LUDWIG WILHELM WALL

DESIGN AND SYNTHESIS OF DIGITAL MECHANICAL METAMATERIALS

DESIGN AND SYNTHESIS OF DIGITAL MECHANICAL METAMATERIALS

LUDWIG WILHELM WALL

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in IT Systems Engineering

Human-Computer Interaction Group
Hasso Plattner Institute
University of Potsdam

October 2016

Ludwig Wilhelm Wall:
Design and Synthesis of Digital Mechanical Metamaterials
October 2016

ADVISOR:
Prof. Dr. Patrick Baudisch
Alexandra Ion

ABSTRACT

In this thesis, we explore how to create simple mechanical computers as part of 3D-printed objects. We are building on 3D printed objects that are subdivided into a regular 3D grid of cells, where each cell can be configured individually. These structures have also been referred to as metamaterials. We introduce a new type of cell that contains a spring that can be loaded after fabrication and that stays loaded until a mechanical impulse arrives at a specific side of the cell, defined as its input "port". When the impulse arrives, the cell "triggers", i.e., the spring discharges and produces an impulse at its output "port". Concatenating such cells therefore implements a digital signal. Through forking or blocking such signals, we implement simple logic functions. We call the resulting structures "digital metamaterials" and discuss the specifics of the underlying logical paradigm, such as the absence of a clock and the resulting implications on concurrency.

The main value proposition of the resulting mechanisms is that they allow implementing simple sense/process/actuate functionality based on mechanics alone and without the electronics usually deployed. This also allows our mechanisms to be fabricated using 3D printers alone.

We present a system designed to allow users to design and fabricate simple interactive objects. Its main component is a custom editor that allows users to model 3D objects, route signals through them, and verify the correctness of "circuitry" by simulating the signal flow. Additionally, the editor automatically synthesizes cell patterns that implement user-defined logic functions, and exports 3D-printable files. Using this editor, we have created simple interactive objects, including a door latch with combination lock.

ZUSAMMENFASSUNG

Diese Masterarbeit erforscht die Erstellung von einfachen mechanischen Computern als Teil von 3D gedruckten Objekten. Wir verwenden 3D gedruckte Objekte die in ein regelmäßiges 3D Zellgitter unterteilt sind, wovon jede Zelle individuell konfiguriert werden kann. Solche Strukturen sind auch als Metamaterialien bekannt. Wir führen eine neue Art von Zellen ein, welche eine Feder enthält, die nach der Fabrikation gespannt werden kann und dann so lange gespannt bleibt, bis ein mechanischer Impuls an einer spezifischen, als "Eingangsschnittstelle" definierten, Seite eintrifft. Wenn der Impuls eintritt, "löst die Zelle aus", d.h., dass die Feder sich entlädt und an einer anderen als "Ausgangsschnittstelle" definierten Seite einen neuen Impuls erzeugt. Verkettungen solcher Zellen implementieren daher ein digitales Signal. Wir implementieren einfache logische Funktionen mittels Signalspaltung und -blockierung. Wir nennen die resultierenden Strukturen "digitale Metamaterialien" und erörtern Details des zugrundeliegenden logischen Paradigmas, wie das Fehlen eines Taktsignals und der daraus resultierenden Auswirkungen auf Nebenläufigkeit.

Das Hauptnutzenversprechen der resultierenden Mechanismen ist, dass sie es erlauben einfache Funktionalitäten von Sensoren, Prozessoren und Aktuatoren nur mittels Mechanik zu implementieren, ohne die üblicherweise verwendete Elektronik. Das erlaubt die Fabrikation unserer Mechanismen mittels eines 3D Druckers allein.

Wir präsentieren ein System, welches dazu ausgelegt ist, Nutzern zu erlauben einfache interaktive Objekte zu entwerfen und anzufertigen. Die Hauptkomponente des Systems ist ein anwendungsspezifischer Editor, der Nutzern erlaubt 3D Objekte zu modellieren, Signale durch sie hindurch zu leiten, und die Korrektheit eingebauter "Schaltkreise" zu überprüfen, in dem ihr Signalfluss simuliert wird. Zusätzlich synthetisiert der Editor automatisch Zellanordnungen, die nutzerspezifische logische Funktionen implementieren. Weiterhin erlaubt der Editor den Export von Dateien die 3D gedruckt werden können. Unter Verwendung des Editors haben wir einfache interaktive Objekte gebaut, unter anderem eine Türklinke mit eingebautem Kombinationsschloss.

ACKNOWLEDGMENTS

First of all, I would like to express my sincere gratitude to my advisor Prof. Dr. Patrick Baudisch for his guidance and vision regarding this and previous projects and for giving me the continued chance to work on relevant research during the last years of my studies.

Thank you Alexandra Ion for co-advising this thesis, enabling me to focus on the most important aspects of the system and providing me with insights beyond the scope of the project.

I thank Robert Kovacs for his competent assessment of the mechanical solutions I envisioned and his encouragement.

I would like to thank Stefanie Müller for providing lasting expertise on how to study and how to write research papers successfully.

I thank Johannes Filter for his help with the implementation of the 2D signals drawing tool.

I thank Anna Seufert for proofreading the thesis.

I would also like to thank the other members of the HCI chair for their valuable feedback and insights and for giving me the opportunity to experience and to participate in other research projects as well.

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

Personal fabrication machines, such as 3D printers, allow users to make custom objects. The functionality of objects is often defined by their external shape [Weichel]. To increase the functionality, static 3D printed objects are often augmented with electronics [10].

Researchers recently started exploring 3D printing to alter objects by designing their inside, e.g., to make them spin reliably or move their center of gravity. Pushing this further, researchers created objects that consist internally of a large number of 3D cells arranged on a regular grid. Since each cell is designed to perform a specific deformation, objects that entirely consist of such cells literally offer thousands of degrees of freedom. Such structures are also known as metamaterials. These thematically related projects and their influence on this thesis will be presented in chapter 3.

While metamaterials were initially understood as materials, researchers recently proposed to think of them as machines [7]. Such metamaterial mechanisms consist of a single block of material, the cells of which play together in a well-defined way in order to achieve macroscopic movement. In this thesis, we explore how to extend this concept towards digital processing. We do so by combining metamaterial mechanisms with concepts from mechanical computers, as well as mechanisms for signal propagation.

We present an editor for this kind of metamaterial and demonstrate how it can be used to create a prototype of an entirely mechanical door latch with integrated combination lock that might otherwise have been implemented using sensors, actuators, and electronics. Figure 1 shows the editor and the printed design of the mechanical door handle that was designed with it. The pins below the door handle are used to input the code that unlocks the door handle, which is otherwise fixed to its resting position. Chapter 2 demonstrates the creation of this object using our editor.

1.1 DIGITAL MECHANICAL METAMATERIALS

Digital metamaterials are interactive objects based on a cell grid structure that contain mechanical signal propagation inside the object alone, i.e., without sensors, actuators, and electronics. Custom cells allow the device to select, decide, and compute and to send this informa-

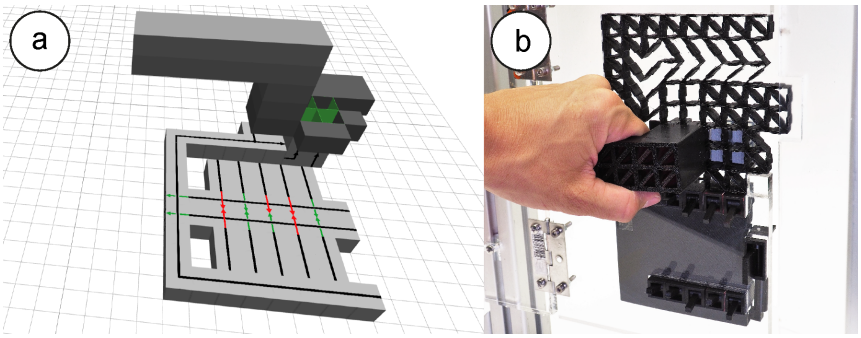


Figure 1: (a) The editor supports the user from the creation of the computation enabled object, over testing the implemented logic, to (b) printing the final 3D model.

tion to other locations by means of transmission cells. Specialized output cells can alter the properties of other metamaterials, such as their permeability or their ability to shear. Other output options include acceleration of projectiles (e.g. a pinball) or locking traditional mechanical parts in place.

These characteristics can be used to replace electronic parts in electromechanical objects where both input and output of the functionality reside in the mechanical domain, and the computation is only a small part of its function (cf. 2). This can sometimes circumvent the need to adjust prototypes after fabrication, e.g. to add electronic parts, completely.

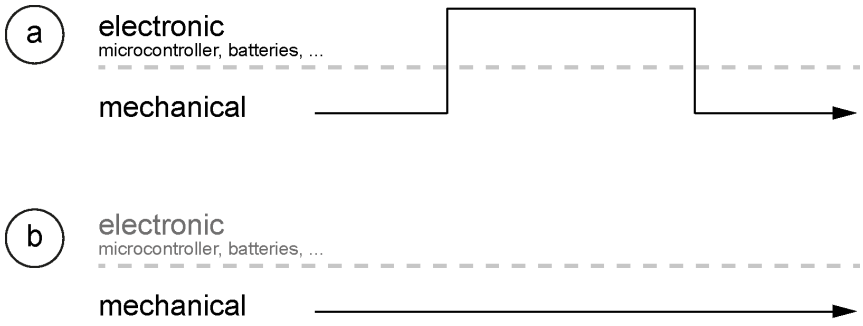


Figure 2: (a) Usually, interactive mechanical systems are controlled by electronics, causing a transition from the mechanical domain to the electronic domain. We propose (b) staying in the mechanical domain by integrating signal transmission and simple computation into the material.

Though the class of objects where our process is applicable is still limited as of today, some useful objects can already be built with our system. We created a lamp shade that can be configured to emit light in a variety of patterns as seen in figure 3 and a plant pot that

calculates its own density to provide more or less water depending on the user input settings for plant size and water consumption.

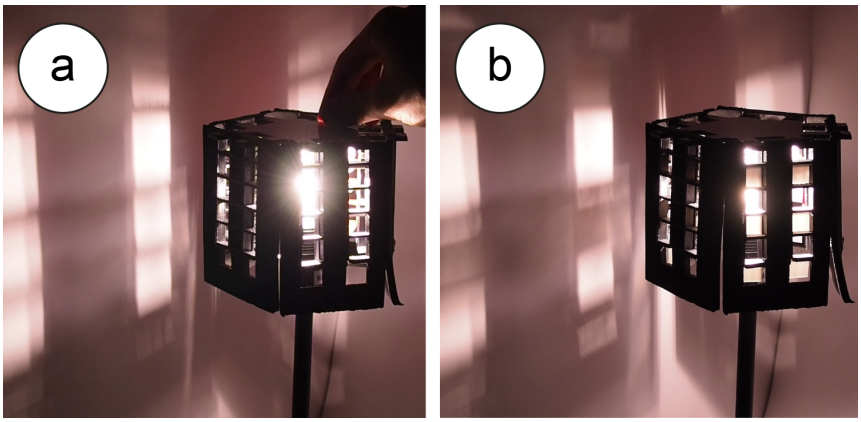


Figure 3: (a) The user manually activates columns of cells, which configures the lamp shade (b) to display different lighting patterns.

1.2 COMPUTATION IN THE MECHANICAL DOMAIN

This thesis introduces a system that fulfills basic mechanical and computational problems in fabricated objects without leaving the mechanical domain. It describes a novel type of metamaterial, the basic building block of which is a cell containing a bistable spring. This cell serves both as energy storage to power the system, as well as a three-state logic switch used to implement combinational logic. Similarly, the mechanical signal we propose serves both as a medium carrying information, as well as an energy source to actuate system outputs. Details of the spring design and the physical foundations that enable a reliable mechanical signal will be depicted in chapter 4. We employ rod logic [11] to implement combinational logic based on the three-state logic cells, as described in chapter 5.

As mentioned before, our system is applicable if the computation part of the task is small. A large number of in- or outputs however can be handled well. One example of a use case with many outputs is changing the properties of an object, since every part of the object may require a specialized output signal or force. Metamaterials theoretically maximize the number of such outputs, as every cell within the object can be viewed as a separate part, and changing object properties becomes changing the properties of the material itself. Since the actuators in our system are metamaterial cells themselves, they scale in a similar fashion as the metamaterials they actuate. Our system thus synergizes well with other types of metamaterial.

1.3 EDITOR FOR METAMATERIALS THAT INTEGRATE LOGIC

Given the ability to print an object with functional mechanical computation and actuation, it is still a very challenging task to create a *mechanical circuit* that implements the desired logic and connects all in- and outputs. This thesis therefore further presents an editor that supports the user in creating digital mechanical metamaterials. Besides a variety of tools intended to create metamaterial mechanisms based on [7], the editor also offers instruments to help the user in implementing combinational logic with digital mechanical metamaterials. The base software already supports the user with tools like a finite element analysis simulation that displays the expected deformations of the material when forces are applied, to assist mechanism design in metamaterials. Arranging signal lines in a 3D environment is supported via intuitive drawing tools and pathfinding capabilities, such as the *add* and *signals* modes in figure 4a. Predefined brushes to place common circuitry directly, such as gates and multiplexers, and a logic synthesizer assist in designing the computation (cf. figure 4b).

Details and implementation of editor features will be discussed in chapter 5. Finally, the editor exports the digital representation of the object to an OpenSCAD script, which in turn converts it to a printable 3D model. The editor user interface supports this via a simple button click (cf. figure 4c).

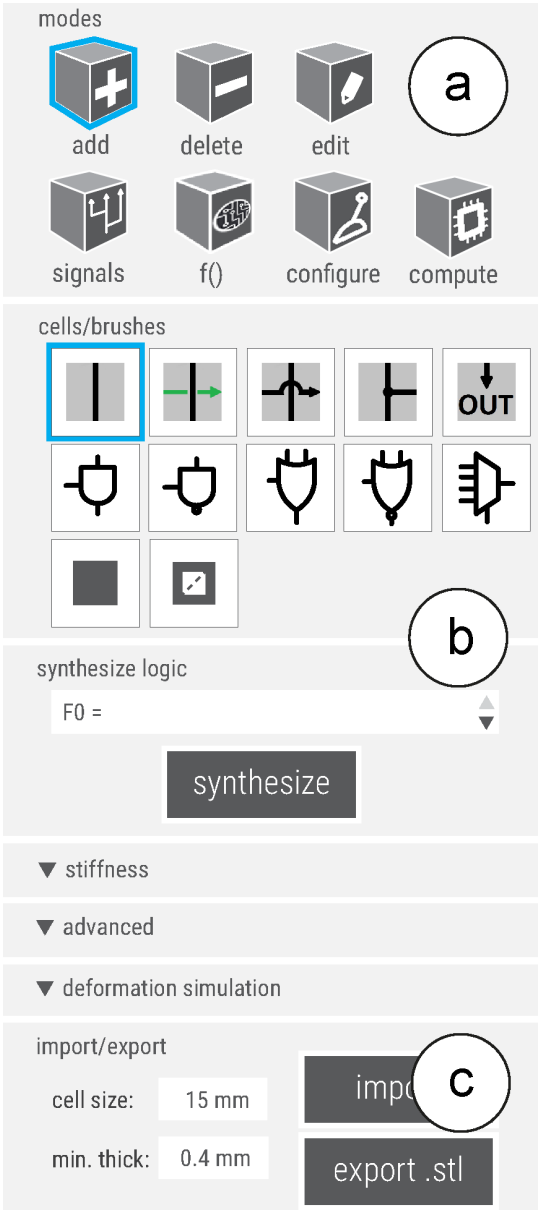


Figure 4: The menu of the user interface of our editor. Functions focussing on editing metamaterials alone are folded down, e.g. the stiffness settings. (a) The basic interaction modes of the editor. (b) Brushes for logic cells of logic gateways. (c) The export function for generating 3D printable files.

2

WALKTHROUGH

The following walkthrough will introduce the functionality of the system, by showing the process used to create the door handle in figure 1, while also explaining the operation of the resulting physical objects that would be created by the editor. Demonstrating both in parallel shall provide a better understanding of the overall system.

Both designing functional metamaterial mechanisms and circuit design are challenges in themselves. The editor presented in this thesis is based on the voxel-based editor from *Metamaterial Mechanisms* [7]. This type of editing environment was chosen, since metamaterials are based on cells, which are represented well by voxels. Voxels are visualization elements in a volume, i.e. on a three dimensional grid. We help users to create functional circuits while abstracting from and making use of the rod logic computation paradigm (cf. section 3.5). Building on the previous work, designing computational logic and physical mechanisms can now be combined within one workflow. Through that, changes and adaptions between those two commonly separate tasks can be applied interactively, which can accelerate the overall design process.

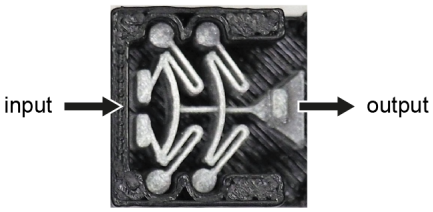


Figure 5: The physical object that is created from a transmission cell in the editor. It is made up of a frame (black) and a bistable spring (silver) and transmits a signal to its output port after receiving a signal at the input port.

2.1 BUILDING THE LOGIC FOR A COMBINATION LOCK

The most basic mode of operation is placing, deleting or modifying single cells by hand. To do so, users select an interaction mode, e.g. the *add* mode used to add cells, and a type of cell to be added, which functions as a brush for the chosen mode. The editor user interface is shown in figure 6 and the physical object that would be created from a single transmission cell can be seen in figure 5.

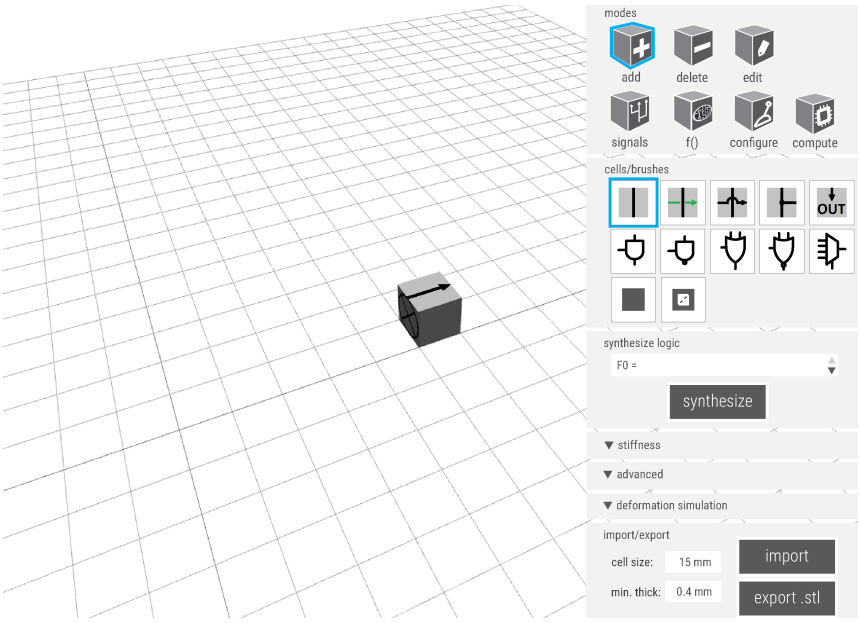


Figure 6: Editor UI: Users selected the add mode and the simple transmission cell brush, both highlighted by a blue frame. Clicking on the grid creates a single transmission cell.

Figure 7 illustrates how users create the computation part for the door lock example from figure 1. (a) They first draw the signal line that evaluates the upper 5 digits by dragging over the ground plane using the *signals* mode. (b) Then, using the same mode, they draw signals perpendicular to the first signal line. When the two signals intersect, the editor automatically draws gate cells. (c) They do the same for the lower row of digits. (d) Users configure the gate cells using the *configure* mode, i.e., they change the initial state of 5 gate cells from initially 'pass' to 'block', which implements the key code.

The line of transmission cells built in figure 7a creates a signal path, because the in- and output ports of these cells are properly aligned. Cells are considered *charged* if their bistable spring is in its second stable state. A push on the input port of the cell causes the bistable spring to rapidly return to its first stable state, thus *triggering* the cell

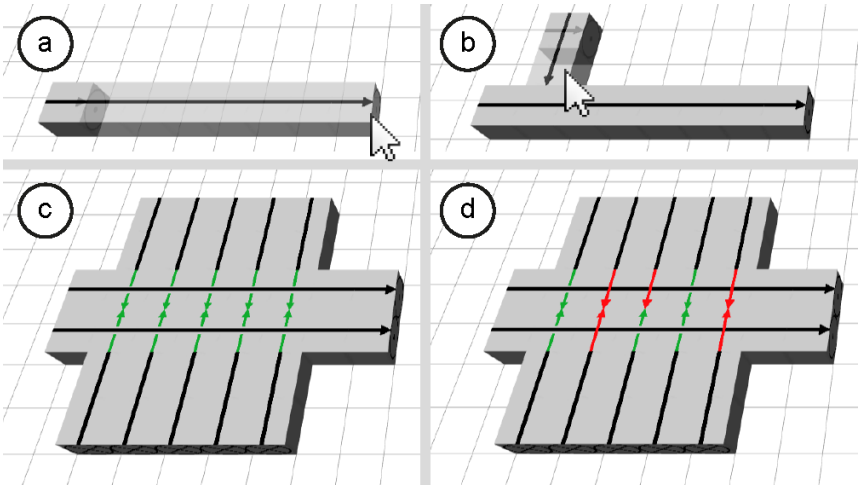


Figure 7: (a) Users draw the signal routing using the *signals* mode of the editor. (b) Once they cross an existing signal route, the editor automatically draws a gate cell. (c) After creating all cells for the digit evaluation, (d) users configure the initial states of the gate cells to define the key code.

and sending a mechanical impulse to the output port of the cell. This causes a chain reaction in all following cells in line (cf. figure 8).

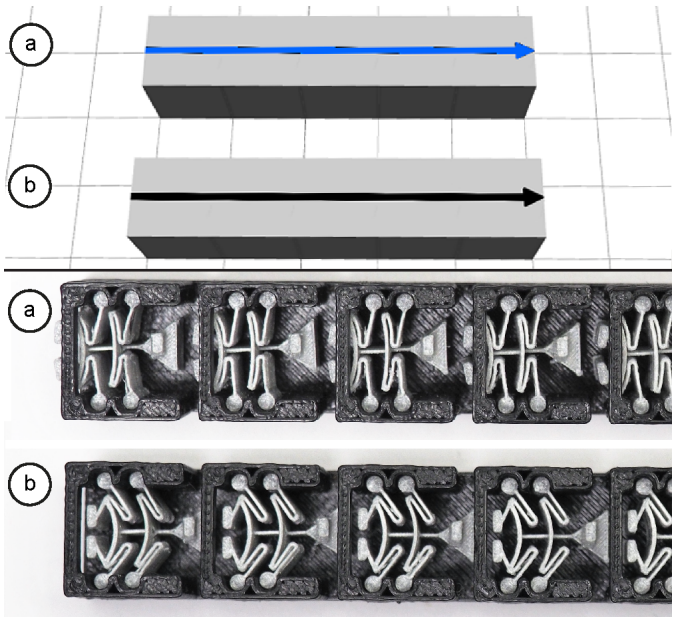


Figure 8: (a) These lines of cells are *charged*. *Triggering* the leftmost cells causes a chain reaction that triggers all following cells, resulting in (b) a line of uncharged cells.

Users start to draw *gate cells* in figure 7b-c. These cells can also transmit signals as normal, but they offer an additional state in which

they physically block any signal that reaches the cell. This state is set by adjacent transmission cells, called *blocking cells*. A row of gate cells functions as an *AND* gate, as a signal can only reach the end of the row, if all of the gate cells are configured to let the signal pass.

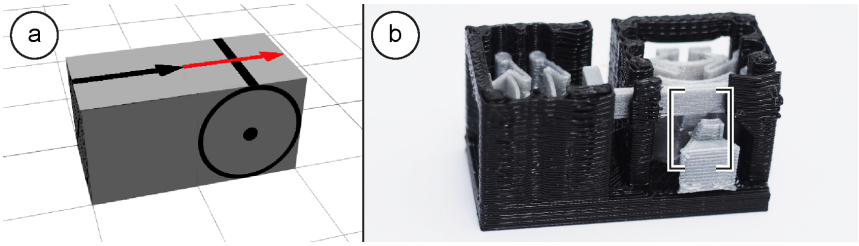


Figure 9: (a-b) The *blocking cell* on the left configures the *gate cell* on the right. If the blocking cell is charged, the signal of the gate cell can pass, otherwise it is blocked.

Figure 9 illustrates how a signal is blocked, while (a) shows the editor view and (b) the printed result. When the cell on the left is triggered, it moves a blocker in the way of a pushing extrusion of the cell on the right. When the cell on the right is then activated, the blocking rod is in the way of the pushing extrusion, preventing the spring from reaching its relaxed position. The cell is prevented from triggering, thus blocking the signal. However, if the left cell is still charged, as shown in figure 10a, pushing extrusion and signal can pass without issue.

In figure 7d, users set the combination of the lock, which is defined by whether gate cells allow signal transmission before or after the corresponding blocking cells have been triggerend. To achieve this on a physical level, the position of the blocker on the blocking rod is altered. It is set to initially block signals and only let them pass after actuation in figure 10c-d. Other types of cells, such as bifurcation and redirection cell, will be introduced in chapter 4. Section 4.1.1 explains how arbitrary combinational logic can be implemented through simple signal blocking and section 5.3.1 describes how it is implemented in our system.

2.2 TESTING THE LOGIC AND INTEGRATION WITH THE METAMATERIAL DOOR HANDLE

Users continue building the prototype of the door handle by adding a line of springs for the evaluation of the created logic, which ends at the point where the metamaterial door handle itself will be built. They add a specialized type of output cell, an *operational amplifier*, to

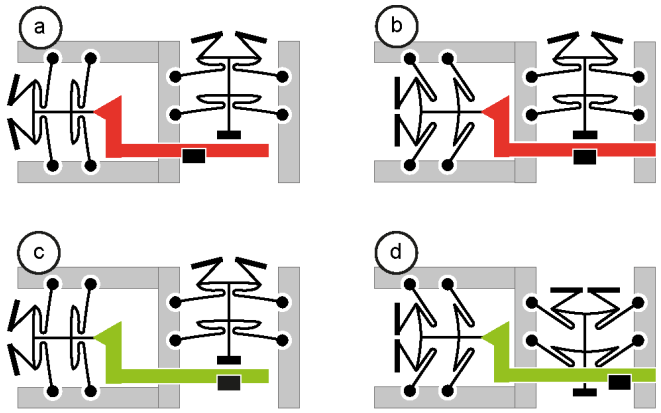


Figure 10: *Gate cells* validate signals and can be configured to block signals (c-d) or let signals pass (a-b) in the tense state of the *blocking cell*.

the end of that signal line. This operational amplifier is an enlarged version of the transmission cell that takes up the space of eight cells and offers a much larger movement of its spring. This large movement is used to move bolts in or out of shearing cells of the door handle, thus allowing it to move as intended or to be locked in place. To activate this large cell, the signal is first pre-amplified, which means that the signal is simply bifurcated right before the operational amplifier is triggered. Users then build the actual metamaterial door handle mechanism on top of the operational amplifier using the methods presented in [7]. The editor result can be seen in figure 11 and the printed result is shown in figure 12.

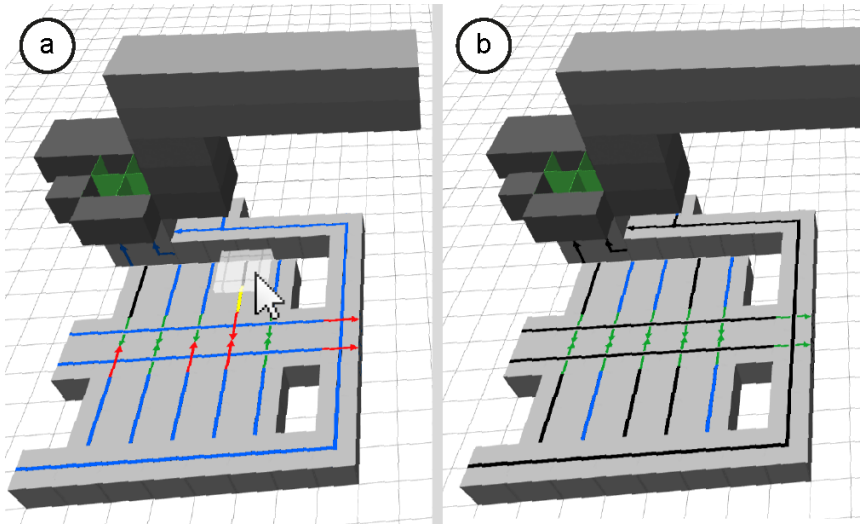


Figure 11: The finished prototype. Users can verify their logic and signal routing. They first charge all springs, then (a) they click the inputs to trigger the signal there, and lastly (b) they trigger the evaluation line and find that the signal passes all the way through to the latch.

Figure 11 shows how users verify the signal transmission in our custom editor. They first charge the cells, which is visualized by turning the signal lines blue. The cells are automatically charged whenever users activate the *compute* mode of the editor. They trigger the inputs and the evaluation signal, as they do on the 3D printed object, and watch if the signal runs through to the door handle. If not, they see where the signal stopped and can correct the error.

All the gate cells in 11b are green, meaning that they let a signal pass through them, after the correct input code has been entered. In the case of a combination lock, this should only be the case if the exact combination of inputs was entered, though the order in which it was entered is not important. The evaluation line however has to be triggered last, as it evaluates the computation given the current state of the system. This evaluation line therefore only unlocks the door if the entire correct code was entered previous to its activation.

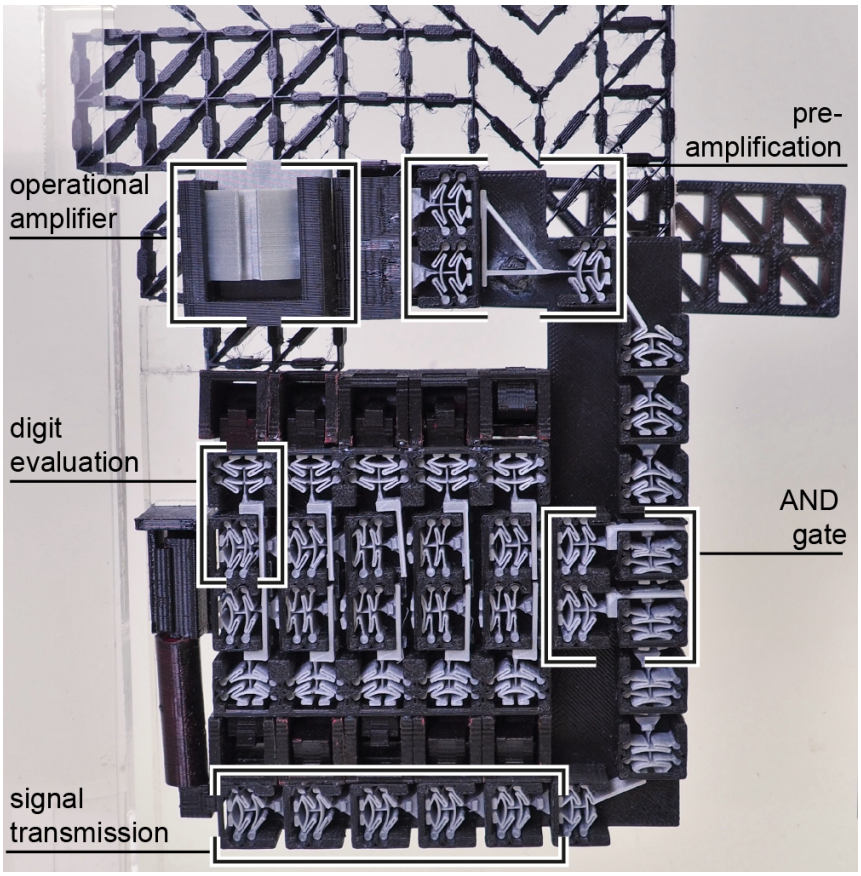


Figure 12: The final door lock consists of 82 cells, which implement the signal transmission, the evaluation of each digit input by the user, an *AND* gate, and one operational amplifier with a pre-amplification step to move the blocking bolts sufficiently far.

Users manually drew the logic that implements the combination lock in the previous section. The editor however also provides the functionality to synthesize these logic cells automatically. Different methods for synthesis are available depending on in which form the logic function is known to the user. These options and the logic synthesis itself will be discussed further in section 5.3.

The first line of logic that evaluates the input digits for the door lock in figure 7d is described by the function $F0 = A \,\&\, \neg B \,\&\, \neg C \,\&\, D \,\&\, \neg E$. If and only if the bits A and D are set, the function returns true, opening the lock. Users may input this function directly and let the editor automatically synthesize the necessary cell patterns to fulfill the function.

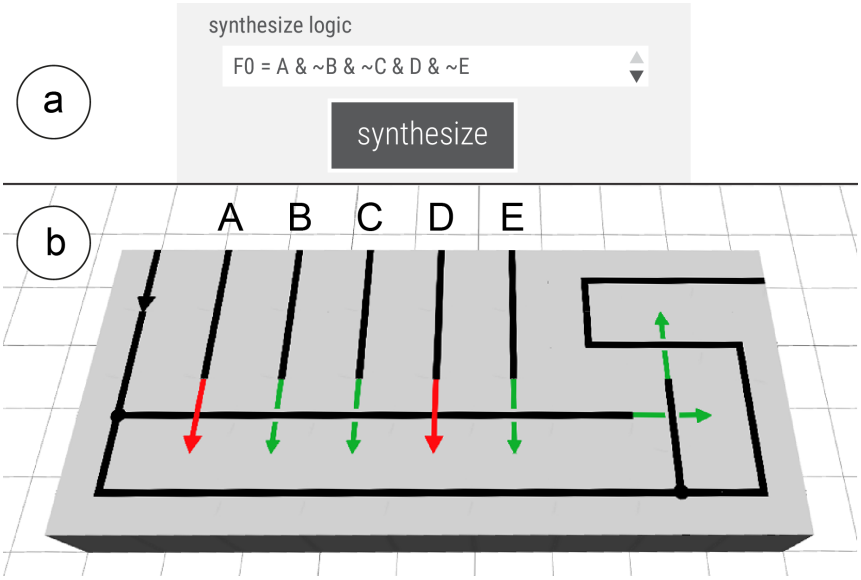


Figure 13: The input function (a) is automatically minimized and a cell pattern (b) is synthesized by the editor.

To acquire a 3D printable file, users simply have to use the export button. The editor then runs a modular OpenSCAD script that automatically generates a .STL file from the voxel based model, which can be used for 3D printing. The final printed prototype can be seen in figure 12. Details of the export functionality will be discussed in section 5.3.4.

3

RELATED WORK

This work is based on previous work in personal fabrication, mechanical metamaterials and analog computers. The ideas of designing the inside of objects to add specific functionality to existing models as well as mechanical signal transmission in fabricated objects have been explored before. Decentralized computation on a grid-like structure relates to cellular automata, while the state-based nature of our cells and the interplay of the cell patterns resemble finite automata and related modeling techniques such as petri-nets.

3.1 PERSONAL FABRICATION

Personal fabrication machines such as 3D printers allow users to make custom objects. Besides printing decorative objects, users often create functional objects the functionality of which is determined by their external shape [12, 24].

To fabricate mechanical assemblies, users can print the structural parts from rigid plastic (e.g., links, axles, bearings, gears, etc.) and assemble them to construct machines [2]. Such assemblies can also be printed in one process [5]. To allow users to go beyond mechanical systems made on their personal fabrication machines, researchers proposed techniques to integrate sensors and microcontrollers into objects. They range from (capacitive) position sensing [9, 23, 20] to sensing light beams for detecting user interaction [25], to complex systems like [19] integrating a camera to track markers. Figure 14 shows an example object with an integrated opening for adding a camera, that converts it into a customized input device.

Printed optics have been added to the inside of an object to enhance both its output and input capabilities in figure 15.

While image processing is not possible in mechanical systems, we argue that simple mechanical information processing can be integrated with a 3D printed object to alter its properties. To do so, the internal structure of the 3D printed object is designed.

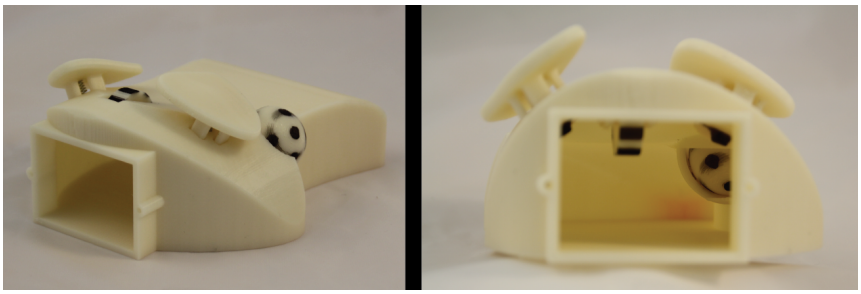


Figure 14: *Sauron* [19] enables inserting a camera into an opening after fabrication to increase the interactivity of the object, allowing it to be used as a trackball mouse. Its movable parts are tracked from within the object and the captured information is processed to be used as mouse input.

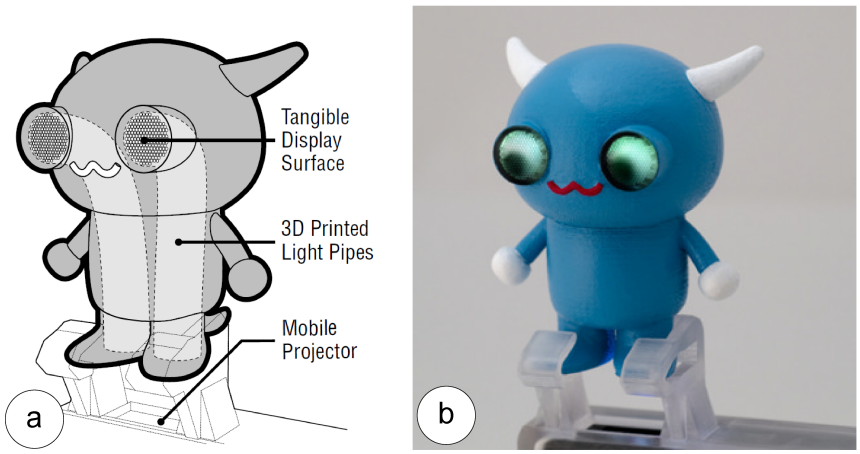


Figure 15: (a) *Printed Optics* [25] integrates light pipes into the model to (b) allow dynamic eye movements of the figure and easily accessible touch input.

3.2 DESIGNING THE INSIDE OF OBJECTS

Researchers recently started to advance the possibilities of 3D printed objects by designing their interior, allowing the created objects to spin reliably [3] or adjust their center of gravity [17]. Both methods take 3D objects with detailed outer shapes and adjust their interior, in this case by carving voxels out of it (figure 16), to add new functionality to the original object. Other approaches discussed in the cited work are to slightly alter the original shape of the object and to use different types of material with different masses for specific voxels to adjust the center of gravity further.

Treating the interior of an object as well-defined configurable cells instead of voxels leads to metamaterials.

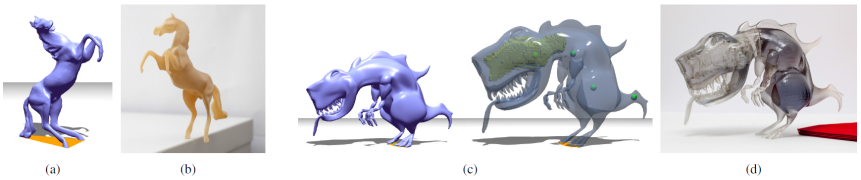


Figure 16: Deformations through *Make It Stand* [17] enable (a) the original horse model (b) to stand on a single leg. (c) Voxels from the inside of the T-Rex head have been removed (d) to allow the printed model to stand without support.

3.3 MECHANICAL METAMATERIALS

Metamaterials are "artificial structures with mechanical properties that are defined by their usually repetitive cell patterns, rather than the material they are made of" [16]. Metamaterials consist of large numbers of 3D cells organized on a regular grid. Since each cell can be designed to deform in a specific way [22, 15], the degrees of freedom they offer is only limited by their number of cells.

Based on this concept, researchers have created objects with unusual behaviors, such as metamaterials that collapse abruptly when compressed [13], that shrink in two dimensions upon one-dimensional compression [6], or objects that arrange layers of varying stiffness (i.e., soft and hard cells) in order to emulate different materials, such as leather or felt [4].

However, metamaterials are usually seen as materials as the name suggests. Using cell structures with well defined deformations, researchers created analogue *machines* (figure 17) entirely from metamaterials, elevating the concept beyond the originally proposed understanding as a novel type of *material* [7].

Making use of the discrete states of bistable springs, this work advances the concept of metamaterials further.

3.4 MECHANICAL SIGNAL TRANSMISSION USING BISTABLE SPRINGS

This work builds on mechanical wave propagation using bistable springs [14]. 3D printed bistable springs transmitting a mechanical signal have been investigated by Raney et al. [18]. In their paper, a chain of bistable springs, which are connected to each other via a zigzag spring or a stiff member, transmits a mechanical signal in conjunction. The length of the chain, i.e. the number of springs that are simultaneously switching between two stable states during signal transmission, varies from 2 to 18 springs depending on parameters defining the stiffness of the springs. The force this chain of springs exerts after activation varies depending on the spring parameters, and

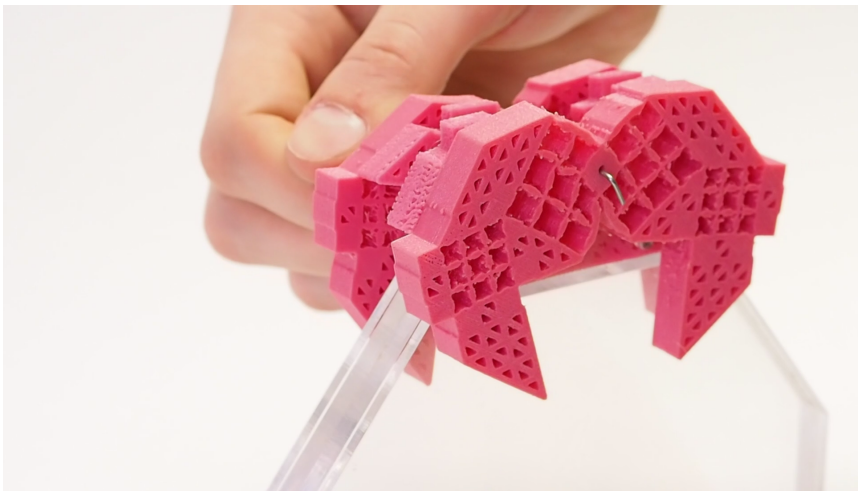


Figure 17: This machine made out of one piece of material was created by *Metamaterial Mechanisms* [7] and converts the rotational input of an axis into a walking motion of its legs.

different force levels are utilized to create basic gateways or diodes. As output forces of the signals change during computation steps (figure 18), additional measures would have to be taken to scale computation by concatenating multiple steps.

This work proposes digital machines instead of analog ones, in which the signal-transmitting cells are self-sufficient units, that can fulfill computation without changes in their force level, thus promoting scalability. The details of cell operation allowing digital signal transmission will be discussed in chapter 4.

3.5 ROD LOGIC

The work [11] initially introduced the concept of rod logic, which describes a system of moving rods in which the position of the rods encodes bits of information. The mechanical movements of the rods encode digital information. It was intended to be used in nanoscale devices, but the presented concepts can also be used in a larger setup to allow computation within objects made through personal fabrication. This thesis builds on concepts of rod logic, but it uses lines of independent cells instead of rods. A rod in their system can be in two meaningful positions, named 1 and 0 here, and transition between the two. The rods are to be constricted to one-dimensional movement and have a spring at the end of the rod that pulls the rod from 0 towards the 1 position.

Along the rods, gate- and probe knobs are attached to it, which can prevent the movement of the rod. Rods are aligned in layers. Rods from the next layer are placed perpendicular to rods from the previ-

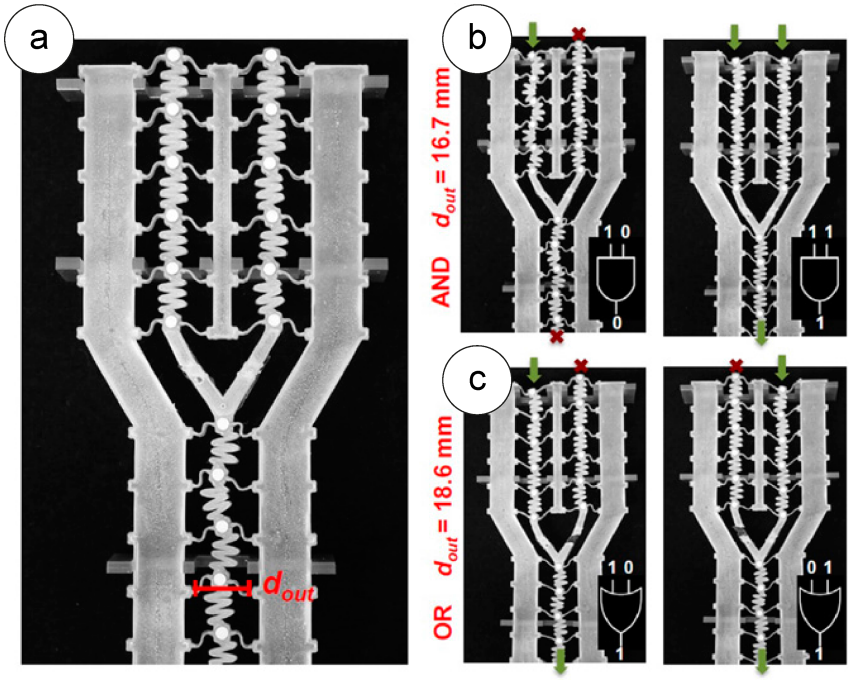


Figure 18: In [18] a signal propagates through soft material using chains of bistable springs. (a) Increasing the width d_{out} decreases stiffness of the output line of springs. (b) The stiffer setup requires the force of both top inputs to activate, implementing an *AND* gate. (c) The softer setup realizes an *OR* functionality since each of the top inputs can activate the bottom output.

ous layer. The rods in the lower layer function as input to the layer above. Probe knobs along a rod can collide with gate knobs along perpendicular rods, thus preventing the movement of the rod and effectively setting the output of the rod to 0. Any rod can be thought of as an AND-/NAND-combination of its perpendicular input rods, since it can only move and therefore reach the 1 position if none of the gate knobs of the perpendicular rods prevent its movement. Depending on the placement of the gate knob, the perpendicular rod has to be in the 0 or 1 position to allow the original rod to move. This realizes a form of inversion of inputs right at the position where the value is used.

Computation that exceeds determining a single (or multiple) AND/-NAND value happens in multiple stages. First, the initial input values of the computation are set by driving the appropriate rods into their 1 positions. Then any number of rods can be evaluated in parallel based on these inputs. Once these rods have reached their final position, their positioning can be used as input for a different set of rods. This process can be repeated as necessary. Once the result has been read, the tension of the drive springs is released and all rods return to their initial positions.

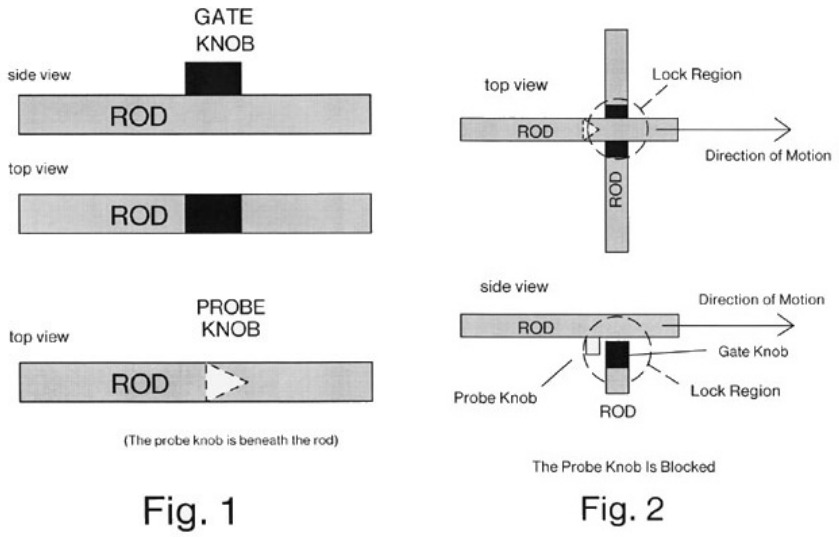


Figure 19: Rod logic knob positioning and blocking operation [11]. The gate knob blocks the probe knob and thus the movement of its rod in this diagram.

3.6 AUTOMATA AND PETRI-NETS

Cellular automata fulfill computation through localized state changes within a system composed of a large number of cells [21]. Changes cascading through the cells of a cellular automaton require discrete steps of progressing time. A clocked timing is therefore inherent to their computation, while this work assumes a simplified timing where all changes affected by one cascade happen instantaneously instead. The computational possibilities of the proposed clock-less system will be depicted in chapter 5.

A different type of automata, communicating finite state machines (FSM) [1], are used to model the behavior of the cells used in our system. The modeling and application details of these FSM's will be discussed in section 5.6.1. Traditional FSM's model the basic cells of this system, and a hierarchical composition of the cells can be used to model the behavior and thus the computed logic they implement. Other modeling techniques such as petri-nets also model the behavior of a system using discrete state transitions [8].

4

HARDWARE AND MECHANICS

The system presented in this thesis makes use of mechanical signal propagation to compute logic in 3D printed objects. The following chapter will depict mechanical challenges and our solutions that allow the signal transmission through 15mm cubic cells. hardware alternatives that enable basic computation and actuation functionality without the need for electronics within the object.

Our prototypes are printed from the commonly available filaments ABS and PLA. Our cells are designed to be printed in an assembled state. We proved the feasibility of this concept by printing 35mm sized cells on the Dimension SST 1200es printer, that has the ability to print soluble support material. The cells were functional after the support material has chemically been removed from them. Later we printed the parts of our prototypes separately: we printed the springs from PLA using the Ultimaker 2+ 3D printer, and the frames that hold the springs from ABS on our Dimension SST 1200es. The higher resolution of the Ultimaker 2+ printer allowed us to produce all prototypes using 15mm cells.

4.1 COMPUTATION USING IMPULSES

Using springs utilizing one-time activation to transmit signals imposes challenges onto the domain of what can be computed and which methods can be used to do so. For example, no clock is present, which restricts computational capabilities, e.g. loops within the program are not possible; restricting the computation to combinational logic. It is also not possible to create an inverter in our system, which inverts a 0 to a 1. (Inverting a 1 to a 0 can implemented simply through blocking the signal.)

These challenges are based on fundamental differences between electric circuits and our mechanical computational. A 'signal' within our system is not an applied voltage, but an impulse, i.e. a mechanical force within the object. This impulse changes the system state by changing physical properties of material, such as the positioning of objects. The impulse is assumed to happen in zero time, and therefore only allows distinguishing between whether an impulse was received at any point in time prior to now, or not.

4.1.1 *Avoiding the necessity to use inverters*

The cells in our system have an internal power source, but they are not connected to an external power supply. Triggering a cell, thus creating a 1 in our system, requires an impulse. Not only is "not receiving a signal" indistinguishable from a dormant system, without an initial impulse, no cell can physically be triggered. It is therefore impossible to create a simple unary inverter within our system (cf. figure 20). Combinational logic however can only be implemented if a basic AND- or OR-gate, as well as a negation of a signal can be achieved. We therefore use an alternative concept that utilizes three-state logic, which effectively results in a binary inverter, to still be able to compute logic.

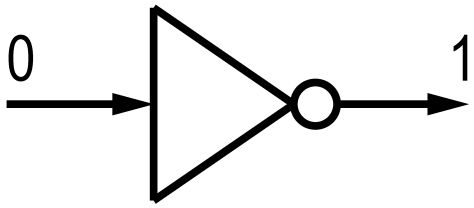


Figure 20: An inverter is a common building part in electronic circuits which we cannot use in our system.

There are two possibilities to deal with this issue without leaving the mechanical domain. The first is a concept called dual rail logic, in which all signals in the system are duplicated. It requires that all decisions concerning signals have to be made explicit. If a button has not been pushed, would normally imply a 0 signal. In dual rail logic however, all input "buttons" become an input "switches", and users have to explicitly set all inputs to either true or false. They trigger either the "0-signal" or the "1-signal" and since the 0 in dual rail logic is denoted by an explicit signal, in can easily be inverted, e.g. by switching the signals out, as shown in figure 21. We did not implement dual rail logic due to the large overhead necessary to duplicate all signals and adapt all other parts of the computation accordingly.

The second possibility is to integrate the inversion into logic functions, e.g. by using a *NOR* instead of an *OR*, and using one secondary computation step, including a new signal, for the evaluation of all logic computations. Integrating the inversion into the function means that we always only need one additional signal to evaluate all "0-signals". The start of the computation is initiated by a new user input, and this resulting new impulse is used to evaluate the whole logic computation in our system. An "inverter" in our system is thus a logic

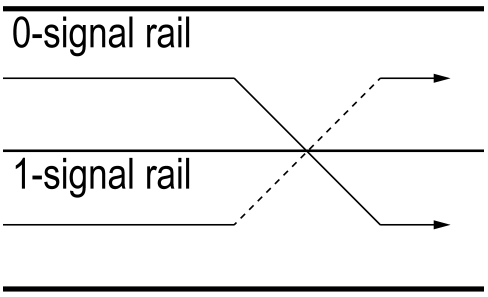


Figure 21: 0 and 1 signals switch places, thus inverting their connotation.

building block with two inputs instead of one, as shown in figure 22. This solution does not have a large overhead, as we always only need to add one additional input, no matter how complicated the function or the composition of functions is.

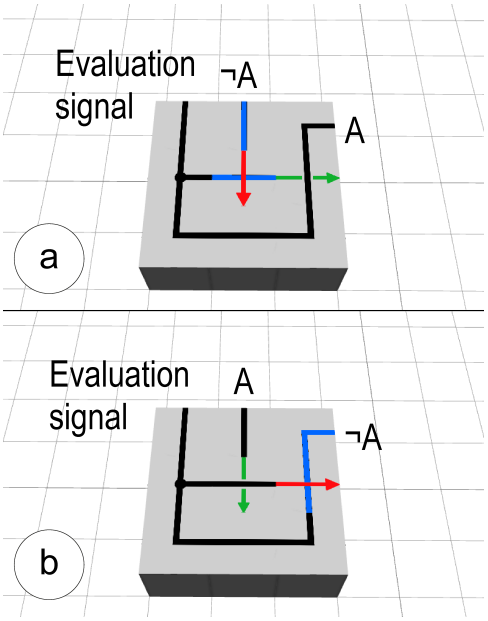


Figure 22: We created a setup that uses one additional input to create inverted signals, since the unary inverter is impossible to build. (a) The input A has not been set. The output of the computation is A. (b) The input A has been set, and the output of the computation is ¬A.

The gate cells used to build this logic have two inputs and perform three-state logic, in which the third state is defined as 'high impedance'. One of these inputs configures the cell, setting it to high impedance or to let the second signal pass unaltered. The three possible values are therefore 1 or 0, as given by the second input signal, or high impedance, as given by the first input signal. Since blocking is

handled within each spring cell as mentioned before (cf. 4.2), all cells within our system that store energy perform three-state logic cells.

4.1.2 *Clockless computation*

Our springs have to be recharged after using them, so we cannot make use of a clocking signal, as it would have to be activated in every clock cycle. We therefore cannot use a system clock as a method of synchronization in our system. This is however not necessary for simple calculations since we use a different property of our system to overcome this challenge; the persistence of state changes in our system.

The state of the system in an electric circuit is commonly lost after the electric current that powers the system is removed. Our system instead implements a static system, due to the fact that once a change has been made to the physical state of our system, it theoretically stays in that state forever, unless an external force introduces new changes. Since these physical changes are persistent, other parts of the system can "read" this information at any time. For example, once a blocker has been moved in front of a spring, it does not matter when the spring will be activated, the blocker will still be in the same position, blocking the signal. We utilize this fact and apply a simplified synchronization for our computations by assuming that all inputs have been set before functions that use these inputs are calculated. This assumption is valid due to the fact that computation in our system starts with a separate new user input, as explained in the previous section. Due to the high speed of the mechanical system compared to that of user input, we further assume, that new user input can only occur after the previous input has already been fully handled.

Race conditions *within* operations can be resolved through the length of signal lines in number of cells. Timing details on this level are explained in section 5.4.

4.2 TRAVERSING THE GRID

Realizing computation on a grid requires the ability not only to transmit a signal from one cell to another, but also to maneuver around the grid to be able to reach all cells, i.e. to change the direction of a signal. Based on geometric necessities, blocking of cell signals is handled within each cell. Redirection and duplication of signals however is managed between cells. "Empty" cells, that is cells that do not have a spring to store energy, create the necessary space for a previous cell

to access other neighbors than the one directly in front of the cell. This open space is then used to redirect and duplicate signals.

As shown in figure 23, we redirect a signal by 90° by adding a beam to the arms of our bistable spring. This beam rotates with the arm of the spring and transmits the impulse to the top right cell.

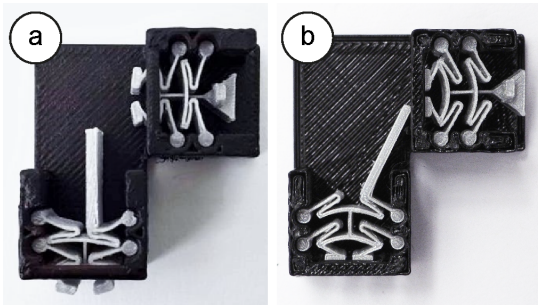


Figure 23: We use a new type of output port to redirect the signal by 90°. We exploit the rotational movement of the spring and attach a strut that hits its neighboring cell.

Figure 24 shows how we can route signal in 3 dimensions. By simply rotating the receiving cell, we can redirect signals from, e.g., the x/y plane to the x/z plane, as shown in figure 24a. With one more such redirection, we get to the y/z plane, as shown in figure 24b. This allows us to route signals in 3 dimensions within a 3D printed object.

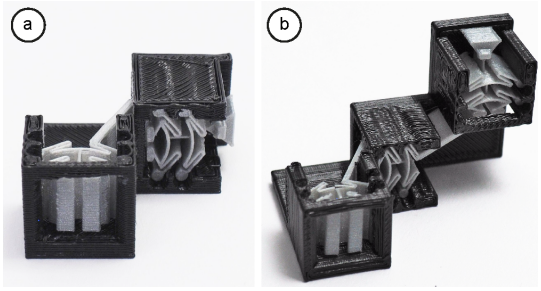


Figure 24: (a) To route signals from one plane to another, we redirect the signal by 90° and rotate the receiving cell. (b) Concatenating three redirecting assemblies allows us to route signals in 3D.

We can also route signals as to cross each other, as shown in figure 25. We attach a crossbar at the output port of the left cell that spans across the middle cell and actuates the right cell.

Figure 25: We cross signals by running a crossbar across another cell.

The previously explained three-state logic can terminate signals, but splitting/duplicating signals must also be possible to enable computation. To bifurcate signals, we exploit the fact that our bistable springs require less energy to be triggered than they actually output. Figure 26 shows two different cell types that bifurcate signals, for all of them we trigger two cells from one. Depending on the requirements for the signal route, we can (a) trigger two parallel signal lines while keeping the signal direction, or (b) trigger two signal lines in opposite directions.

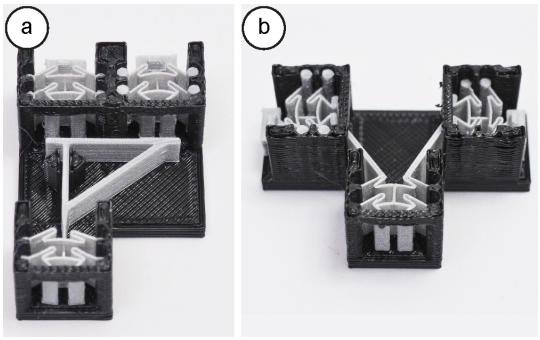


Figure 26: We can bifurcate signals (a) in a parallel manner or (b) let the two signal run in opposite directions.

Inverting the process of bifurcation, i.e. two cells transmitting a signal to the input side of one other cell, results in a physical implementation of an *OR* gate. This cell arrangement is shown in figure 27.

4.3 RECHARGING

The springs in our system are printed in their relaxed state. Fused deposition modeling 3D printers heat up thermoplastics during printing and the heat relaxes the material while it becomes ductile thus printable. Therefor the form it is printed in is always the relaxed form. This means they have to be charged before they can be used for sig-

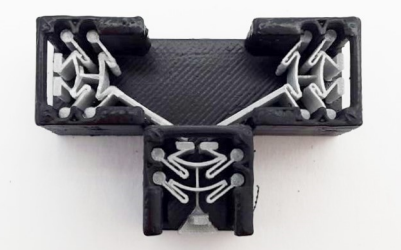


Figure 27: We use the opposite assembly to merge signal as we did to bifurcate them. This implements an OR gate.

nal transmission and calculation. They expend their energy storage during use, so they have to be recharged each time users want to use them again. Charging by hand would be tedious and most often impossible, once cells are hidden within the object. This work introduces a recharging mechanism that uses a rotation around one of the cell edges to transform a pushing motion from the top of the cell downwards into a pushing motion in the opposite direction of the signal direction. Every cell has its own recharger, which is rotated in com-

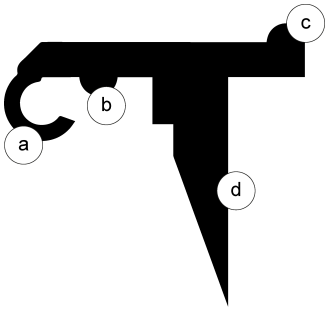


Figure 28: (a) The hook functions as a flexible bearing around an axis at a cell edge, allowing rotation. (b) A knob pushes the recharger up, out of signal line, while not in use. (c) Another knob focuses the pressure from above, creating a long lever for the rotation. (d) These "teeth" push the spring backward when rotated.

pliance to the rotation of the cell. The recharger requires a push from above the cell to work, regardless of the Z-rotation of the cell. That means that all cells on one plane (even multiple layers parallel to the plane) can be charged with one motion.

Two pushes are sufficient to build objects with outputs in all six possible directions of the cubic cells. To achieve this, all outputs in the X and Y directions can be fulfilled by cells which roofs point upwards. All other outputs, i.e. in the Z direction, can be acted out by cells the roofs of which point sideways, while all of those cells agree on one side.

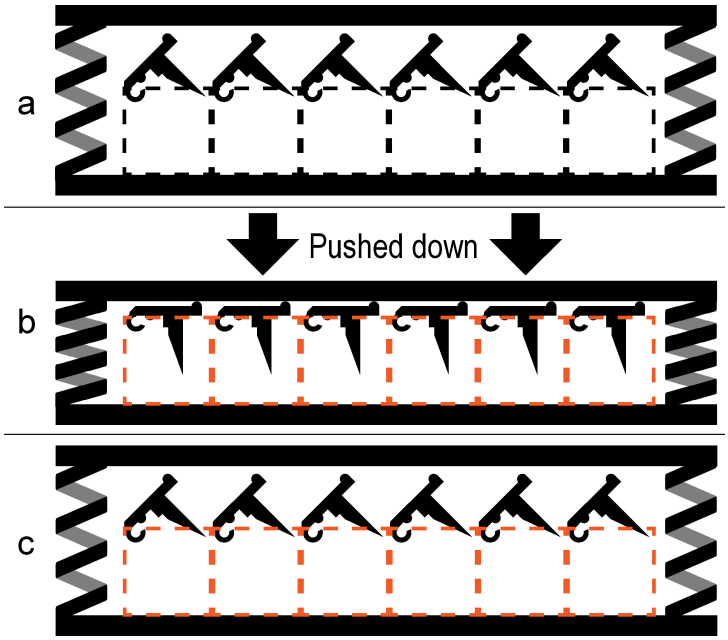


Figure 29: (a) All cells are in their relaxed state. (b) A push from above charges the cells. (d) The bottom knobs on the rechargers force them back into their resting position.

4.4 OPERATIONAL AMPLIFIERS

As will be explained in section 4.5.1, to enable signal transmission, the energy output of every cell always has to be at least slightly bigger than the required energy input E_i necessary to activate the following cell. Thus the energy output E_o is $E_o = E_i + \epsilon$ with $\epsilon > 0$. Depending on the size of ϵ, this fact can be used to have a cell activate a more powerful cell. This stronger cell can then in turn activate another even stronger cell, realizing cascading operational amplifiers. The current E_o increases in each step of this process, and the growth factor is limited by ϵ. Assuming the energy efficiency of cells is unchanged or greater when increasing cell size, the growth factor is at least constant, resulting in at least polynomial growth.

$$E_{o,n+1} = E_{o,n} * (1 + F(\epsilon))^n \tag{1}$$

Such escalating cascades can be utilized to achieve a very strong output signal while the input signal stays minuscule in comparison. Our system relies on operational amplifier cells that are eight times as big as the previous cell level (twice the length/width/height of the cubic cells). Bigger cells can also achieve a longer stroke, thus not only increasing the output energy but also directly the motion range of the output. Though it is possible to activate even such a significantly bigger cell with a single cell of previous size with out system,

this requires the input energy to be applied fairly centrally to the operational amplifier cell. To ensure an even distribution, though one of the smaller cells only covers a quarter of a side area of a bigger cell, we bifurcate the activation signal once right in front of the operational amplifier cell and thus cover two quarters of the operational amplifier cell input side, which is sufficient for a reliable activation.

4.5 PHYSICAL BACKGROUND OF MECHANICAL SIGNAL TRANSMISSION AND ENERGY STORAGE

Transmitting a signal through material requires energy. We argue that mechanical signal transmission and storage can be a serviceable alternative to electronics in the context of home fabrication, as all commonly used building materials in this context are non-conductive thermoplastics.

Similar to ohmic resistance in electronics, friction limits how far a mechanical signal fed from one energy source can travel and fan-out. We therefore utilize cells that store power themselves and can discharge it to trigger at least two other cells. Friction then only has to be resolved on a local level, as the supply of energy is replenished at every activation of a cell. This theoretically allows scaling the system infinitely. Triggering two adjacent cells per activation allows a quadratic chain reaction, also known as fan-out of two. Such a reaction can quickly activate the system as a whole if desired. User interaction can therefore be as simple as a single push of a button to activate the entire system.

4.5.1 *Springs as energy storage*

We use loaded bistable springs to store energy. A requirement for such a designs is, that the stored energy E_s is bigger than twice the energy necessary to trigger another spring cell E_t after the reduction through environmental effects such as friction μ has been applied to it.

$$E_s > (2 * E_t) * \mu \tag{2}$$

We explored a number of designs where a loaded spring was held in position by a latch. We found however that the increased complexity from having two moving parts per cell was not well suited for creating cells intended for scaling down well. Instead, we employed bistable springs as a design alternative. They combine the functional-

ity of the spring and the latch in one part, thus reducing the number of necessary moving parts.

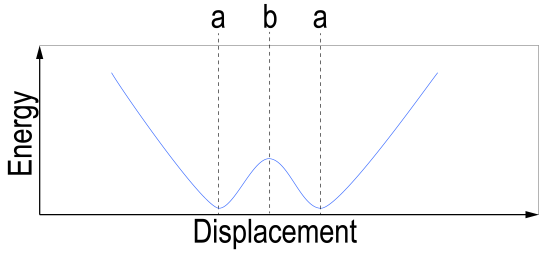


Figure 30: This symmetric bistable system has two minima (a) for potential energy, separated by a local maxima (b).

Bistable systems are characterized by having two minima for potential energy, separated by a local maximum (cf. figure 30). Once enough energy has been spent to pass the local maximum, the system transitions from one stable state to the other stable state. This can be visualized by imagining a credit card, which is being held and pushed from two opposing edges, until in buckles to one side or the other. If enough force is exerted onto the buckled surface of the card, the direction of buckling will change rapidly. This movement is rapid because once the local maximum of potential energy has been traversed, the card will convert this potential energy into kinetic energy and accelerate, until the second minima is reached. A clamped credit card is a symmetric system.

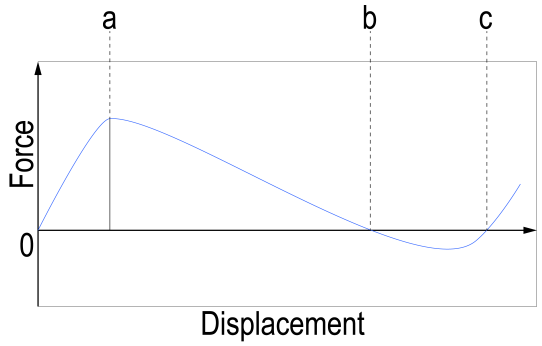


Figure 31: The force-displacement diagram illustrates the snapping behavior of the asymmetric bistable springs. When pushed from the left, the spring will snap after passing (b) to the position at (c). When pushed from the right, it will snap from (b) to the origin, while exerting a large output force.

For energy storage and rapid dispensing at a later time, an asymmetric system is more attractive. An asymmetric bistable system still has two minima of potential energy separated by a local maximum. However one of these states has a much higher potential energy well than the other, which has a value of zero in our case, as this is the state the spring was printed in. The force-displacement diagram in figure 31 illustrates this. For the purposes of this thesis, the state with low potential energy is called "relaxed state". This state is represented by the origin position of the diagram. Both displacement and reaction force of the spring are 0, since this is the position it was printed in. If the spring is pushed until the position b, it will snap to the c position by itself. This c position is the second stable state, with high potential energy, and is called "charged state". The area under the curve to the right ob the b position is the energy necessary to trigger the spring. Triggering it means that it will snap back to the origin of the diagram and exert output energy equivalent to the area under the curve to the left of the b position. It is beneficial to place cells and their bistable springs in a way that a currently triggering cell spring will hit the following charged spring in the b position of the snapping process. The force emitted by the spring is biggest at that point, maximizing the impulse that is transmitted to the next spring.

4.5.2 *Bistable spring design*

Simple bistable springs are designed as a simple buckled beam constrained by the surrounding walls. Such designs however usually have a very high width to length ratio, which does not utilize the space within a cubic cell well; hindering minimization. The ability of the material to withstand stress is another limiting factor for miniaturization. A stiff bistable spring introduces high stress into the mechanism when it is forced into its second stable state. The thickness of the spring is a crucial factor to control its stiffness. 3D printing only offers a limited minimum material thickness. The presented clamped spring design lowers the stiffness of the spring compared to a clamped buckled beam by increasing the length of the spring while better utilizing the available space within a cubic cell. It further offers a longer stroke, that is a longer total movement of the middle of the spring, than a simple buckled beam.

If a spring is deformed, an elastic force acts to return the spring to its relaxed form. A clamped spring becomes bistable if the clamping force exerted by the surrounding walls exceeds the elastic force of the spring at that displacement. The presented design was chosen to facilitate this circumstance by increasing compression forces against the walls and along the length of the members of the spring. Doing so lowers the extent of internal deformations for the spring to reach

its bistable state. If the deformations become to large, they become plastic instead of elastic deformations, thus permanently damaging the spring.

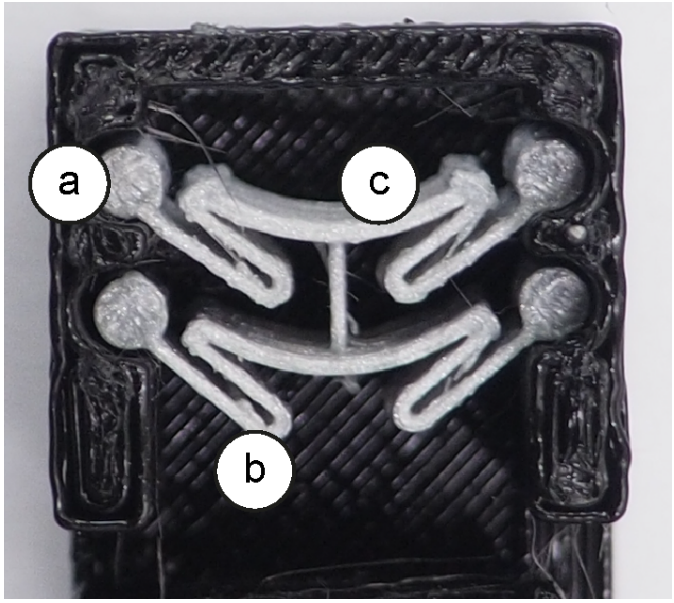


Figure 32: (a) The bistable mechanism is mounted in a bearing to allow large rotations. (b) Angles between beams are minimized to avoid energy loss due to bending when converting compression/tensile forces. (c) Pre-bent beam increases compression/tensile forces further by increasing its width when bent.

The attachment of the spring to the outer walls of the cell, shown in figure 32a, has to permit large rotational motions while avoiding large deformations of the material. This can be achieved using a bearing mount, which allows rotation without bending, unlike a living hinge.

The secondary beams holding the pre-bent beam in the middle are angled at 180° to each other (cf. 32b) to facilitate transmission of compression forces to the walls. When the mechanism is pushed in, the beams connected to the bearing cylinders rotate around the bearing axles and their tips converge. The middle part of the mechanism tries to resist the necessary deformations that allow the tips to converge, thus pushing the outer beams holding it outwards against the walls.

The distance r between the two beams is given by $g + l * sin(\alpha)$. The resulting torque is:

$$Mr = F * r * sin(\alpha) \tag{3}$$

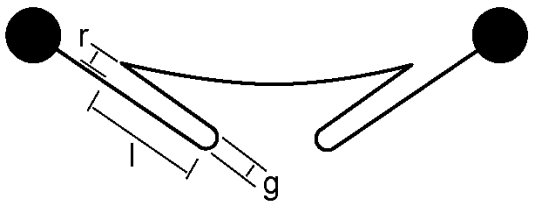


Figure 33: The distance r between the two beams is minimized if the angle between them is 180°, i.e. if they run parallel, and if the gap g is zero.

And the translational moment is defined by:

$$Mt = F * r * cos(\alpha) \tag{4}$$

A high torque would promote bending of the beams. A large translational moment however compresses the beam along its length, which does not result in large deformations. Choosing a 180° angle and the smallest possible gap between the two beams minimizes the resulting torque, thus converting compression/tensional forces most efficiently.

Similar to the simple buckled beams commonly used for bistable mechanisms from, the pre-bent beam, called bridge, in the middle of the mechanism in figure 32c bends in a direction opposite to the influencing force. This effectively straightens the bridge when pushed through to the second stable state of the spring. Since the bridge is already printed pre-bent, this widens the beam. The additional width increases compression within the bridge and tension in the connected beams accordingly. Choosing an inverse buckling curvature with an amplitude identical to the one that would be reached in the second stable state of the mechanism maximizes the increase in compression force. Additionally, increasing the thickness of the middle bridge impedes bending on this part of the mechanism, hence forcing the other beams to bend more. This can be utilized to substitute large (plastic) deformations in this part of the mechanism with smaller (elastic) deformations throughout the mechanism.

4.5.3 *Spring parameterization*

Depending on the use case, different qualities of bistable springs can be desirable. Sometimes it might be necessary to employ a spring that has a very long stroke, i.e. a very long distance between its two stable states, for example to move output actuators over a longer distance. The parameters of our spring design can be altered to achieve different qualities without losing its bistability. We defined three parameters: the arm angle, the length of the bent bridge in the middle, and the strength of the bridge, varied trough changing its buckling

magnitude and its thickness concurrently. Figure 34 visualizes these parameters. We do not alter the cell size and we do not change parameters that are fixed due to environmental circumstances such as resolution of the used printer. This means that these parameters can be changed without the need to make changes to the overall setup, such as using a new printer.

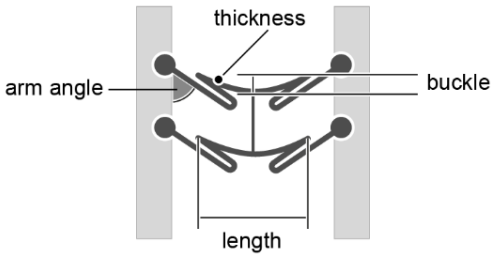


Figure 34: All of these parameters affect the spring constant, but they have varying impact on stroke length and output energy of the spring.

We tested the effects of these parameters empirically and found that the output energy of the spring is affected most by the strength of the bridge and least by the arm angle. In turn, the stroke length is affected most by the arm angle and least by the strength of the bridge. The parameters can also be varied, for example, to increases the fault-tolerance of the system with regards to unwanted activation, e.g., by dropping the object. This can be done by increasing any of the parameters slightly, since all of them increase the spring constant, thus stiffening the system.

5 SOFTWARE

We extended an interactive editor for the creation of metamaterial mechanisms, which was first introduced by [7]. The previous editor version allows placing voxels on the 3D grid, simulates deformations of metamaterials, and employs custom shader code for better rendering performance. The enhanced version features the combination of analog mechanisms and digital computation within the same environment. We added functionality that allows drawing logic by hand, as well as automated synthesis of logic cells. It supports the user in the creation and layout of functional mechanical circuitry through simulation and visualization of the computations. It furthermore provides features that simplify connecting the digital and analogue mechanisms through pathfinding and automated alignment and crossing of signal transmission cells.

The editor is based on a node.js[1] javascript framework and uses the three.js[2] graphics framework for rendering basic geometries. Custom WebGL 2[3] shaders are used to render all voxels quickly directly on the graphics processing unit (GPU).

After depicting the overall architecture of the software system, this chapter will first explain the tools that the editor offers in more detail than before, and then expose implementation specifics of different parts of the software.

5.1 ARCHITECTURE

The software system is composed of a voxel based editor and three separate external components, which handle minimization of logic functions, 3D rendering after the export, and model deformation simulation through finite element analysis. Figure 35 shows the network setup. The finite element analysis works as described in [7], by employing the finite element solver karamba[4], which is a plugin for Rhinoceros/Grasshopper. The functionality of the Python server responsible for logic minimization will be discussed in section 5.3. 3D rendering in OpenSCAD for the export function is explained in section 5.3.4.

1 https://nodejs.org/en/docs/es6/
2 https://github.com/mrdoob/three.js/
3 https://www.khronos.org/webgl/
4 http://www.karamba3d.com/

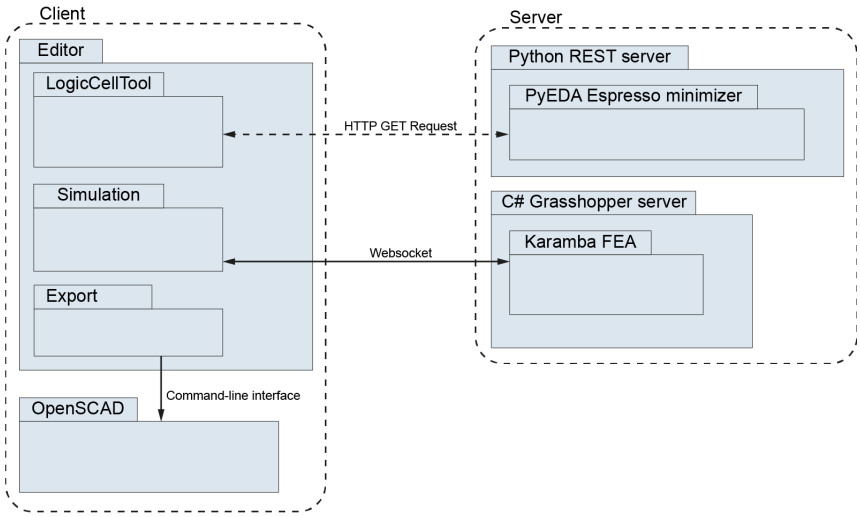


Figure 35: The system is intended to run on two PC's. PC1 handles user interaction, while PC2 works as a server to avoid delays when using the system.

The structure diagram in figure 36 shows the main components of the editor, while omitting details to focus on the parts storing and rendering logic cells. A voxel in our system is a visible entity on the grid used to construct metamaterial models. A logic cell on the other hand is merely a data construct. When built, every logic cell creates a voxel specific to the details of the cell as a visual representation for the user of the editor.

The geometry buffer prepares the voxel geometries for rendering directly on the GPU using a custom shader defined by the buffer. Rendering details are explained in section 5.7.

5.2 MANUAL SIGNAL AND LOGIC DESIGN

We implemented a set of specialized tools to help the users of our editor in the creation of digital mechanical metamaterials. A common task for doing so is connecting existing parts of models with each other or in- and outputs of the object with the computation parts. Depending on how complex the model already is, this might require drawing long and constricted signal paths. The following drawing tools simplify the task of drawing continuous signal lines.

5.2.1 *Drawing*

The basic *add* mode of the editor allows the user to add single voxels, lines and even volumes of voxels quickly. Since all of the created

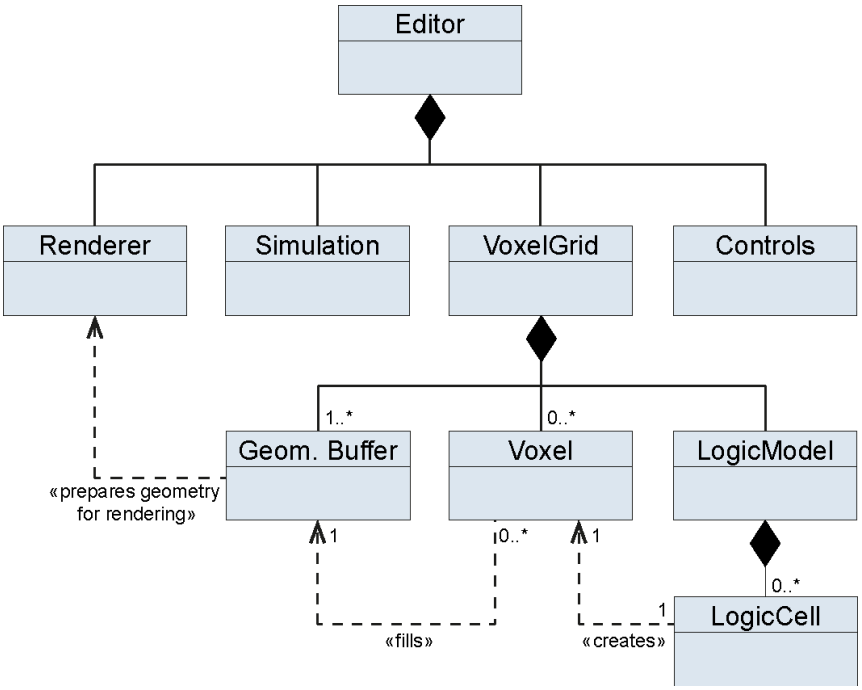


Figure 36: Overview of the internal structure of our editor.

voxels are identical, this can be used to draw straight signal lines in one motion. For signal lines that are not straight, we provide other drawing tools instead.

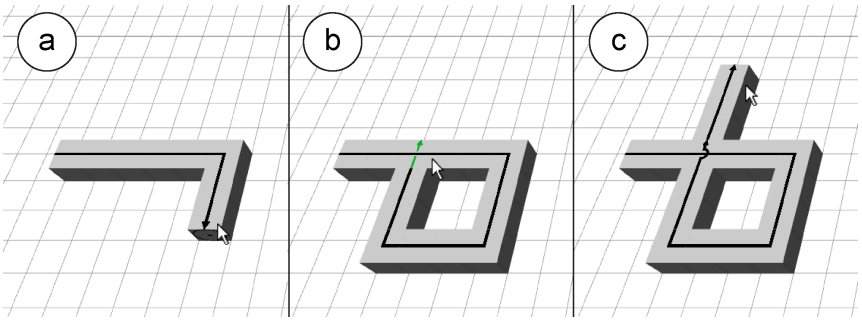


Figure 37: The *signals* mode offers a freeform drawing tool that creates a signal line along the path of the mouse cursor.

The editor offers the *signals* mode, which places logic cells according to the movements of the mouse pointer, as long as the left mouse button is pushed (cf. figure 37a). It therefor functions as a freeform line tool, and the cells that were placed along the line transmit a signal from the start point of the mouse click until the point where the user stopped pushing the button. The tool also adjusts existing cells, when the drawn line intersects existing signal lines. If the user stops

drawing at the intersection of the lines, this is interpreted as the new line blocking/unblocking the existing signal, thus creating a gate cell at the intersection, as shown in figure 37b. Figure 37c shows the user continuing to draw beyond the intersection of the two signal lines. This interaction is instead interpreted as signal crossing, and the gate cell is automatically replaced by a crossing cell. This version of the signals tool however only works in 2D, on one XY-layer of the grid.

The *signals* mode of the editor offers an alternative way to draw signal lines that also works in 3D. This alternate way of drawing is activated by holding the *CTRL* button while selecting voxels. It connects two cells via the shortest path automatically while making sure that the signal line is not interrupted along the way, e.g. that all transmission cells have the correct rotation. An example can be seen in figure 38. This mode makes use of a pathfinding algorithm. The A* pathfinder we employed is part of a 3D version of the PathFinding.js library[5] and was modified to allow using cells multiple times for perpendicular paths. If beneficial, the pathfinder will cross existing lines. Cells that cannot be used for transmitting another signal, such as cells that already implement a signal crossing, are simply bypassed completely.

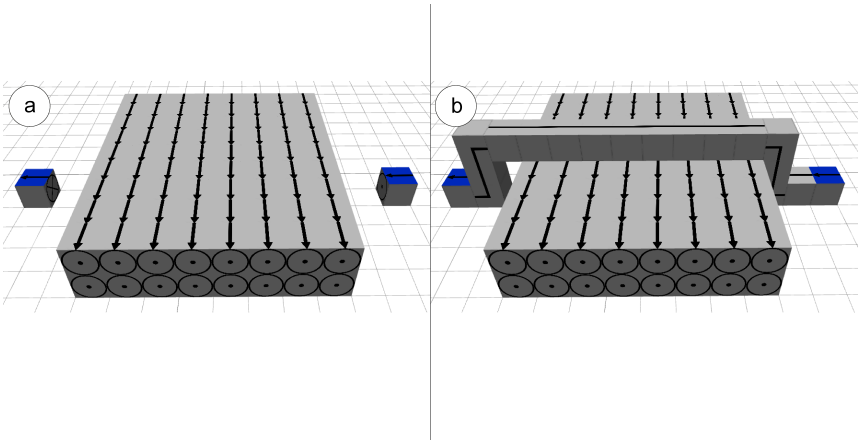


Figure 38: (a) Users select the two blue cells, which are then (b) automatically connected via the *signals* tool.

5.2.2 *Furcated signal paths*

Instead of connecting a single start and a single end cell, it is often desirable to connect one signal line to a set of output cells in a $1 : n$ link. To achieve such behavior, the signal has to be forked $n - 1$ times along

5 https://github.com/schteppe/PathFinding3D.js

the way (cf. figure 39). Similarly, $n : 1$ connections, where multiple input cells can activate one output are sometimes desired. Assuming an implicit *OR*, this can be achieved by having two or more cells activate the same input side of one cell, functioning as an inverse bifurcation. Other ways to interpret $n : 1$ connections, such as an implicit *AND*, or a voting mechanism, are instead implemented through explicit boolean computation. In this case, the tools that assist the user by synthesizing logic should be used.

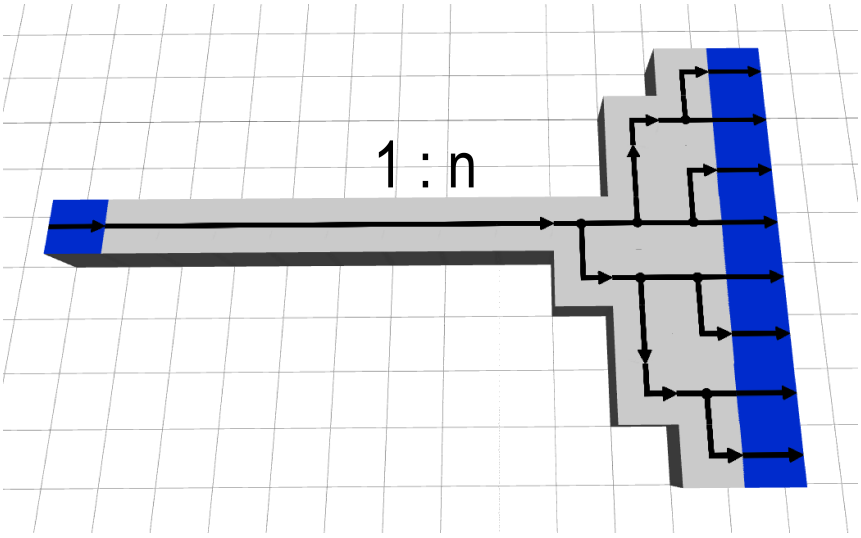


Figure 39: Users want to connect the blue cell on the left to all blue cells on the right, so that the right ones are all activated at nearly the same time. It requires less cells to do so when furcations happen close to the larger set of cells, i.e. close to the right end of the signal paths.

To facilitate such furcations, we want to improve the pathfinder of the *signals* mode of the system to be able to create furcations that minimize the number of necessary cells automatically. A contrived algorithm to do so is depicted in section 6.1.

5.2.3 *Advanced brushes for creating logic: Circuit blocks*

Expert users can create logic functions by placing the proper cells by hand, but this process is slow and thus not well suited for complex logic. Our system supports users to design logic quickly. To achieve this, it offers a range of options to input logical functions, i.e. in the form of logic functions, as truth tables on cell level, and as logic gateways.

Users with a background in digital circuits might prefer designing logic by placing basic logic building blocks such as logic gateways or multiplexers, similar to the workflow in a boolean logic editor, such as LogicFriday[6]. Our editor offers this mode of operation by providing special brushes, that when used with the *add* mode of the editor, place predefined groups of logic cells at once. These specialized cell arrangements perform the desired boolean logic using the rod-logic computation paradigm. The editor already offers brushes for each of the four basic two-input gates and a 4-to-1 multiplexer (cf. figure 40a). Figure 41 shows the signal flow for the 2 input *AND* gate.

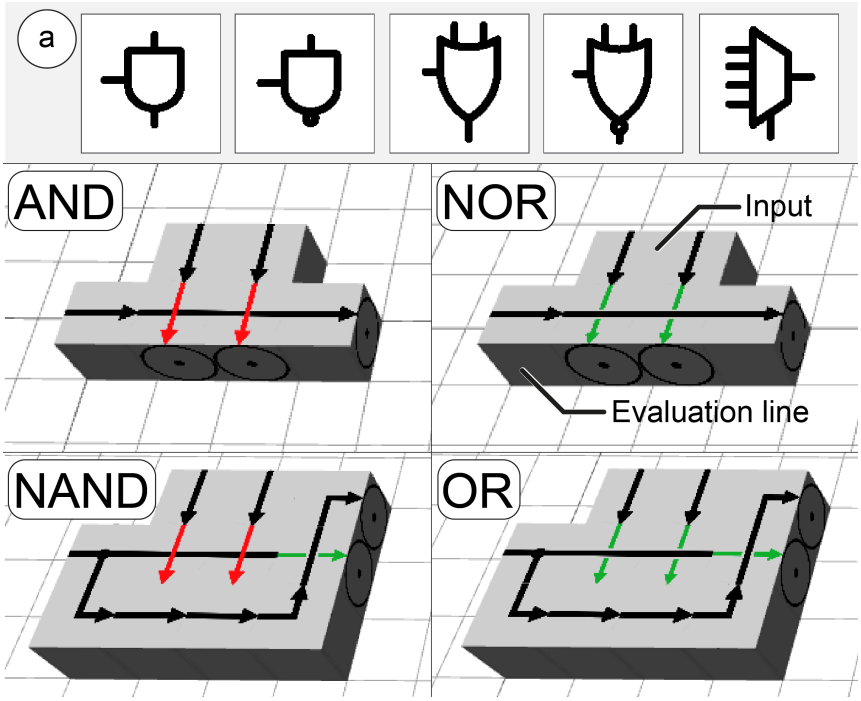


Figure 40: Each of these gates has two inputs coming from the top and an evaluation line coming from the left. The output of the computation is located at the (top) right.

The NOR and the AND gates share great similarity, just like the OR and NAND gates. The only difference in the similar arrangements is that both inputs either block or unblock a signal line. OR and NAND gate are more complex, as this cell arrangement performs two computational steps: First the result of NOR or AND respectively is calculated, and this result is then negated. Why these exact cell arrangements were used to build the basic logic gates will be clarified in section 5.3.1.

6 http://sontrak.com/

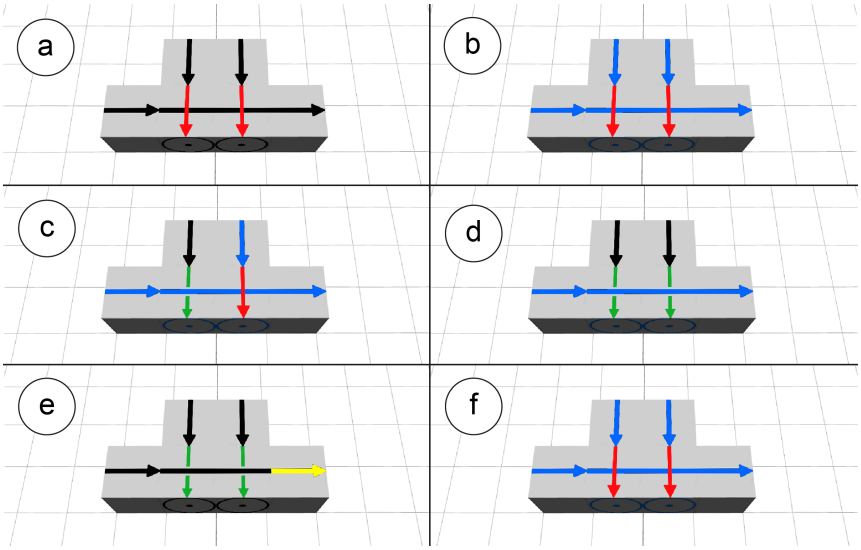


Figure 41: (a) Users can place this *AND* gate directly. (b) Switching to the *compute* mode charges all cells. (c) Users activate the first input. The second gate cell still blocks the evaluation signal. (d) Users activate the second input. (e) Users trigger the evaluation line, which reaches the end, since both inputs of the *AND* gate were activated. (f) Charging the cells also resets the state of the gate cells.

5.2.4 *Undo/Redo*

A single operation within a voxel editor can fill the entire grid space with voxels, replacing all previously set voxels. Such an operation would therefore effectively destroy any existing model. This makes the undo functionality of such an editor a very important feature, besides the gain in efficiency that is to be expected. Undo and redo are implemented on a cell level, but they operate on the more abstract user interaction level. This is achieved through implicit grouping of cell changes, based on a timestamp added to every operation at the time of execution. Whenever an undo/redo operation occurs, the system checks whether the following stack item has been created within a marginally small time frame around the previous one. The length of this time frame has been chosen shorter than the time that passes between a mouse double-click at 100ms. This way, operations occuring from two separate user interactions will not be grouped together. All batch processes however, e.g. when a group of cells is created for synthesizing a logic function, are treated as one operation. This process works, since the processing time necessary to create, delete or edit cells is near constant and well below 100ms. Figure 42 shows the message sequences of adding a cell and the undo operation involving the undo/redo stacks.

The implemented version of this functionality is very lightweight, as only the previous state of the actually altered data objects are

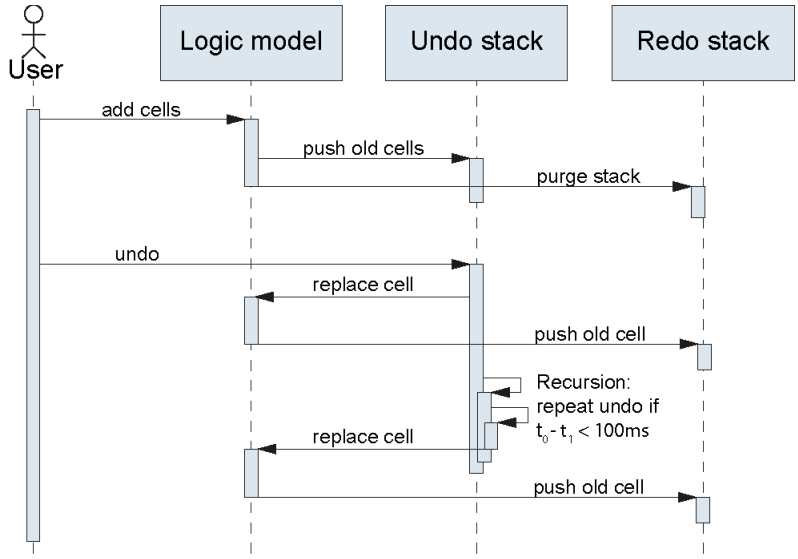


Figure 42: Message sequence chart displaying undo and one previous operation.

stored per step, instead of for example the whole system state. In addition, actual logic cell data is only stored when cells are changed or deleted, the system otherwise only adds empty cell data to the stack. Since the redo stack is purged during user operations, and a stored object is always either in the undo or the redo stack, the combined size of undo and redo stack has an upper bound. The bound is defined by the higher number of currently existing cell objects, either before or after the last user interaction. This means that in the worst case, the undo/redo functionality doubles the number of cell objects that have to be stored in total.

5.3 SYNTHESIZING LOGIC

An even more advanced way to generate cells that implement logic is available if the desired logic function is known. The user can enter this function, which is then minimized internally and a cell pattern that executes this specific logic function is synthesized automatically. To do so, the function is first minimized by the *Berkeley Espresso Minimizer*[7]. The resulting function is then parsed within the editor and split up into parts that can be directly converted to compositions of our logic cells.

As logic minimization is an extremely complex computational problem, we decided not to execute it in our javascript environment. Instead, we employ a very high-performance C implementation, pro-

7 https://embedded.eecs.berkeley.edu/pubs/downloads/espresso/index.htm

vided by PyEDA[8], a Python library for electric design automation. In our system, PyEDA is running on a RESTful Python server. That means that it only provides stateless operations, which are accessible in their entirety over a descriptive URI. In actual use, our editor encodes the logic function that should be minimized as a parameter in a URI for the Python server and then sends this request via the HTTP GET protocol to the server. PyEDA then parses the received logic function internally. This means, that as long as the Python server is running, the user can input specific functions, supported by PyEDA, that have a more succinct representation than the general form, including for example the *implication* function. The parsed function is then minimized using the *Berkeley Espresso Minimizer* and the JSON encoded result is returned to the editor to answer the initial GET request. The *Berkeley Espresso Minimizer* uses heuristic problems to find near optimal solutions to the NP-complete problem of logic minimization. It devides the input variables into subsets and finds locally optimal solutions, which are then merged to a globally near optimal solution.

If the logic minimization server is currently not available, i.e. when the HTTP GET request timed out or returned an error state, the editor instead continues working with the original function without minimizing it. Either way, the function is then parsed to identify the literals (i.e. the variables of the function) and is split up into terms. Using these terms and literals, the cell arrangements that fulfill the desired logic function can be generated. The understand the specifics of these cell arrangements, we first have to explain how combinational logic using rod logic concepts on a cell grid works.

5.3.1 *Cell based computation using rod logic concepts*

The basics of rod logic are described in the related work chapter in section 3.5. Calculation within rod logic happens between interlocking rods, which are either free to move or have their movement blocked by another rod. We propose a system in which a rod is replaced by a line of cells. Signal paths can cross, and the intersections can be outfitted with probes and blocking knobs. The end of one such line of cells is therefore an *AND* array of all the intersections along its path. Only if all intersections are unblocked, the signal will reach the end of the line. *OR* functionality can be achieved by forking a signal and having the two lines run in parallel. The final output can then be activated if any of the duplicated signals reaches its destination. Combining these two methods can implement more sophisticated functionality, such as voters or a XOR-gate with three inputs. In fact, all combinational logic functions (or their negated forms) can

8 https://pyeda.readthedocs.io/en/latest/index.html

be realized in such a fashion in one computation step. Furthermore, multiple computation steps can be chained, in which the previously computed outs can be used as inputs for the following steps.

Since *OR* functionality can be run in parallel and *AND* functionality most easily runs along a line, a disjunctive normal form (DNF) can be arranged in a very compact pattern. An array of disjunctions is run in parallel, while all inputs intersect these lines perpendicular to them. In other words, the editor uses the terms of the function to construct a compact disjunction of minterm conjunctions. A minterm is a minimal conjunction of the input literals that returns true.

This arrangement can be seen in figure 43. Depending on the configuration of the gate cells at the signal intersections, all functions of the form $F = (\pm A \ \& \ \pm B \ \& \ \pm C) \mid (\pm A \ \& \ \pm B \ \& \ \pm C)$ can be calculated with this arrangement of cells.

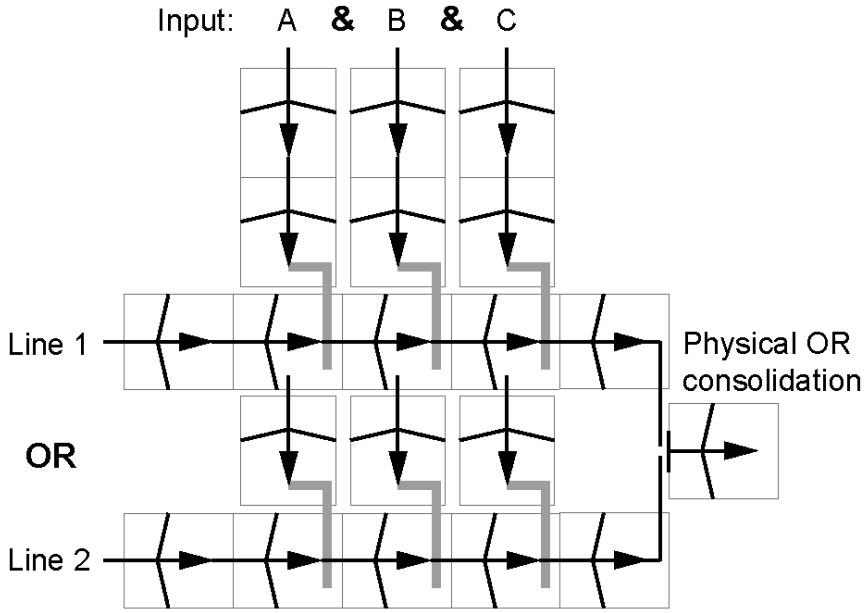


Figure 43: All of the horizontal input lines have to have the right states to block or unblock each of the activation lines

Since the *OR* lines should run in parallel, they are activated by the same signal which is bifurcated along the path, as shown in figure 44a. Instead of a physical consolidation of the *OR* signals that run in parallel, it is also possible to consolidate them using cells with a specific arrangement, similar to that of the original computation. To do so, the signal that activated the *OR* evaluation lines is used again to evaluate all of the parallel *OR* lines. The signal is furcated along the path once more (cf. 44b) in a way that this furcated path would block the original path later (cf. 44c). This blocking signal can

however be blocked and thus stopped by either of the *OR* signal lines. This method scales gracefully with large numbers of *OR* lines that run in parallel, as it adds a fixed number of cells to the total length and width in cells of the computation area. The physical *OR* consolidation would instead add $\log_2 n$ cells with $n = number\ of\ OR\ lines$ to the width of the area.

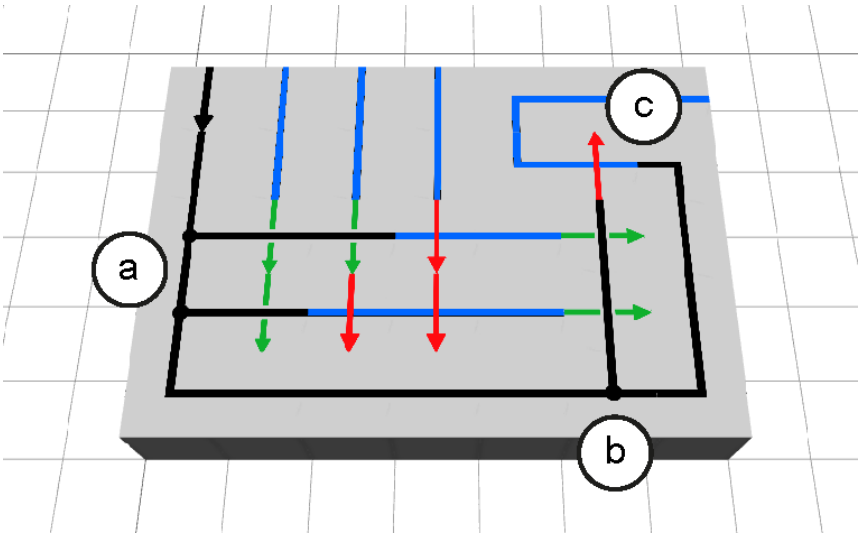


Figure 44: A logical *OR* can replace the physical version. It reuses the evaluation line, that was originally used to (a) trigger the parallel *OR* lines themselves. (b) It is forked a second time, and as a result might (c) block its own signal path depending on the result of the logic function.

Figure 44 displays the function $F0 = (\neg A \,\&\, \neg B \,\&\, C) \mid (\neg A \,\&\, B \,\&\, C)$. In it, the evaluation line was triggered, but it blocked itself, since no input was activated, which does not fulfill the function. Thus, the output is 0. It is notable that in this version, there are no perpendicular cells in between the two *OR* lines. The editor currently assumes, that the rods used for blocking/unblocking the signals can span any number of consecutive cells, which only holds true in theory. In practice, at some point, the blocking cell that moves the blocking rod will not be able to overcome the friction caused by the mounds around the very long blocking rod anymore.

5.3.2 *Form and shape of logic cell arrangements*

The form a logic function is presented in can greatly affects its perceived complexity. As our logic synthesis algorithm is based on the terms of the function, this can also affect the complexity of the cell arrangements that are created. We employ a second custom optimiza-

tion step to avoid using unfavorable representations. Consider the function based on pairs of inputs, which returns true if at least one of the inputs of each pair of inputs is set to true (also known as the Achilles Heel function). The DNF of this function results in figure 45a and the notation in logic terms would be equally large.

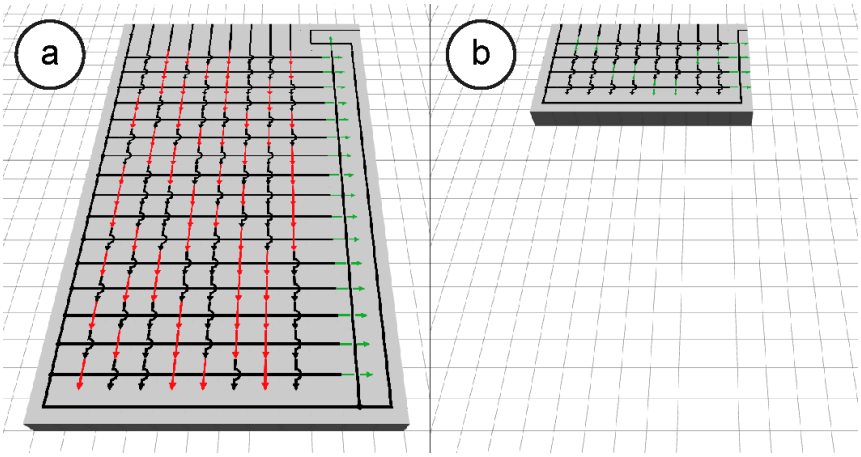


Figure 45: A logical *OR* can replace the physical version. It reuses the evaluation line, that was originally used to trigger the parallel *OR* lines themselves.

Trivially, $DNF = \neg(\neg DNF)$ holds true. We make use of this fact by negating the DNF before minimization, which results in the conjunctive normal form (KNF) of the function. The minimizer is set to bring the input function back into the DNF, and we negate the minimized result again afterwards within the cell arrangement. The final cell pattern can be seen in figure 45b. Note that the end of evaluation line in figure 45b does not have a bifurcation step that could result in self-blocking. Removing this part is effectively negating the result of the computation. Also note the cells within the computation that do not block or unblock a signal. These cells just cross the perpendicular signal lines, as these inputs should have no effect on these minterms of the function, since this input literal is not part of that minterm.

To utilize this method of negating the function twice, we send both versions to the minimizer and compare afterwards which method produces the result that requires less cells to implement. Since the HTTP GET request to the python server that handles minimization is an asynchronous call, we use the callback of the request for synchronization. In detail, the second request to the minimizer is sent within the callback of the first one if it returned a valid result. This way we can avoid any network synchronization issues.

The length and width of a computation pattern might still be too big for the current use case even after all the available minimization steps have been applied. For simple prototypes, such as those that have been created for this thesis, this has not been the case though. A function to handle this potential issue thus has not been implemented into the editor yet. We plan to add the functionality to change the shape of the computation cell arrangement while keeping the internal logic intact in following versions of the editor. To do so, the layer of cells that was created as a 2D surface could be folded up like a sheet of paper by 180° along any of its edges. The total number of cells however would increase, since additional cells would have to be placed at the edges to redirect the signal along the third dimension. The process could then be repeated until the desired dimensions are met.

5.3.3 *Logic synthesis using truth tables*

In case the actual logic function is unknown, but the user is aware which cells should be activated depending on which input cells are active (in other words, the logic truth table is known), the system can synthesize the function through this knowledge as well.

To make use of this method, the user selects all input and output cells and their desired states, i.e. whether they have to be triggered or not triggered for the function to return true. After inputting all necessary combinations, the logic cell pattern implementing the truth table can by minimized and synthesized automatically as before (cf. figure 46).

5.3.4 *Generating geometry for 3D printing*

Our editor uses the OpenSCAD scripting language for rendering the finished prototypes to export a 3D printable STL file. OpenSCAD[9] is a script based modeling tool for the creation of solit 3D models. It is an easy to use open source tool and thus offers a broad range of community created extensions. We used for example an extension for creating bezier-curves[10] within the program. Since it can be run from the command line and the input files are text based, integration into an existing workflow is easy. The script can be started through our editor automatically, outputting the final STL file directly.

Though the programming capabilities of OpenSCAD are limited, it is still a great tool for the creation of objects with a large number of

9 http://www.openscad.org/
10 https://github.com/chadkirby/BezierScad/

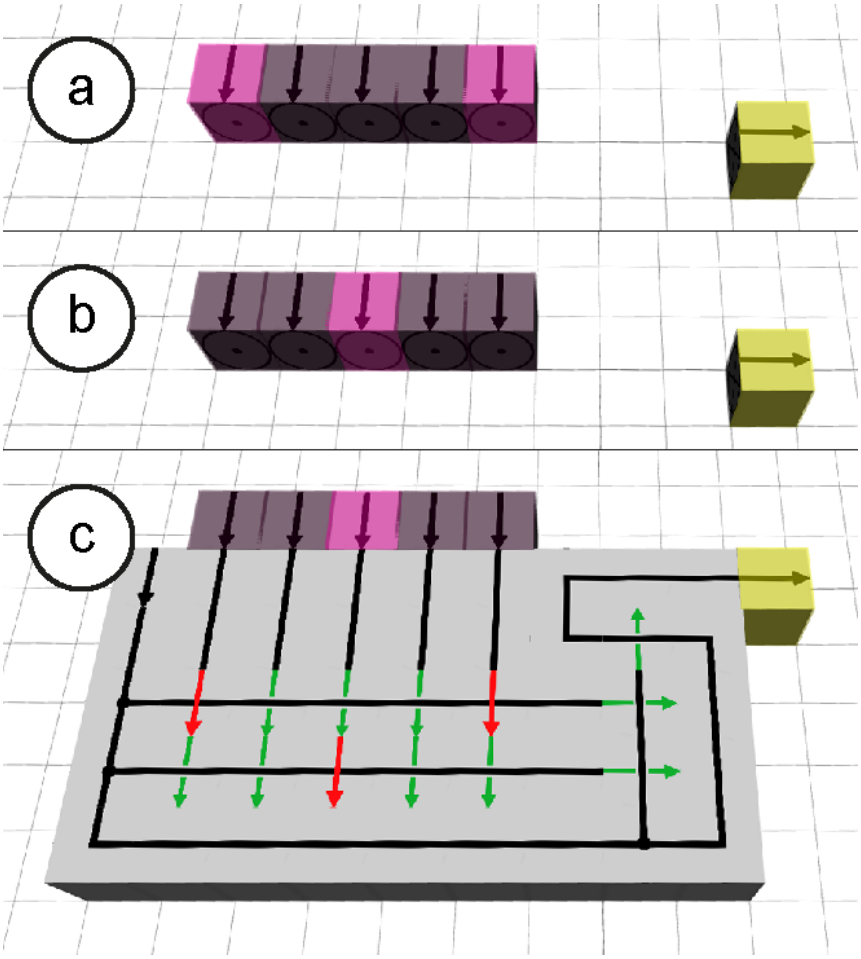


Figure 46: (a) The violet and pink cells have been marked as input cells. To fulfill the function, the pink cells have to be triggered, but the violet ones not. The yellow cell is an output cell. The user has selected the first minterm of the function here. (b) The user selects the second minterm by marking the appropriate cells. (c) The corresponding logic has been generated.

parameters describing them. Our design of a bistable spring is fully parametrized. That way all parts of the spring adapt automatically to changes of any other part. For example changing the thickness of the spring moves all the spring members, to both make sure that they still fit together as expected and also that the distance between them stays the same, so that the parts do not stick together when they are 3D printed (cf. figure 47). With this setup, testing parameters is not a design challenge, but merely a printing task.

The internal export process for objects created with our editor is shown in figure 48. The cell configuration details are exported to a text file containing the position and other necessary information (e.g. rotation, cell type, outputs, scaling, ...) to build the metamaterial cells with integrated logic. The specifics how the cells themselves are designed are contained in separate OpenSCAD modules and a config-

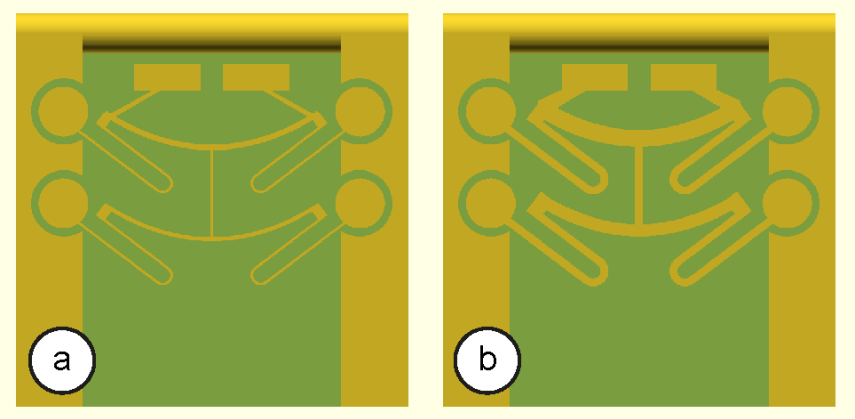


Figure 47: (a) shows a spring with a minimum material thickness of 0,1mm and (b) was rendered with a minimum material thickness of 0,3mm set. The script adapts the geometry automatically to avoid overlapping parts or unwanted translations.

uration file. The configuration file contains different sets of configurations, which can be chosen and applied on the fly. This allows for example to have cells of different sizes (four grid voxels large instead of one) or those that have a particularly long stroke or a high spring constant.

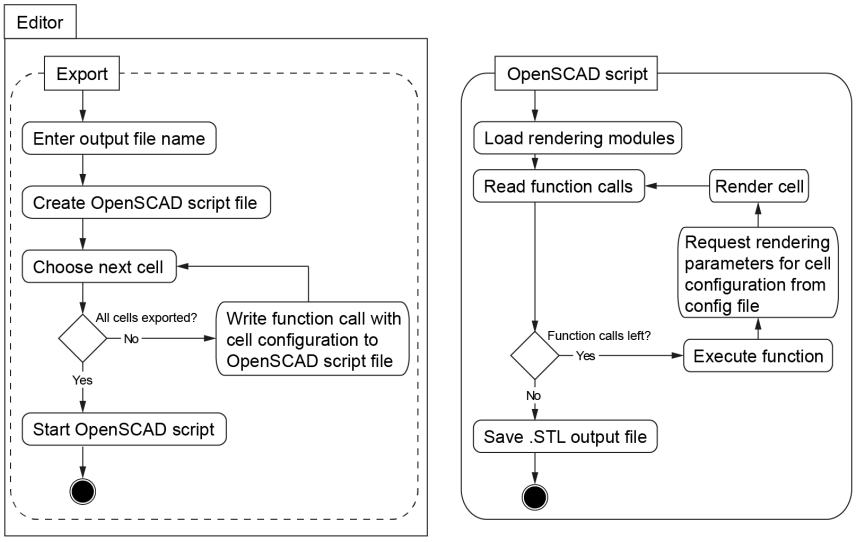


Figure 48: To export STL files, the editor handles the user interaction, i.e. request the output file name and path from the user in a standard Windows *FileSaveDialog*, then prepares and executes the rendering script.

The configuration file fulfills a peculiar role in the rendering process. Unlike most OpenSCAD configuration files, it does not simply store a set of variables that can be used from other files. It instead

stores multiple sets of variables for different configurations. It offers functions instead of variables to provide the stored data. Files depending on the data don't access variables directly, but instead call a function which takes the requested configuration as a parameter. The function then supplies the appropriate data value. It therefore acts as a static server.

5.4 SIMULATING LOGIC CIRCUITS REGARDING TIMING ASSUMPTIONS

The objects created with our system are designed to be able to compute combinational logic, which is time independent. Verifying this kind of logic by itself could easily be executed by a global software component that computes the logic and displays the result. There are however a number of reasons why such a solution is not a viable option.

- It is for example possible to build logic cell arrangements, where the results of some parts of the computation change how the computation of other parts work. There, not only which input was set but also the order in which they are activated influences the outcome of the computation. Such case-by-case analysis would complicate the functionality of, and user interaction with the simulation component.
- A desirable functionality of the system is that the user can observe the actions within the material, instead of just receiving the final output, especially when parallel execution is involved. Figure 49 shows a simulation of signals traversing the material in parallel. Also interacting with the system during simulation might be favored for testing purposes.
- Another reason is, that the time it takes to trigger a cell and snap its bistable spring varies slightly from cell to cell (due to fabrication inaccuracies) and it can also vary depending on other factors, such as cell temperature or possible material fatigue over time. For two mirrored signal lines that run in parallel and have the same length it is still mathematically impossible that they arrive at exactly the same time at the same destination. Since computation in our system is based on impulses, a synchronicity assumption should not be made. Following this reasoning, computation should be cell based, as it also is in the real world objects.

One way to achieve this would be to emulate the cells as a cellular automaton. A cellular automaton is however a clocked system, which does not fit our paradigm well. Furthermore, as in a cellular automa-

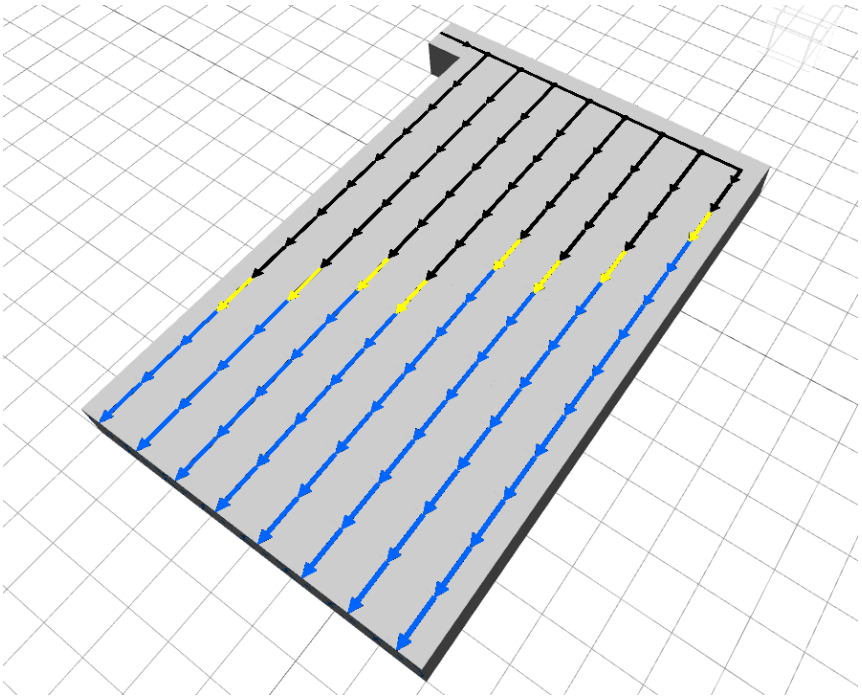


Figure 49: The cells shining yellow are currently active in this simulation. Multiple bifurcations have triggered parallel signal lines. Simulating parallel execution helps identifying and testing race conditions in prototypes.

ton all cells are evaluated simultaneously in every step, in almost all cases this evaluation would have no result, since our cells only fire once, and are silent during the rest of their lifetime. Evaluating all cells at once is unnecessary since signals only cause local changes and it does not scale well.

An option that scales better would be to utilize a FIFO queue of cells that are to be triggered. However, the order in which cells are triggered would be implementation dependent and fixed. When two mirror signal lines running in parallel are simulated, the one whose first cell was added to the queue first would always also arrive at the end first. This ordering is demonstrated in figure 50.

We implemented a solution similar to that of the FIFO queue. A queue is implicitly given through the order of function calls. Every cell stores a delay inherent to that cell, describing how long this cell is active before it triggers following cells. This delay varies per cell and also per activation of the cell. It can vary up to twice the original delay. During the delay, other cells can be triggered and computed by the system. In addition, the delay also helps the user to follow the process since a computation without delay would be too fast to understand or even see. Regarding the setup in figure 50, both signal lines could arrive first, as would be the case in the physical world. The cells are functionally triggering each other, though they are not directly com-

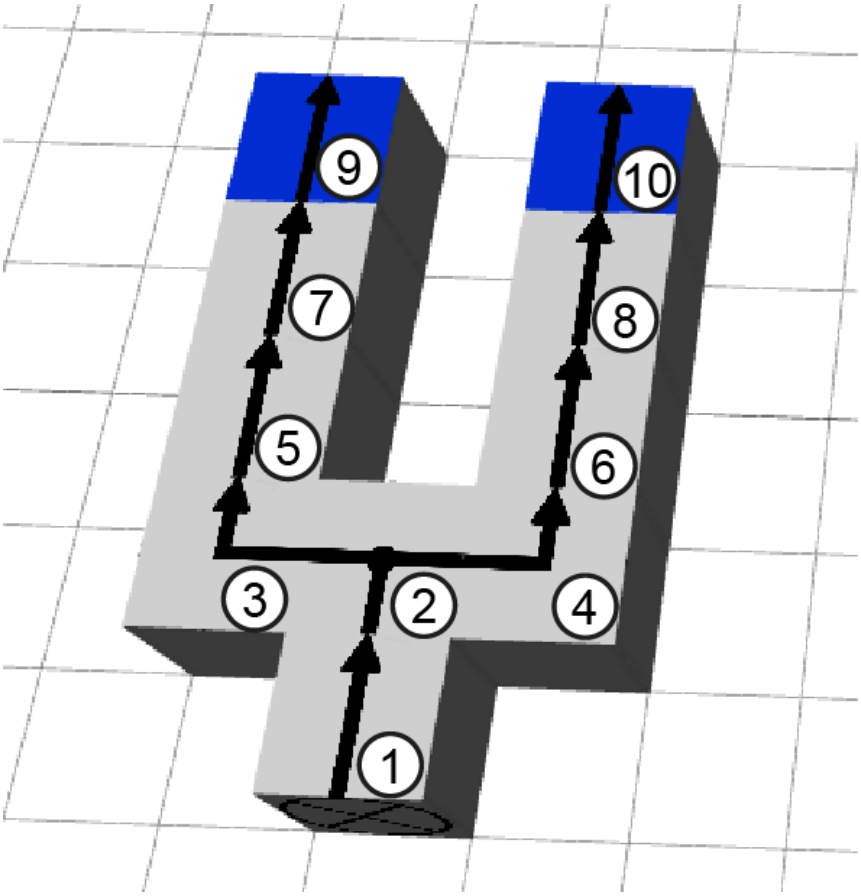


Figure 50: Triggering cells using a FIFO queue results in a fixed ordering depending on the implementation. Here, the left path will always arrive first.

municating with each other, to avoid having to store neighborhood information within each cell.

5.5 MODULAR SYSTEM DESIGN

The software we built is an extension to existing software and possible further extensions are conceivable. To support this possibility, it offers modularity in different parts of the system. The actual design of cells of our system for example can be changed easily by adapting the OpenSCAD script files used for rendering the printable STL files. The script is divided up into different parts, allowing exchanging some of them without compromising the other. A separate script file calls all modules and arranges the parts created by them to form the final cell design. All modules are customized by a shared set of parameters, which is stored in another separate file. Providing a different file for the spring and frame creation could for example produce the results shown in figure 51. The chosen spring design however offered

the best performance while providing useful options to adapt it to different use cases, such as executing a longer stroke.

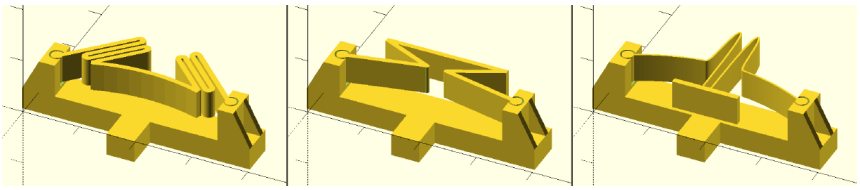


Figure 51: A customized OpenSCAD script creating the frame and three script versions for creating a bistable springs were used to render these 3D designs. A modular system design allows exchanging them quickly.

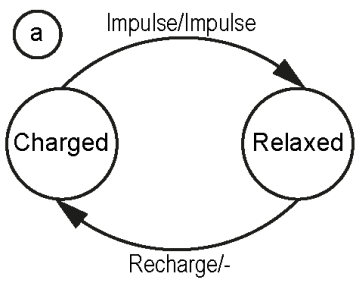
Parts of the editor are also designed in a modular fashion. For example adding new cell types or new cell arrangements for the user to employ in his objects can be achieved by simply adding new brushes to the system. Some useful predefined cell patterns are already implemented as brushes that can be added to the grid with one click, for example a 2-bit 4-to-1 multiplexer. A 3-bit 8-to-1 multiplexer could be added in the same fashion with little effort. The modes of the editor itself are internally treated as modules. New modes can be added efficiently, as long as they provide an *updateVoxel* method, that complies with the expected parameters for this class of functions.

5.6 UNDERLYING MODEL

For a deeper understanding and analytic possibilities in future work, we present two models that capture the states and behaviors of our system well and how they could be used not only for analysis but also for operational use in the software system. Some modeling options have already been mentioned in section 3.6, and here we will discuss the application of finite state machines and petri-nets to our system in detail, starting with a FSM model for our most basic cell (cf. figure 52).

5.6.1 *Finite state machines*

The transmission cell as a FSM is an automaton with two states. If it is charged, it can be triggered by an impulse to output an impulse of its own. If it is not charged, a recharge action from the user brings it back to its charged state, ready to be used once more. In this model, every cell is represented as one simple FSM.



b

State \ Input	Impulse	Recharge
Charged	Relaxed/Impulse	Charged/-
Relaxed	Relaxed/-	Charged/-

Figure 52: (a) The state diagram for a simple transmitting cell illustrates the two states of the cell/spring. (b) Shows the corresponding state transition table.

Such an automaton is then treated as a sub-FSM of a hierarchical setup, where multiple FSM's communicate via impulse messages. The amount of states within the composite FSM is large, as it is fully defined by the combination of states of all sub-FSM's. The number of states is the multiplication of the number of states of all sub-FSM's. However, not all of these states are useful or even reachable, leaving room for optimization. Using this model, one can furthermore easily abstract from the internals of the automata and only regard the automaton as a whole. Following this concept, even a setup of multiple blocking and gate cells that function as a logic gateway could be viewed as one unit, in the form of a hierarchical FSM (cf. 53). This results in a scalable interface where both a broad overview and a detailed view on the internals are available as desired.

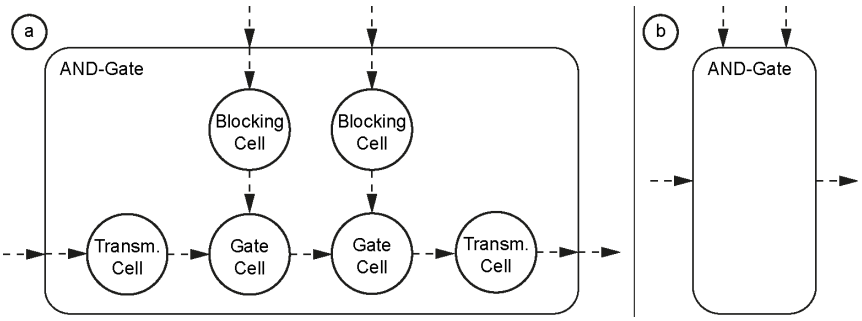


Figure 53: (a) Shows the internal FSM's of the AND-Gate automaton, but already abstracts from their internal states. (b) Takes the abstraction further, hiding unnecessary details from the user.

To create a FSM for a gate cell based on the transmitting cell model, one would replicate the existing states. The first set of these state would function as before, and the second set would ignore the impulse, thus blocking the signal, instead of emitting an impulse itself. One would however have to differentiate between the impulse that

triggers the cell and the impulse that configures the blocking state, by naming them differently.

To allow a nonrestrictive execution of the model however, these impulses should only be distinguished by the direction they are coming from. Yet traditional FSM's do not discriminate between positions of parts of the model. One would have to extend the modeling environment through the notion of a grid, e.g. by constraining the state diagrams to only utilize arrows with fixed lengths of multiples of the cell size, as well as the constraint to only use 90° turns. Given this grid layout and some additional FSM properties, it would allow generating and simulating digital mechanical metamaterials directly from within the model. The other necessary properties are the *greediness property* of the automaton and *simultaneous reactions*.

The *greediness property* describes the convention that in every execution step, the maximum number of possible transitions and static reactions always have to be taken. This means that all FSM's that can undergo a transition have to do so. The order in which transitions are processed should be determined by the method presented in section 5.4.

The property of *simultaneous reactions* closely relates to the *greediness property*. It assumes that the answer to any external stimulus happens already in the same step the input arrived. Whenever the state of the model changes, it has to be evaluated whether new transitions are possible, and those transitions in turn have to be used. This process then has to be repeated until no more transitions can occur. Both properties together denote for example that when a user triggers a cell, the resulting signal may traverse the entirety of the object and provide a result instantly, i.e. before the user has the option to execute another input, as is the case in the real world.

5.6.2 *Petri-nets*

It is also possible to model the parts and the behaviour of our system using petri-nets. Instead of modeling every cell as one FSM, here each transition describes one cell. Figure 54 depicts the same logic as the composite FSM in figure 53. Colored tokens are necessary to implement our system in a petri-net. Charging the system in the context of the petri-net would mean filling all of the red places in 54 with one token. The representation of the logic state of the system is very succinct through the usage of the blue places in 54. Each pair of blue places denotes one state bit in the system, that is either set to true or false. The logic state of the system is encoded purely by the blue places, so hiding all other details gives a good overview of the logic system state.

To make sure, that the order of execution of transitions is the same as triggering cells in our editor, we have to utilize different concepts of timed petri-nets in our models. Transitions can have a "delay of transition firing" property, which can be used to model the triggering delay of the cells in our editor. The tokens of the petri-net can also be extended with the "token age" property. This can be applied to the green impulse tokens in figure 54, since these tokens are not allowed to linger in a place longer than the maximum activation time of a cell, thus forcing the following transitions to fire. Petri-nets offer options to abstract from complexity, such as a hierarchical architecture, similar to FSM's. The tool support for colored petri-nets is however limited to tools like CPN Tools[11], which offers hierarchical colored petri-nets, but not the necessary timing functionality to fully model our system.

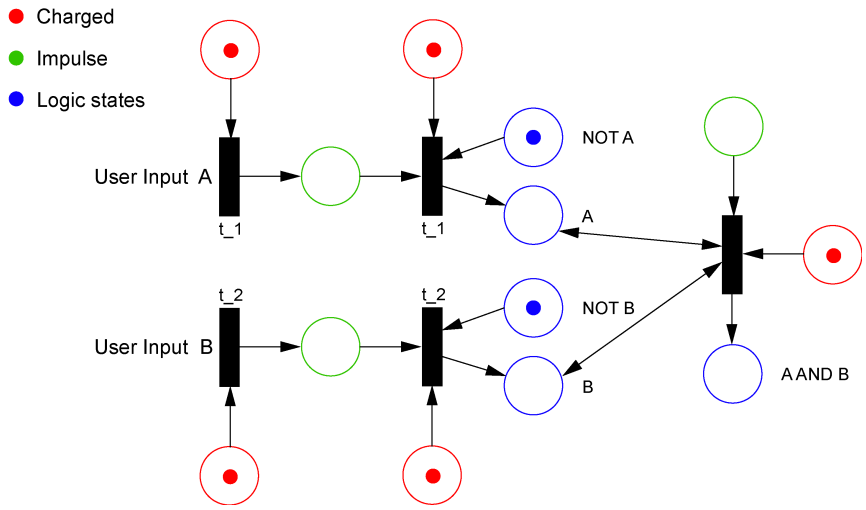


Figure 54: This petri-net shows an AND-gate. Transitions model cells, so they can only fire if they have a red *Charged* token. Blue logic tokens are not consumed when evaluated, as the physical state remains unchanged and other transitions may evaluate them again.

5.7 RENDERING

To render metamaterial and logic cells efficiently, the editor uses custom shaders that follow the OpenGL 2.0 specification. The utilized vertex and fragment shaders are written in the OpenGL Shading Language (GLSL). The editor from [7] is able to render single colored voxels. We built on this rendering architecture and adapted the shader code to be able to also render voxels with custom textures and tex-

11 http://cpntools.org/

ture rotations. Based on the position data of the vertices, the shader reflects and rotates the texture appropriately for each of the two polygons per face of the voxel. The custom rotations are applied afterwards, which allows reusing the same textures independent of cell rotation. To increase performance, the rendering process terminates after the first texture is drawn per pixel. The texture is manipulated during rendering through state dependent color changes as visualized in figure 55) and shadowing, which is an adapted version of the original shadowing functionality.

Using one mesh for all cells avoids redundancies in the allocation of memory, since for example all necessary textures only have to be loaded and stored once. Any DirectX-10 capable GPU offers at least 16 texture units and thus allows storing and using up to 16 different textures per mesh. Modern GPUs offer a higher number of texture units, but to ensure compatibility we limit the number of textures used per mesh and thus per geometry buffer to 16. As shown before in figure 36, the geometry buffers prepare the voxel geometry data for the graphics card and provide the shader code that is used by the GPU to render the cells. We employed a total of three geometry buffers, each rendering a group of cell types, where the cell types within the groups share similar sets of textures. The animation of the cell colors during the simulation is not realized through additional textures with adjusted colors. Instead, the colors of the textures are altered ad hoc during rendering. For all pixels of a cell that have black color (as most symbols on the used textures do), the color vector is changed depending on the state of the cell. In conclusion, charged cells are rendered blue, active ones yellow, and uncharged cells use the texture base color, i.e. black (cf. figure 55).

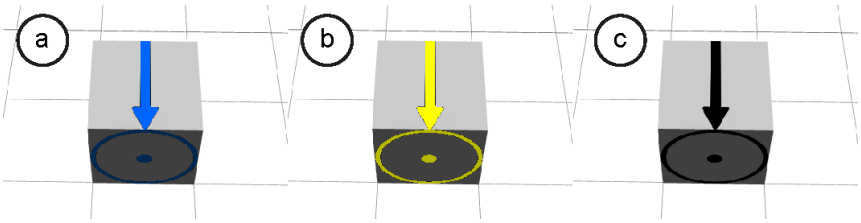


Figure 55: (a) The black parts of the voxel textures are rendered blue to signify that the cell is charged. (b) Active cells are rendered yellow instead. (c) Uncharged cells are rendered without color changes.

6

CONCLUSION AND FUTURE WORK

We built an interactive editor that enables the user to create metamaterials that integrate digital computation. The editor supports designing with specialized metamaterial cells that transmit signals and compute combinational logic purely on a mechanical level without added electronics. The cells contain an efficient customizable bistable spring that can trigger springs of adjacent cells while it changes from its tense to relaxed state itself, implementing signal transmission, redirection and bifurcation. It support truth tables and logic functions for the synthesis of these logic cells and also offers the option to create mechanical circuits by placing basic building blocks such as logic gateways. Pathfinding capabilities facilitate interconnecting this circuitry, which can furthermore be simulated within the editing environment. The logic cells can be integrated with traditional metamaterials and exported to a printable STL file that does not require assembly after printing. We created a recharging mechanism that simplifies tensing the bistable springs within the printed logic cells.

6.1 AUTOMATIC FURCATION GENERATION FOR 1:N SIGNAL CONNECTIONS

We want to enable users to create the necessary signal lines to connect one input with an arbitrary number of outputs or vice versa quickly, through automated generation of furcating signal lines, in the next version of the editor. We contrived a process to find efficient furcated paths for $1 : n$ or $n : 1$ connections using a heuristic pathfinding algorithm, such as the A* pathfinder used in the editor. First, find an initial path as usual from one start to end cell. Lower the weight used in the heuristics of the pathfinding algorithm of all cells that belong to this path (for the duration of this process). A lower weight means that is is 'less expensive' to use these cells. This rewards following the existing path in following iterations of the pathfinding algorithm, thus ensuring furcations happen close to where they are needed. The pathfinder will always prefer using the cells with lower cost/weight and only leave the existing path, thus forking the signal, when necessary, right before the target cell. Continue using the pathfinding algorithm for cells in close proximity to the previously chosen cell. In all further iterations, lower the weight of all newly created cells as

before, but declare cells with their maximum number of furcations as blocked, i.e. set their weight to infinity.

6.2 SUSPENSION FOR LOGIC CELLS

In their current design, the logic cells that transmit signals within our system have a stiff frame. The future version of our system will feature cells, that are suspended within their vicinity, to allow them to compress or shear similar to other types of metamaterial (cf. figure 56). Such a suspension will enhance the integration with existing metamaterial mechanisms such as [7]. The logic cells will then not only connect to and configure e.g. shearing cells, but also act like shearing cells themselves.

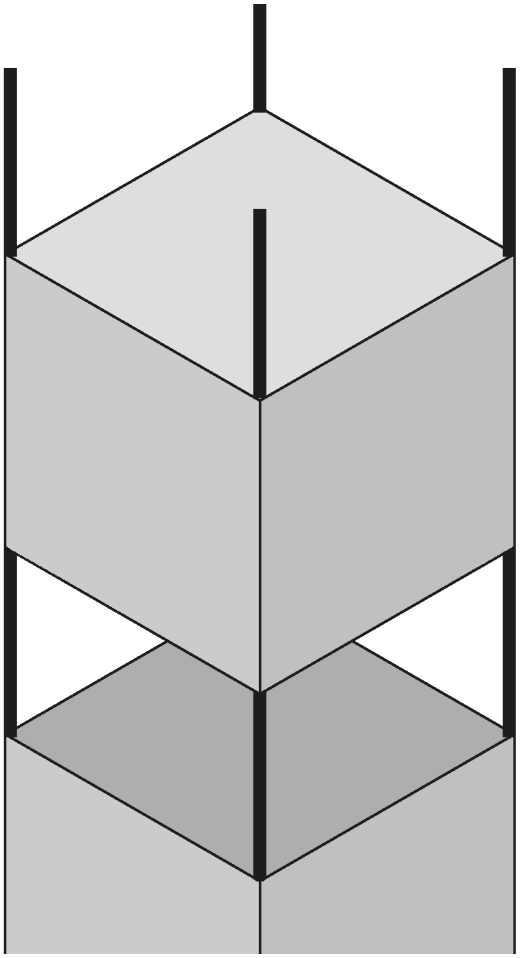


Figure 56: The suspension of the next version of our cell frames allows the cells to shear and it enables the system to place cells around curved surfaces.

They will then also be able to lock themselves in place, prohibiting the shearing motion, depending on their own state, i.e. whether their spring was triggered or not. This version will feature cells that can trigger adjacent cells at an angle, and we want the cells to dynamically fixate this specific angle at their time of activation.

6.3 MEMORY CELLS AND USER POWERED SYSTEM CLOCK

The logic cells of our system are designed to implement combinational logic, which is a stateless system. The work [11] described the idea of memory cells, that can store bits of information beyond one cycle of the system.

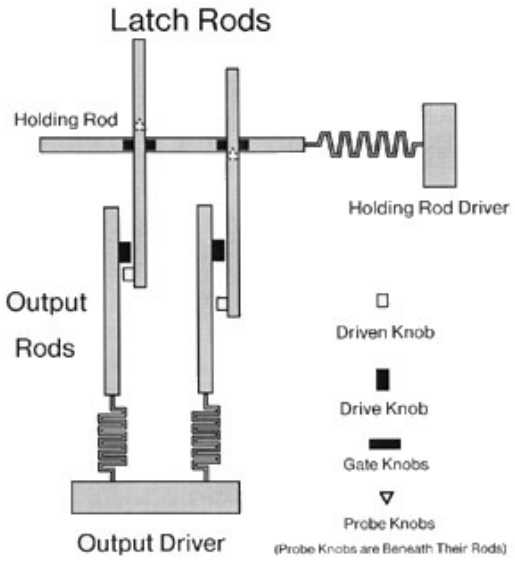


Figure 57: Latch rods in [11] move in conjunction with output rods, but are then locked in place by gate knobs on a holding rod. The previously calculated result of the system can be read through the position of the latch rod in following calculations.

We want to implement a similar mechanism as shown in figure 57 for the next version of our system. Using the possibility to store bits of data using memory cells over multiple iterations of using the system, we want to implement a mechanical clocking system that is powered by the user itself. Such a system could make use of the existing recharging mechanisms, powering it in the first half of a clocking motion, e.g. pulling a shaft. The second half of the clocking motion, e.g. pushing the shaft back to the original position, could be used to trigger all necessary evaluation signal lines.

BIBLIOGRAPHY

[1] Alur, R., Kannan, S., and Yannakakis, M. *Communicating Hierarchical State Machines*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 169–178.

[2] Bächer, M., Coros, S., and Thomaszewski, B. Linkedit: Interactive linkage editing using symbolic kinematics. *ACM Trans. Graph. 34*, 4 (July 2015), 99:1–99:8.

[3] Bächer, M., Whiting, E., Bickel, B., and Sorkine-Hornung, O. Spin-it: Optimizing moment of inertia for spinnable objects. *ACM Trans. Graph. 33*, 4 (July 2014), 96:1–96:10.

[4] Bickel, B., Bächer, M., Otaduy, M. A., Lee, H. R., Pfister, H., Gross, M., and Matusik, W. Design and fabrication of materials with desired deformation behavior. *ACM Trans. Graph. 29*, 4 (July 2010), 63:1–63:10.

[5] Calì, J., Calian, D. A., Amati, C., Kleinberger, R., Steed, A., Kautz, J., and Weyrich, T. 3d-printing of non-assembly, articulated models. *ACM Trans. Graph. 31*, 6 (Nov. 2012), 130:1–130:8.

[6] Elipe, J. C. Á., and Lantada, A. D. Comparative study of auxetic geometrics by means of computer-aided design and engineering. *Smart Materials and Structures 21*, 10 (October 2012), 105004.

[7] Ion, A., Frohnhofen, J., Wall, L., Kovacs, R., Alistar, M., Lindsay, J., Lopes, P., Chen, H.-T., and Baudisch, P. Metamaterial mechanisms. In *Proceedings of UIST'16* (2016).

[8] Jensen, K. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Vol. 2*. Springer-Verlag, London, UK, UK, 1995.

[9] Katsumoto, Y., Tokuhisa, S., and Inakage, M. Ninja track: Design of electronic toy variable in shape and flexibility. In *Proceedings of the 7th International Conference on Tangible, Embedded and Embodied Interaction* (New York, NY, USA, 2013), TEI '13, ACM, pp. 17–24.

[10] Lewis, J. A. Voxel8. http://www.voxel8.com/. [Online; accessed 11-October-2016].

[11] Merkle, R. C. Two types of mechanical reversible logic. *Nanotechnology 4*, 2 (1993), 114.

[12] Mueller, S., Mohr, T., Guenther, K., Frohnhofen, J., and Baudisch, P. fabrickation: fast 3d printing of functional objects by integrating construction kit building blocks. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems* (2014), ACM, pp. 3827–3834.

[13] Mullin, T., Deschanel, S., Bertoldi, K., and Boyce, M. C. Pattern transformation triggered by deformation. *Phys. Rev. Lett. 99* (Aug 2007), 084301.

[14] Nadkarni, N., Daraio, C., and Kochmann, D. M. Dynamics of periodic mechanical structures containing bistable elastic elements: From elastic to solitary wave propagation. *Phys. Rev. E 90* (Aug 2014), 023204.

[15] Panetta, J., Zhou, Q., Malomo, L., Pietroni, N., Cignoni, P., and Zorin, D. Elastic textures for additive fabrication. *ACM Trans. Graph. 34*, 4 (July 2015), 135:1–135:12.

[16] Paulose, J., Meeussen, A. S., and Vitelli, V. Selective buckling via states of self-stress in topological metamaterials. *Proceedings of the National Academy of Science 112* (June 2015), 7639–7644.

[17] Prévost, R., Whiting, E., Lefebvre, S., and Sorkine-Hornung, O. Make it stand: Balancing shapes for 3d fabrication. *ACM Trans. Graph. 32*, 4 (July 2013), 81:1–81:10.

[18] Raney, J. R., Nadkarni, N., Daraio, C., Kochmann, D. M., Lewis, J. A., and Bertoldi, K. Stable propagation of mechanical signals in soft media using stored elastic energy. *Proceedings of the National Academy of Sciences 113*, 35 (2016), 9722–9727.

[19] Savage, V., Chang, C., and Hartmann, B. Sauron: Embedded single-camera sensing of printed physical user interfaces. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2013), UIST '13, ACM, pp. 447–456.

[20] Savage, V., Schmidt, R., Grossman, T., Fitzmaurice, G., and Hartmann, B. A series of tubes: Adding interactivity to 3d prints using internal pipes. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2014), UIST '14, ACM, pp. 3–12.

[21] Schiff, J. L. *Cellular Automata: A Discrete View of the World (Wiley Series in Discrete Mathematics & Optimization)*.

[22] Schumacher, C., Bickel, B., Rys, J., Marschner, S., Daraio, C., and Gross, M. Microstructures to control elasticity in 3d printing. *ACM Trans. Graph. 34*, 4 (July 2015), 136:1–136:13.

[23] Vasilevitsky, T., and Zoran, A. Steel-sense: Integrating machine elements with sensors by additive manufacturing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2016), CHI '16, ACM, pp. 5731–5742.

[24] Weichel, C., Lau, M., Kim, D., Villar, N., and Gellersen, H. W. Mixfab: A mixed-reality environment for personal fabrication. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems* (New York, NY, USA, 2014), CHI '14, ACM, pp. 3855–3864.

[25] Willis, K., Brockmeyer, E., Hudson, S., and Poupyrev, I. Printed optics: 3d printing of embedded optical elements for interactive devices. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 2012), UIST '12, ACM, pp. 589–598.

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, October 2016

Ludwig Wilhelm Wall