Interactive Design & Fabrication of Bioinspired Surface Geometry

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University
In Partial Fulfillment of the Requirements for the Degree of
Master of Science

by
Teng Teng
August 2021

Thesis Committee
Committee Chair
Prof. Jenny Sabin(Architecture)
Minor Member
Prof. François Guimbretière (Information Science)

ACKNOWLEDGEMENTS

I would like to express my utmost gratitude to my committee Prof. Jenny Sabin and Prof. Francois Guimbretiere, for their expertise, guidance, and support during my thesis. Their knowledge and mentorship have been irreplaceable for driving this thesis forward.

I would like to dedicate my sincere appreciation to Prof. Jenny Sabin, who helped me during my difficult time. The enthusiasm and encouragement she brought to the program empowered me to pursue my dream without hesitation.

I would like to acknowledge that this thesis is a cumulative project and would not have been possible without collaborative teamwork. To this end, the author would also like to acknowledge my former team partners Benjamin Mian Jia for his extensive contribution to the early stage of scutoid investigation in Fall 2019. I would like to emphasize with gratitude the significant contributions of my collaborators Eda Begum Birol in designing the robotic ceramics extrusion experiments. This thesis would not have been possible without her diligent and thoughtful work. I am more than appreciative to work with a group of talented people in Jenny Sabin Lab at Cornell University, including Mahshid Moghadasi, Madeline Eggers, Karolina Piorko, Kevin Guo, Alexia Asgari, Veronika Varga, Chi Yamakawa, Alexander Wolkow.

I would like to thank my parents for their understanding and for supporting me in finishing this program. I also need to express my sincere gratitude to my life partner, Zhen Cheng, for her encouragement and accompany. I would not accomplish this fantastic journey without their support.

ABSTRACT

With an enormous number of outstanding design works emerging, bio-inspired design strategies have arguably been one of the most popular design subjects in the past decades. These works are often praised for their higher resilience, adaptability, and efficiency. However, in the meantime, bio-inspired design strategies have also led to higher complexity in the morphology of design objects, which makes the design materialization process (fabrication) being particularly challenging. With the continuous development of computational design and digital fabrication technology in the recent past, many innovative tools developments can assist complex form manufacturing. Nevertheless, the transition from design to make is yet situated in a unidirectional execution process. In this conventional process, design and make are two independent steps. Namely, designers use various tools to design and produce design drawings in practice. Then manufacturers take over the drawings and use multiple digital processing tools to materialize the design object. Although the emergence of robotic fabrication reduces the miscommunication between designer and maker in the process, it still challenges at an architectural scale, such as building bespoke robotic tooling for designers and designer-friendly robotic user interface. This thesis addresses the above issue by conducting a case study on designing and fabricating a morphologically complex surface geometry developed with bio-inspired design strategies. The thesis describes the method and process of designing such bio-inspired surface geometry and the means of materializing such geometry by proposing a series of bespoke robotic tooling for designers to fabricate complex geometry. Furthermore, the thesis discusses how the author can merge design and fabrication on an architectural scale to facilitate final product quality through the case study.

BIOGRAPHICAL SKETCH

Teng Teng is a second-year student in the program of Master of Science, Matter Design Computation at Cornell University. He is an interdisciplinary researcher as well as an experienced architect. He holds a Master of Science degrees from University of Washington, and a Bachelor of Engineering degree from Shanghai University.

He was formerly appointed as a research associate in Jenny Sabin Lab at Cornell University and a lecturer in the Shanghai Institution of Visual Art. As a researcher, his primary research interests reside in Robotics, Computational Design, and Human-Computer Interaction. He is particularly interested in HCI issues in the design and fabrication process, focusing on providing intelligent and instructive computational tools for designers and makers.

Teng was also an experienced architect. He worked at a Seattle-based renowned international architecture firm for five years as a project architect and computational leader. He earned various practical experiences in conceptual design to construction administration from more than twenty large-scale mixed-use, retail, and residential projects across Asia, Mid East, and North America.

Table of Contents

INTRODUCTION

As Janine Benyus suggested in her seminal book "Biomimicry," it is necessary for all architects, designers, and engineers to "not be so smart." Especially when investigating the task background, she suggested not seeking to create innovative methods at the early stage. However, the author will benefit from spending time listening and observing nature. Although it starts very slowly, the innovation will emerge when establishing a deeper understanding of the specificity of the situation. Of course, Benyus also asks us to learn from biology, from the specific behaviors of certain animal and plant species that have evolved for millions of years. The abundance of the earth's living beings provides almost endless nourishment for more innovative solutions that are light, benign, and renewable. According to Janine Benyus's book, biology can inspire design and engineering in three levels: 1) mimicking the characteristics of species, such as species' certain morphology that enhanced the long-term survival of the species. 2) mimicking the manufacturing processes that species used to make their body parts without using esxtra resources or producing toxins 3)mimicking species' long-term survival strategies and organizational principles from the social behavior of organisms.

Utilizing natural elements, especially the biological morphology, as design inspiration may have originated from the beginning of artificial built environments and still exists today. The ancient Greeks and Romans incorporated natural patterns into their architectural design, such as tree-like orders. Byzantine arabesque tendrils are stylized versions of scrolling and interlacing foliage tendrils. The Sagrada Familia of Antoni Gaudí, which began the construction in 1882, is a well-known example of using the functional forms of nature to address structural questions. Antoni Gaudí mimicked tree branches to design pillars that solve the statics problem of supporting the vault. In the modern age, the structure of London's famous Gherkin Tower (30 St Mary Axe) was inspired by a marine animal called Euplectella aspergillum. This delicate creature is named after a hollow tubular structure (or "basket") supported by a lattice-like skeleton. The skeleton segments are arranged vertically, horizontally, and diagonally to form a cage-like structure. This organic scaffold provides tremendous mechanical strength, allowing these sponges to live up to 1 km deep underwater. Beyond architectural practice, a lot of successful academic works are also distinguished by applying biomimetic investigation into design research. During the design process of ICD/ITKE Research Pavilion 2013-14, the team found that the protective shell of beetle wings and abdomen, Elytron, is a highly efficient material suitable for construction. The lightweight structure composed of these two-layer systems can be combined with natural fiber composite materials to achieve maximum mechanical performance.

Cells are the smallest unit of life, and cell biology is an important field of life sciences. Cell biology mainly focuses on cell morphology, cell function, genetic regulation, and various relationships between cells and their environment. For designers or those who concern with morphology, the significance of cell biology is that it provides a novel way to reconsider forms. At the micro-level in a living being's body, complex tissue morphology is caused by cells' various growing shapes and dissimilar movements. The changes in cell structure or the interaction between cells within the tissue causes the tissue to stretch, thin, fold, bend, invade or separate into different layers. If the design object adopts the relations between the cell and the tissue-the local units constitute the global geometry-the global form can be determined by individual units. Dialectically，the global form can be subdivided into the corresponding units as well.

However, the morphology innovation triggered by biology is often accompanied by an increase in the difficulty of fabrication. It requires the architecture team to propose a unique material, method, and fabrication craft in support of morphology innovation. For example, Even in the early years of Sagrada Familia's construction, reinforced concrete was not invented yet. Gaudí was the first architect who planned to use something similar to reinforced concrete to cast the pillars, which led to the emergence of self-compacting concrete after his death. Furthermore, Gaudí pioneering installed railways on-site to transport the material in wagons or building cranes to handle heavyweights. For achieving the Gherkin Tower's diagrid structural system, Arup proposed a new welding method that welds three steel plates to nodes at different angles. In ICD/ITKE Research Pavilion 2013-14 project, designers developed a robotic winding technology for modular double-layer fiber composite structures to maintain geometric freedom while reducing the number of the required templates.

The fabrication strategy of cell biology-inspired local-global geometry usually depends on materializing each unit and assembling the materialized units into a whole. When the global geometry is complex, which leads to each local unit has a unique form, the conventional method for fabricating such forms is to create a numerous number of processing files/templet for each unit. Nevertheless, depending on the fabrication method, such processing files/templets usually cost extra resources to produce. For instance, if the units are made by a milling machine, preparing a large number of toolpaths is necessary, despite it takes longer time than milling. Or, if the units are made by casting, then mold making process is costlier than casting. Indeed, in architectural practice, modularization and optimization technologies can convert a large number of local units with various shapes into modularized units with a certain number of modules. The manufacturer only needs to work with a limited number of modulized molds or templates. However, when considering the increasing complexity of advanced geometries constructed in research lab environments without the manufacturer's resources, fabrication of the limited number of modularized and optimized templates technologies is still less feasible.

Situating from the fabrication of a bio-inspired complex surface geometry in research lab environments mentioned above, the research question the author aims to address through the thesis is how to efficiently design and fabricate a bio-inspired and morphologically complex surface geometry, and how to embed interactive fabrication in the design process to inform the bio-inspired surface design loop at an architectural scale. The area of bio-inspired design-fabrication workflow and Interactive Fabrication Informed Design remains partially unexplored. My goal is to establish a systematic understanding of the area as well as to develop working prototypes that could embody the vision the author seek to explore.

The conventional digital design-make workflow can rapidly materialize product design. Fabrication tools are designed to precisely manufacture the artifacts prototype at late design phases to implement design output, where it targets at final products' precise dimensions, positions, and tolerances. However, the instantiation process divides the making process with the makers of the original representation (Carpo 2011). The growing distance between digital and physical, design, and making, leads to a monodirectional linear process. The process requires designers to create a conceptual model in a graphic user interface and convert it into a fabrication-oriented optimized scheme, which later needs to be encoded into a sequence of machine-readable scripts such as G-code for final production. When the final product is found to have any issue that isn't predicted in the design stages, the aforementioned process will be repeated and adjusted until the final product is ideal.

However, the author argues that this process needs to be upgraded into an interactive workflow. At an early stage, the design representations (drawings/digital models) are not intended to guide fabrication directly and inherently. Instead, it is produced to verify the design concept. As the design is pushed forward in later stages and goes through several prototype iterations, more constraints are considered, such as structure stabilities, material properties, and fabrication equipment features. Modern architectural and civil engineering software can conduct structure computing that provides scientific predictions to designers to avoid structural issues in the early design phase. The material properties can be tested through experiments and digitize data to be used in Computer-Aided Design software (CAD). However, in architectural practice, the fabrication equipment features are less acknowledged in the design phase.

Therefore, the author aims to establish a novel design strategy that integrates design and fabrication into an interactive loop by predicting fabrication results through acknowledging fabrication equipment features to inform design intent and generate new iteration.

Design is an abstract thinking activity that connects materialized design results through the mediating influence of fabrication (Gürsoy and Özkar, 2015). Working with a fabrication result allows designers/makers to acknowledge its properties in depth. The strategy will be implemented

by developing a series of design/fabrication tools and exploiting them to conduct design/fabrication tasks.

Malcolm McCullough (1998) suggests that an essential stepstone to achieve this vision is to develop an interface that combines interactive fabrication with real-time feedback. Outstanding works from fields of Human-Computer Interaction, Computer Science, and Design contribute to the subject of interactive fabrication technologies. A prominent early example of such design strategy linking real-time feedback with interactive fabrication is architect Frazer's Modular Intelligent Modeling system in the 1980s. The Tangible Media Group in MIT developed a considerable amount of projects on tangible modeling systems that provide real-time feedback. (Leithinger and Follmer 2014) More recently, the new emphasis on robotics in creative communities such as architecture, art, and design has been raised by the recent democratization of digital fabrication tools. With the rise of Fab Labs and maker culture (Anderson 2012), open-source hardware and software lower the accessibility for creative communities. Interactive fabrication researches are conducted via the application of robotic arm. Robotic interactive fabrication such as RoMA (Peng et al., 2018) and FormFab (Mueller et al., 2019) demonstrate the potential of robot integration in ad hoc design-while-making systems. RoMA allowed the user to create and manipulate 3D geometry in AR while a robot simultaneously prints the physical model in the same space. It rapidly provides designer tangible feedback to re-think the design object. In addition to HCI's work, many user-friendly robotic programming tools were developed by computational designers such as HAL, KukaPRC, and Mechina, enabling designers to work with robotic arms in real-time to conduct fabrication tasks. All these precedent works done in different fields are my stepstones to established the theoretical foundation of embedding interactive fabrication to inform the design process. However, few works connect interactive fabrication with design quality and rationality at an architectural scale.

CHAPTER 1

Scutoid Brick: Applying Epithelial Cell Inspired-brick in Masonry shell System

Abstract

This chapter focuses on the design of individual bricks in a masonry shell system that are inspired and informed by the reorganization of epithelial cells within tissues. Starting from a newly discovered shape called "Scutoid", the author first investigated how epithelial cells within living animals are packed three-dimensionally within tissues. the author focused on the living mechanisms within these cells that facilitate tissue curvature in the creatures' organs, skin, and blood vessels. By utilizing this generative geometric approach, the author created a series of parametric generators and modeling kits to represent this mechanism and process. the author then explored the potential for adopting this mechanism into larger-scale settings. Meanwhile, the author discovered that the deformation of individual epithelial cells during the bending process generates an intriguing triangular connection along the bending direction. the author translated this unique feature to the architectural scale as a joint system for connecting bricks in a masonry shell structure. Based on the above findings, the author designed and fabricated models for the masonry shell structure generated from scutoid bricks and this unique joint. The geometrical characteristics of scutoid bricks allows the packing of four bricks with just two joints. The work that the author have generated thus far contributes to solving isell design and fabrication issues from the perspective of individual units. The result of the shell structure model demonstrates that applying the epithelial cell inspired-block masonry system is a feasible approach for constructing shell structures.

*This chapter's work has been published and presented at the 38th annual conference of Education and Research in Computer Aided Architectural Design in Europe as:

Teng, Teng. Jia, Mian and Sabin, Jenny. (2020), "Scutoid Brick - The Designing of Epithelial cell inspired-brick in Masonry shell System" in Anthropologic: Architecture and Fabrication in the cognitive age - Proceedings of the 38th eCAADe Conference - Volume 1, TU Berlin, Berlin, Germany, 16-18 September 2020, pp. 563-572

Introduction

Due to the limitation of imaging technology at the nanoscale, a comprehensive visualized description of epithelial cells' three-dimensional appearance has been missing from the field until recently. Most biological researchers understood the shape to be similar to columnar prisms or a frustum shape. In 2018, through the approach of mathematical modeling, xxx unexpectedly predicted that as the curvature of these epithelial tissues increases, some of the epithelial cells would likely develop into forms other than columnar prisms and frustum shapes. The research claims this unique three-dimensional geometry is a transition volume between a pentagon and a hexagon with an added vertex. Researchers named the shape "scutoid" because the backs of some insects (Protaetia speciose) have a similar mini-triangular shell. Additionally, researchers successfully found that some of the epithelial cells present the same shape in several highly curved epithelial tissues in various creatures such as drosophila and zebrafish's renal tubule and thyroid follicles. The discovery of the scutoid has brought new geometric inspiration to understand the three-dimensional structure of epithelial tissues. Epithelial cells are one of the most critical cells in the early stages of every animal. They are capable of reorganizing themselves into different shapes to envelop cavities. This capability allows epithelial tissue to establish barriers for the creature's body from the external environment and deform the barriers to fit with the environment.

There are two parallel goals in this chapter. First, the author aim to investigate further the deformation mechanism of epithelial cells to enrich our overall understanding of how the epithelial cells present the scutoid shape by setting up a series of parametric models and visualized generators from the design research perspective instead of cell biology. Specifically, this paper looks into dialectical relations between global morphology and individual units at a micro-scale. Meanwhile, situating this from a design perspective, the paper aims to adapt the deformation mechanism of epithelial tissues and the concept of the scutoid into a novel design application. Notably, inspired by Scutoid geometry, the author designed a new type of brick termed *Scutoid Bricks*. The bricks can be used to enhance a joint system in a masonry structure, as the scutoid shape in epithelial tissue exhibits certain impressive features that the author can potentially enlarge to a macro scale.

Epithelial cells deform and proliferate themselves into multicellular tissues during embryonic development (Lecuit, Thomas, and Pierre-Francois Lenne., 2007). These tissues will eventually become various organs in our body. Primarily, serving as a medium to isolate different biological tissues as epithelial cells constitute the surfaces of creatures' organs and blood vessels. They form the skin of the creature, establishing a barrier for the creature's body from the external environment. Epithelial cells have other chemical and physical mechanisms, such as regulating the exchange of chemicals inside tissues and body cavities, etc. Our project focuses explicitly on the mechanism that allows epithelial tissue to deform itself into complex geometry. To achieve the wide range of functions mentioned above, epithelial tissues have evolved to develop complex and diverse cellular structures that exhibit unique structural features at multiple dimensions. These features allow epithelial tissues to form into a complex surface to fit with a given environment.

In 2D space, considering a single layer of epithelial tissue as two-dimensional lamellar, it can be represented by a two-dimensional sheet. Studies found that the deformation capability of epithelial tissue as a 2D

lamellar is achieved through its cell rearrangement. As the primary mechanism of tissue deformation, epithelial cells rearrange themselves to a new position without being separated from the adjacent cell according to a topological T1 transition (Kong, D., 2017). A single layer of epithelial tissue is composed of adjacent and closely connected epithelial cells. In this two-dimensional cell model, isolated cells can be approximated as circles. When two cells engage with each other, their common boundary can be represented by a straight line. The length of this common boundary depends on the distance between the geometric centers of the two cells. The closer the two cells are, the longer the common boundary is. When four cells are packed together, cells squeeze each other in both horizontal and vertical directions. If two of the four cells are neighboring with each other horizontally, the other two cells are isolated vertically. The distance of isolated and neighboring cells are inversely proportional. The closer the geometric centers of neighboring cells are, the further away the isolated cells are (Figure 11).

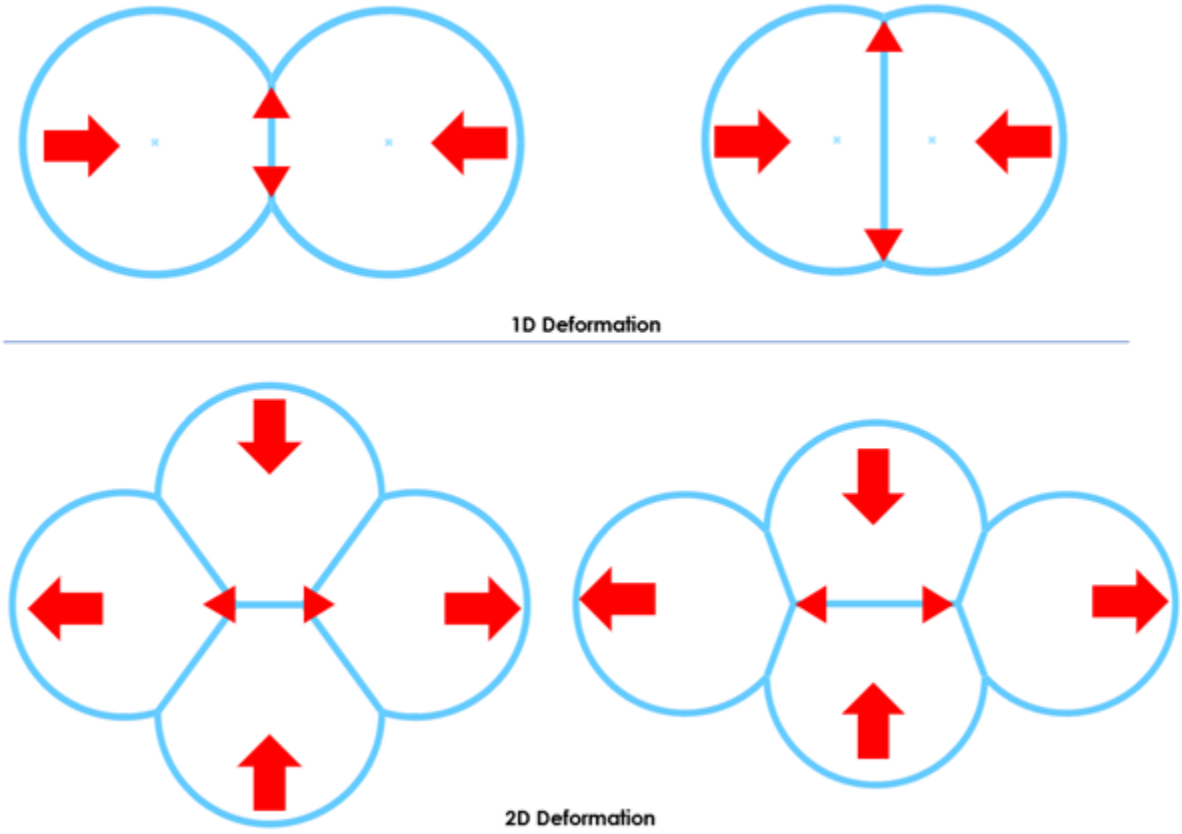


Figure 11, 1 and 2-dimensional deformation of cells

Expanding this into 3D space, for most healthy epithelial cells, the columnar appearance of epithelial cells is one of the critical morphological features. Geometrically, a columnar epithelial cell is composed of basal and apical surfaces, and the deformation process of epithelial tissue is a surface that, with thickness, bends at a particular position. When the tissue is bending, the basal and apical surfaces associatively perform the T1 transition to rearrange its position. Epithelial cells often have significant polarity, which is not only presented in the different functions and structures of basal and apical surfaces but also the various movement tendencies. The basal and apical surfaces of any pair of adjacent cells, that share one common

boundary, will have a different motion tendency. If the tissue is bent toward the basal surfaces (Figure 12), the basal surfaces of the two cells will two-dimensionally squeeze each other. At the same time, the apical surfaces will tend to separate from each other. However, due to the presence of adhesion forces between cells, adjacent cells will not be immediately isolated and their common boundary will not disappear quickly. In the process of tissue bending, with the increasing curvature of the tissue towards the basal surface, the boundary line between the two adjacent apical surfaces will gradually become shorter (Figure 13). Until the curvature reaches a certain level, both apical surfaces are separated. The basal surfaces that two-dimensionally squeeze against each other will produce longer boundary lines. (Figure 15) When four such cells are packed together, two cells that don't align along the bending direction will also have different motion tendencies on the basal and apical surfaces. The two apical surfaces that get closer with each other will occupy the space that is vacated after the detachment of apical surfaces. As the tissue bends, the basal surfaces that squeeze each other will occupy the space left by another pair of basal surfaces.

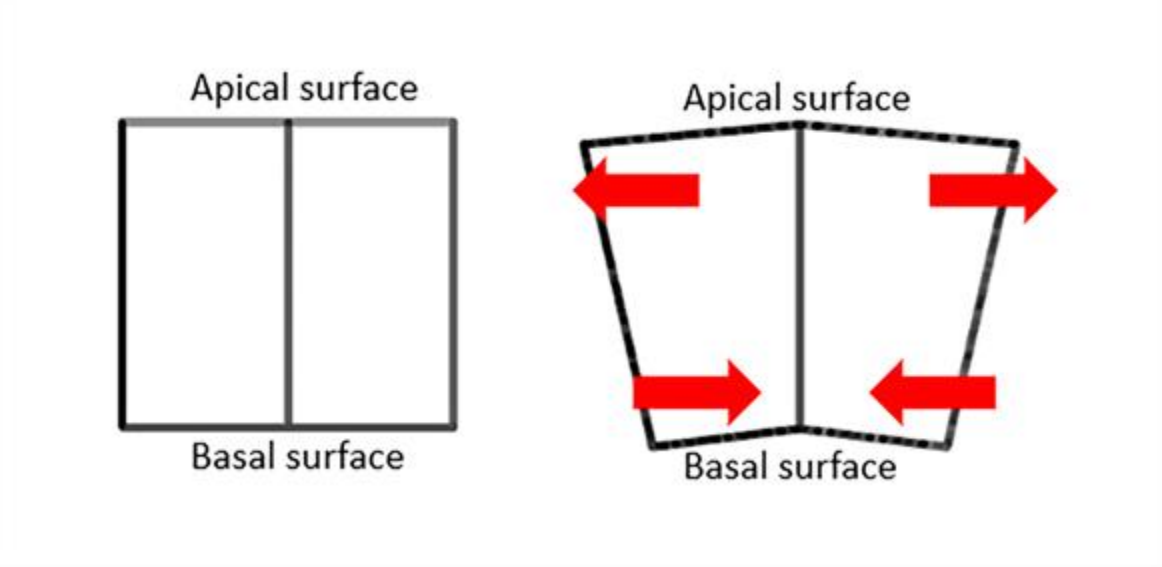


Figure 12, The tissue is bent toward the basal surfaces

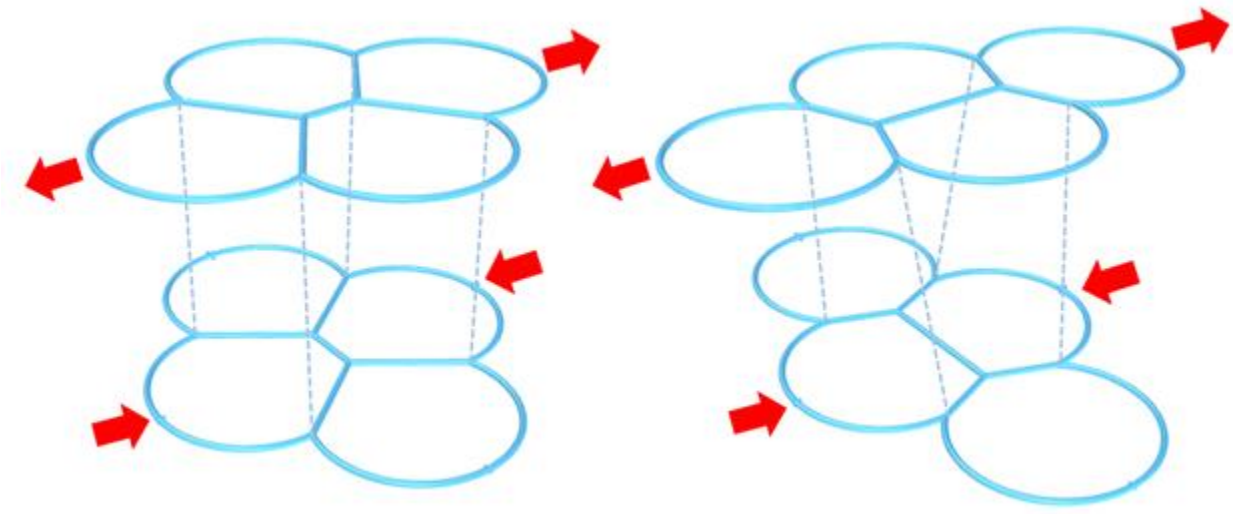


Figure 13, 3d deformation on apical and basal surfaces

Studies (Rupprecht, Jean-Francois, et al. 2017) also found that such polarity can not only be observed on basal and apical surfaces. This is observed inside of the epithelial cell cluster with the help a microfluidic device. The common boundary between adjacent cells exhibits various lengths at different depths (Figure 14). Given the fact that cells are competing for space under geometric constraints, the author realized that at a certain depth within a group of cells, the length of the common boundary would reach 0. It means the initial adjacent cells are connecting through a vertex, while the formerly separated cells that are perpendicular to the bending direction are also connecting through the same vertex. Our initial modeling strategy is based on the geometrical feature described in (G´omez-G´alvez, Pedro, et al, 2018), where the scutoid is a volume that transitions from a pentagon to a hexagon. the author found that the vertex mentioned above plays a critical role e modeling a given scutoid shape. Deeper within the cells, two of the hexagonal neighboring vertices on a surface merge to the mentioned vertex. By doing so, the hexagon on the apical surface converts to a pentagon in the middle of the cell's volume and keeps this formation until it reaches the basal surface.

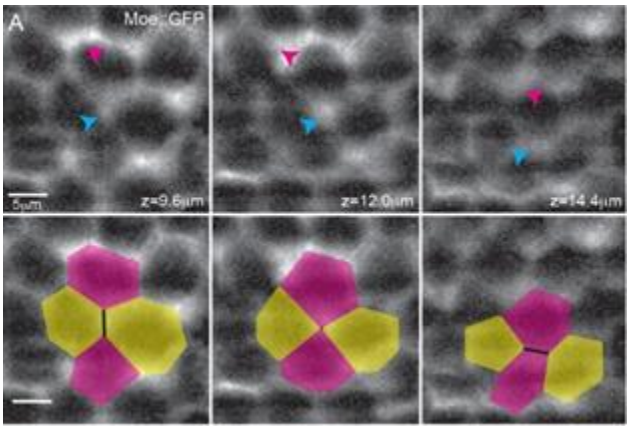


Figure 14, The common boundary between adjacent cells exhibits various lengths at different depths. Photo credits to Rupprecht, Jean-Francois, et al (2019)

The discovery and research above allows us to establish a framework for building a parametric model of Scutoid geometry. Our initial model is composed of three layers. Besides the two necessary outer layers to form the basal and apical surface, the author also set up an intermediate layer to arrange the key vertex. On both outer layers, the author defined eight fixed vertices and two dynamic vertices to form a pair of adjacent hexagons, where the length of the common boundary can be changed. Perpendicular to two adjacent hexagons, the author defined another four fixed vertices to form the associated pentagons which are separated from each other. However, on the intermediate layer, the author identified 13 fixed vertices and formed four pentagons which are sharing the center vertex. The formation of these four pentagons on the intermediate layer won't be impacted by the movement of vertices on the outer layers. By connecting the center vertex on the intermediate layer with the dynamic vertices on both outer layers, and then connecting the remaining fixed vertices, the author established a parametric wireframe model of scutoids that represent four epithelial cells packed together (Figure 16). The morphology of each cell in the cluster is determined by the length of the common boundary of adjacent hexagons. When the cluster model starts folding towards the basal surface, according to the T-1 transition, two hexagons on the apical surface detach from each other. The common boundary gets shorter and the pentagons on the same layer merge closer. Conversely, the

common boundary of adjacent hexagons on the basal surface becomes longer as hexagons are further apart, while the pentagons on the basal surface separate from each other.

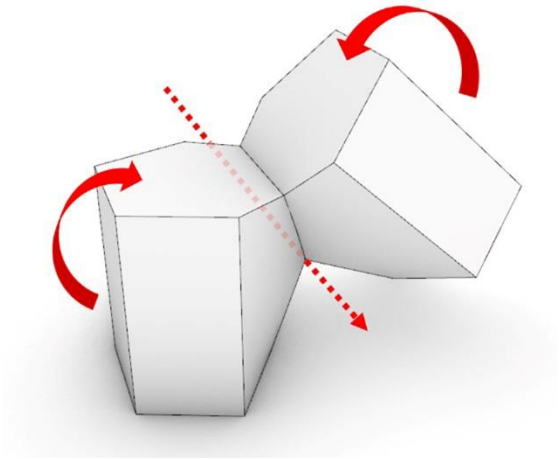


Figure 15, Cells vertically revolve along the horizontal axis resulted in different movement tendencies on apical and basal surfaces.

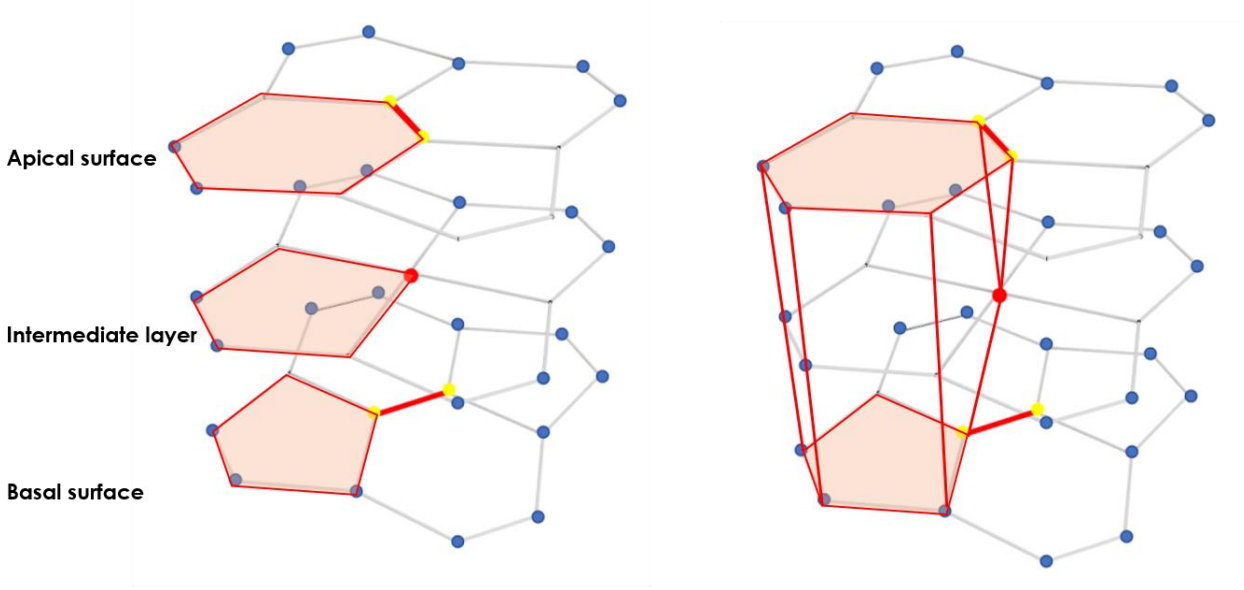


Figure 16, The framework of the parametric model of Scutoid

Related Work

As a recent discovery, scutoid related studies are still at an early stage. Yet, there are still some associated works that positively support our investigation. (Mughal, A., et al., 2018) successfully reproduced the scutoid-like soap bubbles between curved surfaces on a larger scale, as they inject soap water between two highly curved surfaces, which have an 18mm distance in between. The scutoid bubbles can be observed directly with the naked eye in the soap water. But in the controlled experiment, no scutoid-like soap bubbles can be found in between two flat surfaces. As the discovery of scutoids can only occur in highly curved epithelial tissues, this experiment verified that curved space is a necessary condition of generating scutoid. In other words, locally converting columnar prisms to scutoids can facilitate the overall curvature of global geometry that is packed by individual units. Furthermore, this study validates the existence of scutoid in a non-bio setting yet with less discussion of practical potentiality or quantitative description of scutoid.

Parallel to experimental verification, (Subramanian, Sai Ganesh, et al, 2019) started looking at the potential applications of scutoid in a more practical spectrum. They proposed a 3D space-filling approach by applying scutoid as its modular structure. They hope to develop new component-based space-filling tiles to compensate for the limitations on the structure's reliability and strength that is led by traditional prismatic filling units (like rectangular brick). Researchers have developed an algorithm that takes as input two planes containing Voronoi tessellations and inserts tiling between the top surfaces based on the distribution of points, then lofted each contour profile generated through the previous step into a 3D volume. This project didn't realize the fact that the form generation of apical and basal surfaces isn't conducted through the Voronoi algorithm in epithelial tissue, although it looks very similar. Also, this project proposed a 3D space-filling approach without giving an application scenario. But still, in terms of morphology, the author realized that the scutoid as a widely existing geometry should not be limited to the initial description of the transition from the pentagon to the hexagon. More importantly, the number of polygon segments at the basal and apical surfaces of a prismatoid are an essential factor that determines the scutoid. The author defined two conditions that are required for prismatoid-like geometry to be recognized as scutoid. First, the numbers of the polygons segment are not equal on apical and basal surfaces; second, two vertices on one surface can be merged into one vertex on the intermediate layer to equalize the polygons segment.

As we aim to adapt epithelial tissue deformation into a design space, we also examined previous work that borrows findings from biological models into architectural design research. Taking the mammary gland as a starting point, Sabin and Jones (2008) investigated the mechanism of how mammary glands respond to global and local stimulation to change its structure by taking advantage of dynamic parametric 3D modeling. Furthermore, the authors expanded the observations into the design of a set of deployable structures and simulated how the surface structure responds to environmental factors. It is truly a novel approach and workflow that adapts the cell behavior into an architectural structure and design process to bridge the gap between micro-scale biology and macro-scale architecture application.

Method

By developing the scutoid parametric model, the author propose that one of the necessary features of the scutoid is that either two vertices on the basal or apical surfaces merge into the center vertex on the intermediate layer. The connection between the three vertices creates a triangle that is partially bridging two epithelial cells. (Figure 17)When four epithelial cells are packed as a cluster and bend toward the apical surface, the common boundary between the pair of adjacent hexagons becomes extended, and the area of the connection triangle is larger. This mechanism creates an intriguing joinery method where two pairs of epithelial cells connect with each other in a bending environment. Greater curvature within the cell cluster also generates a larger area of triangular connection, thereby creating a more stable connection between two epithelial cells. The features mentioned above are well suited for application in masonry shell systems as one of the main concerns when designing a masonry shell is to avoid sliding failure within the bricks (Rippmann, M. and Block, P., 2013).

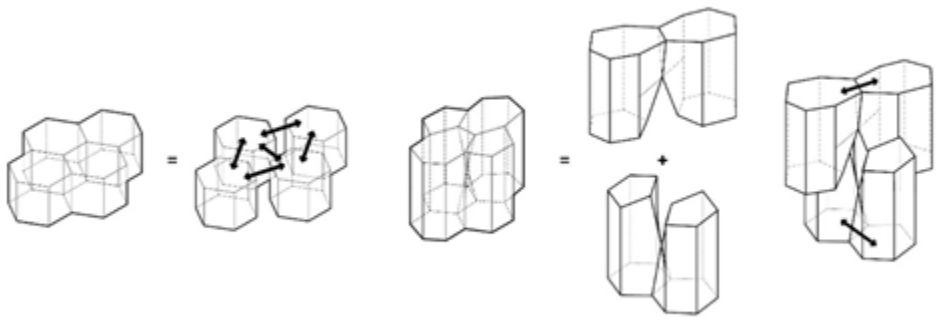


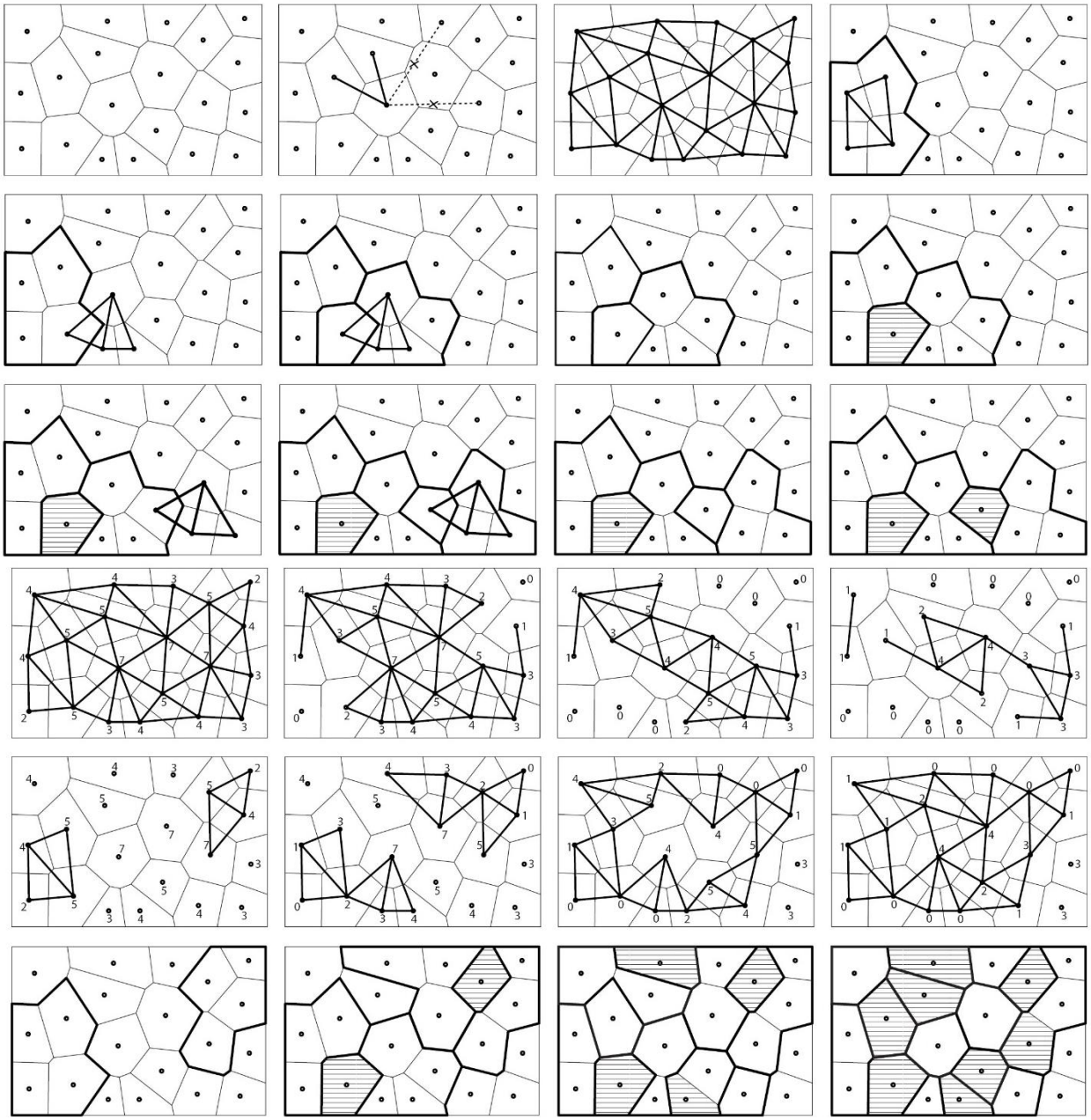
Figure 17, Two pair of scutoids are stacking with each

We propose a novel design and assembly of bricks for the fabrication of a masonry shell with the geometric features of Scutoid. Beyond building a parametric model based on four epithelial cells packed together, the author further explored the subdivision of a free form surface into unit blocks embedded with scutoid features that can be connected and assembled together. Our exploration can be summarized in the following two methods.

Voronoi-Based Scutoid Generator

The first method is a Voronoi-based subdivision solution. To achieve the optimal solution, the author utilizes this well-known algorithm as a geometric filter to select any four cells as a cluster in 2D space to then generate scutoids from. The eventual goal of the selection is to attain the optimal configuration of the cell clusters. Cells that have specific eigenvalues are eligible to merge as a scutoid cluster. The author

executes step-by-step to verify the number of neighboring cells based on the eigenvalues of each cell. The author chooses those cells that have two eigenvalues as the first batch to generate Scutoids. When one cell has been merged into a Scutoid cluster, the eigenvalues will be affected by minus one. And if continuing this process, the selection process will eventually reach the optimal result when there are no more eligible cells in the Voronoi diagram.



The author then generated an automatic workflow to form a three-dimensional scutoid embedded surface subdivision and converted this into a C-sharp code component in Grasshopper and Rhino environment. Based on the selection process the author have developed above, the scutoid generation starts with an initial input that changes the curvature of the surface. In this component, it is necessary to implement a simple associative data structure. It is a spring force model of a dynamic system combined with nodes and edges

in this case. The surface is interpreted as a simple non-interacting particle system where each element is a simple point whose location changes over time as a response to external forces applied to it obeying Newtonian physics. Some collections of objects can be updated in real-time. The initial spring force structure contains multiple polygons that are optimized from the Voronoi diagram. The cells on the margin are fixed. Then, external forces are applied to each node in the spring structure, which makes it bend gradually. During this process, for cells with edges and nodes that are overextended, a trigger function is added to generate scutoids (two vertices on the outer surface merge into one vertex in the middle of the cell). Hence, the Scutoid generator becomes functional when the overall surface is about to lose the stability of the original form.

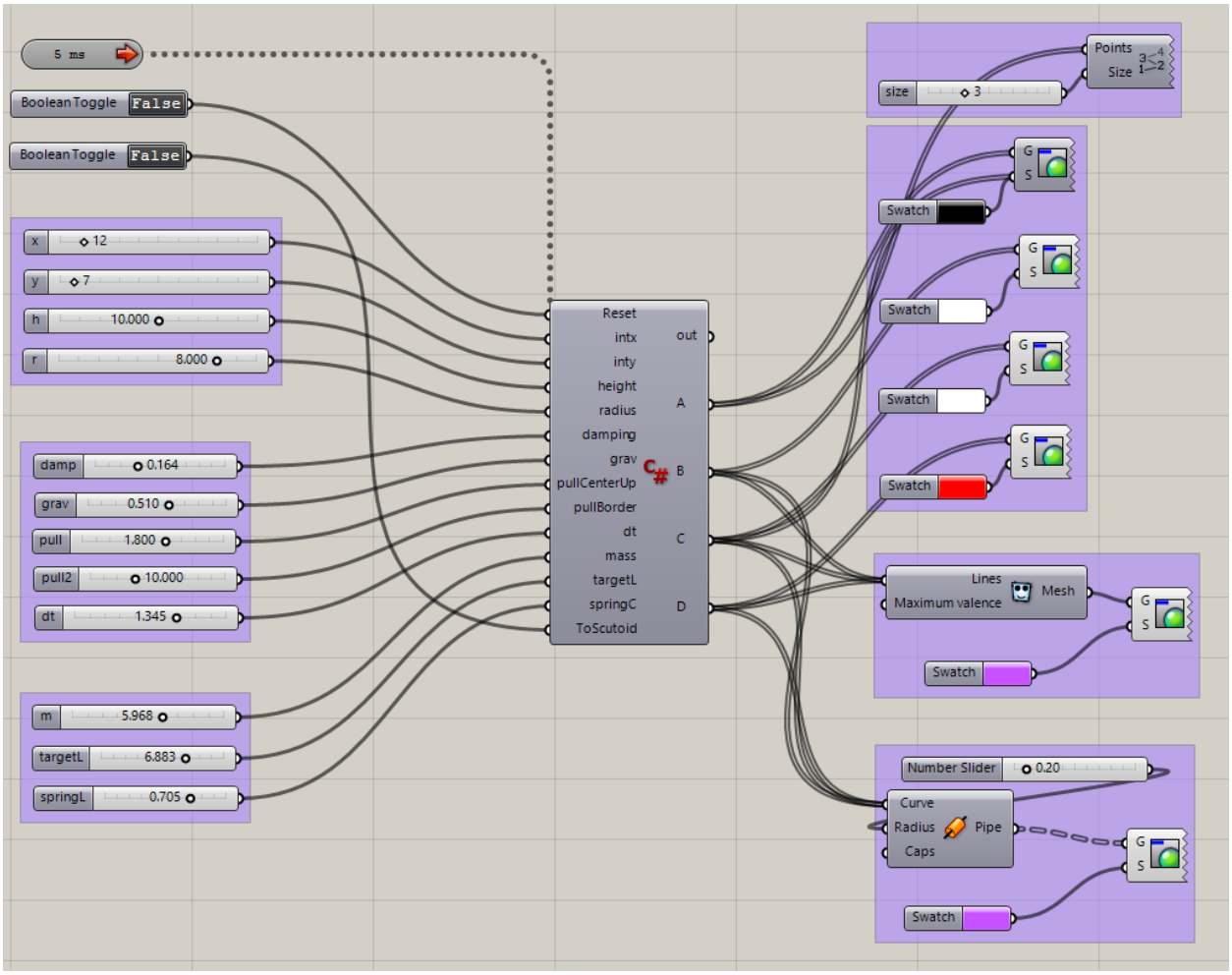
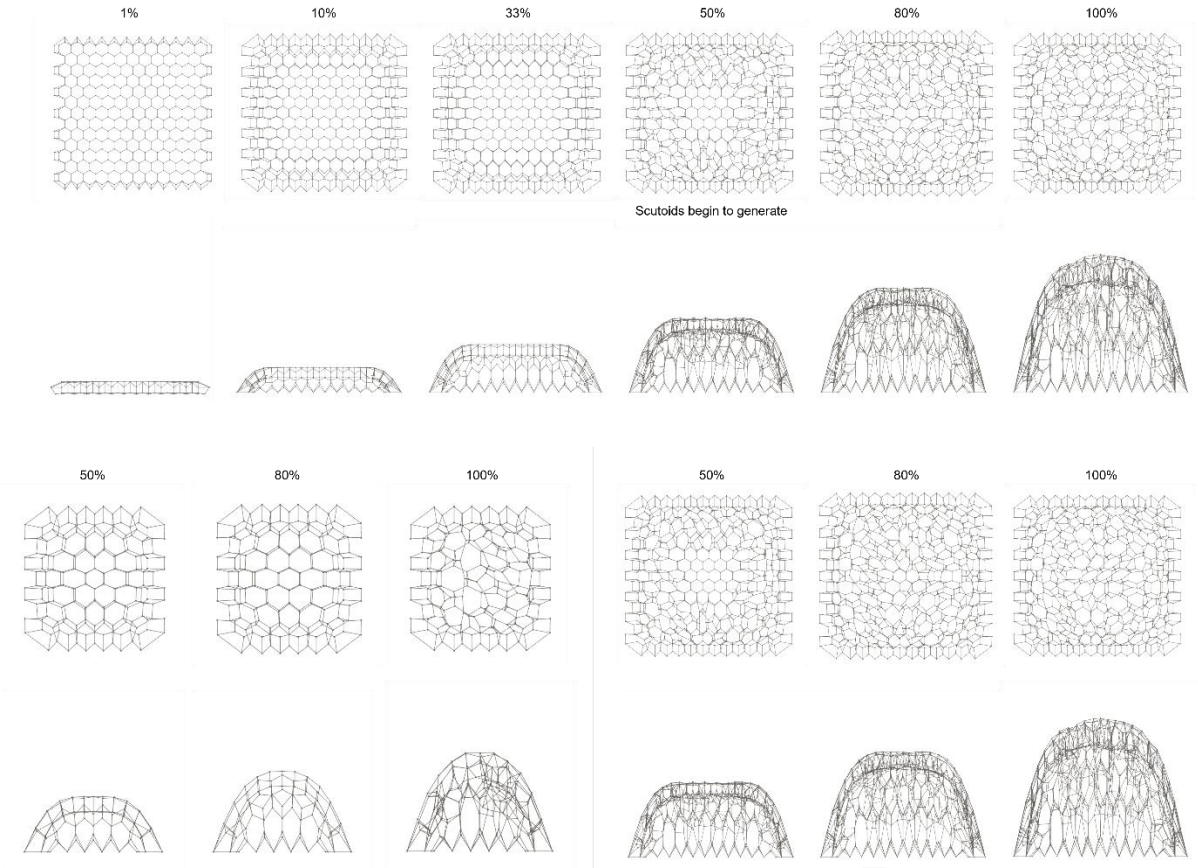
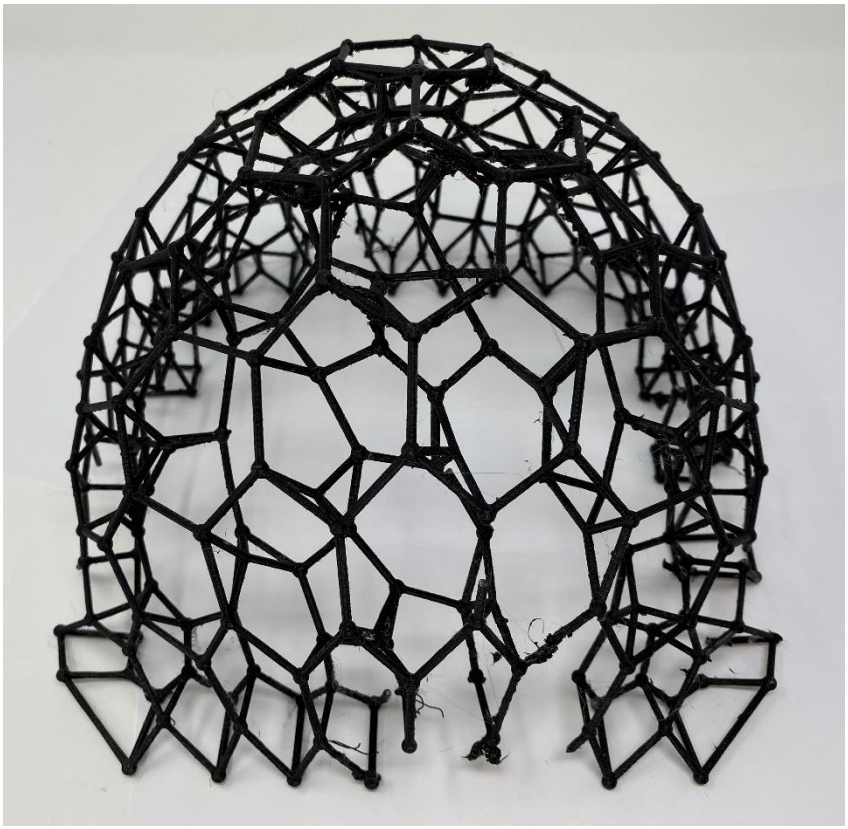


Figure 6, the interactive C# based Grasshopper plugin for scutoid generation.

After successfully establishing a Voronoi-based scutoid generator to create scutoid shapes within a curved surface, the author designed several trials to prove this generator's efficiency and optimize the workflow. the author conducted a comparison when some specific input elements are changed.


1%
10%
33%
50%
80%
100%
Scutoids begin to generate
50%
80%
100%
50%
80%
100%

A preferred model was printed as it best reflects the morphology relationships between local connections of Scutoid bricks. However, from a simulation perspective, the above method has less practical potentials.



Rational-Based Subdivision Scutoid Generator

Compared with the Voronoi diagram-based subdivision solution, the second method attempts to subdivide the overall surface rationally and practically instead of starting from random points. The method creates more accessibility for fabrication, as the sub-divided components are unified into variable modules. This method aims to generate scutoid bricks for a static double-curved shell so that blocks can be interspersed with each other without the risk of sliding failure.

Previously, the author defined that the parametric model of a scutoid cell cluster can be recognized as a three-layer structure. Two essential conditions for determining a scutoid are: 1) the number of the polygonal segments are not equal on apical and basal surfaces; 2) two vertices on one surface can be merged into one vertex on the intermediate layer to equalize the polygon segments. Based on the above rules, the author first extracted the three layers of a given shell geometry. Then, the author sub-divided the apical surface into hexagons and the intermediate layer into a variable grid of diamond shapes with hexagons and basal surface into a grid of diamond shapes and octagons (Figure 18). Finally, the author merged the two vertices of each hexagon on the apical surface with the vertex of each diamond shape on the intermediate layer. the author connected the same vertex of each diamond shape with the two vertices of the octagons on the basal surface.

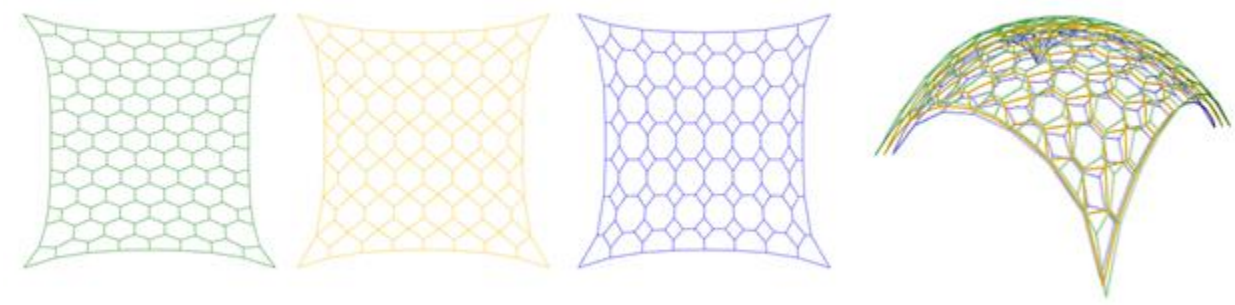


Figure 18, Three layers of sub-division of shell geometry

The above sub-division and connecting method creates two types of scutoid bricks: one with the arrangement of hexagon-diamond shape-diamond shape, and the other with the arrangement of hexagon-diamond-octagon (Figure 19). Despite the two different arrangements, both types of scutoids have two connection triangles allowing these scutoids to be continuously assembled.

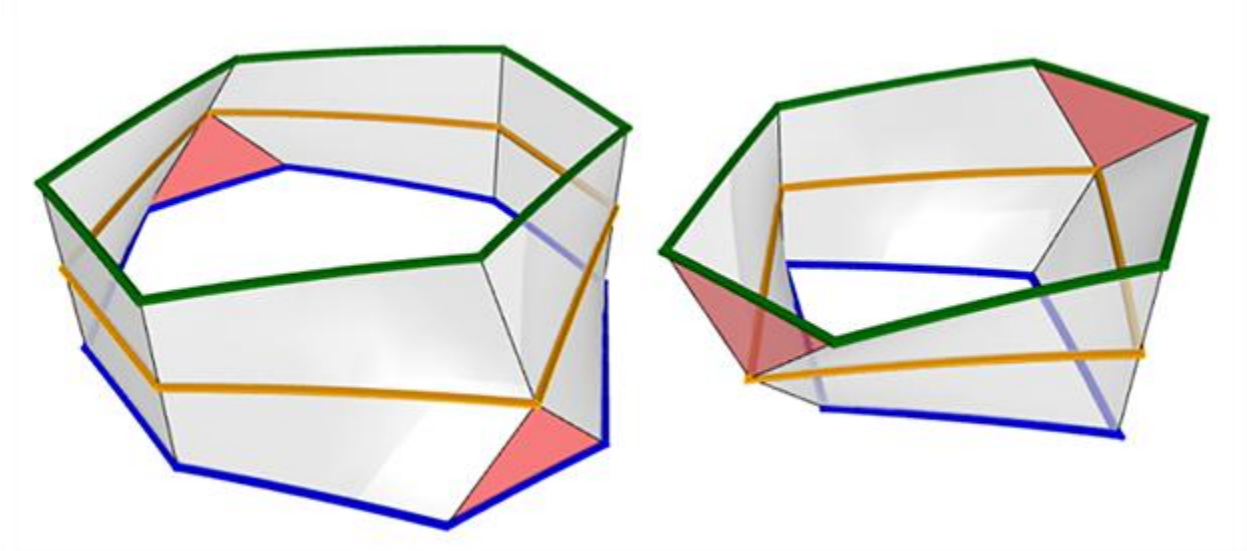


Figure 19, Two types of scutoid bricks

Parallel to the Voronoi-based scutoid generator, the author built a rational-based subdivision scutoid generator component in Grasshopper/Rhino environment. Several physical models of 3D printed shells are fabricated and assembled for evaluation purposes.

In addition to the advantage of the connection triangles preventing sliding failures, the author also discovered another benefit of the scutoid bricks in shell systems when making physical models. The connection triangles of both types of scutoid bricks (hexagon-diamond shape-diamond shape, and hexagon-diamond-octagon) aligned with two bending directions (U and V), as the given shell geometry is a doubly-curved surface. The two types of scutoid bricks share a bridge of connection triangles along the respective axes. Meanwhile, the connected scutoid bricks on both directions act like multiple arches that are

perpendicularly mortising together. The above configuration formed a space grid structure with greater structural stability (Figure 20).

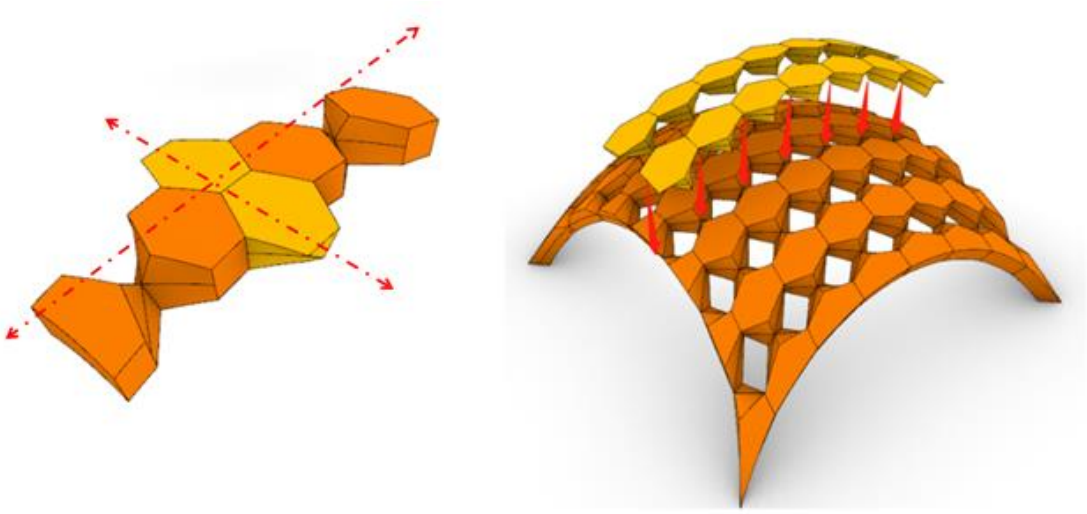


Figure 20, The connected scutoid bricks on both directions act like multiple arches that are perpendicularly mortising together

For testing the structural stability, the author conducted finite element static analysis on the selected shell model with ANSYS, and mainly focused on the total deformation and equivalent stress. In the controlled experiment, the author calculated the force situation for a non-masonry shell. the author found in a particular area that the total deformation is significantly higher than average. However, in the experiment of the masonry shell that is made up of scutoid bricks, the author found the deformation ratio on each scutoid brick of the shell is relatively even (Figure 21). Meanwhile, the author also conducted a stress test on a 3D printed and assembled scutoid masonry shell model. Each scutoid brick of the masonry shell model is 3D printed with PLA with a 20% infill, and the total net weight is 1.8 lb. The model surprisingly supported the payload of a pair of 40-pound dumbbells for 5 minutes without any damage (Figure 22). According to the results of FEA analysis and physical model testing, the author can assume that the use of scutoid bricks in

the masonry shell system can distribute a sudden change in stress more evenly than a non-masonry shell.

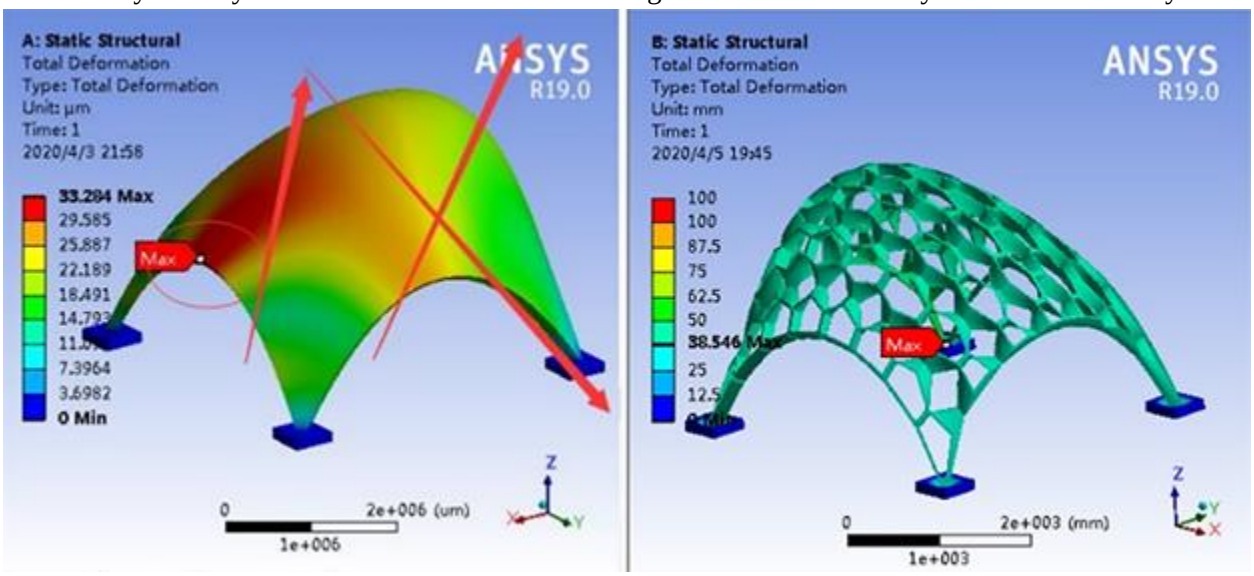


Figure 21 FEA of shell model

Figure 22, Testing with physical model

Future work will include full-scale testing and digital fabrication accompanied with detailed design and technical drawings. Fabrication methods and material studies for constructing a masonry shell with semi-rigid and rigid scutoid bricks will be investigated at full scale. Notebly, this study of scutoid bricks shows us that this unique geometric system does not only pertain to biological models. A comprehensive investigation requires interdisciplinary teamwork.

CHAPTER 2

Design and 4d printing of Epithelial cell-inspired Programmable Surface Geometry

Abstract

This chapter aims to provide a novel surface geometry design and fabrication strategy as a design research project. As the foundation, this session discusses and investigates cellular epithelial tissue's deformation mechanism and geometric features, especially the generation of the newly discovered scutoid shape. Subsequently, the author utilize the mechanism to design and fabricate programmable physical surface geometry that can change shape autonomously based on external stimulation. the author summarize the work the author have conducted thus far into two aspects: First, inspired by the deformation mechanism of epithelial cells the author propose a new design strategy for generating complex surface geometry from transformable individual units; Second, the author also develop a new 4d printing method, which allows the surface geometry to be programmed on demand and to emulate the generative and bio-inspired design model analogically.

*This chapter's work has been accepted by the 39th annual conference of Education and Research in Computer Aided Architectural Design in Europe as:

Teng, Teng and Sabin, Jenny (2021), "The Design and 4d printing of Epithelial cell-inspired Programmable Surface Geometry as Tangible User Interface" in Towards a New, Configurable Architecture: The 39th International Conference of Education and Research in Computer Aided Architectural Design in Europe (eCAADe 2021)- 8-10. September 2021.

Introduction

Surface modeling and design in the digital environment has become a significant part of computational design and fabrication planning. Designers can efficiently obtain access to digital surface design toolkits, now readily available through online forums such as “Food4Rhino”. Nevertheless, designers and makers still have fewer available methods for fabricating complex surface geometry. The repetitive operation of designing complex surface geometry is inevitable as designers often face more parameters and notation systems to optimize and organize the freeform surface morphology than conventional cubic architectural geometry. Additionally, the author propose a bottom-up strategy for coordinating complex surface design that starts with local component-based rules and transformations to inflect a global change in surface curvature . This is in contrast to more traditional optimization methods that frequently privilege a top-down methodology where global surface change is organized into a family of non-standard discrete components post-facto.

The success of the surface design development process heavily depends on the design medium that the designers use. While the digital model helps to visualize the design intent, comparatively, it is reported (Sun, Lei, et al. 2014) that the physical model provides designers higher accuracy and acceptance in terms of spatial understanding and usability. Given the higher uncertainty and complex geometric property of freeform surface design, it is necessary to develop a new physical design media and shape-changing interface. This new interface can change surface geometry morphology based on the designer’s intent and manual, analog input to facilitate a designer’s spatial understanding. Various projects have explored and employed the ability of shape-changing interfaces in the domain of design. However, the biggest challenges of shape-changing design media lie in two aspects: First, most of the precedent works rely on electronic parts to power and control the shape, which limits a designer’s degree of interaction while conducting real-time design activities and updates; Second, the possibility of an interactive design process is subject to the limited movement of a shape-changing interface since most such assemblies are implemented through the actuators' linear or rotary motion. In this case, the new shape-changing interface for surface geometry design proposed in this paper needs to satisfy two features: A, the changing shape should not be driven by any electrical actuator; B, the physical interface needs to be easily deformed and programmed to any degree of freedom. In this chapter, combining the two above expectations, the author explore two methods as a new design foundation.

Our work focuses on the surface geometry bending mechanism. Specifically, the author are inspired by the epithelial tissue's bending mechanism. Epithelial tissue that is constituted with tightly packed epithelial cells envelopes into complex cavities to cover the majority of the surface area of living being's bodies, including skin, organs, and blood vessels. Such features allow epithelial cells to deform and proliferate locally and adapt to different environmental conditions globally. This intriguing feature motivates us, as design researchers, to develop a novel design strategy that generates complex surface structures in both digital and physical surface geometry through the combination of individual units. Scientists (Gómez-Gálvez, Pedro, et al, 2018) recently discovered that epithelial tissue’s deformation capability is contributed to the emergence of scutoid geometry at the micro-scale. And scutoid is an ideal 3d package solution for a highly curved environment. Scutoid, as three-dimensional geometry, bridges two parallel surfaces and is often described as a combination of the frustum and a prismatoid as it has a different number of edge segments on its basal surface and apical surface. One of this paper’s targets is implementing the deformation process

of epithelial tissue at a material scale and developing physical prototypes to implement the deformation. Then to eventually take the prototype as a design media to facilitate design activities.

By integrating the bio-inspired surface geometry bending mechanism, the author aim to use smart material to fabricate the physical surface geometry that can change its shape autonomously based on external stimulation without any electric input element. Such work is implemented through 3 dimensionally printing with shape memory polymer. The long-term development of 3d printing technology has dramatically reduced the application threshold for designers. In recent years, many design practitioners and researchers utilize smart materials' characteristics such as shape memory further to expand the concept of 3D printing into 4D printing. Our work in this chapter focuses on taking advantage of the material behavior to execute the surface geometry bending process and how the author can adequately fabricate such surface geometry.

Related Work

There are related exploratory works that have been done in both trajectories recently.

From the perspective of bio-inspired surface design, by investigating the biological principles in a micro-scale, Sabin and Jones (2008) investigated code-driven parametric design method, models, and tools to gain new insights into how nature deals with dynamics, environment, and feedback within cell and tissue structures, and in turn utilize those into architectural scale. Although the scutoid was discovered and termed in 2018, a few precedents work that links scutoid/epithelial cell with design and fabrication can still be found. (Tsikoliya, Shota, et al.2021) discussed architectural structure molding method that is based on differential growth of epithelial tissue. Inspired by epithelial tissue's deformation and the emergence of scutoid geometry, the authors proposed a continuity folds modeling approach and fabrication planning with concrete for architectural scale experiments. However, the deformability of epithelial cells can only be fully demonstrated over time. Instead, the work they proposed results in a set of static architectural elements, which the author consider do not make full use of epithelial cells' deformability. Some other works paid attention to the cellular structure of epithelial cells. Due to scutoid's structural stability, Dhari, Rahul Singh, and Nirav P. Patel.(2021) proposes a 3D-printed Scutoids-inspired cellular structure for use in lightweight sandwich structure designs that copes compression loads. The structure was made of aluminum alloy and is reportedly to be able to absorb more energy than a regular compressive structure. The work scientifically demonstrates that, from a structural point of view, the bent epithelial tissue and scutoid shape are highly rational. Yet, the work leaves a blank space on scutoid fabrication with soft material. Teng and Sabin (2020) claimed a novel masonry shell system embedded with the geometric features of scutoid. Inspired by scutoids having different adjacent cells on different faces, all bricks in the mentioned masonry shell interlocked each other without external joints needed. The work took one step out in terms of using scutoid geometric features as a standing point of surface design.

As the author are targeting to implement the deformation process at the material scale, the author examined material and methods that can be potentially used to fabricate Epithelial cell-inspired Surface Geometry. Specifically, the author evaluated precedent works of three-dimensional printing objects with active material and 4d printing process. In a lot of Tibbits 's work (2012,2014) , the material is pre-programmed and equipped with actuating abilities and reacting ability. It allows the material to self-deform over time when exposed to external stimulation and morphs into the desired 3d shape. Lee, A.Y., An, J. and Chua, C.K., (2017) comprehensively summarizes the fabrication method of reversibility of 3D-printed shape-memory materials such as depositing with additives or use multiple materials.

Method

Computer Simulation

We firstly investigate the deformation of epithelial cells thorough a scientific literature review. Referring to research in cell and molecular biology, the author learned that the global deformation of the epithelial tissue is proceeded by the position rearrangement of individual epithelial cells in the tissue. the author then computationally simulate the bending process within the tissues through a generative modeling methodology. The simulations demonstrate the emergence of scutoid geometrical shapes within the epithelial cells. the author conclude that the cell rearrangement is achieved through cells revolving and contracting. These morphological transformations create a triangle face on the volume, transiting the extra edge on the apical surface to the basal surface through an extra vertex.

Our previous work has investigated that scutoid's extra edge segments on one of the surfaces merge into vertexes in the intermedia level to equalize the edge segment's number on another surface. the author also discovered that epithelial tissue's curvature positively correlates to the length of the boundary between two adjacent epithelial cells as scutoid cells are generated through cells revolving and squeezing. Bending within the Epithelial tissue causes different motion tendencies on epithelial tissues' apical and basal surfaces. When the epithelial tissue is bent, the adjacent cell surfaces will squeeze each other where the curvature is positive and separate where the curvature is negative. The geometric centers of adjacent cells' profiles move towards each other where the curvature is positive. The closer the two cells' geometric centers are, the greater the squeezing between the two cells is, resulting in a longer boundary line. In summary, the greater the curvature of the epithelial tissue, the longer the boundary between two adjacent cells at the curved position. (Figure 23)Cells in highly curved global epithelial tissue deform along the apical-basal axis. The deformation process results in cells having different neighbors in their basal and apical surfaces.

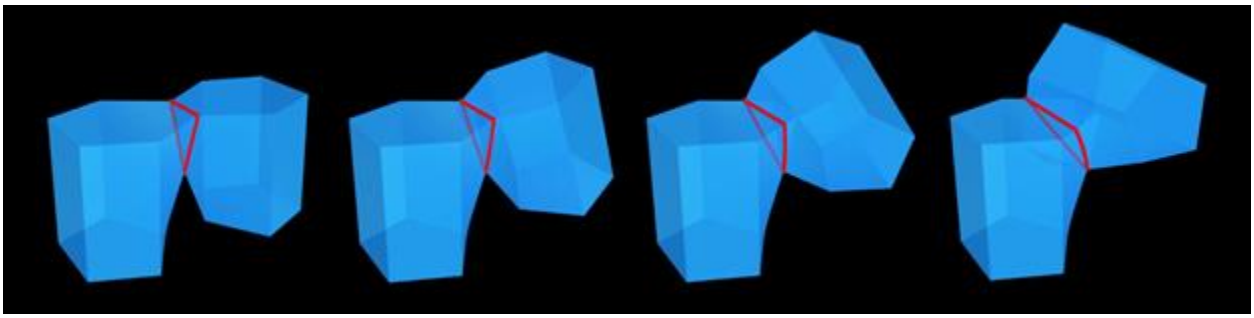


Figure 23 cell rearrangement is achieved through cells revolving and contracting

Physical Prototype

Based on the above findings and geometric relationships, when designing a physical surface geometry with scutoid components embedded, the author look into the dialectical relationhips between global surface geometry and local units. the author hypothesized that if the author can change the length of the boundaries between any pair of adjacent cells within a given model, the author can then change the curvature of global surface geometry. Moreover, when the length of the boundary between any pair of adjacent cells is programmed, the global surface's overall morphology can be precisely controlled. The hypothesis is verified in a later context.

We identified two appropriate materials to be the most suitable deformable material through a series of material experiments: silicon and Shape Memory Polymer, as both materials are reversibly deformable and can translate the cells' behavior to a greater scale. In addition to the findings on the epithelial cells' geometric features that the author discuss, the physical prototype and material studies should also follow the cell's geometric constraints. Study (Rupprecht, Jean-Francois, et al. 2017) investigated that cells compete for space under geometric constraints (Figure 24). The cell's three-dimensional morphology and packing within epithelial tissues is significantly impacted by geometric constraints. According to the literature, the geometric constraints include expansion force from the cytoskeleton and adhesion between cell membranes.

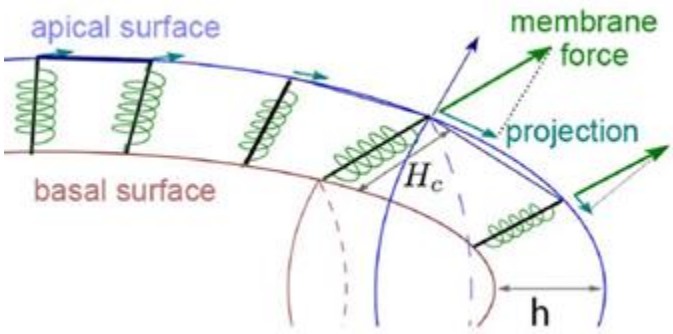


Figure 24, Image credit to: Rupprecht, Jean-Francois, et al." Geometric constraints alter cell arrangements within curved epithelial tissues." Molecular biology of the cell 28.25 (2017): 3582-3594.

Based on these constraints, the author further developed a set of programmable surface geometry prototypes to illustrate the global surface geometry deformation at a material scale using 3d printable shape memory polymer filament and silicon. The heat transfer of liquid-crystalline domains causes the shape memory effect. After the 3d printer heats the polymer to the isotropy state, the nozzle starts depositing the filament layers into the specified 3d shape. Polymer is initially programmed at this moment. As the material remains at a high temperature (temperature A), the printed 3d shape stays in its initial state A and remains soft. When the printing is done, and the 3d shape cools down to room temperature, it becomes firm. Following reheating the shape to temperature B, the shape becomes soft again. The designer can deform the printed 3d shape into state B on demand. After cooling down and changing the environmental temperature between temperature A and temperature B, designers obtained one object with two deformable morphologies at two different times. The forming of silicon in this project is fabricated through casting after the author 3d print the formwork with PLA.

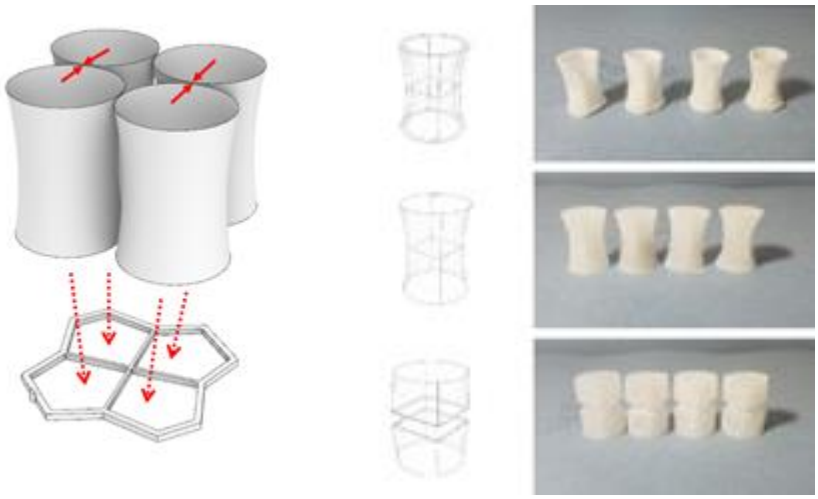


Figure 25 the author developed a set of cell units to be inserted into a constrained frame

Aside from the digital simulation mentioned earlier, two types of deformable surface geometry are then designed and built for serving two purposes: First, the physical prototype verifies if scutoid cells will actually emerge at the local unit when the global surface curvature changes. Second, dialectically, the prototype also verifies if the global surface will be bent when the local units deform into scutoid cells. Both types of prototypes share a common design: To analogically translate the expansive force from the cells' cytoskeleton, the author developed a set of cell units to be inserted into a constrained frame, which simulates the adhesion between cell membranes. The configuration constitutes a surface geometry while maintaining a dense packing of cells. (Figure 25)

Figure 26, silicon unit inserted into Shape Memory Polymer constraint frame.

The first prototype (Figure 26) is generated by inserting cast silicon units into a 3d printed Shape Memory Polymer constraint frame. As active material, the Shape Memory Polymer, which represents the global surface geometry, is deformed according to environmental stimulation, the thermal environment, in this case. the author designed and programmed the Shape Memory Polymer frame to stay flat as the initial state at temperature A and bend towards the apical surface when the temperature reaches B degree. The silicon units representing the local units act as a passive deformation material and are tightly packed by the constraint frame. the author then set the prototype in temperature A and have it stay flat and still. After gradually raising the temperature to B, the flat surface geometry starts bending towards its apical surface. Meanwhile, the local silicon units exhibited the expected behavior. As the global surface geometry prototype bends towards the apical surface, and all local silicon units revolve around the corresponding

horizontal axis, adjacent local units squeeze each other through their boundaries. The length of boundaries becomes longer when the globe curvature increases[JES1] . Such local units' behavior generates the scutoid shapes. This process illustrates how the curvature change of the global surface geometry impacts local cell morphology. (Figure 27)

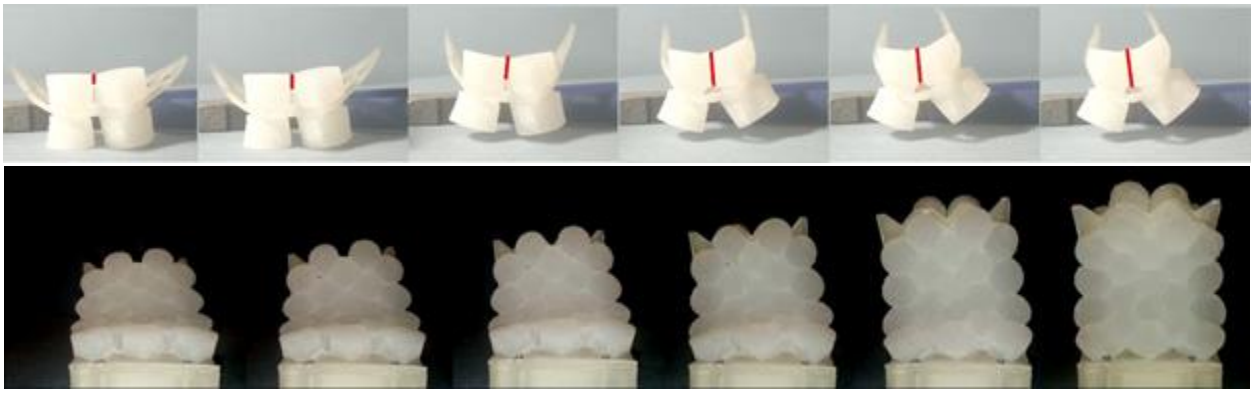


Figure 27, upper image: Two cell's tissue bent towards to apical surface, the length of the common boundary between two cells becomes longer when overall curvature increases. Photo taken from side view. Lower image: 3d printed SMP active constraint frame inserted with silicon unit, when constraint frame bent towards to apical surface, all units deforms into scutoid. Photo taken from front view.

In contrast, the constraint frame for the second surface geometry prototype is made by silicon representing the global surface, and all local cell units are 3d printed with SMP. the author firstly 3d printed all SMP local units as regular frustums with its apical and basal surface flat at temperature A. the author then programmed each SMP unit individually to deform into a scutoid shape at temperature B. Deforming units into scutoid cells means purposely changing the length of the common boundary between two adjacent units and the triangular face's size that transitions the polyline edge into a vertex to determine the expected curvature of the final global surface geometry at that unit's position. First, the SMP units are tightly packed into a global surface with the constraint frame and the surface prototype is placed in the thermal environment. Next, the SMP cell units actively deform from regular frustums into scutoid shapes with various sizes generated at the common boundary length or connection at the triangulated face. Thus, the prototype deformed from a flat surface geometry into a curved and programmed surface geometry. This prototype illustrates how local units impact global geometry deformation and morphology. (Figure 28)

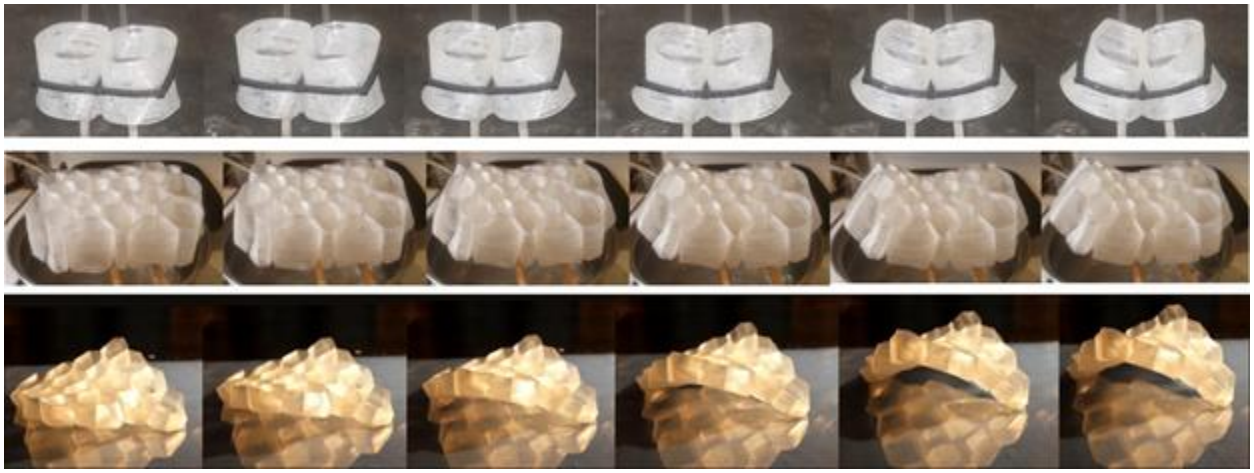


Figure 28, Upper: Two SMP cells tissue bent towards to apical surface. Two cells actively revolving and contracting towards each other results in constraint frame bent. Middle: SMP cells actively deform themselves into scutoids to bend globe surface geometry towards the apical surface. Lower: SMP cells actively deform themselves into scutoid.

To conclude with the above observations from the physical prototype experiment, in a surface geometry (epithelial tissue) constituted by smaller units (epithelial cells), the global surface and local unit's morphology influences each other. To activate such surface geometry, the author either program the overall global surface curvature or program the individual units' morphology.

Application

The work the author has completed thus far has illustrated how the author combine the geometric features of epithelial cells with smart material additive manufacturing to achieve bioinspired programmable surface geometry. Importantly, this work leverages local transformations at the cell/component level to achieve global changes to surface curvature.

However, such programmable surface geometry is still not an ideal shape-changing interface for design applications. As a physical interface, interaction design and responsivity has not been emphasized in this latest set of experiments. As mentioned previously, the purpose of using a physical model in this design research is to demonstrate at the material level, a bottom-up strategy for bio-inspired complex surface design that leverages local component-based rules and transformations to inflect a global change in surface curvature. This construct also benefits the designer through an intuitive methodology that links responsive material interface with complex component-based surface design. The author proposes two ways that the bioinspired programmable surface geometry can be merged into this interact with the designer: First, the designer interacts with the surface geometry by programming its target morphology. When a designer deliberates a complex surface design, it is easier for the designer to shape the desired morphology by manually stretching or bending the material assembly than by modifying it in a digital environment.

Second, the author designed an embeddable flex sensor to attach to the physical surface geometry (Figure 29). When the designer shapes the surface on demand, the flex sensor, which consists of conductive ink, bends with the physical surface. The bending behavior forms a flexible potentiometer in which resistance changes upon deflection. Higher curvature on the physical surface results in higher resistance on the flex sensor[JES2] . A microcontroller (Arduino) connects with the flex sensor and converts the resistance to curvature data in real time, then transmits data to computer modeling software (Rhino/ Grasshopper) to digitize the physical surface.

The above two ways create a feedback loop that allows the designer to incorporate his/her design intent directly with the physical surface geometry. Adjustments to the morphology are immediately input as updated data in the digital modeling environment. (Figure 30)

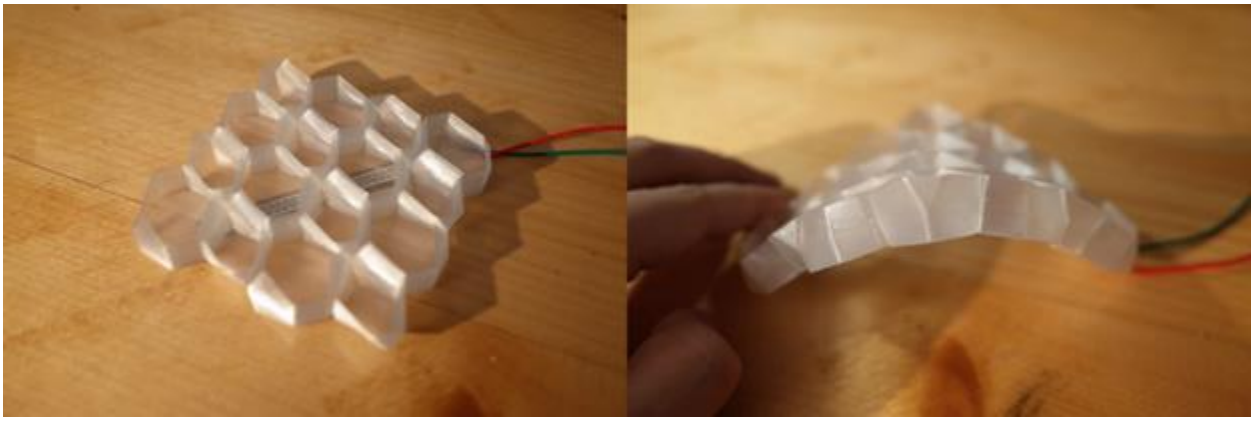


Figure 29, flex sensor attach with the physical surface geometry

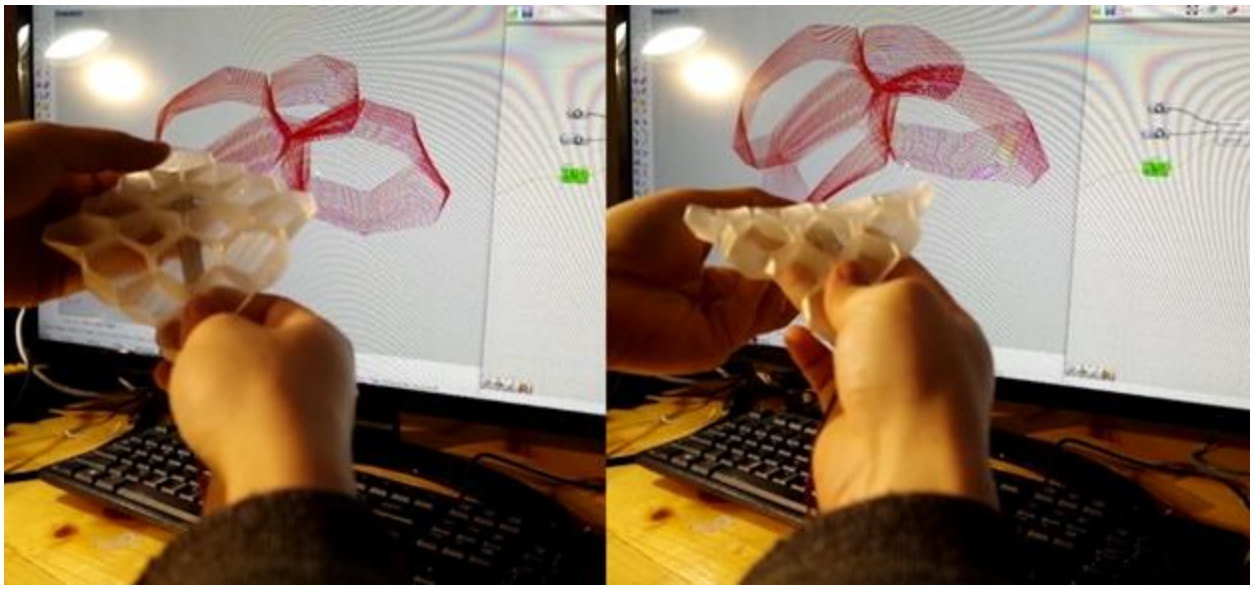


Figure 30, The designer manually shapes the physical geometry, and the flex sensor captures the curvature data to digitize the surface.

Utilizing the proposed 4d printing method in the architectural geometry design stage plays a unique role. As a physical representation, the 4d printed model is the outcome of the design intent that presents the design object's geometric feature through a tangible agent. The deformation process helps the designer to establish a better understanding of the dialectical relations between global geometry and local units. In the meantime, similar to Gaudi's extensive use of hanging chain model to simulate the double-curved surface when he was designing the Sagrada Familia, the forementioned 4d printed prototype attached with flex sensor acting as a tangible user interface that allows the designer to conduct an intuitive form-finding process to inform the design interactively. For example, when testing the physical prototype's coping efficiency with compression, the 4d printed surface geometry can be rapidly and manually deformed into any curvature for testing. In addition, the deformable nature allows it to be used multiple times until the best curvature of the surface geometry is found instead of fabricating multiple static prototypes. The flex sensor digitized all curvature data from each time of testing for further use. Such process saves resources for fabricating multiple prototypes and yet enhancing the efficiency of design development.

Conclusion

The author proposed a materially-based bottom-up strategy for coordinating complex surface design that starts with local component-based rules and transformations to inflect a global change in surface curvature. the author has summarized the work thus far into two aspects: 1.) Inspired by the deformation mechanism of epithelial cells, the author proposes a new design strategy for generating complex surface geometry from transformable individual units; 2.) the author developed a new 4d printing method, which allows the surface geometry to be programmed on demand and to emulate the generative and bio-inspired design model analogically. While the author has not yet conducted a user study to test the functionality of our proposed design interface, the work demonstrated in this paper focuses on the desktop scale as a speculative and interactive design tool where changes in a responsive and physical material assembly have a direct digital output to a Rhino modeling environment.

In this project, the author outlines the methods developed to prototype a bio-inspired programmable surface geometry as a design interface concept. the author has successfully satisfied the two initial requirements that the author proposed: 1.) the author eliminated the electrical actuator, a commonly used driver in actuating a programmable surface to inflect changes to surface geometry. This is achieved by incorporating responsive 3d printed shape memory polymer and through a design methodology that leverages the deformation mechanism and geometric features of cellular epithelial tissues. the author also developed a bottom-up design methodology to work with surface geometry that can be deformed and programmed to any degree of freedom, at local and global levels, by embedding scutoid geometric features. Beyond proposing the programmable surface geometry as a design interface, the work the author have completed provides a novel and interactive strategy for designing surface geometry.

CHAPTER 3

Interactive Fabrication of Complex Bioinspired Surface Geometry

Abstract

The nature of bioinspired surface geometry is often morphologically complex, adaptive, and seemingly chaotic. This chapter mainly outlines the fabrication strategy that matches the above nature of bioinspired surface geometry by demonstrating several newly developed fabrication tools that are aiming for scutoid embedded surface geometry, including 1) Clay Extruding Robotic Arm 3.0 (CERA III), a two-Step Mechanic Piston with Conventional Screw Based Clay Extruder. 2) Scu-bot bed, adaptive printbed that is constituted by a swarm robot system that can bend based on demand. 3) Pinbed, an Interactive and Adaptive 2.5 D Printbed for Robotic Additive Manufacturing.

Introduction

When building surface geometry at an architectural scale in reality, the standard method is to compose prefabricated individual units into a whole. It is a reasonable means when constructing a regular box-like architectural geometry. A limited number of modules is sufficient to cover all units. However, as the author pointed out in the thesis introduction, when the morphology is increasingly complex, especially when the surface geometry is designed through a bio-inspired method, the fabrication of global surface geometry composed of local units is often challenging. Using traditional manufacturing methods takes a lot of time and resources to prepare working files and molds. Even though there are many optimizing methods to modularize units to reduce the cost of working file and mold preparation, it somehow sacrifices the geometric advantages brought by imitating nature to a certain degree.

To this end, the author aims to develop a novel additive manufacturing method that can adaptively fulfill the geometric requirement of different unit shapes with minimum effort and avoid the repetitive work of preparing a large number of processing files making molds. In the meantime, conventional additive manufacturing usually needs a support structure to implement complex geometry, which consumptions a large amount of material and fabrication time. An ideal setting for additive manufacture setting should find ways to eliminate the supportive structure.

Generally speaking, two essential elements constitute an additive manufacturing device: the extruder and the printbed. The extruder is defined as a part that ejects material in the liquid or semi-liquid form to deposit it in successive layers within the 3D volume. The printbed is defined as a surface where an extruder deposits down the materials. In a conventional additive manufacturing setting, the extruder is moved in space along the cartesian coordinate by three perpendicularly placed axes. In some advanced cases, the extruder is moved by a 5 or 6 axes robotic arm which provides more degree of freedom. In contrast to the dynamic nature of the extruder, the printbed needs to be level and flat, and being stable during the manufacturing process to successfully produce layers of material in filament form.

In this chapter, the author proposes a modified additive manufacturing strategy by improving the mentioned two elements: the extruder and the printbed in order to support the fabrication of bio-inspired surface geometry, specifically, the scutoid embedded masonry shell.

Building Extruder

Two Step Mechanic Piston with Conventional Screw Based Clay Extruder

*(*This part of the work has been submitted to ACADIA 2021 conference for blind review. The author contributes to designing and manufacturing hardware of the extruder and programming the control software, but the work presented here cannot be completed without closely collaborated with Eda Begum Birol, Mahshid Moghadasi, Madeline Eggers, Karolina Piorko, Kevin Guo, Alexia Asgari, Veronika Varga, Jenny Sabin)*

CERA III is developed based on the previous generation of robotic clay extruders in Jenny Sabin Lab. The purpose of CERA III development is to convert a industrial robotic arm (ABB IRB 4600) into a robotic 3d Clay printer to fabricate scutoid brick to compose a large-scale scutoid embedded surface geometry.

In CERA III, the extruder utilizes a stepper motor for the mobilization of clay. A clay reservoir is manufactured from a hollow, 6 mm thick aluminum tube with 115 mm inside diameter. Power and clay movement is derived from the stepper motor (Nema 32) , which eventually transfers movement to a steel lead screw with a custom piston attached to the end furthest from the motor. The piston is a two part disc: the surface in direct contact with the clay body is made from UMPHWE (ultra-high-molecular-weight polyethylene) plastic and is given a lightly grooved side profile, which creates “wipers” (offset by 1 mm from the original diameter of the disc). Above the UMPHWE disk, another machined aluminum disk was designed for mounting and stabilizing the leadscrew with an embedded connection fixed via two set screws. The lead screw is passed through the gearbox, which will move upward and downward to control the movement of the clay body.

The extruder design is further developed by introducing an auger to facilitate the clay extrusion at the nozzle. This mitigates the pressure created by the plunger. Hence, this iteration is defined as a “two-step assembly” consisting of feeding and extrusion systems. This decreases the strain on the single motor, controls the high-pressure differentials, and implements a high precision start/stop sequence that was not achieved in previous iterations.

While the core components of the system remain the same, a number of changes are made to decrease strain and failure due to internal pressure and allow for a greater range of print resolution and scale. The diameter of the lead screw is increased from 12mm to 38 mm, and the gear-box is replaced by a screw jack.

NEMA 32, a hybrid closed-loop stepper motor, drives the Worm Gear Screw Jack and converts the motor's rotary motion to linear motion on a 38 mm diameter lead screw. Worm Gear's 1 to 12 reducer ratio creates sufficient torque to overcome the resistance from compressing the clay with less moisture. In order to

achieve a high range of extrusion speeds while maintaining high torque for smooth extrusion, the extruder designer requires a real-time control of the rotary motion speed of the clay compressor motor (NEMA 32). Due to shortcomings of a regular stepper motor with an open-loop control system, such as losing steps during development, a hybrid stepper motor with a closed-loop control system is chosen, as it meets demand for continuous extrusion. Additionally, a hybrid closed-loop stepper motor stops on its own in case of excessive resistance, which prevents damage to the system.

The auger that extrudes the clay from the hose, attached at the robot's sixth axis, is driven by a regular NEMA 23 stepper motor. A 1/4" fiberglass hose connects the aforementioned two parts. Contrary to the clay compressor motor, this motor has fewer constraints. It does not negotiate high resistance generated through clay compression, thus a regular stepper motor with high torque is sufficient.

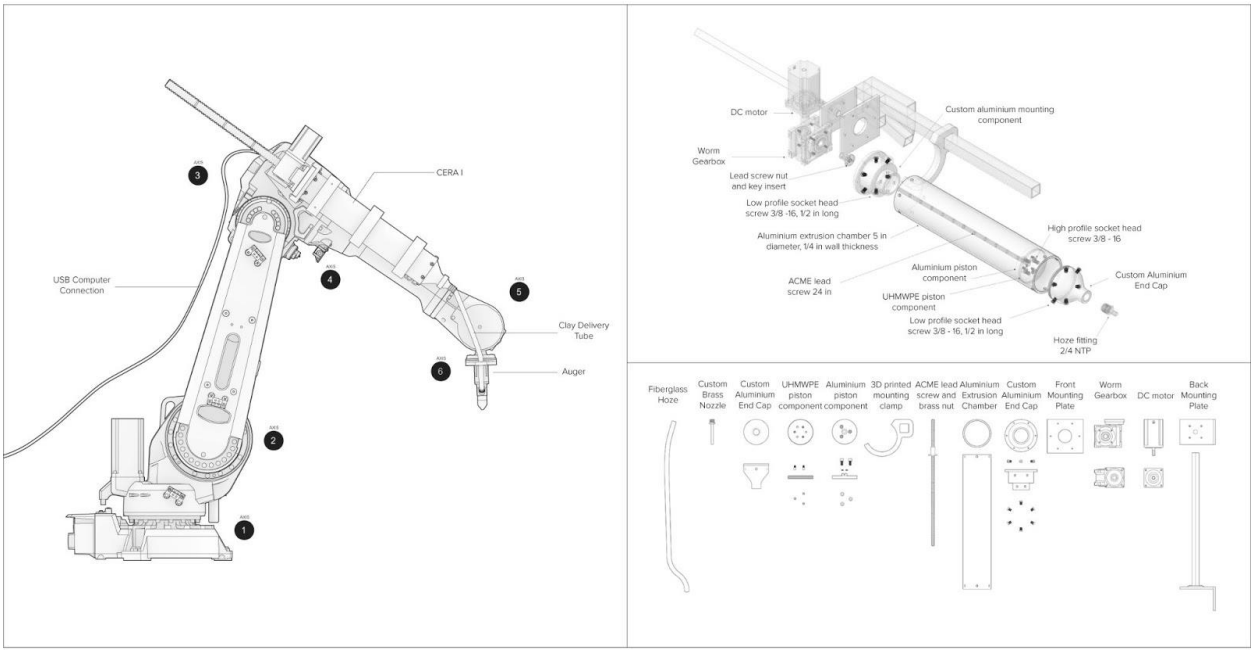


Figure 31: Third and final generation of Clay Extruding Robotic Arm (CERA) novelly developed and assembled in Jenny Sabin lab. The figure demonstrates the assembly mounted on the ABB IRB 4600 robotic arm (left), an exploded axonometric view demonstrating the assembly (upper right), and an exhaustive drawing of all assembly parts used (lower right).

Electronic Control

A command computer is used to control the robotic arm, the bowden extruder system, an external Arduino board, and two stepper motor drivers. The Arduino board is connected to the computer through a serial port for receiving data. (Figure 32)

Both motors used in our system are controlled by electrical pulse signals. The firmware loaded to the Arduino board converts electrical pulse signals into corresponding angular or linear displacements. Due to the pulse control, the angular movement and speed of the motor are strictly proportional to the number of input pulses and the pulse frequency. The angular displacement is controlled by regulating the amount of pulses to achieve accurate positioning, By controlling the pulse frequency to the speed and acceleration of the motor rotation is adjusted. By changing the energization sequence, the direction of motor rotation is changed. The worm gear screw jack amplifies torque generated to reach a certain level to extrude the clay from the clay reservoir and into the hose.

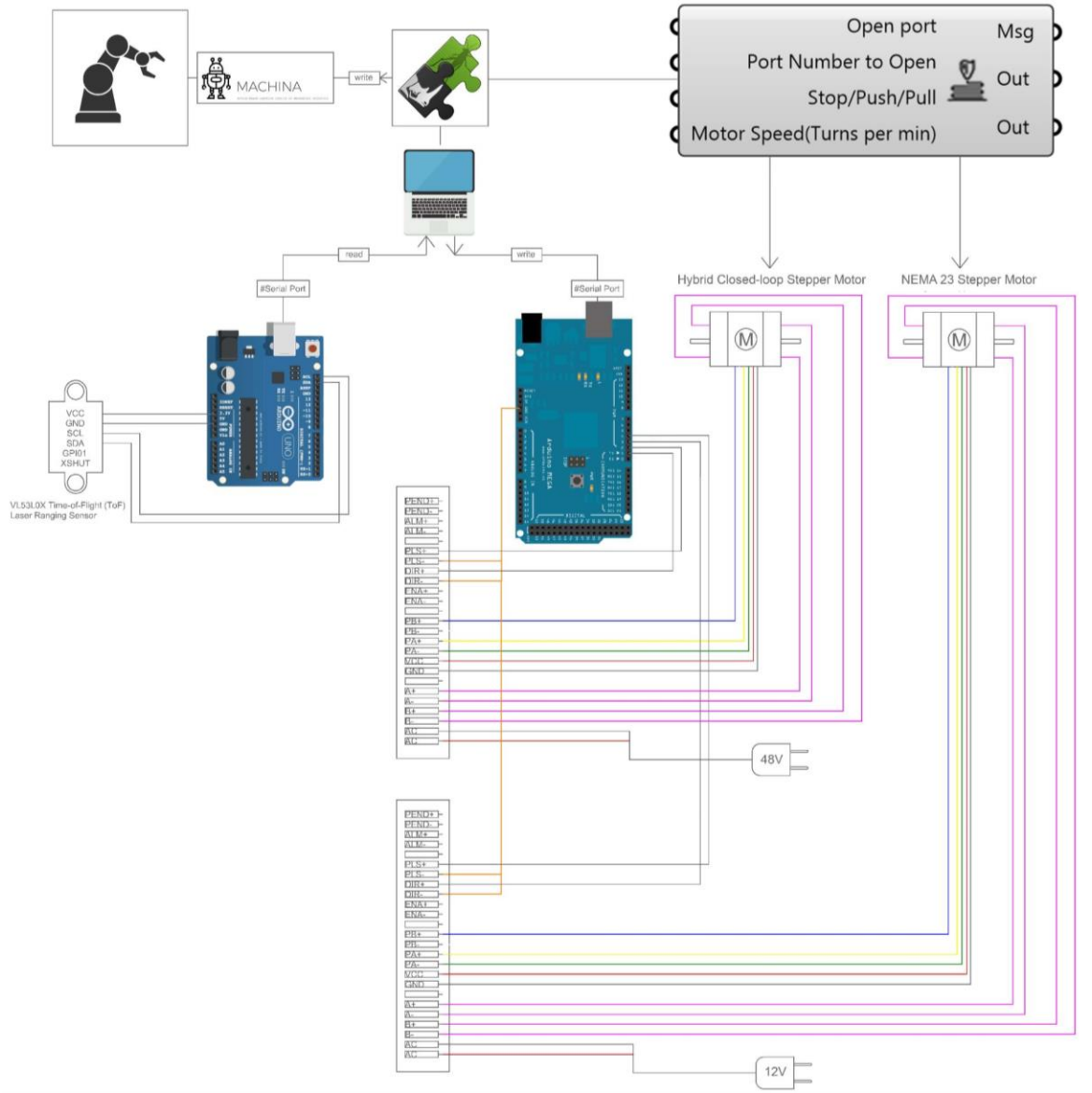


Figure 32, electronic wiring scheme of the third generation two step CERA,

Parametric Modelling for Effective Clay Extrusion

In order to achieve a smooth clay extrusion from the nozzle and mitigate internal pressure, the volume of clay extruded per minute must be equalized to the volume of clay fed into the hose. For this, a relationship needs to be established between the two motors in the system (Figure 33).

Knowing the inner diameter (D_{inner}) of the hopper, allows the calculation of the cross-sectional area (S) of the hopper inner chamber, that is:

$$S=\pi(D_{inner}/2)^{\wedge 2}$$

The volume of clay extruded from the hopper per minute can be described as the piston lead screw linear displacement (H) multiplied by inner cross-sectional area, namely:

$$V_{hopper}=H* \pi(D_{inner}/2)^{\wedge 2}.$$

The worm gearbox drives both the piston and lead screw linearly. Each rotation of the worm gear pushes the lead screw by distance (L) (Figure 33). As the author use a 1: 12 worm gearbox to amplify the torque, each turn on the motor drives the lead screw for L/12. If the speed of the clay compressor motor is $T_{Compressor}$ (Turns per minute)(Figure 6), then the total piston displacement is:

$$H= T_{Compressor} * L/12$$

Therefore:

$$V_{hopper}= T_{Compressor} * L/12*\pi(D_{inner}/2)^{\wedge 2}.$$

As the clay is compressed into the hose, the auger drill must push out the same amount of clay through the nozzle. Therefore, the choice of the auger, especially the dimension of the auger flute and the effective helix length of the drill, will affect the rate of extrusion. In our configuration, the author use an 8mm diameter general-purpose twist drill with two flutes, where the pitch (P)(Figure 6) of each flute is approximately 10mm. The inner chamber of the auger case is 10mm wide, making the effective linear length (L_E)(Figure 6) of each auger flute to carry clay to be 10mm long.

The cross-section area of the two flutes on the auger drill occupies approximately half of the total cross-section of the 8 mm auger drill. If the rotation speed of the auger motor is T_{auger} (Turns per minute), the volume of clay extruded by the auger extruder per minute is:

$$V_{auger} =L_E*\pi(D_{auger}/2)^{\wedge 2} *T_{auger}$$

Since the volume of clay extruded every minute equals the volume the compressor hopper is feeding to the hose, the author can conclude that:

$V_{hopper} = V_{auger}$,
Namely,

$T_{Compressor} * H*\pi(D_{inner}/2)^{\wedge 2} = L_E*\pi(D_{auger}/2)^{\wedge 2} * T_{auger}$

By substituting all known factors into this equation($H=0.5mm/turn, D_{inner}=117mm, D_{auger}=8mm, L_E=10mm$), it is concluded that to achieve a smooth extrusion, the proportional relations between the auger's and compressor's motor speed (turns per minute) are approximately 22 :1. Namely, each turn on the clay compressor motor will need 22 turns on the auger motor to achieve the clay volume balance. A set of extrusion tests has verified the result.

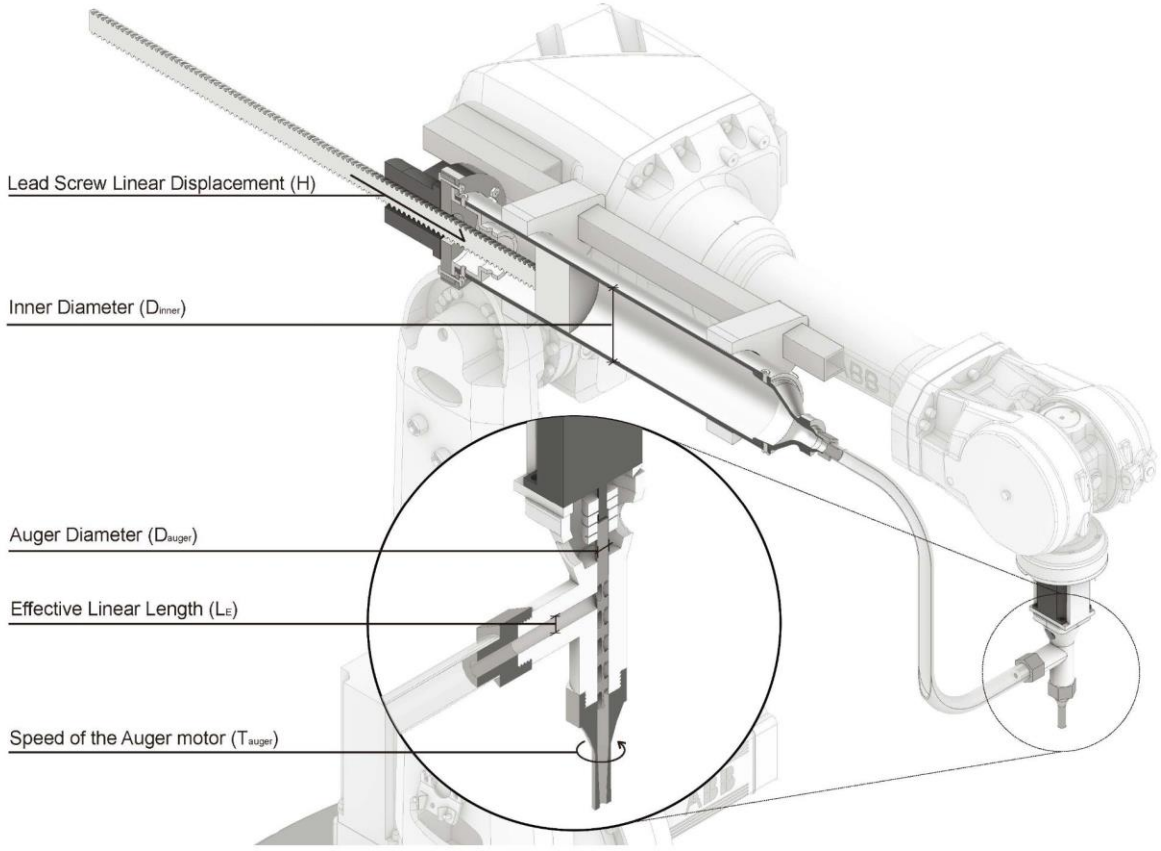


Figure 33: Cross sectional axonometric of the third generation two step CERA assembly mounted onto ABB IRB 4600. The detailed cross sectional axonometric showcases the bowden extrusion achieved by an auger assembly. The auger assembly consists of a NEMA motor driving the motion of an auger. The assembly is held together and mounted onto the robot through custom designed and additively manufactured auger support. This part also functions to feed the extruded clay from the hose into the auger system. The assembly allows the user to easily shift between various nozzles that can be screwed onto the custom 3D printed auger support.

Ubiquity in clay printing is only achievable through development of a comprehensive workflow. A modular assembly, combined with effective software interfaces and fabrication aware toolpathing are all essential components of this workflow. Hence, the author place great emphasis

on developing a designer-friendly user interface within a Rhino/Grasshopper environment. For this, a set of in-house Grasshopper components are developed.

The novel “PulseControl” component transforms integers into pulse frequency and energization sequences. By simply adjusting these parameters in Grasshopper, the user can directly control the motor's rotor speed and direction in real time. The component's input includes the serial port's index, a switch turning the port on/off, motor rotation speed (pulse frequency), and motor direction (energization sequence). By optimizing the Arduino firmware and conducting a set of motor calibration experiments, the author set the speed input of the Grasshopper component to be an absolute integer. Unlike a relative analog value, it accurately represents and controls the number of turns on the motor per minute.

Additionally, a laser distance module sensor is added inside the extrusion hopper. This prevents the collision of a lead screw piston with the extrusion hopper. The sensor reads the distance between the piston and the end of the hopper tube and informs the system to stop the motor when necessary.

CERA III has proven to be successful as it has printed large-scale scutoid bricks smoothly on a flat printbed. (Figure 34, 35)

Figure 34: Scutoid bricks are printed with CERA III

Figure 35: Scutoid bricks are printed with CERA III

Building Printbed

As the author has emphasized in the previous chapters, the surface geometry designed through the bio-inspired method is consequently complex. Such geometric complexities are reflecting in multiple dimensions. Hence, additive manufacturing only in two dimensions (X and Y) is not sufficient to cover the complexities of bio-inspired surface geometry fabrication. Therefore, taking advantage of the third dimension (Z) is essential. In additive manufacturing, the third dimension is related to the printbed. However, most conventional additive manufacturing devices are utilizing a flat surface as printbed, which leads to the limitation of geometric complexities of fabricated objects.

A lot of the practice and research project has demonstrated the possibility of using static and temporary curved surface to support the additive manufacturing of advanced geometry. Tam et al. (2015) use a robotic arm to print thermoplastics filament onto a carved wooden mold which is made through CNC-milling. Seyedahmadian et al.(2015) placed fiber by robot along a curved CNC milled wood mold surface as well. The aforementioned projects use single static surfaces as molds for robotic additive manufacturing, the method is not feasible when there are too many units with various morphology. In addition, the surface molds used in both projects are relatively simple. Ko, Minjae, et al. (2018) attached a hot wire to the robot arm and use it to cut a large number of free-from foam molds, then deposit clay along each mold to print

complex geometry. The mold cutting process requires additional work for the designer, which is lowering the efficiency of the fabrication task. Doerstelmann et al. (2014) used an inflated membrane to form the mold then deposit fiber along the membrane. The dynamic feature of the inflated membrane from this project is intriguing as it can be reused and re-inflated into other shapes.

Therefore, the author argues for efficiently printing complex bioinspired surface geometry, an adaptive mold (Printbed) is essential. This part of the thesis discusses two attempts of making adaptive printbed. While to fit with various curvatures of the bio-inspired surface geometry, both attempts are designed to be adaptive, in which the printbed can bend itself autonomously based on the geometric feature of the design object.

Scu-bot Bed

The first prototype is named Scu-bot bed, an adaptive print bed that is constituted by a scalable swarm robot system that can bend based on demand. The basic unit of the scu-bot bed is an individual scu-bot, which mimicking the bending mechanism of a four-epithelial cells cluster.

Epithelial tissue is composed of a considerable number of epithelial cells. While each cell is preserving its basic features, it also has its unique factor differentiating them. In this dialectical relationship investigation between the overall morphology of the tissue and the individual cells, the author found that the changing length of the common boundary of a pair of adjacent epithelial cells will embody the curvature of the general tissue morphology at some level. Thence, inspired by the morphological property of scutoid, the author designed a modularized wireframe robot called “Scu-bot” that imitated how four epithelial cells packed together.

Such Scu-bot is made up of a set of inner wireframe structures, a set of tensile wrapping cables, a dual-axis core, four sides holders, two servos, a pair of pulling ropes, a 9v battery, four electromagnets, four iron screw washers, and a NodeMCU board. (Figure 36) The inner wireframe structure of each unit represents a fixed-length dimension in a group of four epithelial cells, and it is wrapped around by the tensile cable that helps the structure to remain in position. (Figure 37) Besides, the author designed a dual-axis core to reflect the changing length of the common boundary of a pair of adjacent epithelial cells. The NodeMCU board has an IP address and serves as a controller to receive the signal through Wi-Fi. A command computer generates digital information such as on and off or rotation degree in Rhinoceros/Grasshopper environment and sends this data through UDP protocol to the IP address of NodeMCU. When the NodeMCU receives the data, it will control two servos mounted on the side holders, rotating the shaft to a certain degree. (Figure 38)

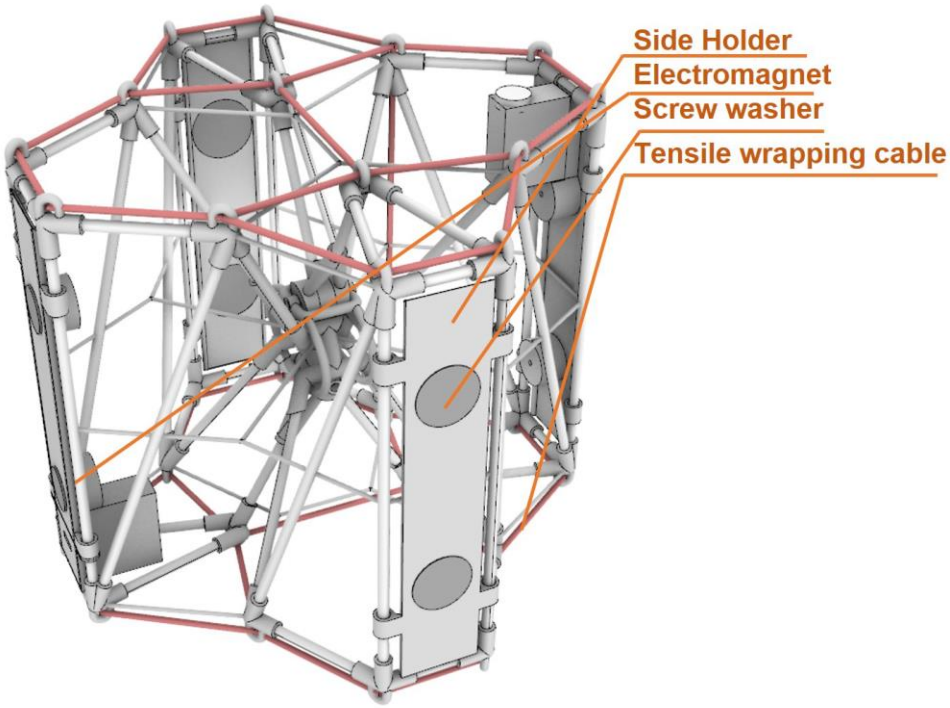


Figure 36, the configuration of scu-bot

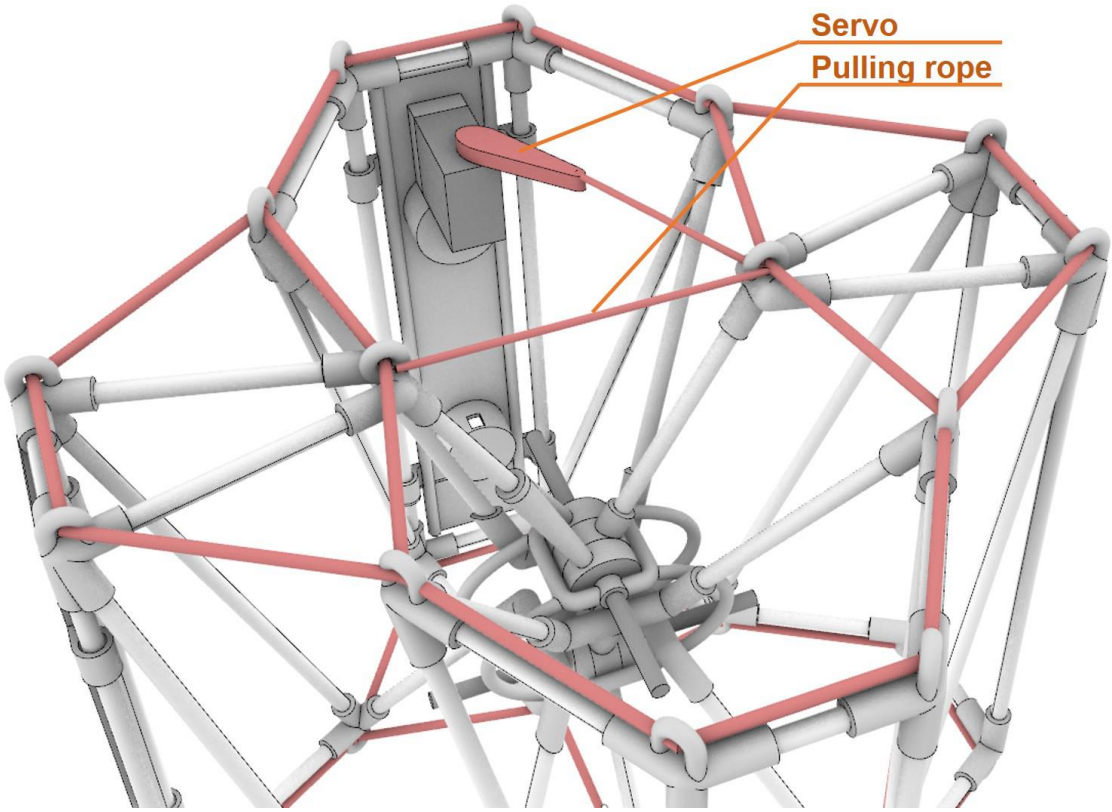


Figure 37, the tensile cable of scu-bot

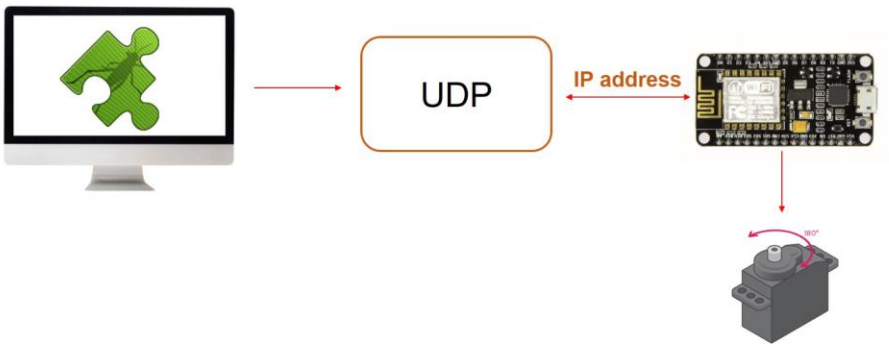


Figure 38, the communication method of scu-bot

The dual-axis core allows the wireframed structure to be rotated along with two directions. Meanwhile, the rotation of each pair of the wireframed structural unit is driven by a rope that is pulled by the servo mentioned above. The rope that connects a pair of wireframe structural units represents the changing length of the common boundary of a pair of adjacent epithelial cells. When the servo rotates, the shaft pulls the rope through a fulcrum, which shortened the remaining length of the rope between two structural units. By doing so, the overall structure bents toward to the apical surface. (Figure 39)

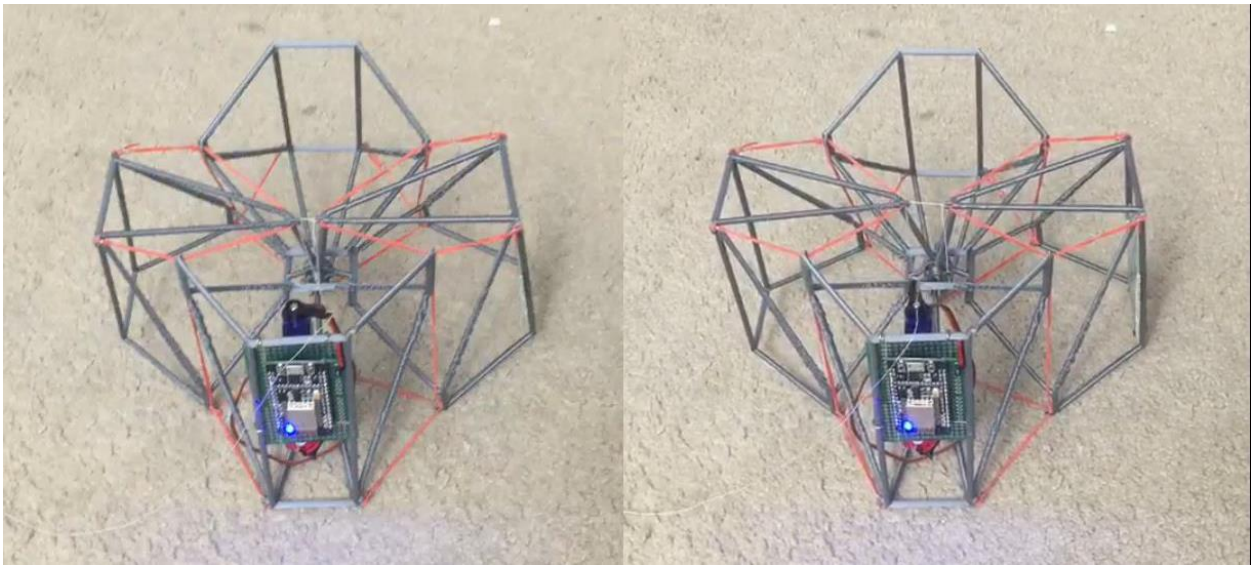


Figure 39, When the servo rotates, the shaft pulls the rope through a fulcrum, which shortened the remaining length of the rope between two structural units. By doing so, the overall structure bents toward to the apical surface.

We designed this modular robot system to be able to assemble from the individual robot into a whole as the individual robot changes the length of the connecting rope through the rotation of the servo. it causes the deformation of each module. If multiple modules are connecting through electromagnets and iron screw washers, the deformation of each module will change the overall morphology of the whole structure. The wireless communication between NodeMCU and the command computer can also occur in multiple channels. The command computer can send the data to various IP addresses, which makes distributed control of multiple Scu-bot feasible. the author can send different rotating angles in the same time to

multiple modular robots connecting and controlling the whole system to form a complex surface. Figure 41, 42, 43

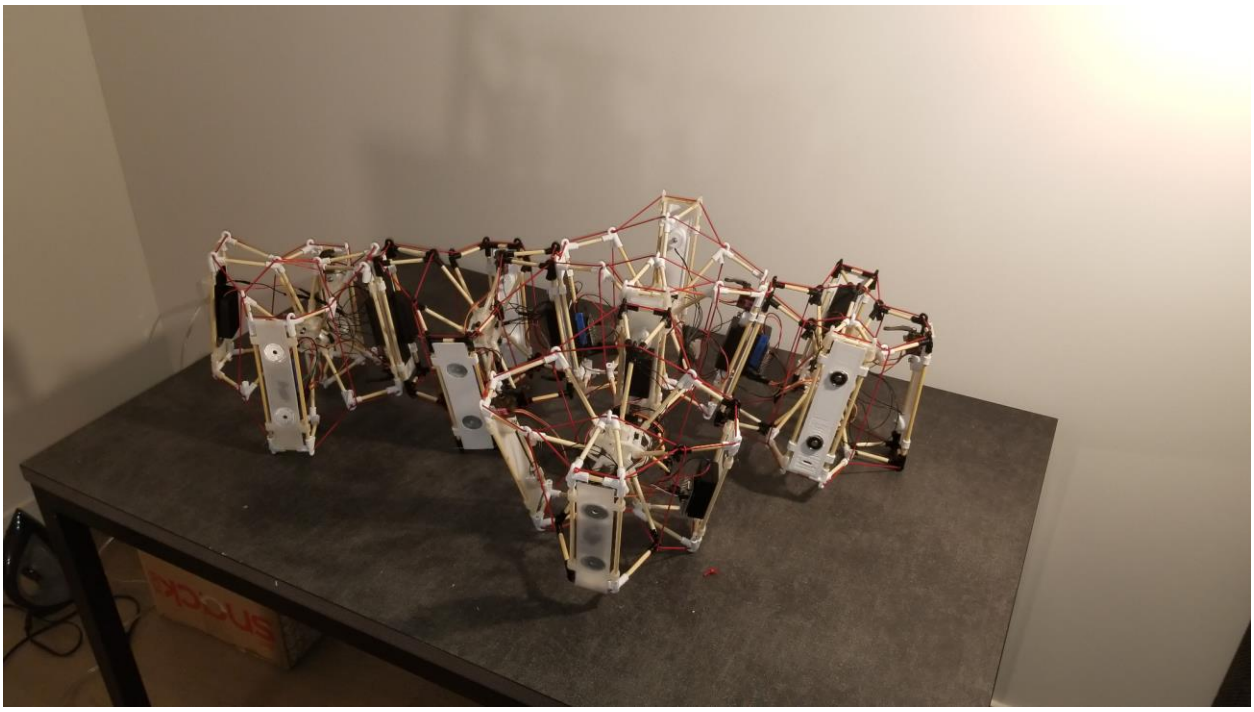


Figure 41, The physical prototype of scu-bot system

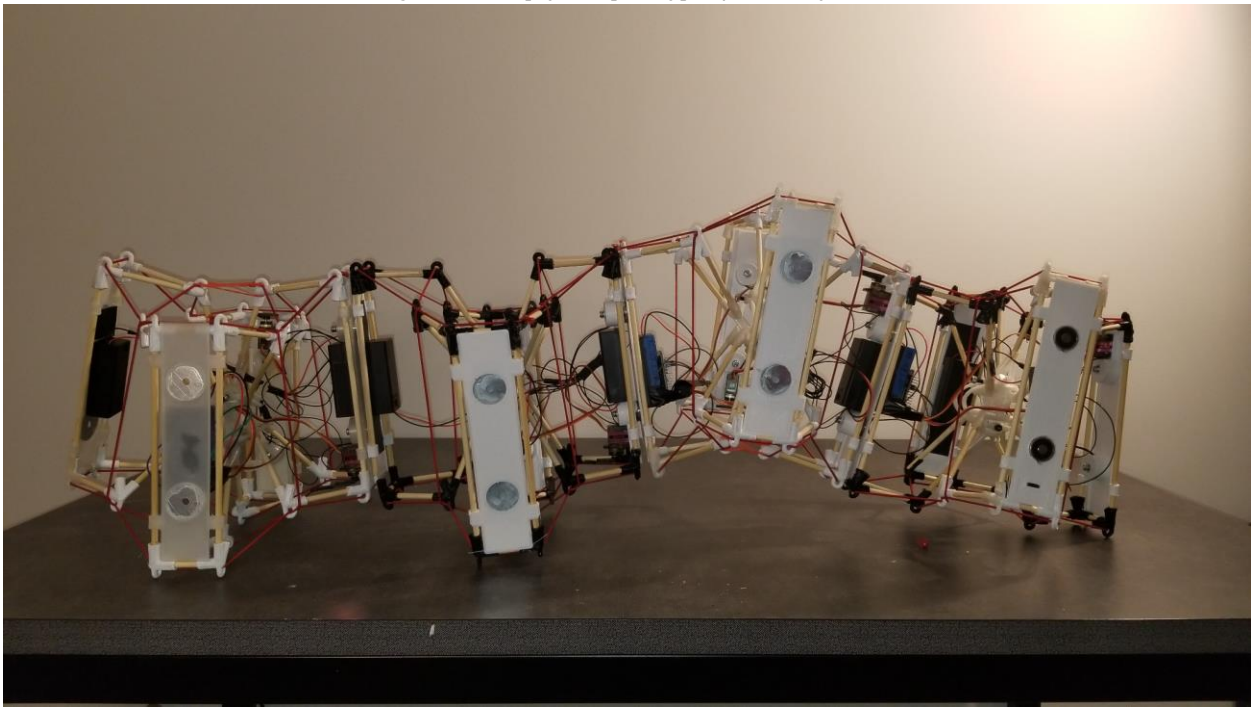


Figure 42, The physical prototype of scu-bot system

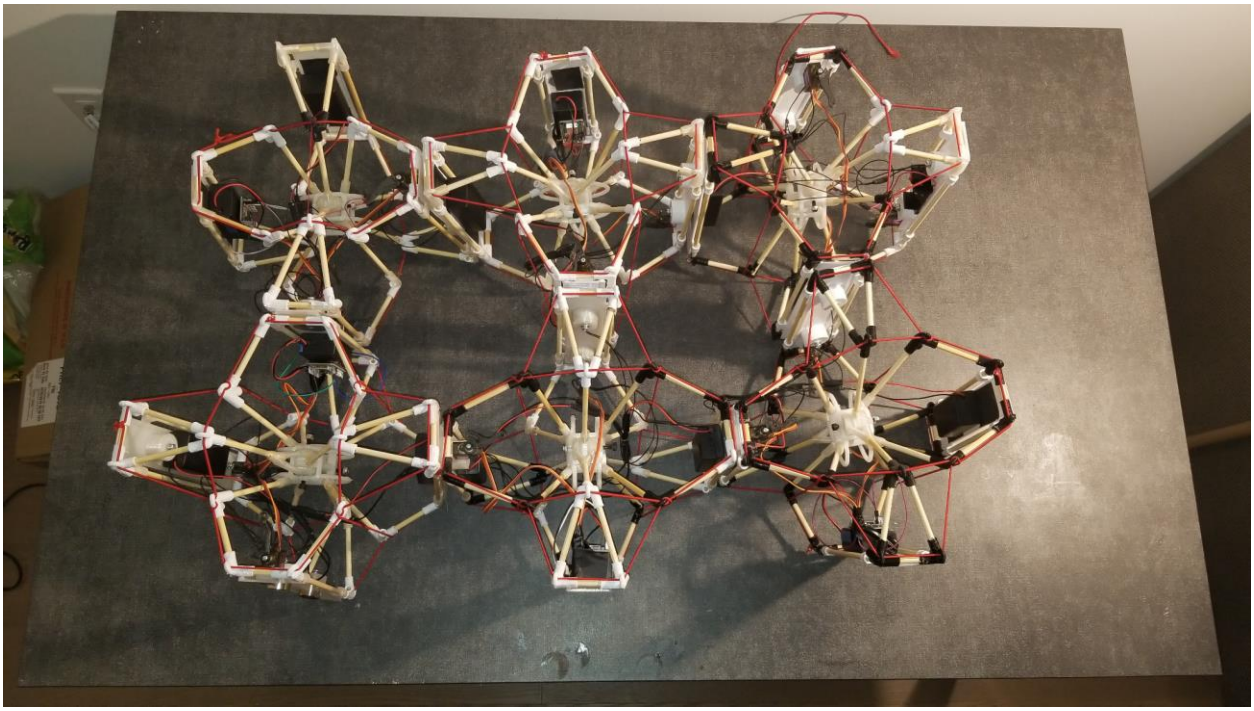


Figure 43, The physical prototype of scu-bot system

As the individual scu-bot constitutes a whole wireframe structure, the length changing of the connecting rope results in the changing curvature of overall geometry. When the structure is covered by a flexible sheet, in theory, it can be used as a dynamic and curved printbed. If the design object demands a larger printbed, the system allows a new scu-bot to be attached to form a larger structure. (Figure 44)

However, during the development process, it is found that pulling the connecting rope to change the morphology of each scu-bot requires a significant amount of torque for servo, which is challenging. In addition, when depositing material on the sheet that covers the whole scu-bot system, the weight of the material leading to the damage of the scu-bot structure. Therefore, the author proposes another attempt to build the dynamic printbed.

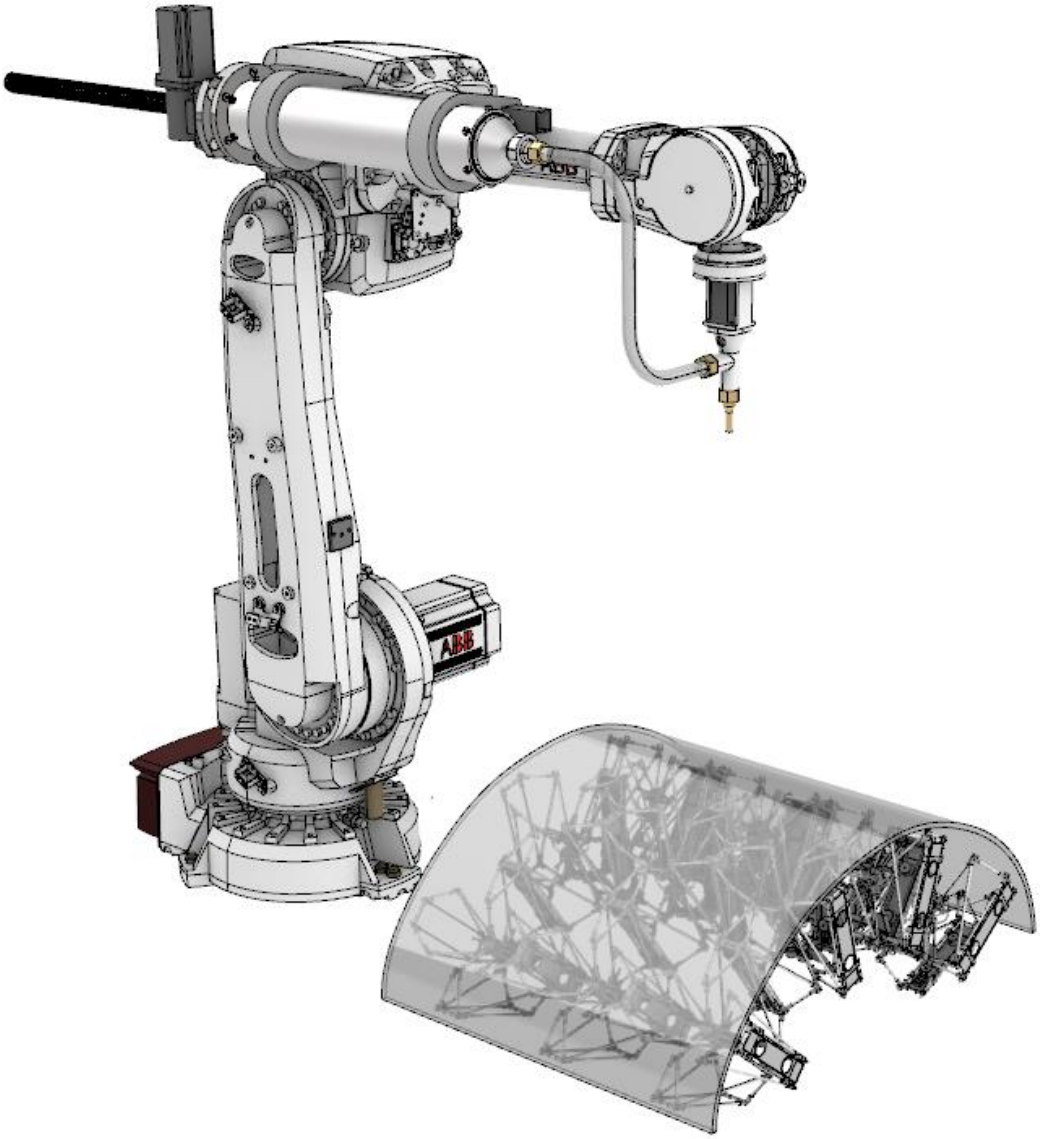


Figure 44, using the scu-bot system as a dynamic and curved printbed.

Pinbed

The project aims to address the problem by developing an easy use and reconfigurable 2.5-D printbed to help the robot print complex bioinspired surface geometry components. The limited capability of depositing material on nonplanar surfaces has been recognized as one of the universal challenges for additive manufacturing. Much work in the field has been done to solve this issue, but they are mostly focusing on modifying current FDM 3d printer firmware or taking advantage of the multiple detentions of freedom of a robotic arm to print along statically curved surfaces for small scale experimental prototyping. But there are fewer works that have been done at a large scale for producing architectural components or fail to provide an adaptive solution of generating different printing surfaces. The project aims to address the problem by developing an easy-to-use and reconfigurable 2.5-D printbed to help the robot print complex bioinspired surface geometry components.

The idea of this reconfigurable 2.5-D printbed was inspired by multi-point forming technology that has been wildly used in aerospace manufacturing and shipbuilding for precisely forming sheet metal. The author found this technology suits well for robotic additive manufacturing to generate nonplanar printing surfaces.

The building process of the reconfigurable 2.5-D printbed is conducted through two stages, the pilot study and the final prototyping.

Pilot study

During the pilot study, the author designed an Inflatable printbed as a proof of concept. The printbed designed here draws from precedent thesis research by Gergana Rusenova (ITECH University of Stuttgart. Supervisors: A. Menges, K. Dierichs, E. Baharlou, 2015) who designed, constructed, and employed a dynamic formwork for the creation of volumes within an aggregate of designed granular particles. When inflated, the balloons provide a void within the formwork which the particles aggregate around - forming structure. When deflated, the aggregate is left to stand on its own. In our design, an air compressor is connected to a manifold, branching into four individually connected tubesin turn to balloons. In the middle of each tube is a solenoid connected to Arduino controlling their on/off state. This setup allows the inflation/deflation of individual balloons beneath an elastic textile cladding which forms a lofted surface between the underlying balloons.

Physical Construction

The Inflatable pilot printbed is made up of metal struts, balloons, a spandex membrane, and a system of custom-printed balloon holders, tubes, solenoids, and an air compressor. The UNISTRUT strut channel framing is strong enough to hold the fabric in tension and allow for a fully adjustable balloon bed. The solenoids control the airflow that regulates the inflation and deflation of the balloons under the spandex. In doing this, the fabric relates to the elastically growing nature of the balloons under it and is easily replaceable after each test. (Figure 45)

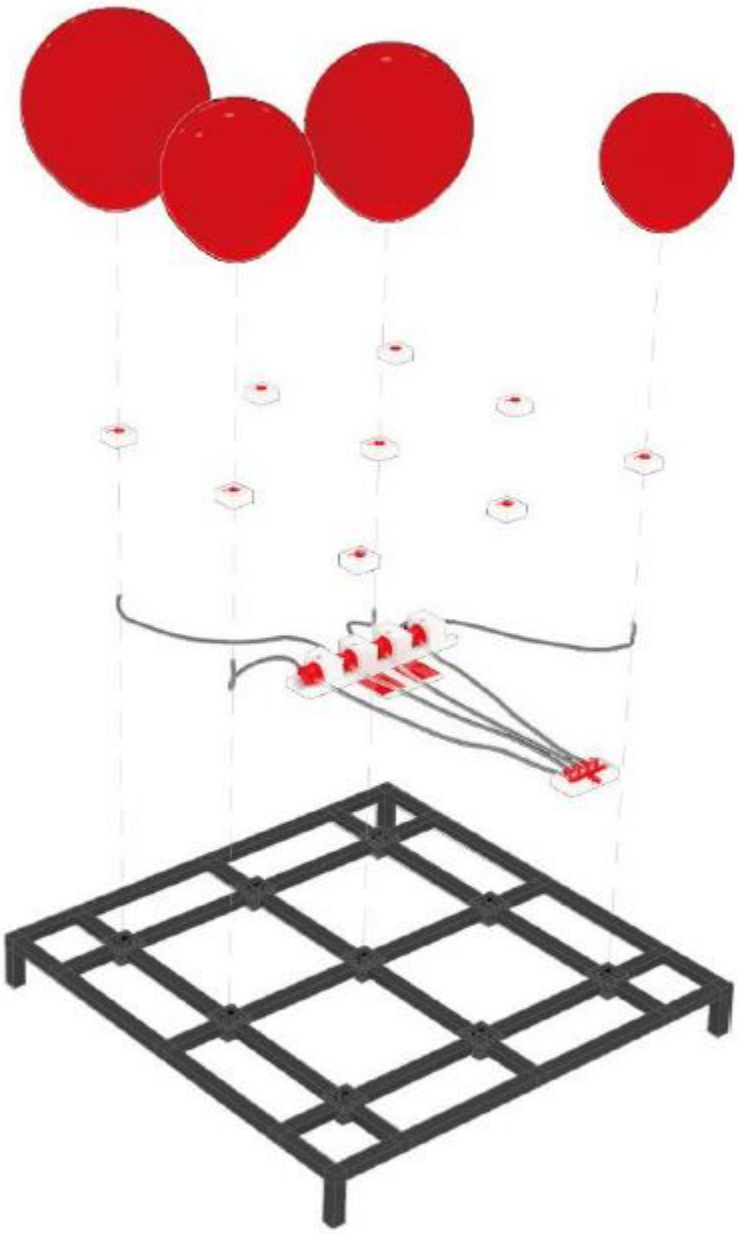


Figure 45, Physical Construction of inflatable pilot printbed

To control the solenoid valve with the Arduino, simply set the pin to a high level for the appropriate time. However, it is dangerous that the solenoid works at a different voltage from the Arduino. Connecting the Arduino and solenoid directly will lead to damages. In this case, the TIP120 transistor is needed to act as a bridge. The TIP120 allows a lower dc voltage (5V from Arduino) to be pressurized to a higher voltage (12V) and to drive the solenoid valve. It can be considered as a switch that applies current to B and allows current to flow between C and E. The diode connected to the solenoid (1N4007) allows the current to flow in only one direction. When the current is turned off, the solenoid valve attempts to continue the current. The diode feeds this current back to the solenoid valve until it dissipates. Since the transistor is doing all the heavy work in the circuit, the coding is relatively simple. When Arduino pins are set to high power, this connects the transistor collector to the emitter of the transistor, which activates the solenoid valve. When using Grasshopper to control the solenoid valve, we need to send a series of digital signals (high or low) to a serial port to activate the solenoid through Arduino.

The inflatable printbed is activated by High and Low values that’s sent from Arduino. So it can be easily controlled with a wide range of input means. For example, all solenoid valves can be controlled by a gesture sensor as the input device connected with a computer. When a user is lifting hands away from the work surface, it actives the HIGH value, which leads the balloon to be inflated, but when the user's hands are moving closer to the work surface, it actives the LOW value and leads to the balloon deflating.

Figure 46, Gesture sensor inflation control

Sensing

The pipeline for creating a responsive robotic toolpath involved five key phases: establishing a simple toolpath to be altered, reading the environment, translating coordinate systems, evaluating the toolpath curve, and refreshing the toolpath values.

Phase one (Figure 47) was to establish a simple toolpath, such as a line tracing across the printbed. The robot followed this line, continually changing its z-value to correspond to the height of the balloon it read at its current location along its toolpath. Phase two (Figure 48) consisted of using a Kinect to read the environment as a point cloud, to then send to the robot as its means of perceiving its target -- in our case, the continually varying printbed topography. Phase three (Figure 49) involved remapping the points read by the vision system -- at a random location and scale in space -- to the real-world cartesian coordinate system in which the robot would be operating. This step placed a QR code on each corner of the printbed and remapped the field of points read by the Kinect to the four corners of a box in space in front of the robot, where we could then accurately match the digital toolpath to the real bed. Phase four (Figure 50) evaluates the toolpath curve according to time and travel speed: the robot's XY position along the toolpath curve at t=0, t=1, etc. A straight, two-dimensional toolpath curve in the XY axis was generated by intersecting a point cloud with a vertical plane coinciding with the simple toolpath established in phase one, interpolating between the intersection points, and evaluating the resultant (now three-dimensional) curve along its length. This had the effect of projecting a line to an actively updating point cloud. At its current location along the line, the robot would receive a z-value read from the point cloud and update its current target z-value accordingly, actively adjusting to the height of the bed. Phase five performed the active updating of the toolpath in the robot's proprietary software. Using python scripts altered from Robosense 2.0 (Bilotti, Norman, & Rosenwasser, 2018), the z-value was sent out of Grasshopper and into a text file, to be retrieved by a different python script, which would unpack the coordinates sent out in RAPID code and replace the current z-value with the contents of the text file -- the new z-value-- at every 50 ms interval. The robot's RAPIDcode targets actively update using the open-source software OpenABB, and the robot is able to responsively change its height based on its "sensed" environmental inputs. Unfortunately, phase five ultimately had too many issues to be fruitful, so controlled, hardcoded tests were conducted using the same principles outlined above to simulate active updating and explore the gap between perceiving and reacting.

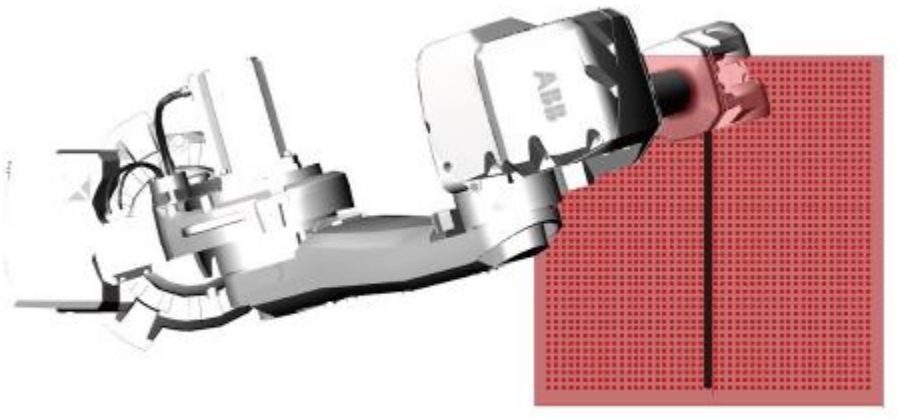


Figure 47, Phase one: simple line toolpath established for the robot to follow along a surface

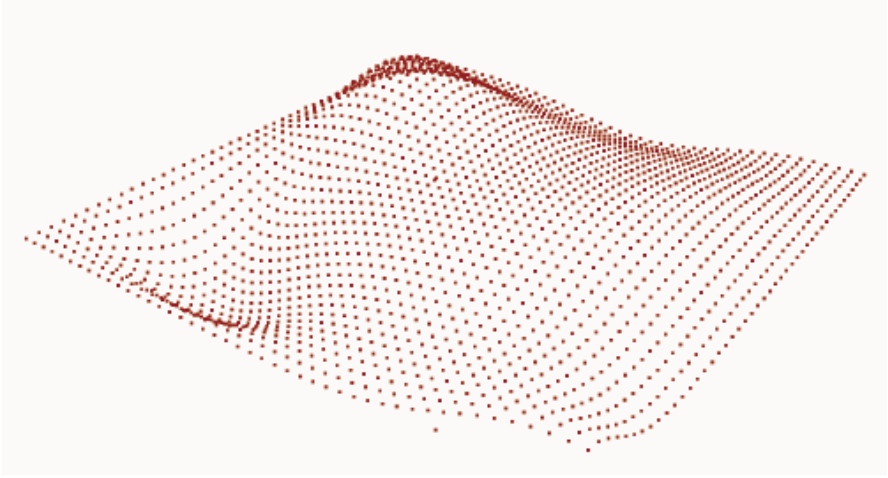


Figure 48, Phase two: Kinect reads environment as point cloud

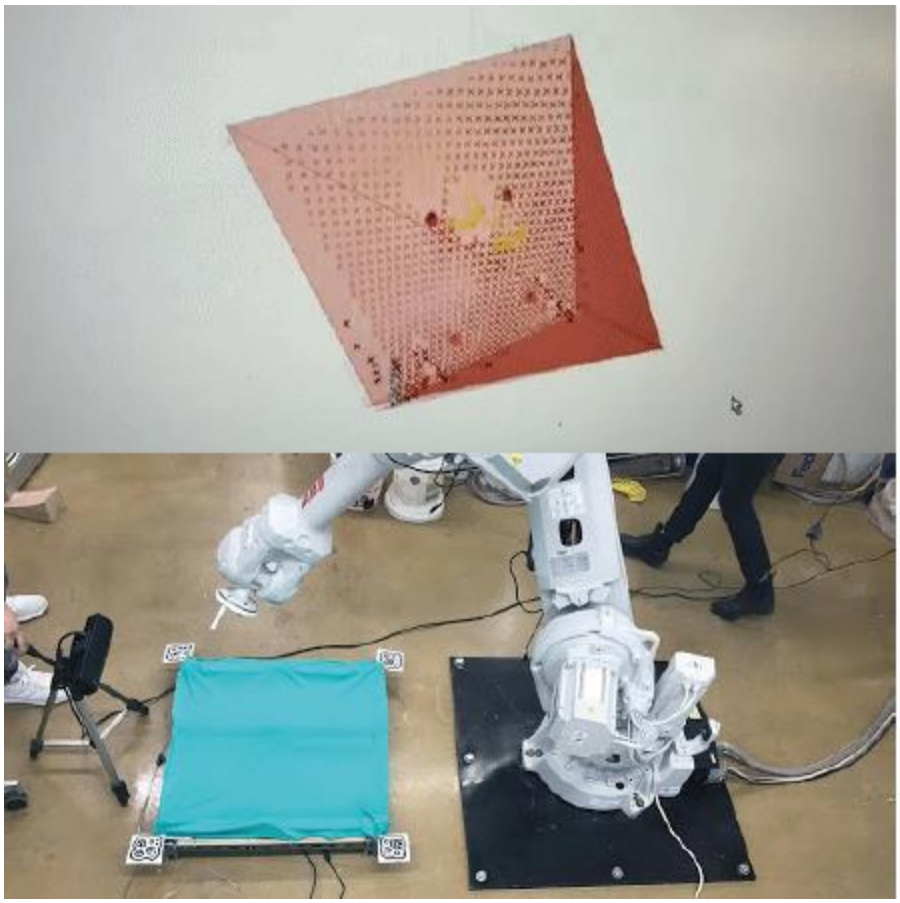


Figure 49, Phase three: QR code mapping as method of coordinate system matching.

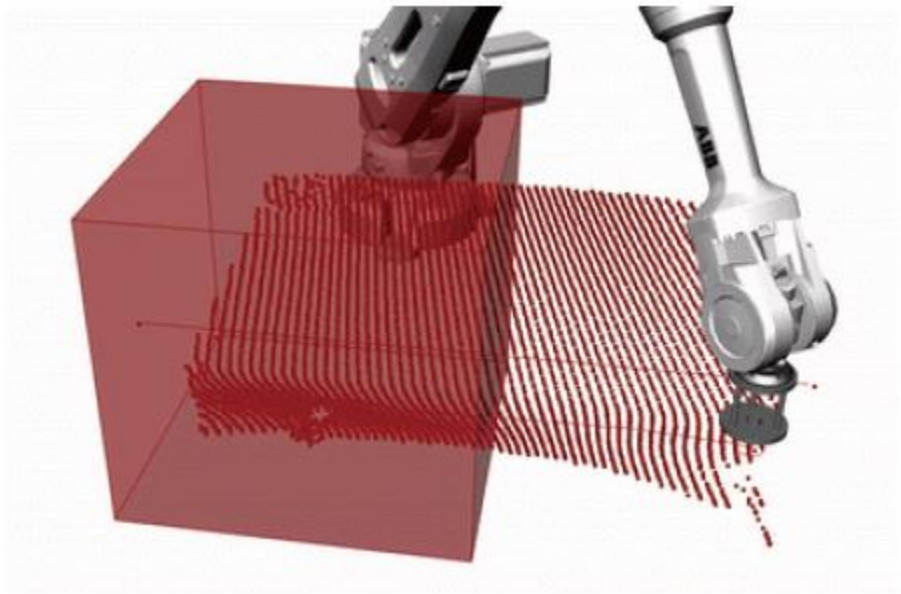


Figure 50, Phase four: toolpath curve evaluated according to distance traveled over, z value extracted

Final prototyping

After conducting the pilot study and building an inflatable printbed, it is found that the inflatable printbed is less ideal for coping with a large volume of robotic additive manufacturing, as the inflatable balloons are not able to afford the weight of the material. In addition, the curvature control of the printbed surface through the support of inflatable balloons is not accurate. Therefore, in the final prototyping, the author switched from inflatable balloons to the linear actuator to achieve accurate curvature control and allow large volume printing.

This reconfigurable 2.5-D printbed approximately has a 30-inch by 30-inch working area that contains 36 linear actuators with 6 inches vertical working space in a 6 by 6 array, a DC motor with gearbox reducer drives each actuator. (Figure 51)

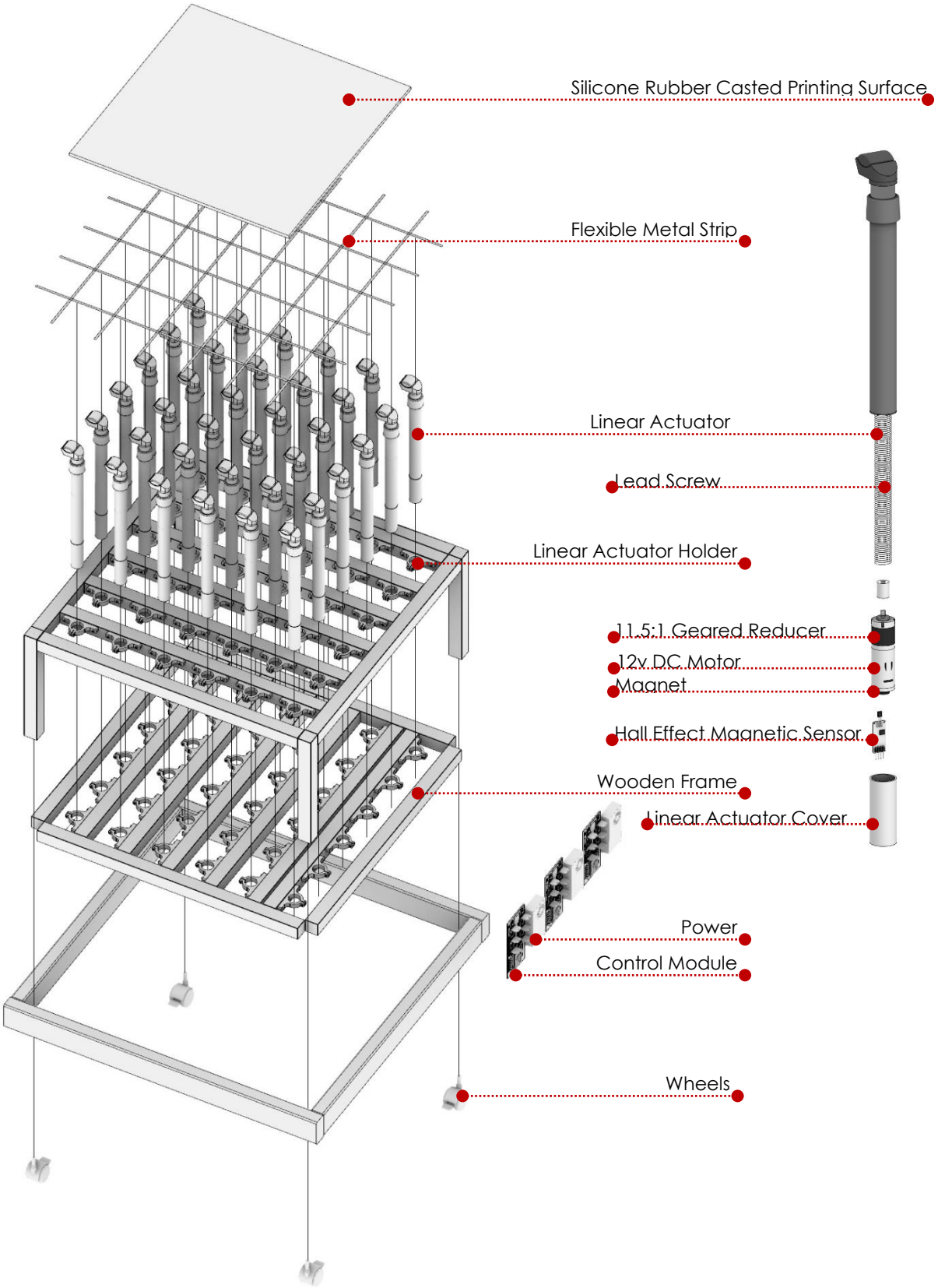


Figure 51, The physical configuration of reconfigurable 2.5-D printbed

So far, the hardware design and assemble of the reconfigurable 2.5-D printbed has been done, including wiring and programming work. the author has also developed a grasshopper plugin (Figure 52) to connect and control the printbed so that the robot arm can associativity synchronizing with it—the orientation of the sixth axis of the robotic arm (end effector) aligns with the normal of the surface on the depositing position. The grasshopper plugin sends two types of data: the linear actuator's speed and the number of rotations.

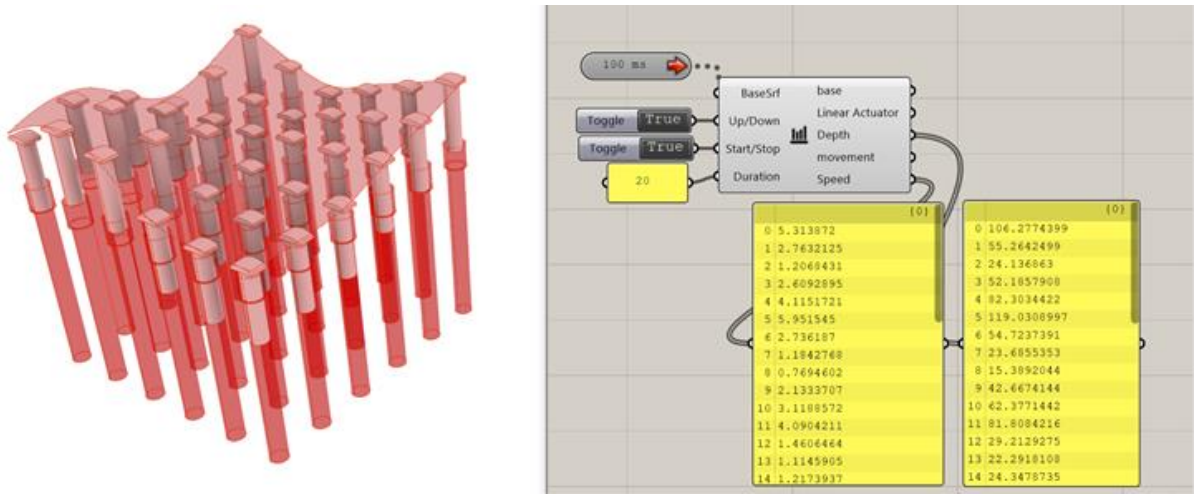


Figure 52, the grasshopper plugin sends two types of data: the linear actuator's speed and the number of rotations to each linear actuator

The author has designed and fabricated 3 PCB and made 3 control modules (Figure 53) out of them. Each control model contains 6 H bridges, that being said, each control module is capable of controlling 12 linear actuators. If the project demands a higher resolution reconfigurable 2.5-D printbed, the user just needs to attach more linear actuators with another control module.

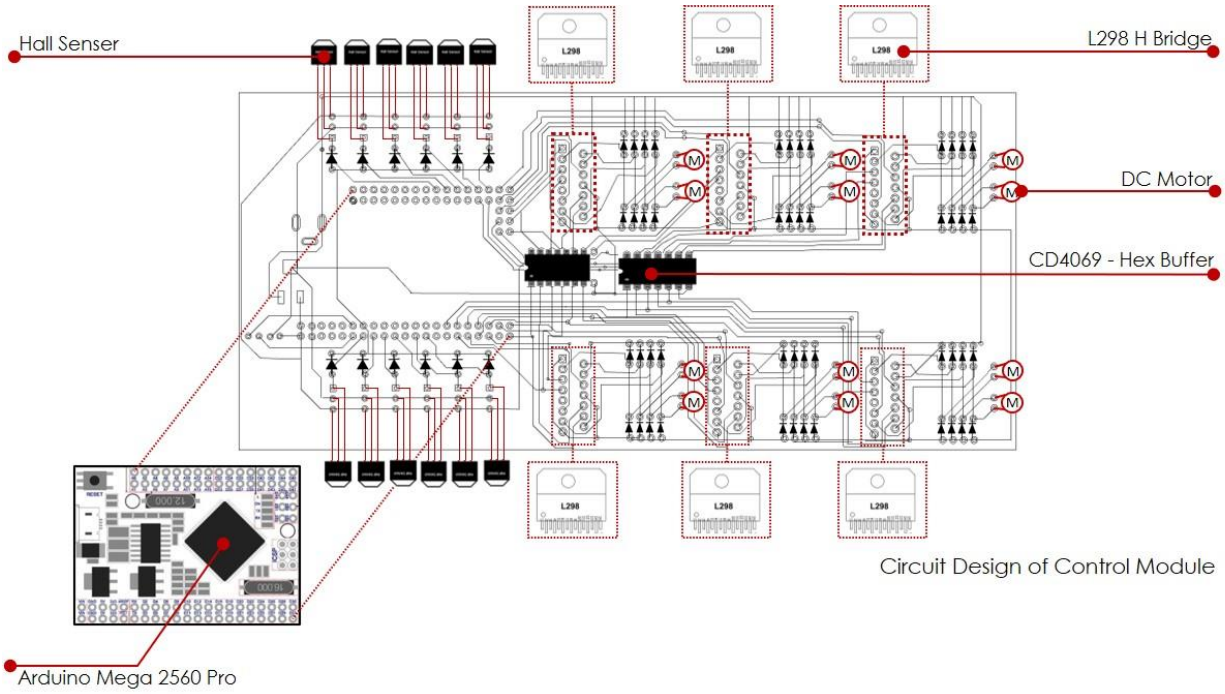


Figure 53, Control modules with Arduino Mega

A grasshopper plugin was also developed to connect and control each control module. The Grasshopper plugin converts a surface into a 6*6 points grid and sends the height information to the serial port. The information sends to Arduino wirelessly from the serial port through TTL, which gives the printbed more mobility. Arduino is receiving two types of data: the linear actuator's speed and the number of rotations. While the H-bridge drives the motor, the hall sensor attached to the bottom of the motor reads the number of rotations so that the system knows how far the linear actuator has traveled. Meanwhile, the surface in Grasshopper can be used for generating a toolpath for the robot arm, so the robot and the printed could be synchronizing together to conduct additive manufacturing. (Figure 54)

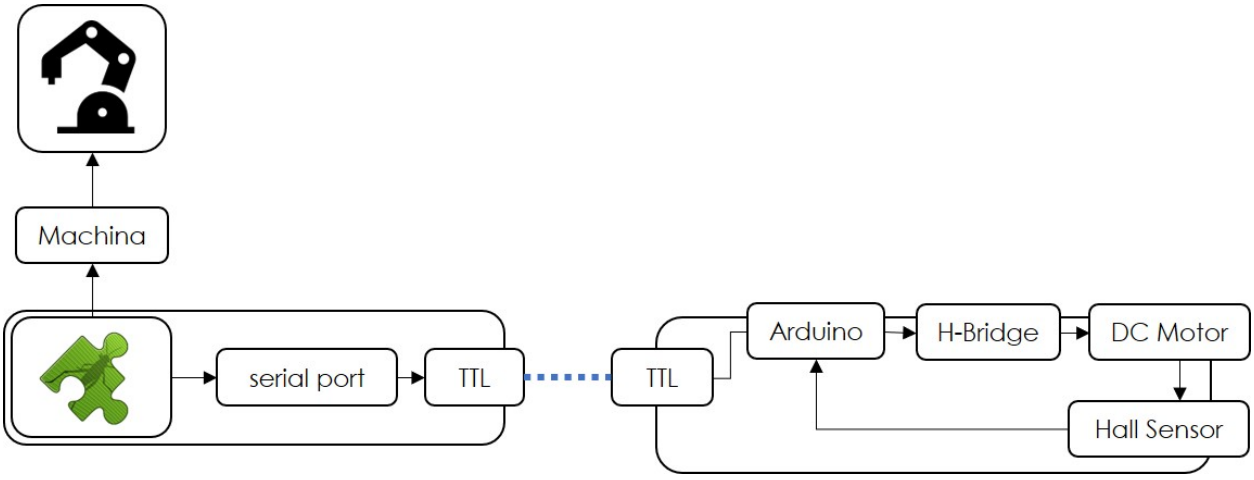


Figure 54, the workflow of control modules driving linear actuator

Due to some uncertain issues, by the time of thesis written, the pinbed is not fully functional yet to automatically control all 36 linear actuators at the same time to form a curved surface for printing. For the proof of concept purpose, the author manually controls each linear actuator to extend to a certain height to form a curved printed surface. A set of initial printing test has been conducted. The test results look very promising for later work of manufacturing a full-scale prototype. (Figure 55)

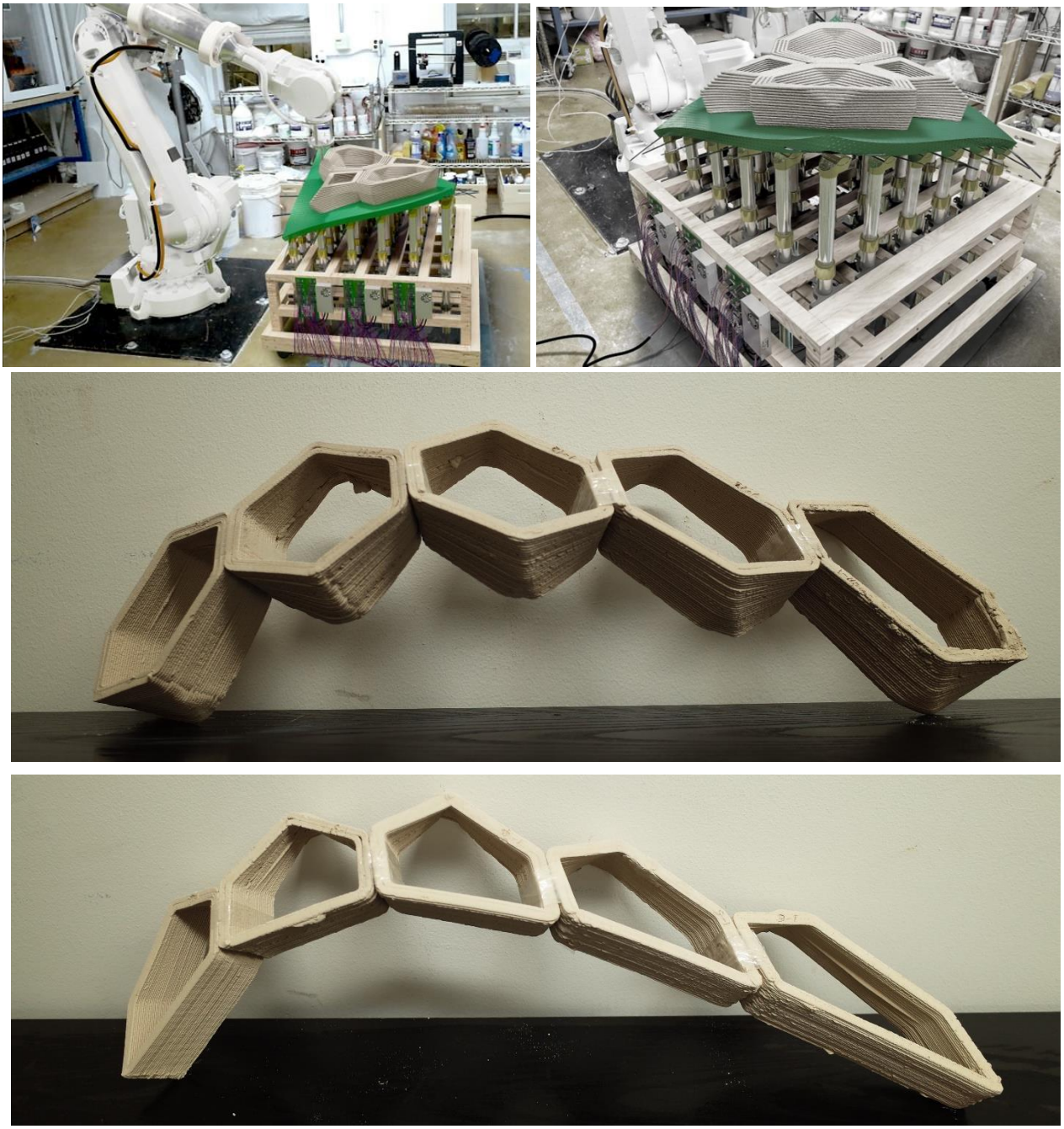


Figure 55, the printed scutoid brick prototypes with CERA III extruder and Pinbed

Discussion

The research question of this thesis lies in two aspects: how to efficiently design and fabricate a bio-inspired and morphologically complex surface geometry and how to embed interactive fabrication in the design process to inform the bio-inspired surface design loop at an architectural scale.

To answer these two open-ended questions, situating from a designer's perspective, the author proposed a comprehensive workflow that contains bio-inspired surface design strategy, design software development based on the strategy, bespoke robotic tooling for designers, and a designer-friendly user interface.

All the work done in this thesis are built upon an in-depth understanding of the deformation mechanisms of epithelial cells,

In the first chapter, the work that the author has generated thus far contributes to and enhances our understanding of how deformation mechanisms within epithelial tissue structures local and global change due to changes in local geometry and global surface curvature. The author also opened up a new channel that builds upon rigorous observations of cell behavior to then apply these findings in design research for architectural potential. Most importantly, the author contributed a novel approach for shell structure design and fabrication from the perspective of individual scutoid units. These outcomes demonstrate that the epithelial cell inspired-brick can be applied in a masonry shell system and is a feasible and innovative approach for fabricating shell structures. Utilizing such deformation mechanisms is feasible for designers to design a complex bioinspired surface geometry. The Grasshopper plugin developed contributes to packing such mechanisms into a parametric tool that can easily generate complex surface geometry.

In the second chapter, the work situates in-between the design and fabrication of complex surface geometry as an interface that transitions the digital information to guide physical fabrication and uses the physical information to support new design iteration. The 4d printed scutoid embedded surface geometry developed as a design medium encodes the digital model to the materialized object. While the 4d printed scutoid embedded surface geometry acts as a tangible user interface that allows designers to rethink the design with intuitive means, it also can be used as an input device to drive the fabrication equipment at a later stage of production. The 4d printed prototype shadows the main discussion around the formalism and generalizable computational approach learned from the experiments. When writing this thesis, the author realized that the readers might expect the digital input to change the physical form, not first modeling the physical structure and using it as an interface for changing the digital form. This approach seems inefficient, considering the affordances of both physical and digital modeling have not been considered in creating a set-up to interactively experiment with ‘deformable’ surfaces composed of Scutoid units. The limitation of the work is explicit. As the author hasn’t conducted any user study to test the usability of the proposed interface.

In the third chapter, the author developed several bespoke robotic tooling for designers expected to support bioinspired surface geometry fabrication. The author acknowledges that this is the hardest part of the thesis as the author encountered quite a few technical problems. The final prototype pinbed is still not fully functional yet. The future work will include further developing the unfunctional pinbed and depositing clay alone to build curved scutoid bricks.

Conclusion

Efficiently design and fabricating a bio-inspired and morphologically complex surface geometry is a systematic process. As a starting point, when designing bio-inspired surface geometry, it is essential to conduct a deep investigation in an interdisciplinary environment. Instead of situating ourselves as a designer who focuses merely on form and functions, we ought to learn from biology professionals and understand the field's basic knowledge. A thorough understanding of a specific biology subject helps designers establish a comprehensive parametric framework that is to be further developed later. It is also essential for designers to acknowledge that each bioinspired geometry is unique due to its biodiversity. In addition to designing forms using various types of design mediums, it is the designer's duty to propose and develop the proper fabrication method and craft that matches the geometric feature of the bioinspired geometry. Meanwhile, due to the complexity of bioinspired geometry, repetitive work is inevitable. Therefore, the design and fabrication method has to be adaptive and interactive, which leaves space for necessary adjustment.

This is a cumulative thesis that is composed of multiple projects. The author first investigated the deformation mechanism of epithelial cell situating from the cell biology literature review and computational geometry. To take advantage of the mentioned mechanism, the author then proposed a novel design application of an architectural surface geometry constituted by individual units. Specifically, a scutoid embedded masonry shell structure is designed in this case. By further investigating the epithelial cell and the dialectical relations between local and global surface geometry, the author materialized scutoid embedded surface geometry with 4d printing technology and Shape-Memory Polymer at a material scale. The author discussed utilizing such 4d printed scutoid embedded surface geometry as a tangible design user interface to facilitate the surface geometry design process. In the end, for fabricating such bio-inspired surface geometry at an architectural scale, the author developed a set of experimental fabrication tools supporting bio-inspired surface geometry additive manufacturing. Specifically, a robotic ceramic extruder is built, and a set of dynamic experimental printbeds is designed and developed.

The thesis developed thus far is not the ending of the work. On the contrary, the work scope of designing and fabricating bio-inspired surface geometry expanded. Beyond bio-inspired surface geometry, it opens up a possible chapter about integrating design and fabrication better in any design subject. Meanwhile, the thesis is a combination of prototyping-orientated research works, all the tools developed for the thesis, including the design tools (grasshopper plugin of scutoid shell generator, 4d printed tangible user interface) and fabrication hardware (extruder, printbed) have not been evaluated through a rigorous user study. As the author proposes a design and fabrication strategy to improve bio-inspired surface design quality and rationality, the proposed strategies and tools' efficacy must be evaluated. The efficacy needs to be tested through quantitative evaluation. That is to say, the usability and efficiency of the tools developed need to be tested. Such tests will follow the typical user study methodology in the HCI field.

BIBLIOGRAPHY

http://en.wikipedia.org/wiki/Epithelium.

Behl, Marc, and Andreas Lendlein. "Shape-memory polymers." Kirk-Othmer Encyclopedia of Chemical Technology (2000): 1-16.

Bilotti, J., B. Norman, David Rosenwasser, Jingyang Liu and J. Sabin. “Robosense 2.0. Robotic sensing and architectural ceramic fabrication.” (2018).

Coelho, M. and Zigelbaum, J., 2011. Shape-changing interfaces. Personal and Ubiquitous Computing, 15(2), pp.161-173.

Dhari, Rahul Singh, and Nirav P. Patel. "On the crushing behaviour of scutoid-based bioinspired cellular structures." International Journal of Crashworthiness (2021): 1-10.

Gergana Rusenova, Master Thesis, Emergent Space: Simulations and Analyses of Spatial Aggregate Formations, https://www.ehsanbaharlou.com/?p=452

Gibson, Matthew C., et al. ”The emergence of geometric order in proliferating metazoan epithelia.” Nature 442.7106 (2006): 1038.

G´omez-G´alvez, Pedro, et al. ”Scutoids are a geometrical solution to three-dimensional packing of epithelia.” Nature communications 9.1 (2018): 2960.

Khoo, Zhong Xun, Joanne Ee Mei Teoh, Yong Liu, Chee Kai Chua, Shoufeng Yang, Jia An, Kah Fai Leong, and Wai Yee Yeong. "3D printing of smart materials: A review on recent progresses in 4D printing." Virtual and Physical Prototyping 10, no. 3 (2015): 103-122.

Kong, D., Wolf, F. and Großhans, J., 2017, 'Forces directing germ-band extension in Drosophila embryos.', Mechanisms of development, 144, pp. 11-22.

Lecuit, Thomas, and Pierre-Francois Lenne., 2007, 'Cell surface mechanics and the control of cell shape, tissue patterns and morphogenesis.', Nature reviews Molecular cell biology, 8.8 , p. 633.

Lee, Amelia Yilin, Jia An, and Chee Kai Chua. "Two-way 4D printing: a review on the reversibility of 3D-printed shape memory materials." Engineering 3, no. 5 (2017): 663-674.

Martin, Adam C., Matthias Kaschube, and Eric F. Wieschaus. ”Pulsed contractions of an actin–myosin network drive apical constriction.” Nature 457.7228 (2009): 495.

Mughal, A., et al. ”Demonstration and interpretation of” scutoid” cells in a quasi-2D soap froth.” arXiv preprint arXiv:1809.08421 (2018)

Panozzo, D., Block, P. and Sorkine-Hornung, O., 2013. Designing unreinforced masonry models. ACM Transactions on Graphics (TOG), 32(4), pp.1-12.

Rippmann, M. and Block, P., 2013. Rethinking structural masonry: unreinforced, stone-cut shells. Proceedings of the Institution of Civil Engineers-Construction Materials, 166(6), pp.378-389.

Rupprecht, Jean-Francois, et al. ”Geometric constraints alter cell arrangements within curved epithelial tissues.” Molecular biology of the cell 28.25 (2017): 3582-3594.

Sabin, J. & Jones, P. “Nonlinear Systems Biology and Design: Surface Design” in Acadia 2008: Silicon + Skin, Biological Processes and Computation, ed. Kudless, A., Oct. 16-19, pp. 54-65, 2008.

Subramanian, Sai Ganesh, et al. ”Delaunay Lofts: A Biologically Inspired Approach for Modeling Space Filling Modular Structures.” Computers Graphics (2019).

Sun, Lei, Tomohiro Fukuda, Toshiki Tokuhara, and Nobuyoshi Yabuki. "Differences in spatial understanding between physical and virtual models." Frontiers of Architectural Research 3, no. 1 (2014): 28-35.

Teng, Teng. Jia, Mian and Sabin, Jenny. (2020), “Scutoid Brick - The Designing of Epithelial cell inspired-brick in Masonry shell System” in Anthropologic: Architecture and Fabrication in the cognitive age - Proceedings of the 38th eCAADe Conference - Volume 1, TU Berlin, Berlin, Germany, 16-18 September 2020, pp. 563-572

Teng, Teng and Sabin, Jenny (2021), “The Design and 4d printing of Epithelial cell-inspired Programmable Surface Geometry as Tangible User Interface” in Towards a New, Configurable Architecture: The 39th International Conference of Education and Research in Computer Aided Architectural Design in Europe (eCAADe 2021)- 8-10. September 2021.

Tibbits, Skylar. "Design to Self-Assembly." Architectural Design 82, no. 2 (2012): 68-73.

Tibbits, Skylar. "4D printing: multi-material shape change." Architectural Design 84, no. 1 (2014): 116-121.

Tsikoliya, Shota, Imro Vaško, Petra Sochůrková, and Daniel Sviták. "Tectonics of Differential Growth. Folds in Additive Fabrication and Moulding for Architectural Design." In Formal Methods in Architecture, pp. 29-35. Springer, Cham, 2021.

Appendix

The original C# code of the interactive scutoid generator plugin of Grasshopper in chapter 1.
Author : Benjamin Mian Jia and Teng Teng,

Produced for fulfilling course requirement of ARCH 7151, Fall 2019 at Cornell University,
Instructor: Prof. Jenny Sabin

```
using System;
using System.Collections;
using System.Collections.Generic;

using Rhino;
using Rhino.Geometry;

using Grasshopper;
using Grasshopper.Kernel;
using Grasshopper.Kernel.Data;
using Grasshopper.Kernel.Types;

using System.IO;
using System.Linq;
using System.Data;
using System.Drawing;
using System.Reflection;
using System.Windows.Forms;
using System.Xml;
using System.Xml.Linq;
using System.Runtime.InteropServices;

using Rhino.DocObjects;
using Rhino.Collections;
using GH_IO;
using GH_IO.Serialization;

/// <summary>
/// This class will be instantiated on demand by the Script component.
/// </summary>
public class Script_Instance : GH_ScriptInstance
{
#region Utility functions
  /// <summary>Print a String to the [Out] Parameter of the Script component.</summary>
  /// <param name="text">String to print.</param>
  private void Print(string text) { /* Implementation hidden. */ }
  /// <summary>Print a formatted String to the [Out] Parameter of the Script component.</summary>
  /// <param name="format">String format.</param>
  /// <param name="args">Formatting parameters.</param>
  private void Print(string format, params object[] args) { /* Implementation hidden. */ }
  /// <summary>Print useful information about an object instance to the [Out] Parameter of the Script component. </summary>
  /// <param name="obj">Object instance to parse.</param>
  private void Reflect(object obj) { /* Implementation hidden. */ }
  /// <summary>Print the signatures of all the overloads of a specific method to the [Out] Parameter of the Script component. </summary>
  /// <param name="obj">Object instance to parse.</param>
  private void Reflect(object obj, string method_name) { /* Implementation hidden. */ }
#endregion

#region Members
  /// <summary>Gets the current Rhino document.</summary>
  private readonly RhinoDoc RhinoDocument;
  /// <summary>Gets the Grasshopper document that owns this script.</summary>
  private readonly GH_Document GrasshopperDocument;
  /// <summary>Gets the Grasshopper script component that owns this script.</summary>
  private readonly IGH_Component Component;
  /// <summary>
  /// Gets the current iteration count. The first call to RunScript() is associated with Iteration==0.
  /// Any subsequent call within the same solution will increment the Iteration count.
  /// </summary>
  private readonly int Iteration;
#endregion
```

```
  /// <summary>
  /// This procedure contains the user code. Input parameters are provided as regular arguments,
  /// Output parameters as ref arguments. You don't have to assign output parameters,
  /// they will have a default value.
  /// </summary>
  private void RunScript(bool Reset, int intx, int inty, double height, double radius, double damping, double grav, double pullCenterUp, double pullBorder, double dt, double mass, double targetL, double springC, bool ToScutoid, ref object A, ref object B, ref object C, ref object D)
  {

    if (onetime){
      g = new Graph(intx, inty, height, radius);
      onetime = false;
    }
    onetime = Reset;

    g.Move(ToScutoid, intx, inty, radius, mass, targetL, springC, damping, grav, pullCenterUp, pullBorder, dt);

    A = g.GetCntPoints();
    B = g.GetTopEdgeLines();
    C = g.GetBotEdgeLines();
    D = g.GetCntEdgeLines();
  }

  // <Custom additional code>

  Graph g = new Graph(1, 1, 1.0, 1.0);
  bool onetime = true;

  //1.Node
  class Node{
    public Node(Point3d p0, Vector3d u0, double mass) {
      p = p0;
      m = mass;
      u = u0;
    }
    public List<Edge> Edges = new List<Edge>();

    public Point3d p = Point3d.Origin;
    public Vector3d f = Vector3d.Zero;
    public Vector3d u = Vector3d.Zero;
    public double m = 0.0;
    public bool Fixed = false;

    public List<Node> GetNeighbours()
    {
      List<Node> nn = new List<Node>();

      foreach (Edge ee in Edges)
      {
        if (ee.n0 == this && ee.n1 == this) continue;

        if (ee.n0 == this) nn.Add(ee.n1);
        else nn.Add(ee.n0);
      }
      return nn;
    }

    public void Move(double dt, double damping)
    {
      if (Fixed) return;
      //acceleration
      u *= damping;
      u += f * (dt / m);
      //position move
      p += u * dt;

      //if (Fixed) p.Z = 0.0;;
      if (p.Z < 0.0) {
        p.Z = 0.0;
        if (u.Z < 0.0) u.Z = -u.Z;
      }
    }
  }

  //2.Edge
  class Edge {
    public Edge(Node _n0, Node _n1) {
      n0 = _n0;
      n1 = _n1;
      n0.Edges.Add(this);
      n1.Edges.Add(this);
    }

    public Node n0;
    public Node n1;
    public bool Scutoided = false;

    public double Length()
    {
      return n0.p.DistanceTo(n1.p);
```

```
  }

  public Vector3d Direction()
  {
   return n1.p - n0.p;
  }

  public Point3d MidPoint()
  {
   return (n0.p + n1.p) * 0.5;
  }

  public double TargetLength = 0.0;
  public double SpringConstant = 0.0;

  public void ApplySpringForce() {
   Vector3d dp = n1.p - n0.p;
   double dist = dp.Length;
   dp.Unitize();

   if ((n1.p.DistanceTo(n0.p)) > TargetLength){
    n0.f += dp * (dist - TargetLength) *
SpringConstant;
    n1.f -= dp * (dist - TargetLength) *
SpringConstant;
   }
  }
 }

 //3.Graph
 class Graph {
  public List<Node> topnodes = new
List<Node>();
  public List<Node> botnodes = new
List<Node>();
  public List<Node> cntnodes = new
List<Node>();

  public List<Edge> topedges = new
List<Edge>();
  public List<Edge> botedges = new
List<Edge>();
  public List<Edge> cntedges = new
List<Edge>();

  //3-0.Gragh generation
  public Graph(int mx, int my, double z, double
mr) {

   double dx = 0.5 * Math.Sqrt(3) * mr;
   double dy = 0.5 * mr;

   //top 六边形:ok
   for(int j = 0; j < my; ++j) {
    double fh = j * mr;

    //1
    for(int the author = 0; the author < mx; ++i)
{
     int nownum = (4 * mx + 2) * j;
     double nx = dx + the author * 2 * dx;
     double ny = j * 2 * mr + 0 * dy + fh;
     Point3d newp = new Point3d(nx, ny, z);

     TopAddNode(newp, Vector3d.Zero, 0.0);
     //if ((i>0)&&(j == 0)){
     //  TopAddEdge(topnodes[nownum + i],
topnodes[nownum + the author - 1]);
     //}
     if (j > 0){
      TopAddEdge(topnodes[nownum + i],
topnodes[nownum + the author - mx]);
     }
    }

    //2
    for(int the author = 0; the author < mx + 1;
++i) {
     int nownum = (4 * mx + 2) * j + mx;
     double nx = the author * dx * 2;
     double ny = j * 2 * mr + 1 * dy + fh;
     Point3d newp = new Point3d(nx, ny, z);

     TopAddNode(newp, Vector3d.Zero, 0.0);
     if (i == 0) TopAddEdge(topnodes[nownum
+ i], topnodes[nownum + the author - mx]);
     if ((i > 0) && (i < mx)) {
      TopAddEdge(topnodes[nownum + i],
topnodes[nownum + the author - mx - 1]);
      TopAddEdge(topnodes[nownum + i],
topnodes[nownum + the author - mx]);
     }
     if (i == mx)
TopAddEdge(topnodes[nownum + i],
topnodes[nownum + the author - mx - 1]);
    }

    //3
    for(int the author = 0; the author < mx + 1;
++i) {
```

```
        int nownum = (4 * mx + 2) * j + (2 * mx +
1);
        double nx = the author * dx * 2;
        double ny = j * 2 * mr + 3 * dy + fh;
        Point3d newp = new Point3d(nx, ny, z);

        TopAddNode(newp, Vector3d.Zero, 0.0);
        TopAddEdge(topnodes[nownum + i],
topnodes[nownum + the author - mx - 1]);
       }

      //4
      for(int the author = 0; the author < mx; ++i)
{
        int nownum = (4 * mx + 2) * j + (3 * mx +
2);
        double nx = dx + the author * 2 * dx;
        double ny = j * 2 * mr + 4 * dy + fh;
        Point3d newp = new Point3d(nx, ny, z);

        TopAddNode(newp, Vector3d.Zero, 0.0);
        TopAddEdge(topnodes[nownum + i],
topnodes[nownum + the author - mx - 1]);
        TopAddEdge(topnodes[nownum + i],
topnodes[nownum + the author - mx]);
       }
     }

     //bot 六边形:ok
     z = 0.0;
     for(int j = 0; j < my; ++j) {
      double fh = j * mr;

      //1
      for(int the author = 0; the author < mx; ++i)
{
        int nownum = (4 * mx + 2) * j;
        double nx = dx + the author * 2 * dx;
        double ny = j * 2 * mr + 0 * dy + fh;
        Point3d newp = new Point3d(nx, ny, z);

        BotAddNode(newp, Vector3d.Zero, 0.0);
        //if ((i > 0) && (j == 0)){
        //   BotAddEdge(botnodes[nownum + i],
botnodes[nownum + the author - 1]);
        //}
        if (j > 0){
          BotAddEdge(botnodes[nownum + i],
botnodes[nownum + the author - mx]);
        }

      //2
      for(int the author = 0; the author < mx + 1;
++i) {
        int nownum = (4 * mx + 2) * j + mx;
        double nx = the author * dx * 2;
        double ny = j * 2 * mr + 1 * dy + fh;
        Point3d newp = new Point3d(nx, ny, z);

        BotAddNode(newp, Vector3d.Zero, 0.0);
        if (i == 0) BotAddEdge(botnodes[nownum
+ i], botnodes[nownum + the author - mx]);
        if ((i > 0) && (i < mx)) {
          BotAddEdge(botnodes[nownum + i],
botnodes[nownum + the author - mx - 1]);
          BotAddEdge(botnodes[nownum + i],
botnodes[nownum + the author - mx]);
        }
        if (i == mx)
BotAddEdge(botnodes[nownum + i],
botnodes[nownum + the author - mx - 1]);
      }

      //3
      for(int the author = 0; the author < mx + 1;
++i) {
        int nownum = (4 * mx + 2) * j + (2 * mx +
1);
        double nx = the author * dx * 2;
        double ny = j * 2 * mr + 3 * dy + fh;
        Point3d newp = new Point3d(nx, ny, z);

        BotAddNode(newp, Vector3d.Zero, 0.0);
        BotAddEdge(botnodes[nownum + i],
botnodes[nownum + the author - mx - 1]);
      }

      //4
      for(int the author = 0; the author < mx; ++i)
{
        int nownum = (4 * mx + 2) * j + (3 * mx +
2);
        double nx = dx + the author * 2 * dx;
        double ny = j * 2 * mr + 4 * dy + fh;
        Point3d newp = new Point3d(nx, ny, z);

        BotAddNode(newp, Vector3d.Zero, 0.0);
        BotAddEdge(botnodes[nownum + i],
botnodes[nownum + the author - mx - 1]);
```

```
        BotAddEdge(botnodes[nownum + i], botnodes[nownum + the author - mx]);
      }
    }

    int topnum = (4 * mx + 2) * my;

    //从上往下链接
    for (int the author = 0;i < topnum;i++){
     //Edge ce = new Edge(topnodes[i], botnodes[i]);
      // Node cn = new Node(ce.MidPoint(),Vector3d.Zero, 0.0);
      CntAddEdge(topnodes[i], botnodes[i]);
    }

    //border fixed
    for (int the author = 0;i < botnodes.Count;i++)
    {
      if(i < mx) {
        botnodes[i].Fixed = true;
      }
      for (int j = 0;j < my;j++)
      {
       if (i == mx + j * (4 * mx + 2)) {botnodes[i].Fixed = true;}
       if (i == 2 * mx + 1 + j * (4 * mx + 2)){botnodes[i].Fixed = true;}
       if (i == 2 * mx + j * (4 * mx + 2)){botnodes[i].Fixed = true;}
       if (i == 3 * mx + 1 + j * (4 * mx + 2)){botnodes[i].Fixed = true;}
      }
      if(i > 3 * mx + 1 + (my - 1) * (4 * mx + 2)) {botnodes[i].Fixed = true;}
    }
  }

  //3-1.Gragh Initialize
  public void initialGraph(int rx, int ry, double r, double imass, double iTargetLength, double iSpringConstant){

    foreach (Node n0 in topnodes)
    {
      n0.m = imass;
    }
    foreach (Node n0 in botnodes)
    {
      n0.m = imass;
    }
    foreach (Node n0 in cntnodes)
    {
      n0.m = imass;
    }

    //1.heightEdge will be more loose(X)
    foreach (Edge e0 in topedges)
    {
      e0.TargetLength = iTargetLength;
      e0.SpringConstant = iSpringConstant;
    }
    foreach (Edge e0 in botedges)
    {
      e0.TargetLength = iTargetLength;
      e0.SpringConstant = iSpringConstant;
    }
    foreach (Edge e0 in cntedges)
    {
      e0.TargetLength = iTargetLength * 0.75;
      e0.SpringConstant = iSpringConstant;
    }
  }

  //3-2.Add
  //AddNode
  public Node TopAddNode(Point3d _p0, Vector3d _u0, double _mass) {
    Node n = new Node(_p0, _u0, _mass);
    topnodes.Add(n);
    return n;
  }
  public Node BotAddNode(Point3d _p0, Vector3d _u0, double _mass) {
    Node n = new Node(_p0, _u0, _mass);
    botnodes.Add(n);
    return n;
  }
  public Node CntAddNode(Point3d _p0, Vector3d _u0, double _mass) {
    Node n = new Node(_p0, _u0, _mass);
    cntnodes.Add(n);
    return n;
  }

  //AddEdge
  public Edge TopAddEdge(Node n0, Node n1)
{
    if (n0 == n1) return null;
    Edge ed = TopFindtheEdge(n0, n1);
```

```
    if (ed != null) {
      return ed;
    }
    Edge e = new Edge(n0, n1);
    topedges.Add(e);
    return e;
  }
  public Edge BotAddEdge(Node n0, Node n1)
{
    if (n0 == n1) return null;
    Edge ed = BotFindtheEdge(n0, n1);
    if (ed != null) {
      return ed;
    }
    Edge e = new Edge(n0, n1);
    botedges.Add(e);
    return e;
  }
  public Edge CntAddEdge(Node n0, Node n1)
{
    if (n0 == n1) return null;
    Edge ed = CntFindtheEdge(n0, n1);
    if (ed != null) {
      return ed;
    }
    Edge e = new Edge(n0, n1);
    cntedges.Add(e);
    return e;
  }

  //3-3.Remove
  //RemoveEdges
  public void TopRemoveEdge(Edge e)
  {
    e.n0.Edges.Remove(e);
    e.n1.Edges.Remove(e);
    topedges.Remove(e);
  }
  public void TopRemoveEdges(List<Edge> _e)
  {
    foreach (Edge ee in _e)
    {
      TopRemoveEdge(ee);
    }
  }

  public void BotRemoveEdge(Edge e)
  {
    e.n0.Edges.Remove(e);
    e.n1.Edges.Remove(e);
    botedges.Remove(e);
  }
  public void BotRemoveEdges(List<Edge> _e)
  {
    foreach (Edge ee in _e)
    {
      BotRemoveEdge(ee);
    }
  }

  public void CntRemoveEdge(Edge e)
  {
    e.n0.Edges.Remove(e);
    e.n1.Edges.Remove(e);
    cntedges.Remove(e);
  }
  public void CntRemoveEdges(List<Edge> _e)
  {
    foreach (Edge ee in _e)
    {
      CntRemoveEdge(ee);
    }
  }

  //RemoveNodes
  public void TopRemoveNode(Node n)
  {
    if (n.Edges.Count != 0)
    {
      List<Edge> nodeedges = new List<Edge>();
      nodeedges.AddRange(n.Edges);
      TopRemoveEdges(nodeedges);
      BotRemoveEdges(nodeedges);
      CntRemoveEdges(nodeedges);
    }
    topnodes.Remove(n);
  }
  public void TopRemoveNodes(List<Node>
nodelist)
  {
    foreach (Node nn in nodelist)
    {
      TopRemoveNode(nn);
    }
  }

  public void BotRemoveNode(Node n)
  {
    if (n.Edges.Count != 0)
    {
```

```
      List<Edge> nodeedges = new List<Edge>();
      nodeedges.AddRange(n.Edges);
      TopRemoveEdges(nodeedges);
      BotRemoveEdges(nodeedges);
      CntRemoveEdges(nodeedges);
     }
     botnodes.Remove(n);
   }
   public void BotRemoveNodes(List<Node>
nodelist)
   {
     foreach (Node nn in nodelist)
     {
       BotRemoveNode(nn);
     }
   }

   //3-5.NodeSelection
   //FindNodesOfValence
   public                              List<Node>
TopFindNodesOfValence(int v)
   {
     List<Node> selectnodes = new List<Node>();
     foreach (Node nn in topnodes)
     {
       if (nn.Edges.Count == v)
       {
         selectnodes.Add(nn);
       }
     }
     return selectnodes;
   }

   //3-6.EdgeSelection
   //FindtheEdge
   public Edge TopFindtheEdge(Node _n0, Node
_n1)
   {
     foreach (Edge ee in topedges)
     {
       if ((ee.n0 == _n0 && ee.n1 == _n1) || (ee.n0
== _n1 && ee.n1 == _n0))
       {
         return ee;
       }
     }
     return null;
   }
   public Edge BotFindtheEdge(Node _n0, Node
_n1)
   {
     foreach (Edge ee in botedges)
     {
       if ((ee.n0 == _n0 && ee.n1 == _n1) || (ee.n0
== _n1 && ee.n1 == _n0))
       {
         return ee;
       }
     }
     return null;
   }
   public Edge CntFindtheEdge(Node _n0, Node
_n1)
   {
     foreach (Edge ee in cntedges)
     {
       if ((ee.n0 == _n0 && ee.n1 == _n1) || (ee.n0
== _n1 && ee.n1 == _n0))
       {
         return ee;
       }
     }
     return null;
   }

   //FindtheNodeNum
   public int FindtheTopNodeNum(Node goaln)
   {
     for(int the author = 0;i < topnodes.Count;i++)
     {
       if ((topnodes[i] == goaln))
       {
         return i;
       }
     }
     return -1;
   }
   public int FindtheBotNodeNum(Node goaln)
   {
     for(int the author = 0;i < botnodes.Count;i++)
     {
       if ((botnodes[i] == goaln))
       {
         return i;
       }
     }
     return -1;
   }

   //FindtheEdgeNum
```

```
  public int FindtheTopEdgeNum(Edge goaledge)
  {
   for(int the author = 0;i < topedges.Count;i++)
   {
    if ((topedges[i].n0 == goaledge.n0 && topedges[i].n1 == goaledge.n1) || (topedges[i].n0 == goaledge.n1 && topedges[i].n1 == goaledge.n0))
    {
     return i;
    }
   }
   return -1;
  }
  public int FindtheBotEdgeNum(Edge goaledge)
  {
   for(int the author = 0;i < botedges.Count;i++)
   {
    if ((botedges[i].n0 == goaledge.n0 && botedges[i].n1 == goaledge.n1) || (botedges[i].n0 == goaledge.n1 && botedges[i].n1 == goaledge.n0))
    {
     return i;
    }
   }
   return -1;
  }
  public int FindtheCntEdgeNum(Edge goaledge)
  {
   for(int the author = 0;i < cntedges.Count;i++)
   {
    if ((cntedges[i].n0 == goaledge.n0 && cntedges[i].n1 == goaledge.n1) || (cntedges[i].n0 == goaledge.n1 && cntedges[i].n1 == goaledge.n0))
    {
     return i;
    }
   }
   return -1;
  }

  //3.7 Copy
  public List<Edge> GetTopEdges()
  {
   List<Edge> ecopy = new List<Edge>();
   ecopy.AddRange(topedges);
   return ecopy;
  }
  public List<Edge> GetBotEdges()
  {
   List<Edge> ecopy = new List<Edge>();
   ecopy.AddRange(botedges);
   return ecopy;
  }
  public List<Edge> GetCntEdges()
  {
   List<Edge> ecopy = new List<Edge>();
   ecopy.AddRange(cntedges);
   return ecopy;
  }
  public List<Node> GetTopNodes()
  {
   List<Node> ncopy = new List<Node>();
   ncopy.AddRange(topnodes);
   return ncopy;
  }
  public List<Node> GetBotNodes()
  {
   List<Node> ncopy = new List<Node>();
   ncopy.AddRange(botnodes);
   return ncopy;
  }

  //3.8 info
  //getheight
  public double getHeight(){
   double h1 = topnodes[0].p.Z;
   foreach(Node n in topnodes) {
    if(n.p.Z > h1) h1 = n.p.Z;
   }
   return h1;
  }
  //getCenter
  public double[] TopgetCenter(){
   double[] center = new double[4];
   double x0 = topnodes[0].p.X;
   double y0 = topnodes[0].p.Y;
   double x1 = topnodes[0].p.X;
   double y1 = topnodes[0].p.Y;

   foreach(Node n in topnodes) {
    if(n.p.X < x0) x0 = n.p.X;
    if(n.p.X > x1) x1 = n.p.X;
    if(n.p.Y < y0) y0 = n.p.Y;
    if(n.p.Y > y1) y1 = n.p.Y;
```

```
  }
  center[0] = x0;
  center[1] = x1;
  center[2] = y0;
  center[3] = y1;
  return center;
}
public double[] BotgetCenter(){
  double[] center = new double[4];
  double x0 = botnodes[0].p.X;
  double y0 = botnodes[0].p.Y;
  double x1 = botnodes[0].p.X;
  double y1 = botnodes[0].p.Y;

  foreach(Node n in botnodes) {
    if(n.p.X < x0) x0 = n.p.X;
    if(n.p.X > x1) x1 = n.p.X;
    if(n.p.Y < y0) y0 = n.p.Y;
    if(n.p.Y > y1) y1 = n.p.Y;
  }
  center[0] = x0;
  center[1] = x1;
  center[2] = y0;
  center[3] = y1;
  return center;
}

//3.9 Branch
//OpenEdge
public void OpenEdge1(Edge e)
{
  Node n0_1 = e.n0.GetNeighbours()[0];
  Node n0_2 = e.n0.GetNeighbours()[0];
  Node n0_3 = e.n0.GetNeighbours()[0];
  int jishu = 0;
  foreach(Node nn in e.n0.GetNeighbours()){
    if (FindtheTopNodeNum(nn) != -1){
      n0_3 = nn;
    }
    else{
      if (nn != e.n1){
        jishu++;
        if (jishu == 1) n0_1 = nn;
        if (jishu == 2) n0_2 = nn;
      }
    }
  }
  Node n1_1 = e.n1.GetNeighbours()[0];
  Node n1_2 = e.n1.GetNeighbours()[0];
  Node n1_3 = e.n1.GetNeighbours()[0];
    jishu = 0;
    foreach(Node nn in e.n1.GetNeighbours()){
      if (FindtheTopNodeNum(nn) != -1){
        n1_3 = nn;
      }
      else{
        if (nn != e.n0){
          jishu++;
          if (jishu == 1) n1_1 = nn;
          if (jishu == 2) n1_2 = nn;
        }
      }
    }

    Edge tope = TopFindtheEdge(n0_3, n1_3);
    if (tope == null) return;

    Vector3d dir = e.Direction();
    Point3d midpoint = e.MidPoint();
    Point3d topmidpoint = tope.MidPoint();
    //crossproduct to get normal
    Vector3d cntdir = tope.MidPoint() -
e.MidPoint();;
    Vector3d dirnormal =
Vector3d.CrossProduct(dir, cntdir);
    dirnormal.Unitize();
    dirnormal *= 0.25 * e.Length();
    Point3d cntpoint = midpoint + cntdir * 0.5;

    Node cntnode = CntAddNode(cntpoint,
Vector3d.Zero, 0.0);
    Node newn0 = BotAddNode(midpoint +
dirnormal, Vector3d.Zero, 0.0);
    Node newn1 = BotAddNode(midpoint -
dirnormal, Vector3d.Zero, 0.0);

    Edge botee = BotAddEdge(newn0, newn1);

    CntAddEdge(cntnode, newn0);
    CntAddEdge(cntnode, newn1);

    CntAddEdge(cntnode, tope.n0);
    CntAddEdge(cntnode, tope.n1);

    if (newn0.p.DistanceTo(n0_1.p) <
newn1.p.DistanceTo(n0_1.p))
    {
      BotAddEdge(newn0, n0_1);
      BotAddEdge(newn1, n0_2);
    }
```

```
    else
    {
     BotAddEdge(newn0, n0_2);
     BotAddEdge(newn1, n0_1);
    }

    if (newn0.p.DistanceTo(n1_1.p) < newn1.p.DistanceTo(n1_1.p))
    {
     BotAddEdge(newn0, n1_1);
     BotAddEdge(newn1, n1_2);
    }
    else
    {
     BotAddEdge(newn0, n1_2);
     BotAddEdge(newn1, n1_1);
    }

    BotRemoveNode(e.n0);
    BotRemoveNode(e.n1);

    foreach (Edge ee in tope.n0.Edges){
     if (FindtheTopEdgeNum(ee) > -1) topedges[FindtheTopEdgeNum(ee)].Scutoided = true;
     if (FindtheBotEdgeNum(ee) > -1) botedges[FindtheBotEdgeNum(ee)].Scutoided = true;
    }
    foreach (Edge ee in tope.n1.Edges){
     if (FindtheTopEdgeNum(ee) > -1) topedges[FindtheTopEdgeNum(ee)].Scutoided = true;
     if (FindtheBotEdgeNum(ee) > -1) botedges[FindtheBotEdgeNum(ee)].Scutoided = true;
    }
    foreach (Edge ee in botee.n0.Edges){
     if (FindtheTopEdgeNum(ee) > -1) topedges[FindtheTopEdgeNum(ee)].Scutoided = true;
     if (FindtheBotEdgeNum(ee) > -1) botedges[FindtheBotEdgeNum(ee)].Scutoided = true;
    }
    foreach (Edge ee in botee.n1.Edges){
     if (FindtheTopEdgeNum(ee) > -1) topedges[FindtheTopEdgeNum(ee)].Scutoided = true;
     if (FindtheBotEdgeNum(ee) > -1) botedges[FindtheBotEdgeNum(ee)].Scutoided = true;
    }
   }

   public void OpenEdge2(Edge e)
   {
    Node n0_1 = e.n0.GetNeighbours()[0];
    Node n0_2 = e.n0.GetNeighbours()[0];
    Node n0_3 = e.n0.GetNeighbours()[0];
    int jishu = 0;
    foreach(Node nn in e.n0.GetNeighbours()){
     if (FindtheBotNodeNum(nn) != -1){
      n0_3 = nn;
     }
     else{
      if (nn != e.n1){
       jishu++;
       if (jishu == 1) n0_1 = nn;
       if (jishu == 2) n0_2 = nn;
      }
     }
    }
    Node n1_1 = e.n1.GetNeighbours()[0];
    Node n1_2 = e.n1.GetNeighbours()[0];
    Node n1_3 = e.n1.GetNeighbours()[0];
    jishu = 0;
    foreach(Node nn in e.n1.GetNeighbours()){
     if (FindtheBotNodeNum(nn) != -1){
      n1_3 = nn;
     }
     else{
      if (nn != e.n0){
       jishu++;
       if (jishu == 1) n1_1 = nn;
       if (jishu == 2) n1_2 = nn;
      }
     }
    }

    Edge bote = BotFindtheEdge(n0_3, n1_3);
    if (bote == null) return;

    Vector3d dir = e.Direction();
    Point3d midpoint = e.MidPoint();
    Point3d botmidpoint = bote.MidPoint();
    //crossproduct to get normal
    Vector3d cntdir = bote.MidPoint() - e.MidPoint();;
```

```
    Vector3d dirnormal = Vector3d.CrossProduct(dir, cntdir);
    dirnormal.Unitize();
    dirnormal *= 0.25 * e.Length();
    Point3d cntpoint = midpoint + cntdir * 0.5;

    Node cntnode = CntAddNode(cntpoint, Vector3d.Zero, 0.0);
    Node newn0 = TopAddNode(midpoint + dirnormal, Vector3d.Zero, 0.0);
    Node newn1 = TopAddNode(midpoint - dirnormal, Vector3d.Zero, 0.0);

    Edge topee = TopAddEdge(newn0, newn1);

    CntAddEdge(cntnode, newn0);
    CntAddEdge(cntnode, newn1);

    CntAddEdge(cntnode, bote.n0);
    CntAddEdge(cntnode, bote.n1);

    if (newn0.p.DistanceTo(n0_1.p) < newn1.p.DistanceTo(n0_1.p))
    {
     TopAddEdge(newn0, n0_1);
     TopAddEdge(newn1, n0_2);
    }
    else
    {
     TopAddEdge(newn0, n0_2);
     TopAddEdge(newn1, n0_1);
    }

    if (newn0.p.DistanceTo(n1_1.p) < newn1.p.DistanceTo(n1_1.p))
    {
     TopAddEdge(newn0, n1_1);
     TopAddEdge(newn1, n1_2);
    }
    else
    {
     TopAddEdge(newn0, n1_2);
     TopAddEdge(newn1, n1_1);
    }

    TopRemoveNode(e.n0);
    TopRemoveNode(e.n1);

    foreach (Edge ee in bote.n0.Edges){
      if (FindtheTopEdgeNum(ee) > -1) topedges[FindtheTopEdgeNum(ee)].Scutoided = true;
      if (FindtheBotEdgeNum(ee) > -1) botedges[FindtheBotEdgeNum(ee)].Scutoided = true;
    }
    foreach (Edge ee in bote.n1.Edges){
      if (FindtheTopEdgeNum(ee) > -1) topedges[FindtheTopEdgeNum(ee)].Scutoided = true;
      if (FindtheBotEdgeNum(ee) > -1) botedges[FindtheBotEdgeNum(ee)].Scutoided = true;
    }
    foreach (Edge ee in topee.n0.Edges){
      if (FindtheTopEdgeNum(ee) > -1) topedges[FindtheTopEdgeNum(ee)].Scutoided = true;
      if (FindtheBotEdgeNum(ee) > -1) botedges[FindtheBotEdgeNum(ee)].Scutoided = true;
    }
    foreach (Edge ee in topee.n1.Edges){
      if (FindtheTopEdgeNum(ee) > -1) topedges[FindtheTopEdgeNum(ee)].Scutoided = true;
      if (FindtheBotEdgeNum(ee) > -1) botedges[FindtheBotEdgeNum(ee)].Scutoided = true;
    }
   }

   //3-98.MakingScutoid
   public void MakingScutoid(Edge mide, int ii)
   {
    //can it be scutiod?
    bool cantscu = false;
    if (mide.n0.Edges.Count < 4) cantscu = true;
    if (mide.n1.Edges.Count < 4) cantscu = true;
    foreach(Edge ee in mide.n0.Edges){
     if (ee.n0.Edges.Count < 4) cantscu = true;
     if (ee.n1.Edges.Count < 4) cantscu = true;
    }
    foreach(Edge ee in mide.n1.Edges){
     if (ee.n0.Edges.Count < 4) cantscu = true;
     if (ee.n1.Edges.Count < 4) cantscu = true;
    }
    if (cantscu) return;
```

```
    //making scutoid
    if (ii == 1) OpenEdge1(mide);
    if (ii == 2) OpenEdge2(mide);
   }

   //3-99.AtLast.Dynamics
   public void Move(bool ToScutoid, int rx, int ry, double r, double m, double tL, double sC, double damping, double g, double pulls, double pulls2, double dt) {

    //MakingScutoid
    if ((ToScutoid) && (getHeight() > 6 * rx)){
     for (int the author = 0;i < botedges.Count;i++)
     {
      if ((!botedges[i].Scutoided) && (botedges[i].Length() > 1.2 * r))
      {
       MakingScutoid(botedges[i], 1);
      }
     }
    }
    if ((!ToScutoid) && (getHeight() > 6 * rx)){
     for (int the author = 0;i < topedges.Count;i++)
      if ((!topedges[i].Scutoided) && (topedges[i].Length() > 1.2 * r)
      ){
       MakingScutoid(topedges[i], 2);
      }
    }

    //basic initialize
    initialGraph(rx, ry, r, m, tL, sC);
    foreach(Node n in topnodes) {
     n.f = Vector3d.ZAxis * g;
    }
    foreach(Node n in cntnodes) {
     n.f = Vector3d.ZAxis * g;
    }
    foreach(Node n in botnodes) {
     n.f = Vector3d.ZAxis * g;
    }


    double[] centerp = TopgetCenter();
    Point3d centerpoint = new Point3d((centerp[0] + centerp[1]) * 0.5, (centerp[2] + centerp[3]) * 0.5, 0);

    //pull from center
    foreach(Node n in topnodes) {
     double dx = Math.Abs(n.p.X - centerpoint.X);
     double dy = Math.Abs(n.p.Y - centerpoint.Y);
     double disxy = Math.Sqrt(dx * dx + dy * dy);
     if ((disxy < (centerp[1] - centerp[0]) * 0.5) && (disxy < (centerp[3] - centerp[2]) * 0.5))
     {
      n.f += Vector3d.ZAxis * pulls;
      foreach (Node nn in n.GetNeighbours()){
       nn.f += Vector3d.ZAxis * pulls * 0.2;
      }
     }
    }

    //pull from border
    for (int the author = 0;i < topnodes.Count;i++)
    {
     double notmove = 0.1 * r;
     //if(i < 2 * rx + 1) { //所有边都贴在地上
     if(i < rx) {
      if (topnodes[i].p.Z < notmove){topnodes[i].Fixed = true;}
      topnodes[i].f += (Vector3d.YAxis + Vector3d.ZAxis) * -pulls2;
     }
     for (int j = 0;j < ry;j++)
     {
      if (i == rx + j * (4 * rx + 2)) {
       if (topnodes[i].p.Z < notmove){topnodes[i].Fixed = true;}
       topnodes[i].f += (Vector3d.XAxis + Vector3d.ZAxis) * -pulls2;
      }
      if (i == 2 * rx + 1 + j * (4 * rx + 2)){
       if (topnodes[i].p.Z < notmove){topnodes[i].Fixed = true;}
       topnodes[i].f += (Vector3d.XAxis + Vector3d.ZAxis) * -pulls2;
      }
      if (i == 2 * rx + j * (4 * rx + 2)){
       if (topnodes[i].p.Z < notmove){topnodes[i].Fixed = true;}
       topnodes[i].f += (Vector3d.XAxis - Vector3d.ZAxis) * pulls2;
      }
      if (i == 3 * rx + 1 + j * (4 * rx + 2)){
```

```
        if (topnodes[i].p.Z < notmove){topnodes[i].Fixed = true;}
        topnodes[i].f += (Vector3d.XAxis - Vector3d.ZAxis) * pulls2;
       }
      }
      //if(i > 2 * rx + 1 + (ry - 1) * (4 * rx + 2)) {//所有边都贴在地上
      if(i > 3 * rx + 1 + (ry - 1) * (4 * rx + 2)) {
       if (topnodes[i].p.Z < notmove){topnodes[i].Fixed = true;}
       topnodes[i].f += (Vector3d.YAxis - Vector3d.ZAxis) * pulls2;
      }
     }

     //spring
     foreach(Edge e in topedges) {

      e.ApplySpringForce();
     }
     foreach(Edge e in cntedges) {
      e.ApplySpringForce();
     }
     foreach(Edge e in botedges) {
      e.ApplySpringForce();
     }

     //move
     foreach(Node n in topnodes) {
      n.Move(dt, damping);
     }
     foreach(Node n in cntnodes) {
      n.Move(dt, damping);
     }
     foreach(Node n in botnodes) {
      n.Move(dt, damping);
     }
    }

    //3-99.AtLast.GraphicRepresentation
    public List<Point3d> GetPoints() {
     List<Point3d> c = new List<Point3d>();
     foreach(Node n in topnodes) {
      c.Add(n.p);
     }
     foreach(Node n in botnodes) {
      c.Add(n.p);
     }
     return c;
    }

    public List<Point3d> GetTopPoints() {
     List<Point3d> c = new List<Point3d>();
     foreach(Node n in topnodes) {
      c.Add(n.p);
     }
     return c;
    }

    public List<Point3d> GetBotPoints() {
     List<Point3d> c = new List<Point3d>();
     foreach(Node n in botnodes) {
      c.Add(n.p);
     }
     return c;
    }

    public List<Point3d> GetCntPoints() {
     List<Point3d> c = new List<Point3d>();
     foreach(Node n in cntnodes) {
      c.Add(n.p);
     }
     return c;
    }

    public List<Vector3d> GetForces() {
     List<Vector3d> c = new List<Vector3d>();
     foreach(Node n in topnodes) {
      c.Add(n.f);
     }
     foreach(Node n in botnodes) {
      c.Add(n.f);
     }
     return c;
    }

    public List<Vector3d> GetVelocities() {
     List<Vector3d> c = new List<Vector3d>();
     foreach(Node n in topnodes) {
      c.Add(n.u);
     }
     foreach(Node n in botnodes) {
      c.Add(n.u);
     }
     return c;
    }

    public List<Line> GetTopEdgeLines() {
     List<Line> c = new List<Line>();
```

```
    foreach(Edge e in topedges) {
      c.Add(new Line(e.n0.p, e.n1.p));
    }
    return c;
  }

  public List<Line> GetBotEdgeLines() {
    List<Line> c = new List<Line>();
    foreach(Edge e in botedges) {
      c.Add(new Line(e.n0.p, e.n1.p));
    }
    return c;
  }

  public List<Line> GetCntEdgeLines() {
    List<Line> c = new List<Line>();
    foreach(Edge e in cntedges) {
      c.Add(new Line(e.n0.p, e.n1.p));
    }
    return c;
  }
}
```

```
  // </Custom additional code>
}
```