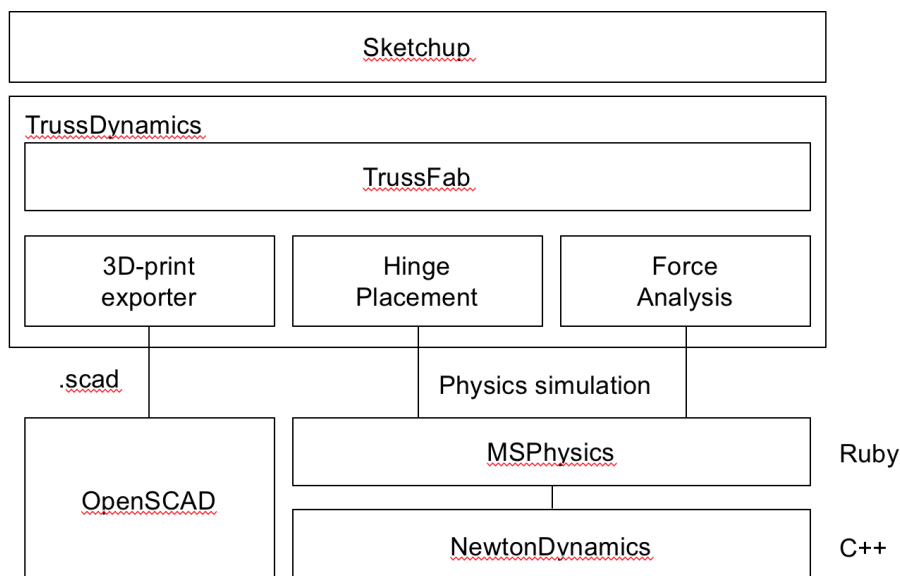


Figure 5.1: TrussFab Architecture

components. 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>

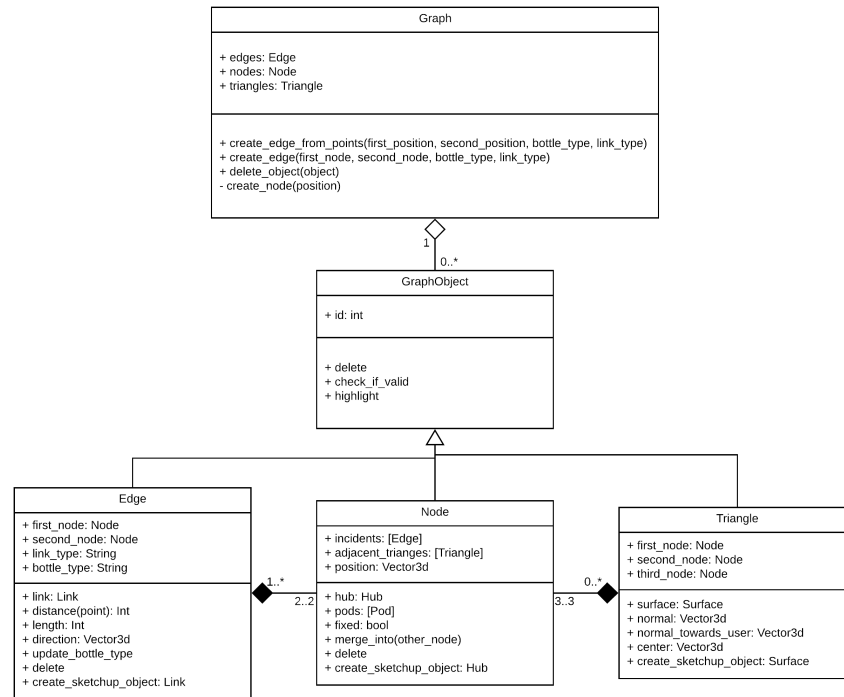


Figure 5.2: Class Diagram showing the high-level Graph Structure of the TrussFab Designer

5.1.1 Designer

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

explain what a Singleton is

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

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

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

Show before and after code snippet

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

implemented in the respective subclass.

Nodes are the connecting components of the structure. *Edges*, as well as *Triangles* are created based on Nodes. Apart from storing adjacent Edges and Triangles, a Node can specify their positions in the SketchUp world. The Nodes' adjacent objects constantly check if their position has changed and update their SketchUp representation accordingly. If the structure is deformed in such a way that a Node will be at the same position as another one, the Node object can automatically merge into the other Node. The Node will iterate over all its adjacent Edges and tell each one, apart from the Edges that run from the other Node to the Edge the Node is hinging around (i.e. the Edge that is opposite the Node), to exchange itself with the Node it wants to merge into. These Edges are removed from its own adjacent Edges and added to the collection of the new Node. The same happens for all adjacent Triangles. As a last step, the Node deletes itself and all remaining adjacent Edges and Triangles (which will be the Edges and Triangles that got merged). The object will then be adapted according to the new positions using the *Relaxation algorithm*, described in section 5.5.4.

```

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).
16     nil?
17     triangle.exchange_node(self, other_node)
18     other_node.add_adjacent_triangle(triangle)
19     merged_adjacent_triangles << triangle
20   end
21   @adjacent_triangles -= merged_adjacent_triangles
22
23   delete
24 end

```

Listing 5.1: Merging of two Nodes

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

The *Edges* are the most visual components of TrussFab. They are visualized by bottles of different lengths, if they are static links, or as

add image

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

IMPROVE!

5.1.2 SketchupObjects

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

Each *SketchupObject* has a *Sketchup::Entity*, which is a class provided by SketchUp that is capable of handling the representation in SketchUp itself. This includes changing the color of the model, hiding and transforming. On creation, each *SketchupObject* is also persisted in the entity.

find out why exactly I did that

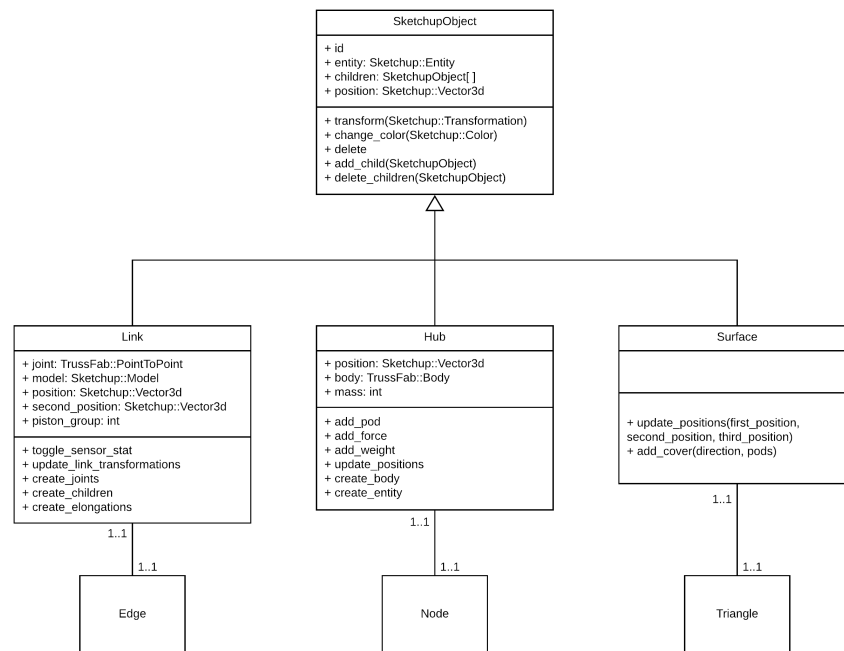


Figure 5.3: Class Diagram showing the UI components of the graph structure

5.1.2.1 Hubs

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

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

5.1.2.2 Links

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

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

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

5.1.2.3 Surface

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

5.2 PHYSICS SIMULATION

TrussFabs' force analysis used to be based on *Finite Element Analysis*, calculated asynchronously on a remote server. This provided fairly ac-

curate results and did not require a powerful computer to run. However, TrussFabs' responsibilities evolved during the course of its life and we decided to implement a real-time physics engine inside of our plug-in.

We decided to use the SketchUp plug-in *MSPhysics* by Anton Synytsia³. MSPhysics is capable of calculating real-time physics on SketchUp elements and creates a customizable physics world in the modeling software. This MSPhysics world has parameters, like gravity, update timestep, and solver model which we can adapt to maximize accuracy and speed of the simulation.

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

```

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

```

Listing 5.2: Simulation `nextFrame` method

During this update, the physics engine calculates new forces on each physics component of the built object. For that, first all static forces are applied to the object. These are forces added by the *Add Force Tool* or static forces calculated by the *GenericPointToPoint* joints. Using these values and all other intrinsic parameters included in the physics objects, we call the entry point to our MSPhysics plug-in. The `@world.advance` function calculates the change of forces and positions from one timestep to another. In our physics world, one timestep cor-

³ <https://github.com/AntonSynytsia/>

relates to $1/60s$, to achieve realistically timed results assuming that SketchUp itself runs with 60 frames per second.

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

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

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

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

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

5.3 MINIMIZATION LOGIC

- elongates and shortens edges so that maximum movement is possible with minimum material use
- uses iterative relaxation algorithm, will be explained in [5.5.4](#)

5.4 EXPORT

5.5 TRUSSFAB DESIGNER

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

5.5.1 User Interface

The user interface (UI) is written in JavaScript. It uses SketchUps' built-in *HtmlDialog* class, which lets us interact with HTML dialog

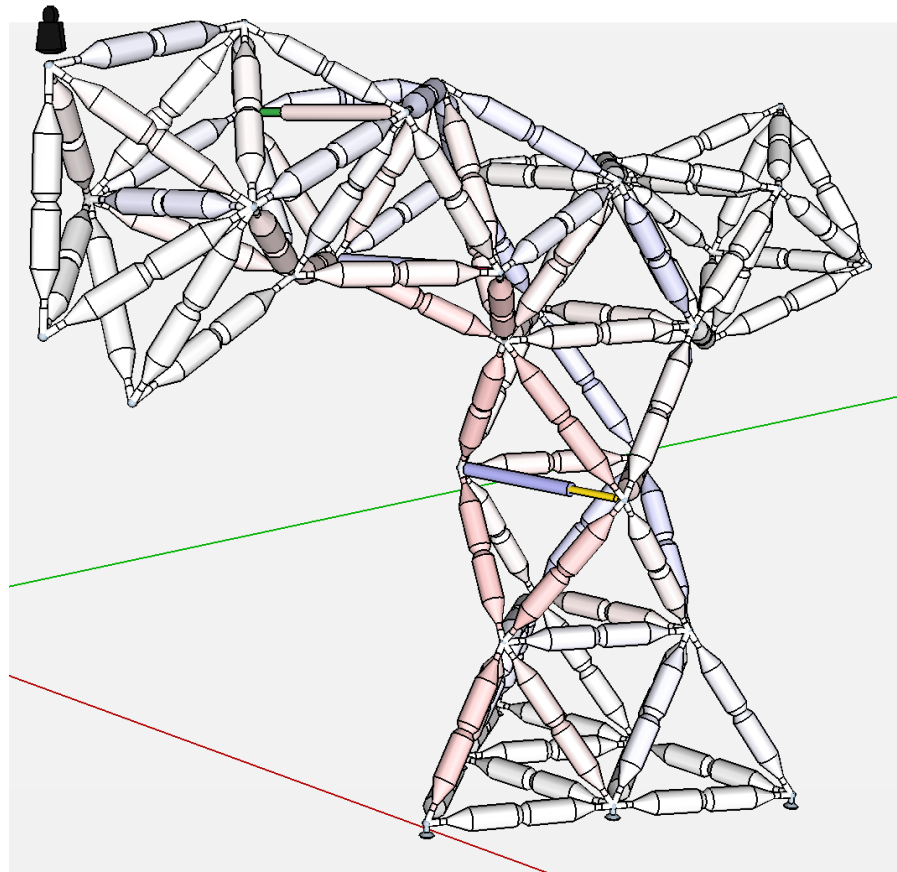


Figure 5.4: Visualization of forces acting on edges. Blue - tension force, red - compression force, white - little or no force

boxes using the Ruby API⁴. The HtmlDialog is a modified version of Google's 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.

⁴ Application Programming Interface


```

1 Class AnimationPane
2   def add_piston(id)
3     @dialog.execute_script("addPiston(#{id})")
4   end
5
6   def stop_simulation
7     @dialog.execute_script('resetUI();')
8   end
9
10  def register_callbacks
11    @dialog.add_action_callback('start_simulation') do |_ctx|
12      if @simulation_tool.simulation.nil? ||
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 5.3: excerpt from UI callbacks

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

5.5.2 Structure Creation

Terminology:

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

iii. PID link

5.5.3 *Modifying the Structure*

5.5.4 *Relaxation Algorithm*

5.5.5 *OpenSCAD Export*

5.6 TRUSSFORMER PHYSICS ENGINE

5.6.1 *Automatic Actuator Placement (if it works soon-ish)*

5.7 FORCE CONTROL

5.7.1 *PID*

CONCLUSION

BIBLIOGRAPHY

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

DECLARATION

I certify that the material contained in this thesis is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Ich erkläre hiermit weiterhin die Gültigkeit dieser Aussage für die Implementierung des Projekts.

Potsdam, April 2018

Tim Oesterreich