

Beyond Shape: Construction-Aware Languages for Design for the Physical World

Amy Zhu

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

Doctor of Philosophy

University of Washington

2026

Reading Committee:

Adriana Schulz

Zachary Tatlock

Gilbert Bernstein

Program Authorized to Offer Degree:
Computer Science & Engineering

© Copyright 2026

Amy Zhu

University of Washington

Abstract

Beyond Shape: Construction-Aware Languages for Design for the Physical World

Amy Zhu

Chairs of the Supervisory Committee:

Zachary Tatlock Adriana Schulz

Paul G. Allen School of Computer Science & Engineering

Design-for-fabrication systems often ingest conventional geometric representations, like meshes. However, these designs must actually be made in the physical world. How the real objects are made is an integral part of design, as it controls performance metrics like fabrication time, material usage, material properties, and fabrication robustness, among other things. Though geometric representations capture shape and behaviour, they cannot capture these construction details; users are left either doing onerous low-level reasoning and brittle editing, or relying on mythical “sufficiently smart compilers” to generate the fabrication plan they already had in mind.

This thesis argues that design systems should instead let users specify construction explicitly, and that doing so requires a language-based design representation built from three components: geometry as a partial input, abstractions of domain-specific fabrication operations, and support for a user-in-the-loop workflow in which the representation is directly editable and results can be previewed end to end.

I develop this argument across three systems. The first, for machine knitting, develops abstractions for the fabrication procedure of knit illusions, and creates a system for users to design their own illusions, including novel double-sided illusions. This project establishes the basic

principles in a scoped setting where the set of fabrication plans is limited. The second extends knitting to 3D layouts and shaping through a scheduling language over compositions of 2D panels, where a schedule specifies how each panel is composed, decomposed, and knitted. The third carries the same ideas beyond knitting, presenting a domain-specific language for animating content on electromechanical displays, where schedule-like motion effects are applied to keyframe inputs. Together, these systems show that using a design representation where construction is a first-class, language-based component enables the design and creation of real objects where using geometry alone would not.

Contents

1	Introduction	1
2	Why care about construction?	5
3	Knitting Background	7
3.1	Knitting Basics	7
3.2	Machine Knitting	10
3.3	Colorwork	12
3.4	Higher-Level Representations	14
4	Illusion Knitting	17
4.1	Introduction	17
4.2	Overview	20
4.2.1	Insights from Knit Microgeometry	21
4.2.2	Defining a Design Space	22
4.3	Foundations of illusion knitting	26
4.4	Fabrication	29
4.4.1	Combined Colourwork Strategy	29
4.5	Evaluation	35
4.5.1	Automating Single-view Illusions	35
4.5.2	Beyond Single-view Illusions	36

4.5.3	Comparison with Other Design Methods	36
4.5.4	Fabrication	40
4.6	Limitations and Future Work	42
4.7	Conclusion	43
5	Scheduling Knitting	45
5.1	Introduction	46
5.2	PLANEL System Overview	50
5.3	PLANEL Embedded DSL Design	53
5.3.1	Specifying Shapes	54
5.3.2	Scheduling Shapes to Fabrication Plans	56
5.3.3	Tracking Merges through Scheduling Operations	60
5.3.4	Language semantics	63
5.4	PLANEL Backend: Codegen	63
5.4.1	Generating KnitGraphs from Planned Panels	66
5.4.2	Joining KnitGraphs	67
5.4.3	Tracing and Chunking	69
5.4.4	Knittability	70
5.5	Evaluation	71
5.5.1	RQ1. Supporting Challenging Objects	71
5.5.2	RQ2. Large-scale evaluation	75
5.5.3	RQ3. Controlling Performance Tradeoffs	77
5.6	Editing and Iteration Case Studies	80
5.6.1	Modify Shape, Maintain Schedule	80
5.6.2	Simplify schedule	80
5.6.3	Transfer schedule	81
5.7	Related Work	82

5.8	Limitations and Future Work	84
5.9	Conclusion and Future Work	85
6	Flip displays	87
6.1	Introduction	87
6.2	Background on display hardware examples	91
6.2.1	Flipdots	92
6.2.2	Split flaps	92
6.3	Design Overview	93
6.3.1	Effects	94
6.3.2	Hardware specification	97
6.4	Implementation details	98
6.4.1	Lowering the front-end graph	98
6.4.2	Transitions	99
6.4.3	Orders	102
6.4.4	Hardware	104
6.4.5	Playground	104
6.5	Evaluation	106
6.5.1	Effects Case Study	106
6.5.2	Effect remixing	110
6.5.3	Emergent effects	112
6.6	Related work	116
6.6.1	Unconventional displays	116
6.6.2	Programmable motion graphics	118
6.6.3	Low-resolution rendering	118
6.6.4	Scheduling	118
6.7	Conclusion	119

7 Conclusion	121
Bibliography	123

Acknowledgments

Thank you first and foremost to my advisors, both of which have influenced my taste in research, and more importantly, a taste for beauty. Thank you for teaching me so much about computer science, about how to frame problems and ideas, and how to connect with people. You have each built amazing communities that I have loved being a part of, and have influenced me to try to speak honestly and from the heart. Adriana, I think I most value your endless energy and enthusiasm for projects that kept me going when even I didn't believe, and your ability to offer time for me whenever I needed it despite your crazy schedule; Zach, your ability to find clarity and simplicity anywhere, and to make me believe in a way forward always. Thank you both most of all for putting my ideas, beliefs, and successes first. I am so lucky to have been your student.

Thank you to the rest of my committee. Afroditi, you are incredibly cool and your art inspires me deeply. I am so appreciative you could be part of my committee and that you are doing amazing work here at UW. Gilbert, I appreciate your need for preciseness and correctness, and especially your earnest attempts to do good in this world, and that you are constantly thinking of others, as evidenced in that you always having something to say to everyone.

Thank you to my other advisors earlier in life for the best summers imaginable. Ivan, for taking me in as an undergrad and teaching me all about distributed systems, software engineering, and research, for bringing me into your lab, and for mentoring me. Much of my impressions of grad school culture are still because of you (juggling, for example). Stew and Vaas, my two closest

friends and labmates at the NSS. I fondly remember a love of Coca-Cola, many small gifts, and conversations that would last all day. Ilya, for making me feel so welcome at NUS, for bringing me to PL research, and for believing in my success whenever we saw each other next. Thank you Andreea, for giving so much advice and for your knowledge equally about synthesis and Singapore. Yunjeong, for traveling with me, for having an inspiring appetite to learn, and for your overwhelming sincerity.

Thank you to my Adobe mentors for the opportunities to work on my own research at Adobe. Maaz, for giving meaningful advice at the right time, and teaching me so much about the PL and architecture community. Alec, for your vast knowledge about both computer graphics and about art, and bringing that perspective and enthusiasm for them to my summer in Toronto.

Thank you to the machine knitting community, for all your help along the way. Especially thanks to Megan, for being the first to teach me everything in your class, and for the continued help even afterwards. It would have taken me a lot longer to get to where I am now without you. And thank you Jenny, for being a wonderful community member and for always answering my questions even though we are far apart.

Thank you to my mentors in grad school, all my “upperclasspeople”. Max, for being a great leader but also a reliable source to talk to about games and especially dating reality TV. Jasper, for your kindred walks about our shared concerns, and for all your introductions to HCDE people. Althea, for bringing me to SESSC and into the world of art. Gus, for always saying the most out-of-pocket things, for throwing amazing house parties, and for dreaming big on art together. Ben, for all the research help, for surviving a summer at Adobe together, and especially for all the movie nights with the sheet on the side of your shed and the bonfire. Felix, for never failing to improve our conditions and speak on behalf of the students, never failing to having a reference of the literature at hand, and providing thoughtful reflections whenever we talk; it’s always enjoyable to hear about your current interests. Yuxuan, for being a tireless collaborator, for being

consistently involved in the community, for taking over shifts running the knitting machine (especially for illusion knitting) and spending countless hours debugging that darn thing. And also for being always so easy to talk to and so supportive, and for all your delicious baking. Chandra, for our years-long friendship. I was so impressed by you the first time we met at PNW PLSE in 2018, and when we met again at visit days, I could feel that we would be friends for a long time. Thank you for all your mentoring in research and in careers, for being the first person I go to when I need advice, and for being able to be silly and chat endlessly.

Marlena, thank you for being the most amazing mentee, for being truly unstoppable in work and life, for always handling whatever is happening without a single complaint. And for simultaneously being exactly as wacky, if not wackier, than the rest of us, and for letting me into that world of yours. I know you will do amazing things.

My dearest friends and labmates in CSE, I could never have done six years without all of you. Every one of you has made me smile, laugh, and taught me things every day. Thank you! And if I left you out, I'm sure it's just because I forgot, having to write this at the last minute as all things are. Audrey, for being so easy to talk to, for having so many of the same interests and enthusiasm, I am amazed by your ability to pick up any craft and do it well, and your prolificness! Yihong, for being so engaged with both labs and also so funny. I often think about your "Is sea... is land" Progress update slide that one time you went to renew your visa. Jesse, for always being a welcoming friend, for your humour, and for all the puzzles along the way. Jennifer, for gracing us with your presence even though we're all weird PL people, and for being so enthusiastic and funny underneath your general calm. Anjali, for being the voice of reason, and for those unexpected and delightful moments when you break that shell of reasonableness and say you'd want a pet raccoon. I'm glad to have been able to get to see that side of you over the years and become friends. Dan, for surviving TAing with me, for teaching me about art and architecture, for being my friend even past the time we were labmates, and for inviting me to Arkansas. I'm so glad I had someone to

share the grad school experience so closely with especially back then. Megan, for your grounding and accepting energy, and for all our shared excitement about strange, potentially canned foods. Ben, for indulging in silly, over-the-top ideas with and for keeping the wacky spirit alive. I will never forget that Zoom background you had in early days with your face. John, for bringing so much culture of music and literature, for teaching me about your tastes, for encouraging me to submit to FARM, and for the project we worked on together. I really enjoyed bringing it to life with you and hearing about your multitudes of endeavors! Saul, for being so fun and easy to talk to and spend time with, for staying at my place during visit days, and for being the best sauna buddy. Kirsten, for taking care of the trees, the swamp, and the lab with all your earth tones and your wisdom, and for being able to roll with every weird conversational punch. Josh, for all the walking and talking with the amazing company of you and your friends, the insights you share, and the creative projects that delight me. Andrew, for your artistic insight, for your ability to become a meme unto yourself, and for all the days of being Sonic fans and Kitchen Nightmare enjoyers together. Nat, for conjuring up a left-field dry and witty statement at any time, and for being on this weird machine knitting journey together. I hope I get to wish you a happy birthday sometime. Vishal, for being my accountability buddy and someone who always provides conversation easily. Ryan, for how you can make anyone feel welcomed, your keen ability to ask questions, and the way you make any room feel brighter and warmer just by being there. Special thanks for your ability to stir the pot and then gossip about it with me. Thia, for your countless incisive, perfectly-accurate observations and your deep and interesting cultural knowledge. I could listen to you talk all day no matter the subject and I hope I will be able to for a long time. Ewin, for unfailingly making me laugh and for embracing all our weird tastes together, especially in perfume and its reviews. Elliott, for making every day in PLSE wild, chaotic, and exciting, for always being ready to jump into a group activity, and for bringing that unabashedly insane energy any time we've met. Andrew, for entertaining me every single day and keeping me

on my toes... thank you for being the only person who is regularly mean to me, and is actually also my friend who cares. I'll still let you use the A-frame, don't worry. Oliver, for the way you manage to mix having the most fun with doing hard work, for inspiring me with all the projects and creative efforts you do on the side, and for letting me help you with some of the important ones! And especially for being ready to hang out, even all the way back when you first moved here. Brett, for your years of friendship all the way from the very beginning, for all our endless threads on Slack, for teaching me all about planes, ships, and roadways. I am confused as to how we talk so much with basically no common interests, but I relish every conversation. Kevin, my partner in crime, my kindred spirit, my bosom buddy. Thank you for always scheming together, for always understanding me (or pretending to), and for finding fun in every corner. I am amazed at the depth of your wisdom and analysis in every setting when you choose to be serious, and enjoy every other moment when you're not. I am confident I could never find anyone remotely like you and I'm honoured to be your friend. We'll fly again! James, I think you have kept me sane ever since we met in 2016. I can't believe we went on this whole computer science journey together. I'm glad we're just as inseparable as we used to be as co-TAs, as co-SDS leaders, as temporary roommates, as friends across a boarder, and now as labmates at UW. Can't wait to give a speech at your own graduation.

To all my friends from childhood, to high school, to undergrad and beyond. Niu, for being the best roommate and friend since (I think) you woke me up during class in first year. Tiger, Daphne, for bringing us on so many adventures in Vancouver, Seattle, and beyond. To the people around Seattle who taught me varied things and furthered my learnings in fabrication, especially Michael for your enthusiasm for gemcutting as the NPSFG president, and our endless conversations, and Linda for providing such an amazing weaving community at the SESSC. To my high school friends, thank you for over a decade of hotpots, food gatherings, get-togethers, events, vacations, and group chats. I'm so glad we're still in touch every day and I hope we will be for the rest of time.

To my family, thank you for always supporting me on whatever path I take and believing in me. Thank you to my dad for inspiring me, at least partly, to do this whole thing at all, and for the guidance along the way. Thank you to my mom for always humouring me, for seeing the light side of things, and for pushing me without being overbearing. Thank you to Kevin, for being my best friend through my whole life, for always wanting to spend time together, and for giving me the best sibling relationship you could imagine.

Finally, thank you the most to Jonathan. You have supported me infinitely throughout these six years, from us moving in together (in a new country! during COVID!), to making sure I am fed hot meals every day, to listening to all my problems and indulging all my whims so patiently. You brought joy to me every day. Thank you for always being excited to spend time with me, no matter where it is or what we are doing. I am excited to see what the next step will be for us, especially after we both worked so hard to get this PhD.

Chapter 1

Introduction

Design and fabrication tools for the physical world shape what creators are able to imagine and make. While many tools focus on producing familiar categories of objects more efficiently, we believe computational support can help creators reimagine what is possible. A knitted object can change appearance depending on how it is viewed; a garment can gain structure and expression from the way its panels are shaped and joined; a display can use the motion of its physical elements as part of the image itself. These examples point to a broader opportunity: by building tools that highlight specialized fabrication processes, we can help creators explore objects that, yes, trade off performance criteria, but are also surprising, expressive, and specific to the way they are made.

As it stands now, creators of design and fabrication systems choose the type of inputs users provide as the design's starting point. Typically, as the final objects live in 3D, the design is expressed via its shape or its material distribution using classic geometric representations like meshes, point clouds, or B-reps.

From this point on, however, there are many choices for how an object will be physically realized. Imagine a tank top knit in a single piece in-the-round. Now imagine that we add a seam to it. This decision changes the drape and naturally encourages it to fold along that line.

We might be able to knit it flat and unlock the ability to produce textures that were not possible in-the-round. On the other hand, an extra seaming step adds manufacturing time and human effort. In 3D printing, the user dictates the direction of printing, the layer height, the infill density and patterns, and the support designs. These choices affect fabrication time, material cost, material properties like stress/strain, appearance, and much more. Thus, to control performance and other characteristics, we need to choose between different fabrication plans.

Since these factors are so important, how do we allow the user to specify, explore, and choose the fabrication plan they want? The first step is: we need design representations that captures construction details, or *how something is done*. Then, we can build a design space using that representation.

Can existing representations encode such construction details? Coarse geometry, like these conventional geometric representations, relies on mythical "sufficiently smart compilers" to correctly interpret a user's intent. Usually, because the compiler relies on a heuristic or optimizer, we also get a very sparse design space where one input shape maps to one plan. Designing for difficult-to-quantify criteria, like human perception, add an extra layer of infeasibility. Embedding the construction plan to create fine geometry, like modelling the layer lines directly inside a mesh, results in a very dense design space where most points are not meaningfully different. Additionally, encoding the plan at such a low level flattens any structure within it. Therefore, conventional geometric representations alone are not suitable.

We think this design representation, then, should represent construction details as a **language that exposes abstractions of domain-specific physical operations**. First, digital fabrication systems ultimately execute machine-level code, so we can intuitively design abstractions over that machine code. Second, a sequence of instructions is already how we communicate things like recipes or how-to guides, and has the same form as imperative languages. This ability to customize our abstractions is especially powerful when applied to nonstandard domains or specialized

subdomains. With this internal representation, given an input shape, the system discovers and shows the user all valid plans to make it.

However, it may be that the set of operations is too large, and the possibilities for composing them too complex, to be show the user every valid plan that satisfies a shape. In that case, we need a second step: a way for users to traverse the design space so that they can directly specify their desired plan and iterate to explore the space of plans. If a user has a concrete vision, they should be able to write it down easily; if that initial attempt isn't right, they should be able to explore easily. We think users should do this by inputting information about the construction plan, so we should create **an input representation that combines partial geometry with construction details**. This representation should be directly editable, and users should be able to preview the result from **end-to-end for ease of iteration (i.e. user-in-the-loop)**.

Thus, this thesis argues that design systems for the physical world need a design representation combines aspects of the geometry with aspects of the construction plan. This representation should be a domain-specific language that with three components: geometry as a partial input, construction details via abstractions of domain-specific physical operations, and support for a user-in-the-loop workflow.

To elaborate, first, these representations interface with geometry by making it a partial input. Shapes and images still contain rich information about the object's appearance and behaviour. Users should specify and author geometry to modify these directly.

Second, they provide abstractions of the construction process via imperative, domain-specific operators. These should expose construction details while being higher-level than machine code. Imperative languages reflect the physical reality of making as an ordered sequence of operations, and enable modularity, invariant maintenance, and realizability guarantees. These details should be used by the system, but should also be specifiable by the user as input.

Finally, they are designed to support user-in-the-loop interaction, so we need support for fast

iterations with previewing and direct editing so for the user to drive the result.

To show how this thesis applies, we look to creating systems for designing in novel, surprising, and creative domains. We establish the first step with one project in machine knitting. We create a system that enables the design of knit illusions, a creative sub-domain of machine knitting, by abstracting over the domain-specific operations involved in creating them. We use images as inputs to the system, and allow users to choose the final fabrication plan.

Then, with a second project in machine knitting, we show how expanding to domains with richer geometry, non-sheet topology, and more varied fabrication operations demands the second step: combined geometry and construction information as input, and a user-in-the-loop workflow to aid users in directly editing that input. We develop *Panel*, a scheduling DSL to describe the shape *and* construction plan of a knit design. *Panel* features 2D panel geometry as the geometric input, and a description of their construction, given as scheduling operators describing how the panels are made and connected, forms the abstraction of domain-specific operations. We show how users are able to edit the input shape or construction details directly to create the object they want.

Finally, we extend the thesis statement beyond knitting by using the same principles to design for a new domain of electromechanical displays. These displays physically move between states as they are being dynamically updated, and so different schedules for how they move affect what viewers see. This project uses animation keyframes as geometric inputs, a customizable set of operators expressed as *Effects* that describe how units physically flip between states, and a user-in-the-loop system that facilitates the creation of novel perception-based designs and emergent effects.

Chapter 2

Why care about construction?

Picture a sock in your head. What does it look like? Maybe a merino hiking sock with a fun outdoors-themed pattern? Maybe a dress sock of a very fine gauge? Maybe a 30-denier nylon sock? I think it's very unlikely that you just imagined something that is "the shape of a foot". Of course, each design is a partial specification, and we'll eventually reify it through iterative decisionmaking into the final actual object. Yet a vision of a design without any information about how it's made feels incomplete.

Today, after millennia of development across all kinds of crafts and industry, we have a full landscape of creation methods, of techniques, of materials, of physical and chemical and computational and embodied knowledge. If you need a piece of fabric, you can do anything from covering some animal hide in fat and smoke it into leather, to taking tufts of wool and agitating them so the fibres stick together as felt, to growing bast fibres, retting, breaking, scutching and hackling them before spinning them into thread and weaving them into cloth.

Even within a specific discipline, methods for construction vary greatly. In weaving, thread weight, density (sett), weave structures, finishes, all have profound effects on the final product. If there are surface patterns on the cloth, are they dyed before weaving (like ikat), dyed after

weaving (like batik), woven inlaid (like jacquard), printed on yardage (like with digital printing), embroidered, or perhaps even applied to the final garment (like screen printing)? Each of these methods has different properties and leaves different traces.

These traces apply down to the specifics of the machine that makes it. Even comparing automatic programmable jacquard looms, flying shuttle looms automated the process of throwing the weft thread back and forth, and then airjet looms streamlined the process further by removing the shuttle and simply propelling a single warp on its own through the warp. The artifacts of the differences in the process are visible in the selvedge (the edges of the fabric): because flying shuttles use one continuous thread, the resulting fabric has tight, traditional selvedges with the yarn turning around and “self-finishing”; airjet looms have individual cut lengths of thread, and so typically have leno selvedges with fringes. These differences may have implications for construction, or they may just be identifying clues as to the object’s origins.

All this to say: if you have a design, there are many ways to realize it. And if you have an object, there are many ways it could have been made. We should encode information about its construction in our formal specification for designs, too.

I attempt to fulfill this idea directly by creating self-embedded knitting programs [Zhu et al.(2023)], where the fabrication program would be explicitly decodable from the design. We viewed this as a quine: *embedding the fabrication program within the fabrication program*. Theoretically, by “executing” the program embedded in the design, we can replicate the object. Importantly, the fabrication plan is necessarily decided with the rest of the design, rather than solved-for afterwards.

However, quines are famously inflexible, and so this experimental exploration is quite limited in the objects that we can express. Therefore, I want to present three other projects that show how our philosophy for a design representation leads to general, flexible methods for expressing construction details.

Chapter 3

Knitting Background

To economize on space, I will give a short introduction to knitting here, which will serve the next two chapters.

3.1 Knitting Basics

Knitting is a structured process of forming fabric from a single continuous strand of yarn by looping it onto itself repeatedly. Figure 3.1 provides a visual representation of the structure of knitted fabric. The fundamental unit of knitting is the *stitch*, which is created by pulling a new loop (the *child* loop) through an existing loop (the *parent* loop). Arranging these stitches in a grid-like pattern forms the fabric: horizontal sequences of stitches are called *courses*, while vertical columns of stitches are known as *wales*. Adjacent loops within the same course are called *neighbors*. Knitting begins with the creation of an initial set of loops. Since there are no parent stitches to pull these loops through, a special type of stitch is used to create them, a process known as *casting on*. As new stitches are formed, the topmost row remains *live*, meaning these loops are unsecured and can unravel if left unfinished. To complete the fabric, the live stitches must be *cast off*, a process in

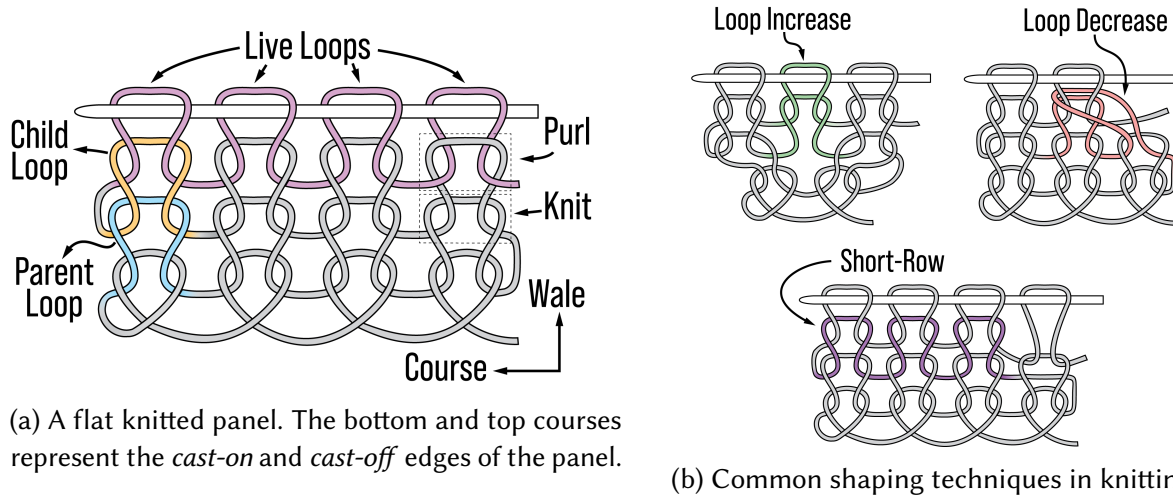


Figure 3.1: The structure of a knitted panel is formed using a single piece of interlocking yarn. Child loops (orange) are pulled through parent loops (blue) to prevent the thread from unraveling. Unsecured live loops (pink) must be held on the needle during construction. Horizontal and vertical sequences of stitches are referred to as *courses* and *wales*. Shaping and curvature can be introduced into the panel by strategically adding and removing loops.

which neighboring loops are pulled through one another to lock the structure in place.

Shaping and seaming

The principles described above enable the creation of more complex shapes beyond rectangular panels. By *decreasing*—pulling one child loop through two parent loops—or *increasing*—creating two child loops from one parent loop—we can alter the number of wales in the fabric. Additionally, *short rows*, where a full row is not completed, can change the number of courses. Together, these techniques (illustrated in Figure 3.1b) form the foundation of shaping and curvature in knitting. To transform a flat piece into a tube, we can either sew the edges together, a process known as *seaming*, or knit continuously in a spiral by joining the loops in a circle, a technique called *knitting in the round*.

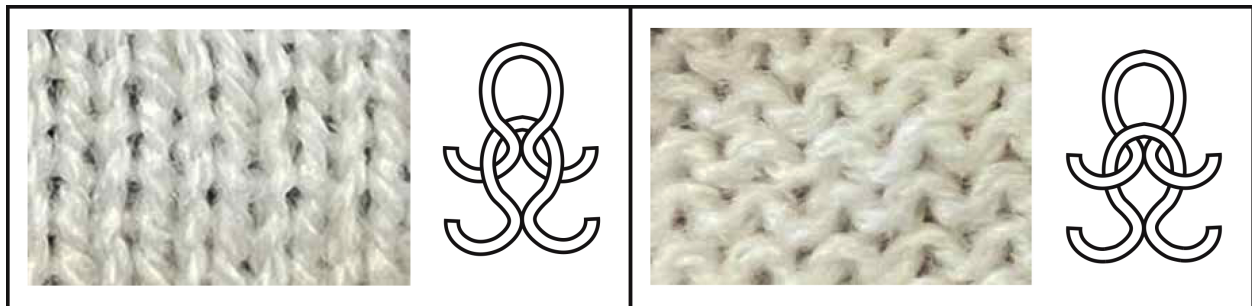


Figure 3.2: A basic view of knits and purls. On the left is a photo of a fabric made of only knits next to a diagram of a single knit; on the right is a photo of a fabric made only of purls next to a diagram of a single purl.

Texture

Surface texture in knitted fabric is introduced by varying how loops interlock. For instance, by alternating **knit** and **purl** stitches (shown in Figure 3.1a and in Figure 3.2), twisting loops, overlapping loops, or making intentional holes we can create surface *textures* such as stockinette, garter, or ribbing. For example, the garter texture alternates courses of knits and purls; the rib texture alternates wales of knits and purls.

Additionally, leaning increases and decreases — where loops are manipulated to slant left or right — can add further visual interest. When aligned in a row, these stitches can form a decorative seam-like texture, distinct from true seams that occur when separate knitted panels are physically joined. Full specifications of knit designs are called *patterns*.

Aesthetics

Several factors affect the appearance of knit objects. Knits and purls are *duals*, meaning that a purl viewed from the back is a knit, and vice versa. Thus, knit fabrics often have a “right side,” or a “technical face,” from which the pattern is designed to be viewed. Typically, designers focus on the aesthetics of the technical face. Additionally, there is a metric consideration since the size

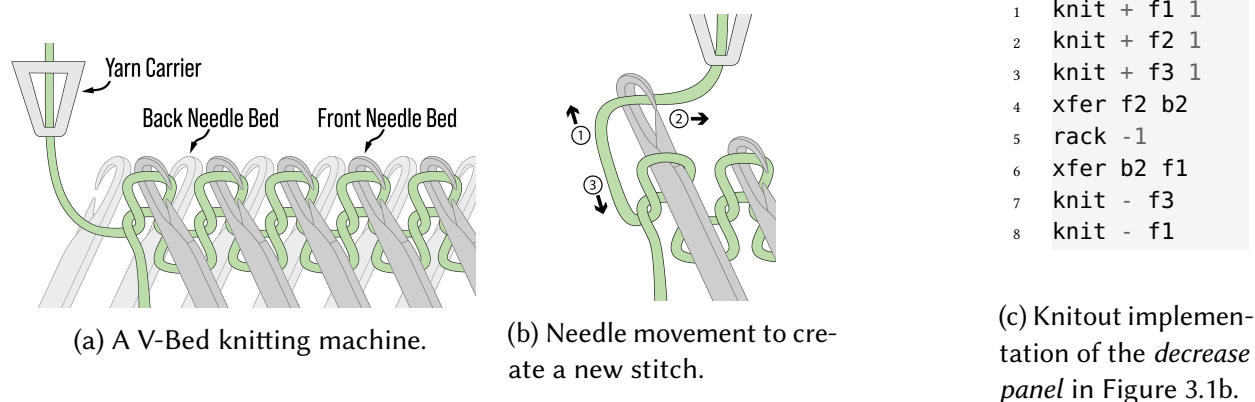


Figure 3.3: V-Bed knitting machines use two opposing beds of needles to construct the garment. To create a new stitch, (1) the needle is lifted and yarn is laid on the hook by the yarn carrier, (2) the needle latch closes around the yarn, and (3) the needle pulls the thread through the live loop it holds to create the stitch. Low-level assembly programs, which specify behavior at the level of individual stitches, are used to operate such machines.

of a knitted object is directly related to the arrangement and size of the loops. The *stitch gauge* describes the number of stitches per unit distance. In hand knitting, the gauge is controlled by the size of the needle used for creating the loops and the size of the yarn, while in machine knitting it is determined by needle size and how close the needles are arranged on the beds.

3.2 Machine Knitting

When knitting by hand, two needles are used to knit course by course, one loop at a time. Needle A holds the loops of the course, while needle B picks up each live loop, replacing the parent loop with a new child loop on needle A. Hand knitting offers great flexibility, allowing knitters to freely increase, decrease, and knit into non-live loops. They can also pick up stitches to work in a different plane. Beyond traditional techniques, hand knitting can incorporate elements from other crafts, like crocheting, to expand design possibilities.

Machine knitting differs significantly from hand-knitting in the way knit structure is created.

While hand-knitters manipulate one loop at a time, adjusting tension and stitch formation as they go, knitting machines automate this process through rigid, predefined movements. Although different types of knitting machines exist, the V-bed architecture (illustrated in Figure 3.3a) is the most common, particularly in industrial knitting.

V-bed knitting machines consist of two beds of needles arranged in an inverted V-shape. Each needle is indexed and holds a live loop of yarn. Yarn carriers run down the middle between the needle beds, causing stitches on the front bed to always form knit stitches and those on the back bed to always form purl stitches. A moving carriage slides back and forth across these beds, activating the needles as required. When activated, the needles pull yarn through the live loops they hold to create new stitches, as shown in Figure 3.3b. The strict separation between knit and purl operations—knits limited to the front bed, purls to the back—creates challenges when programming stitch textures, managing loop transfers between beds, and controlling lateral needle-bed movements (racking).

Knitting machines are operated via programs that specify machine-level operations. For example, Figure 3.3c shows a knit program implemented in the abstract assembly language Knitout [McCann et al.(2016)] that generates the decrease example from Figure 3.1b. The instruction `knit + f1 1` creates a stitch on the live loop held by needle `f1` (front bed, 1st needle), using yarn carrier 1. The `+` symbol indicates that the yarn carrier moves from left to right. The two needle beds on a V-bed machine are initially aligned so that needles with matching indices face each other. Users can adjust this alignment or transfer loops between opposing needles using instructions such as `rack` and `xfer`.

Due to the low-level nature of these instructions and the constraints imposed by the machine architecture, even simple actions, such as creating a single decrease, requires multiple operations. As shown in Figure 3.3c, to create a decrease between two adjacent loops, we first transfer the left loop to the back bed (line 4), update the bed alignment such that the left and right loops are held



Figure 3.4: Examples of colorwork techniques, with the front face (top row) and back face (bottom row). *Doubleknit jacquard* embeds a different image on each side. *Stranded knitting* results in long, unknitted yarn floats across the back. *Intarsia* is seamless from the front, but the yarn-catching used to keep the two different-colored fabrics together is visible along the vertical join on the back. *Plated* produces the inverse of the front design on the back.

by opposing needles (line 5), and finally, we transfer the left loop back to the front bed (line 6), placing it onto the same needle that holds the right loop. Translating high-level designs into such fine-grained instructions is non-trivial; programmers must carefully account for yarn tension and slack at each step, track the state and alignment of needles, and ensure they are positioned correctly to generate the intended stitch patterns. Furthermore, making high-level design changes to existing programs is challenging and often infeasible, as it typically necessitates reworking the entire sequence of low-level instructions.

3.3 Colorwork

The following text on colorwork is taken from the Illusion Knitting paper [Zhu et al.(2024)]. There are many existing techniques for colorwork. *Plating* works all yarns together, producing color patterns by switching which yarn sits in front. *Stranded* knitting passes unused colors behind

stitches, forming strands (a.k.a. floats) along the back. *Intarsia* portions the knitted item into single-color blocks, intertwining the yarns at the borders of the blocks. *Doubleknit jacquard*, arguably the most common technique for machine-knit colorwork, uses all needles on both beds. Importantly, these techniques do not systematically integrate with textures. We now provide a more in-depth explanation of each colorwork technique and illustrate each in Figure 3.4.

Stranded Colorwork In stranded colorwork, the knitter transports all yarns as they knit, even if they are not actively in use. Yarns that are not being used for the current loop lie along the back. These loose strands, or *floats*, end when that color’s yarn is returned to use, but overly long strands are undesirable. To avoid this, stranded designs employ small motifs close together to keep color changes frequent. Thus, stranded colorwork is appropriate for regions with frequent color changes, but it incurs unnecessary material cost and complexity when a carrier is crossing large swaths of other colors. Also, machine knitting stranded colorwork when patterns have purl textures requires special handling because a naive program will create floats in front of any purl stitches on the back bed.

Intarsia Instead of carrying every yarn when it is not in use, intarsia knitting divides up regions of colour such that each region is knit with just one yarn moving back and forth. Sometimes, due to how the colour regions are arranged, the project may require two or more instances of one yarn color. Intarsia produces fabric that matches the drape and feel of a single-color knit, but because it requires a new yarn for each color change in a course, it is meant for designs that have larger color regions and few color changes. In general, carrier positions diverge for intarsia, i.e., carriers may move in different directions and be in different ranges of the needle bed. As a result, carriers may not be appropriately positioned to knit the next course and might require extra instructions to ensure correctness. In addition, knitting each section with a different yarn

produces disconnected pieces that require explicit joining.

Plated Knitting In plating, each loop is made with two yarns pulled through the parent loop. On the machine, we can control the appearance of a plated stitch: whichever carrier is “first” contributes most of the color, with the second carrier barely peeking through. Colorwork pieces can be constructed entirely from plating; however, we avoid it because the thickness of plated stitches interferes with the illusion, making knits much taller than non-plated knit stitches.

Doubleknit Jacquard Jacquard knitting is a very common machine knitting colorwork technique that creates a double-sided stockinette. Notably, the color pattern on the front may completely differ from that on the back. Both the front and back beds are used (and in all-needle jacquard, which is the standard for machine knitting, all needles are used) and the two sides of the fabric are held together when the yarn crosses from one to the other. Because both beds are used, it is impossible to use the back bed needles for purls or transfers at full gauge, which means texture is not easily accomplished.

3.4 Higher-Level Representations

To alleviate the difficulty of programming directly in low-level machine languages like Knitout, researchers have developed higher-level representations of knit structures that can be automatically compiled to languages like Knitout.

KnitGraphs [Narayanan et al.(2018a)] represent knitted structures as graphs, where nodes correspond to loops and edges represent their connections across wales and courses. This abstraction allows users to express design at the level of stitch topology rather than specifying each machine operation manually. However, while these representations abstract away machine-level details,

they still operate at the yarn level, meaning that design reasoning – such as shaping or texturing – must be done by manipulating individual loops.

Remeshing algorithms that automatically convert 3D meshes to KnitGraphs and offer knittability guarantees have been proposed [Narayanan et al.(2018a)]. However, while these black-box approaches are powerful, they offer limited control over the fabrication plan. For instance, the algorithm proposed by [Narayanan et al.(2018a)] only allows users to specify a starting point for knitting, after which the system autonomously produces the entire KnitGraph as a single seamless shape.

Chapter 4

Illusion Knitting

This text is taken directly from the previously-published work [Zhu et al.(2024)].

4.1 Introduction

Unique and visually interesting view-dependent effects have been developed for many fabrication methods and mechanisms, including scratch-based reflection, 3D printing, shadow casting, and more [Shen et al.(2023), Alexa and Matusik(2011), Bermano et al.(2012)]. However, there is currently no support for view-dependent effects in knitting. Intrepid artists have described manual, hand-knitted view-dependent effects (dubbed “illusions” [Zhu et al.(2024)]), but their approaches are tedious, slow to knit up, and limited to single-view; it takes hundreds of hours to design a knitted illusion, and hundreds more to actually fabricate the knit object. Though machine knitting is an obvious choice to speed up fabrication, no compilers for illusion knitting currently exist. Nor are there machine knitting scheduling algorithms that support arbitrary intra-row texture and colorwork, necessary for maximizing design freedom. Furthermore, artists have created only single-view illusions, which depict the same image from both side views, but we imagine enriching



Figure 4.1: We propose a framework for computational illusion knitting that enables simple production of traditional and completely novel illusion designs, and show the fabricated designs generated with our method. On the left, Leonardo da Vinci’s Mona Lisa (public domain) appears when viewed from the side. On the right, Vincent van Gogh’s Self-Portrait with a Bandaged Ear (public domain) appears from one side, his Sunflowers, fourth version (public domain) on the other. The rightmost top two images are the tool inputs; the bottom two are rendered predictions of the illusion pattern.

the design space by creating novel illusions that embed multiple images.

We create the first design system for illusion knitting, enabling design without cumbersome manual effort, pushing the boundaries into multi-view illusions, and correctly compiling to machine knitting instructions.

First, we identify how illusions work. Illusion knitting emerges from stitch-level *microgeometry* on the surface of knits (depicted in Figure 4.2); we contribute a set of observations about microgeometry formalized as constraints, such that we can directly construct these building blocks. These constraints capture both *viewing behavior*, i.e., what image is seen at an angle, and *physical behavior*, i.e., the result of different knitted design and fabrication choices.

Formalizing these constraints provides immediate benefits: for single-view, we can easily satisfy the constraints to generate knittable illusions. Furthermore, as traditional single-view illusions change color only between rows, we require no new machine knitting algorithms. Therefore,

we can quickly design and computationally fabricate single-view illusions, like the Mona Lisa in Figure 4.1, given a user-provided three-color image.

Trying to extend these same solutions for single-view to multi-view illusions, we discover that they are inadequate and so we must invent other strategies. Our key contributions focus on new techniques for multi-view illusions.

Unfortunately, the constraints for multiple images are almost never satisfiable. Our insight is that *we can relax these constraints* and search for the best compromise. We could do this using automated methods, such as optimization or MaxSAT. However, we believe it important to begin with a framework that lets users directly, interactively, and iteratively edit designs, since an end-to-end automated system cannot return a perfect result for every user 100% of the time. Because users employ human perception to gauge whether designs are readable, we create a *designer-in-the-loop framework* that uses insights from craft illusion knitting to simplify the task of improving illusion quality to iterative image editing. This framework relies on a new tile-based system to systematically relax constraints while giving users direct control. Creating a framework also preserves extensibility as we can plug in other automated algorithms.

We then face the issue of new designs intermingling texture and colorwork within rows. Given this task, existing machine scheduling algorithms fail to create objects without unworkable physical flaws. To fabricate knit illusions quickly, we introduce *new scheduling algorithms for texture and color changes within a single row of knitting*, which is necessary for multi-view illusions.

Our constraint-informed, human-in-the-loop design system enables fabricating the first double-view illusions, supporting diverse image types across graphics, text, paintings, and more. Additionally, our new scheduling approach avoids many failures that arise with naive scheduling, producing successful knit illusions.

As evidence, we present multiple complex knit illusions never previously created, examples of which we show in the teaser (Figure 4.1). We compare the output of our iterative, image-editing



(a) A top-down look at a single-view pattern. The knitted version is depicted in Figure 4.3b.



(b) A close up of the microgeometry of single-view patterns. At the bottom is a row of yellow purl bumps, below blue and yellow knits. The yellow purl bumps are raised above the fabric.

Figure 4.2: Physical view of illusion patterns and microgeometry.

design system with other systems. Finally, we visually compare the fabricated results of knitting with and without our new scheduling algorithms, where the latter have notable holes and visual failures.

4.2 Overview

Single-view illusions show that exploiting knit microgeometry can achieve surprising designs, and we intend to push the boundaries to discover even more possibilities using computation. In this work, we consider illusions that embed **two** input images, A and B, where image A is obvious from one side and image B from the other side. However, we design approaches that can theoretically extend beyond two inputs. Additionally, since the landscape of knitted microgeometry is very diverse, we make it navigable by restricting the discussion to illusions created via solid-color microgeometry with two possible heights (flat and raised).

To create a design system, we must first identify what is possible in terms of the appearance of knitted microgeometry when viewed. Doing so involves knowledge of fabrication as well as physical experimentation. We draw on these insights to develop new, constrained design spaces using logical constraints and a new design system for traversing them.

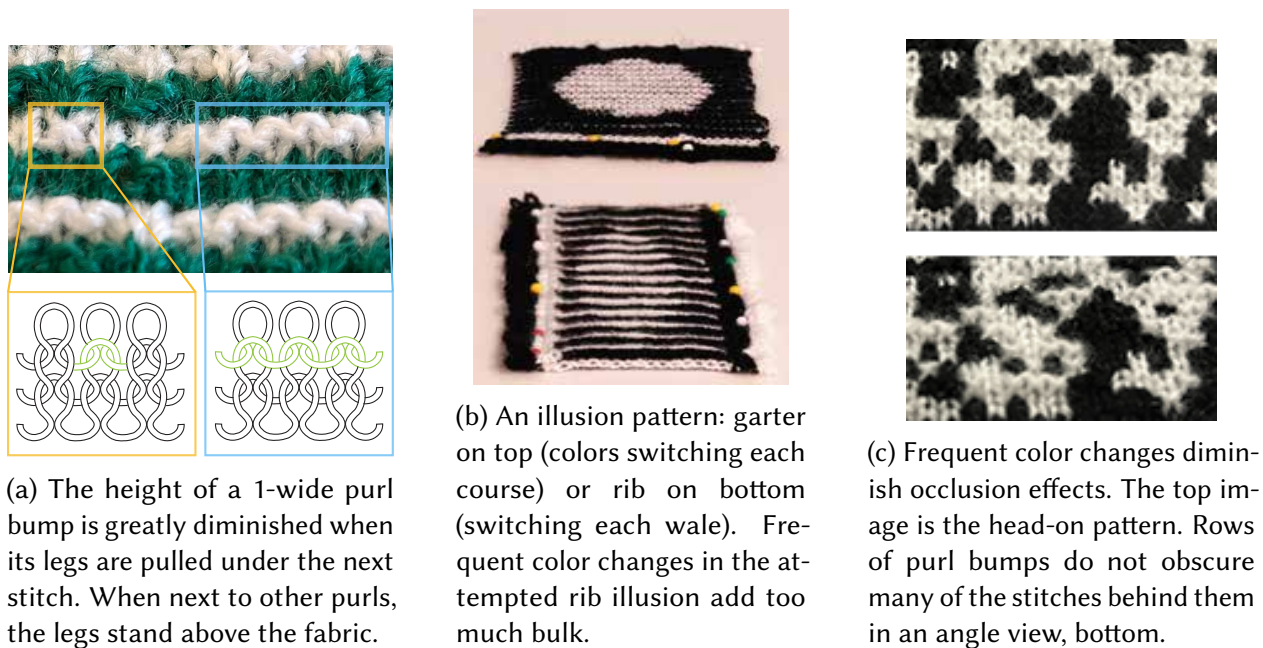


Figure 4.3: Some pattern choices negatively affect occlusion. Without carefully arranging stitches, occlusion effects do not appear.

4.2.1 Insights from Knit Microgeometry

To characterize a knit object’s microgeometry, we examine empirical examples of knit illusions. Our goal is to abstract over the complexities of knitting physics so we can streamline illusion design.

Combinations of knit and purl stitches create raised *bumps* that may occlude previous stitches. A bump is created from a knit with another loop pulled through it front-to-back; in other words, a bump is made of a knit with a purl as a child. Not every combination creates a crisp occlusion effect, and we present some mistakes in Figure 4.3. First, a single bump is not sufficiently tall to occlude a stitch behind it from almost all angles. The legs of a bump flanked by two flat stitches are pulled down, shown in Figure 4.3a. Knit stitches alone remain *flat*. Second, directionality matters. Theoretically, we could view and produce bumps along wales rather than courses, working in ribbing rather than garter stitch. However, we find that height differences are more pronounced in garter, as shown in Figure 4.3b. Third, different programs and techniques might produce excess

bulk, whether in the form of extra “floating” yarn at the back, doubling up the yarn used per stitch, or via other techniques. This bulk reduces the height difference between flats and bumps, diminishing the illusion effect. We can avoid some techniques that accrue extra bulk, but not all: each color change within a row inevitably incurs some added bulk.

From these insights, we draw three conclusions. From the first, we choose to introduce an abstraction over these physical constraints. In this *logical* space, flat and bump *units* are individually compiled to stitches such that the bumps are guaranteed to occlude other stitches at an angle, i.e., each pixel becomes a 2x2 block of stitches. From the next insight, we define our objective to view the first input image from the left side of the rows and the second input image from the right side of rows. From the final insight, we conclude that our designs should mindfully reduce the number of color changes.

4.2.2 Defining a Design Space

To identify which double-view illusion designs can be knit “perfectly” in principle (ignoring practical fabrication concerns), we formalize the constraints for when $N \times M$ pixel images A and B will be obvious within the result of knitting the logical (color and bump level) pattern P , where $C_{i,j}$ and $R_{i,j}$ indicate the color and raisedness of P at row i and column j , respectively. Since the first row will not be occluded by any previous rows for A , and similarly for B with the last row, we require $C_{0,j} = A_{0,j}$ and $C_{N,j} = B_{N,j}$. For subsequent rows, we have $\neg R_{i-1,j} \implies C_{i,j} = A_{i,j}$ and $\neg R_{i+1,j} \implies C_{i,j} = B_{i,j}$ (i.e., if the cell in the previous column is flat, then our current cell will be visible and must match the color of the corresponding cell in the input image), as well as $R_{i-1,j} \implies C_{i-1,j} = A_{i,j}$ and $R_{i+1,j} \implies C_{i+1,j} = B_{i,j}$ (i.e., if the cell in the previous column is raised, then it will occlude our current cell, so the cell in the previous column’s color must match our current corresponding cell in the input image). These constraints implicitly specify visual

rendering restricted to booleans; we present a full rendering function in Section 4.3.

The boolean rendering highlights the fact that multi-view illusion knitting is over-constrained: for the vast majority of pairs of images, no pattern can satisfy all the requirements. Intuitively, without bump units, our images cannot differ since bump units occlude the units behind them and create the illusion. However, *the color of every bump unit must match both images* since it will be visible from both sides.

We can attempt to minimize error using automated techniques, but human perception ultimately decides illusion quality. Therefore, we establish a first human-in-the-loop framework that can support both automated design as well as interactive, iterative editing.

To do so, we abstract over the logical space to create an editing space for designers. This design space enables robust approaches to generating obvious and knittable illusions by *reducing the development of illusion patterns to image editing*.



Consider the pair of inputs from the inset figure. These inputs are unsolvable: one image demands white, where the other demands black.

How can we create a double-view illusion at all?

To begin, we observe that using the logical constraints specified previously, it is possible to have the left and right views differ. Consider a 3 by 1 repeating tile, i.e., white flat, black flat, and white bump. From the right, the white bump occludes the black flat, and the image appears white. From the left, we see the white bump **and** black flat, creating a black and white striped “fill.” Incidentally, we notice that in craft illusion knitting, simple repeating patterns are occasionally used as fills; for example, a striped black-and-white pattern looks like a medium gray against solid black and solid white. Leveraging the idea of black/white stripes as a fill, we can substitute the solid black in both inset inputs with a black/white stripe to make this illusion possible using our new tile, as shown in Figure 4.4.

This approach provides yet another new insight; we can modify input images to make robust

double-view illusions possible. In other words, we must relax the illusion constraints. Doing so requires striking a balance between three factors: (1) constraint satisfaction (i.e., does the pattern actually replicate the input image?), (2) knittability (i.e., is it actually feasible to create the pattern?), and (3) readability (i.e., if we modify the input to improve satisfiability, will the modified input still be perceived as similar to the original input?). For example, relaxing all inputs into a white square would make them knittable and constraint satisfying but not readable as most input images. Or, prioritizing satisfiability, we could use an approach such as MaxSAT or optimization. However, using these relaxation methods might make it more difficult to control knittability and readability, as shown in Section 4.5.3.

To successfully navigate this tradeoff, we can draw on some observations from craft illusion knitting. Knit illusions become less readable when fabricated. With a pixel mapped to 2x2 stitches, the resulting knit objects are large, meaning input images are typically low-resolution. As cloth, the fabric is wavy, so details are often swallowed up. In response, artists segment images into large swaths of each color and avoid dithering, instead employing regular, repetitive patterns to make the borders between colors distinct. We must relax the input simply and *consistently* to preserve readability. This relaxation should be editable so the user has control.

A tile-based system balances our three concerns. Repainting input regions with these new fills makes the constraints within each region satisfiable since the fills are backed by tiles. Tiles are designed to be fabrication-sensitive; our chosen patterns create unbroken courses of color within each tiled region. This property results in shorter fabrication times with less complexity. Tiles also consistently relax the input; all instances of the same colored regions are repainted with a color or a simple, regular fill, maintaining readability. Best of all, relaxation is controlled by the user when they define the color regions of the input and decide how to repaint them. Users can fine-tune elements by editing the quantized design, reducing the problem to image editing, which can be performed in any familiar software.

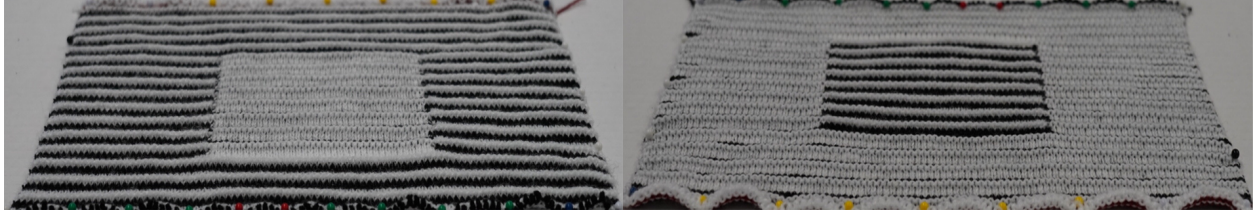


Figure 4.4: A knit version of our recolored input patterns. (Left) The black background and white square have become a striped background and white square. (Right) The white background and black square have become a white background and striped square.

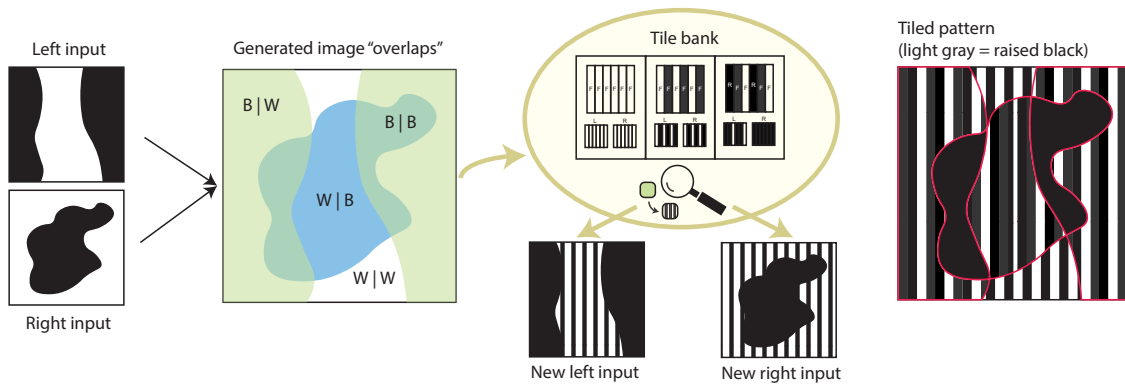


Figure 4.5: The workflow for our tile-editing framework. (Leftmost) Our original inputs for each view, with large, simple color regions. (Center left) The system then overlaps the images, creating many new smaller regions where the two inputs had different colors. For example, the blue region in the middle should be white on the left and black on the right. (Center right) The system repaints the inputs with new fills. It must assign a new fill to each color c in the original left input such that for each overlap region $c|d$, there is a tile that shows c on one side and d on the other, and same for the right. (Rightmost) The final tiling.

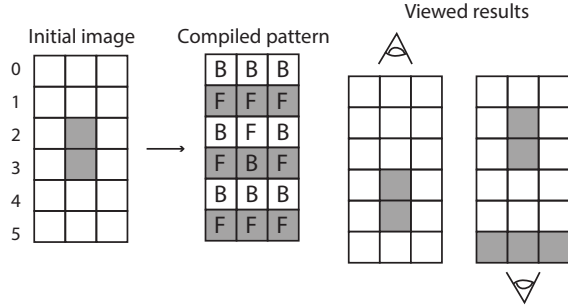
Thus, we propose a *tile-based framework* for creating illusions. We provide a *tile bank* of template tiles that have known fills from each view, and which can be customized and extended. The user provides simplified and quantized input images. The framework then overlays the two inputs onto each other, recording the original color specification. It then repaints the original colors in the image with new fills, drawing from the tile bank for the matching tiles (Figure 4.5). Crucially, multiple tiles might map to the same fill on one side but to a different fill on the other side. The tool then displays the predicted knitted output, and users are free to edit what tiles are used.

In this way, editing becomes as simple as modifying the input image regions with a preferred image editor, knowing that knittability and satisfiability will be preserved.

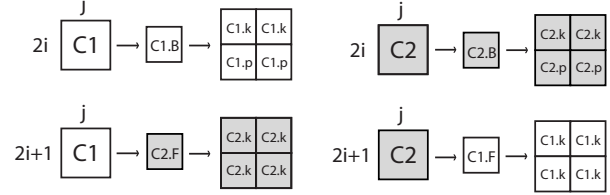
4.3 Foundations of illusion knitting

To reason about correct illusions, we seek to establish a formal understanding of illusion knitting. We start by describing a function called `can_see`, which mathematically identifies visible logical units in the side view. The definition of `can_see` is given by the constraints defined in Section 4.2.2, such that `can_see` from one direction is given by the constraints for A and from the other direction given by the constraints for B. Using `can_see`, we define a “perfect” illusion as one where the pattern viewed from the side exactly matches the input image.

Constraints for Single-View Illusions. We use `can_see` to write an algorithm for generating single-view illusions from a two-color input. Our algorithm, shown in Figure 4.6, is heavily inspired by craft examples. We alternate striped rows of colors 1 and 2 (c_1 and c_2). However, we translate only c_1 pixels to bump units if the stripe row is c_1 -colored, while c_2 pixels in c_1 rows are translated to $c_1.flat$ units. This algorithm produces SAT patterns (modulo boundary rows).



(a) An example of our compilation algorithm and the result of `can_see`. The two rightmost images are the viewed image from the top and bottom, respectively, according to `can_see`.



(b) A visual description of our single-view image-to-illusion compilation algorithm. Each pixel in the input is compiled first to a logical unit and then to a physical unit depending on its row of origin.

Figure 4.6: Illustrations of the single-view compilation algorithm.

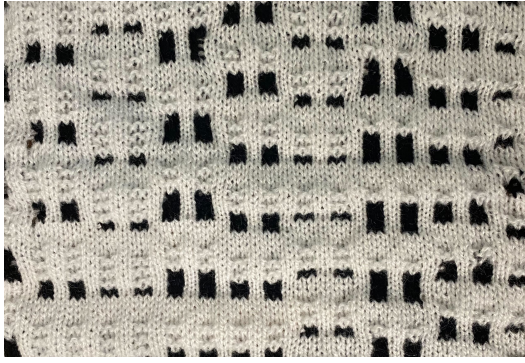


Figure 4.7: Our calibration pattern. We knit sets of various sizes of white bumps next to various sizes of black flat stitches. Our examples use different colored yarns of the same dimensions; however, given varied yarn sizes, we could knit a different version of the calibration pattern, with each distinct yarn replacing the white stitches, to learn the height of purl bumps for each yarn.

While we ignore $c2$ pixels in $c1$ rows, we raise the stitches only where $c1$ is used. So, a $c1.flat$ will never block a $c2.flat$ in the next row. Given that all pixels are either $c1$ or $c2$, $c2$ pixels in row n (when n is a $c1$ row) will appear in $n + 1$ in the knit result.

For inputs with an intermediate third color, we can extend this algorithm to translate $c3$ to two adjacent bump rows. This modification maintains approximate correctness.

Accelerating design for single-view illusions is simpler than our goal of doing so for double-view, but we are still content that our push-button solution for single-view can replace hundreds of hours of work, lowering the barrier to entry for designing such illusions.

Differentiable Renderer Though we have a formal renderer defined via our constraint system, we additionally observe that the angle at which the image is viewed affects perception. Providing a more flexible, accurate view is essential for interactively exploring the design space. In real life, viewers will naturally move around to find the sharpest image. Our new ray-casting renderer is differentiable, enabling optimizations to be built on top of it.

Two simplifying assumptions allow for autodifferentiating stitch occlusion. First, we treat logical stitches as cuboids of fixed height. Second, we fix a camera that is orthographic with respect to the width. As a result, all predictions are rectangular, which aids when fitting an illusion knit pattern to the target input. Thus, viewing rays can only intersect with the leading or top faces of each stitch cuboid within one wale. The 3D rendering problem is thereby reduced to a single 2D problem with wales computed independently.

To render a wale, we analytically compute the contribution of each line segment S_i to each pixel P_i , accounting for occlusion by earlier segments $S_{j < i}$: $| (S_i \setminus \cup_{j < i} (S_j)) \cap P_i |$. Here, S_k and P_k are the projections of the k -th stitch and pixel onto image space. We show a differentiable closed form expression in the Supplemental Material.

To tune the renderer’s parameters, we knit a test pattern, shown in Figure 4.7. We measured the angle at which each purl bump configuration obscured the opposite-color stitches and used this to calculate the height of the purl stitches. The same pattern can be used to tune parameters for a new setup, e.g., when using a thicker yarn or different gauge needles. We show the results of rendered outputs compared to knit outputs in Figure 4.12. The rendered output strongly replicates the real, physical illusion effect, successfully making immediate visual feedback possible.

4.4 Fabrication

The illusion patterns we designed require changes between knits and purls and changes between colours within a single row. In hand knitting, the knitter is entrusted to make decisions handling yarn behaviours, e.g., whether the yarn runs along the front or back of the knit object or whether to twist the yarns around each other when they need to catch. In machine knitting, these subtle details must be concretized and managed. Furthermore, operations that non-experts might expect to freely compose, such as knitting a section in one yarn and a second section with another yarn, do not do so because of the implicit changes in machine state that are not explicitly surfaced to the level of abstraction we use to program the machine. Thus, to automatically compile our designs to knitting programs, we must develop new mixed-colorwork-and-texture-aware scheduling strategies to ensure correctness.

4.4.1 Combined Colourwork Strategy

Single-view knits posed no scheduling issues. With tiles, however, we must account for color changes within a row, which entails mixing colorwork and texture. Existing machine scheduler algorithms [McCann et al.(2016), Narayanan et al.(2019), Lin et al.(2018), Lin and McCann(2021), Lin et al.(2023)] are not designed to support both switching between knits and purls and switching colours within a row, as is needed in illusion patterns. One work [Albaugh et al.(2023)] deals with the mixed colorwork and texture required for *brioche knitting*, a well-known special style of knitting that uses a particular rib texture and usually employs a colour change to highlight it; we still need new strategies for the arbitrary mixed colorwork and knit-purl texture demanded in illusion knitting.

For more background on colourwork, please reference Section 3.3.

We propose a novel colorwork strategy that supports the colorwork and textures found in

illusion knits, based on four key properties of illusion patterns: (1) bumps must be clear so that the illusion is visible; (2) patterns often have many large regions or long runs of a single color; (3) there may be areas with dense color switching, perhaps if there are many small regions; and (4) purls may occur anywhere, and for the knitting machine, where purls are created on the back bed, this means that the back bed cannot freely be used.

From (1), we decided against plating for colorwork since bulky knit stitches interfere with the illusion. Considering properties (2) and (3), neither stranded nor intarsia obviously prevails since one is better suited for frequent and the other infrequent color changes, respectively. We therefore propose a best-of-both-worlds combination of stranded colorwork and intarsia, but we must still address technique-specific challenges. In stranded, floats should not appear on the front of the knit item; in intarsia, different yarns must be securely connected so the piece does not unravel. While solutions exist in typical contexts, they may not apply when combining these techniques or under the constraints imposed by (4). Thus, we introduce three compiler passes to address each concern.

Float Avoidance Floats appear when a carrier knits on one part of the bed, then moves to knit on another section of the bed. Because the yarn is not cut, it runs loosely along the work. Since carriers move in the space between the front and back beds, floats appear on the front of the work when this movement passes in front of stitches on the back bed. To avoid floats, whenever a carrier moves freely (i.e., not while knitting), we check for purls it might cross in front of, and surround the carrier movement with extra transfer instructions such that stitches move to the front bed before the carrier passes and return to the back bed after. This operation incurs two extra carriage passes per free carrier movement if there are back-bed stitches to transfer, so checking that this operation is needed saves time.

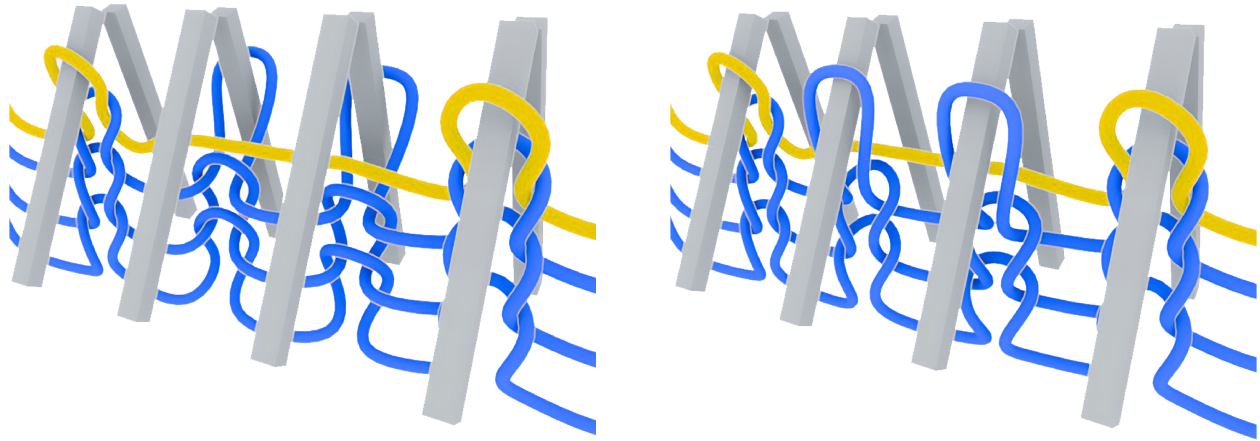


Figure 4.8: We depict a 3/4 view representation of the needle bed, with the front and back needles tented inwards and the front needles closer to the camera. The loops currently on the needles are the most recently created, with the fabric moving down as rows are knit. Left: (Incorrect) Floats passing across loops formed on the back bed appear on the front of the fabric. Right: (Correct) Floats passing across the bed go behind loops on the front bed, so they appear on the back of the fabric.

Algorithm 1 Decide to move purls to the front bed to avoid floats.

```

function FLOATAVOIDANTTRANSFERSGENERATOR(colorCourse, carriageDir, backBed-
NeedlesWithLoops, anchors, program)
    carrier  $\leftarrow$  carrier for loops in colorCourse
    anchor  $\leftarrow$  anchors[carrier]
    firstNeedle  $\leftarrow$  first needle of colorCourse
    pretransfers  $\leftarrow$  []
    posttransfers  $\leftarrow$  []
    for needle between anchor and firstNeedle in carriageDir do
        if needle in backBedNeedlesWithLoops then
            pretransfers += xfer(needle, FrontBed(needle))
            posttransfers += xfer(FrontBed(needle), needle)
        end for
    program += pretransfers
    for loop in colorCourse do
        program += getInstsToKnitLoop(loop, carriageDir, carrier)
    end for
    program += posttransfers
    anchors[carrier]  $\leftarrow$  last loop in colorCourse
end function

```

Yarn Catching Changing colors typically means using a new yarn, and without explicitly ensuring that two yarns are “caught” (meaning entangled or twisted together), doing so causes gaps in the knit. These gaps are especially apparent when they span multiple rows in the same column. Imagine a degenerate example where a knit piece consists of two different-color rectangles side-by-side. Without catching, the two yarns never contact each other, and the piece is therefore simply two disjoint single-color rectangles.

Yarns do not “catch” on their own since the carrier rail of the knitting machine is designed to avoid yarns tangling. There is a folk method that creates a catch using the back bed. Unfortunately, in illusion knitting, the back bed is densely occupied.

Our strategy uses a plating stitch at the beginning of a color change to catch. Though using plating throughout causes overly thick fabric, we have observed that sporadic use is fine. Both yarns are used in the same stitch and are pulled through one loop together, durably ensuring yarn contact and connecting the two sides. We control which color is more prominent by laying it down first.

Algorithm 2 Modify loop data structure to plate at color changes.

```

function ADDPLATINGTOCATCH(loopNeighborPairs, loops)
  currentYarn ← loops[0].yarns[0]                                ▷ First yarn is “main” yarn
  for n1, n2 in loopNeighborPairs do
    if n1, n2 have different yarns then
      n2.yarns += currentYarn
      currentYarn ← n2.yarns[0]
    end for
  end function

```

Yarn Path Planning and Anchoring In our programs, carriers do not always move together, so some scheduling assumptions do not hold. Given a carriage direction, the knitting machine expects the carrier to originate from behind the carriage in that direction. Consider the case where

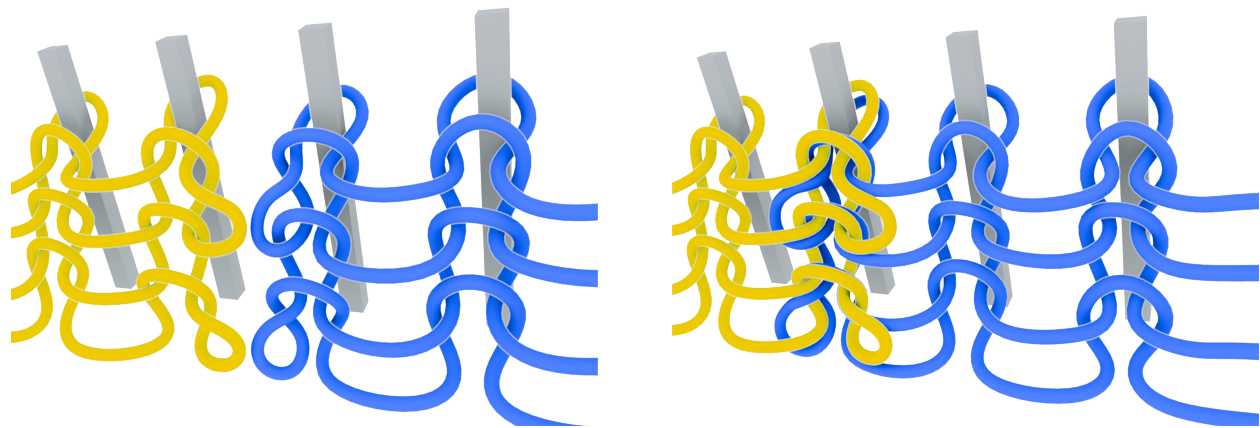


Figure 4.9: We depict a 3/4 view representation of the back needle bed, with the front bed omitted. Left: (Incorrect) Knitting two regions of separate colors back and forth without intervention leads to disconnected pieces of fabric. Right: (Correct) Using plating to knit both colors together at the color change ensures that the two halves are intertwined, with the yarn on top contributing the most to the overall colour of the plated stitches.

a carrier is anchored, i.e., has last knit in the middle of the bed, and, in the next course, is going to be used to knit from the very left side to the very right side of the bed. Regardless of which side the carriage starts from, the carrier will originate from in front of the carriage, so no yarn is placed onto the needles and no stitch is formed. This problem is illustrated in Figure 4.10.

To ensure that every stitch is knit, we propose two different strategies. In the first, we divide the course where the yarn is anchored, first knitting in one direction from the anchor to one side, then turning around and knitting from the anchor to the other side. However, imagine a plated stitch where the carriers involved are on separate sides of the carriage. There is no direction that the plated stitch can be knit such that both carriers will come from behind the carriage. In this scenario, we introduce an *anchoring stitch*. The secondary plating carrier first makes an anchoring stitch (for example, a knit) to anchor it on the same side of the carriage as the primary carrier. Of course, this movement may incur further float-avoidant transfers.

Algorithm 3 Modify carriage direction and course ordering or add instructions to set correct yarn path.

```

function SETYARNPATH(colorCourse, rest, anchor, program)
  if anchor is closer to colorCourse start then
    carriageDir  $\leftarrow$  LeftToRight
  else
    carriageDir  $\leftarrow$  RightToLeft
  if anchor is between colorCourse start and end then
    if strategy is Split then
      flip carriageDir
      left, right  $\leftarrow$  split(colorCourse, anchor)
      if carriageDir is RightToLeft then
        colorCourse  $\leftarrow$  left
        rest += right
      else
        colorCourse  $\leftarrow$  right
        rest += left
    if strategy is Anchor then
      if carriageDir is LeftToRight then
        program += knitWTransfers(start, RightToLeft)
      else
        program += knitWTransfers(end, LeftToRight)
    if carriageDir is RightToLeft then
      reverse colorCourse
  end function

```

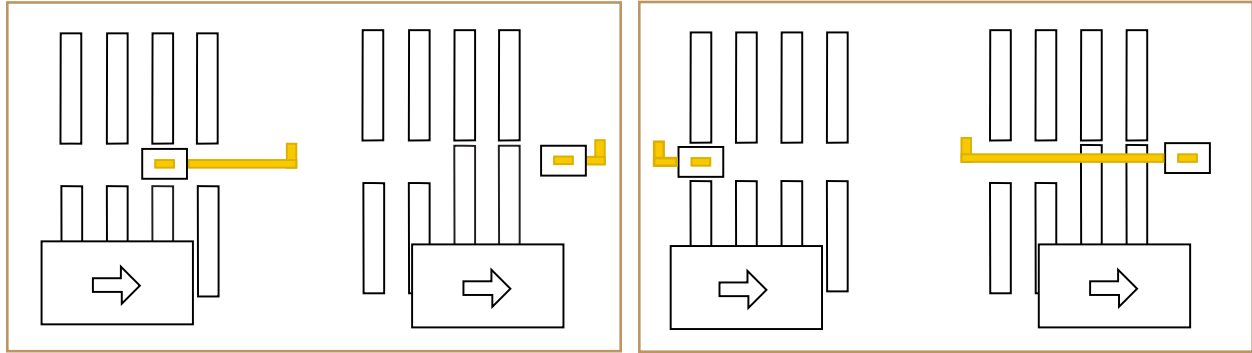


Figure 4.10: From a top-down view, we depict two needle beds with four needles each, which move up into the gap to grab yarn and form loops. In each case, the carriage is moving left-to-right, as indicated by the arrow on it. There is one carrier for yellow yarn. Left: (Incorrect) When the carrier is anchored in front of the carriage in the carriage direction, the carrier stays in front of the carriage, and the yarn retracts rather than being laid over the needles. Right: (Correct) When the carrier is anchored behind the carriage, the needles can move into place before the carrier passes over them.

4.5 Evaluation

To demonstrate that our approach is successful, we created a broad variety of traditional and totally novel knitted illusions, showing that the design burden is low, the resulting illusions obviously exhibit the input images in each view, and actual fabrication is possible.

4.5.1 Automating Single-view Illusions

The Mona Lisa illusion in Figure 4.1 demonstrates our automation of single-view illusions: only one input image is required. Compared to the traditional approach requiring hundreds of hours of manual labor, our framework significantly expedites the process and enables non-experts to create intricate illusion knits.

4.5.2 Beyond Single-view Illusions

This paper expands the range of achievable illusions in knitting; in particular, we have detailed techniques for novel double-view illusions and provided the formal foundations for even more illusion types. We show examples generated with our system, all taking five to twenty minutes of design time to iterate on the input images, yarn choices, tile selection, and editing.

Figure 4.11 shows an example that can be viewed head-on but is occluded in a side view; this novel effect was enabled via tile editing in our design tool. We also produced the first known double-view illusions, where two two-color images are knitted using two yarn colors. In our examples, we choose the automated assignments that replace the background color with a striped pattern. This approach yields sharp results, as in the bunny and teapot (Figure 4.12 (d)) and the ECG pulse and heart (Figure 4.12 (a)). We demonstrate text legibility in a more challenging example, where one side reads “ILLUSION” and the other reads “KNITTING” (Figure 4.12 (b)).

We push the boundaries further with illusions using more than two yarn colors. In Figure 4.12 (c), we show the SIGGRAPH logo in red and blue on one side and an orange logo for SIGGRAPH Asia on the other side. This example uses four yarn colors, with a fixed background and changing logo colors. Finally, to even further push the limits of our techniques, we knit two complex Van Gogh paintings using a three-color quantization and three yarn colors, requiring a total of nine unique tiles that were automatically created and assigned (Figure 4.1). The different views are distinct, and the images are recognizable. This example demonstrates the potential of our technique.

4.5.3 Comparison with Other Design Methods

We also experimented with fully automatic methods for designing illusion patterns. We encoded constraints from Section 4.2.2 as a MaxSAT formula and solved for the lowest-logical-error pattern.

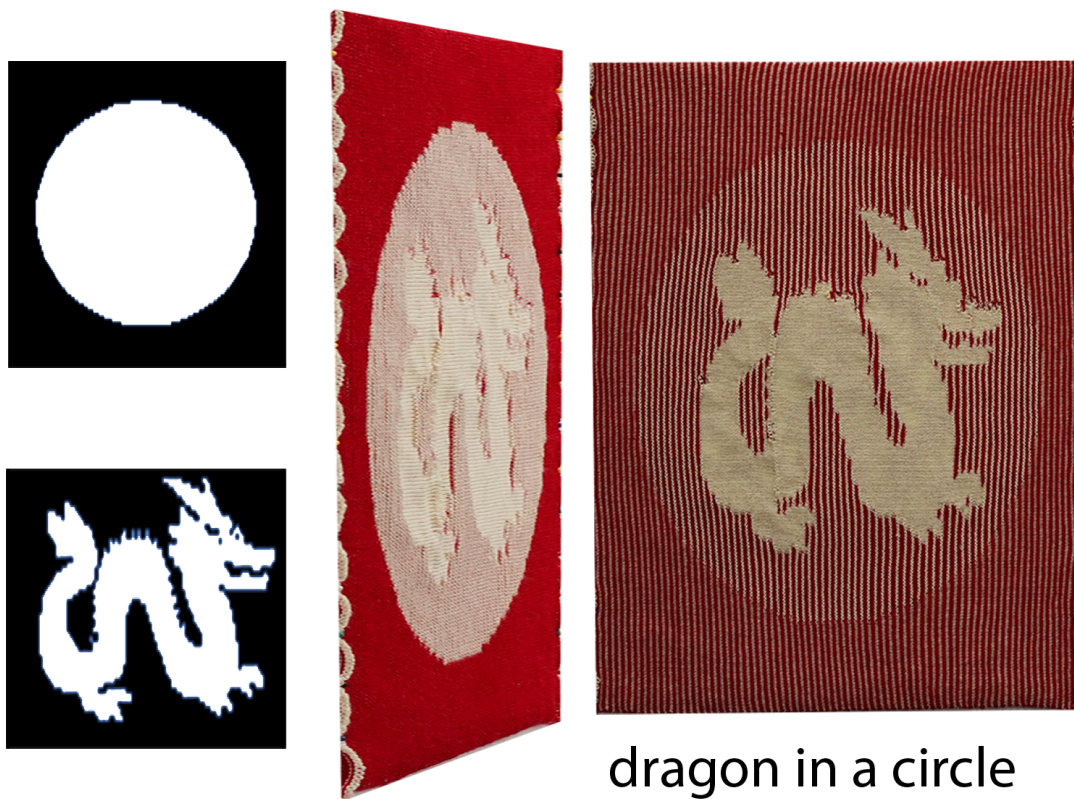
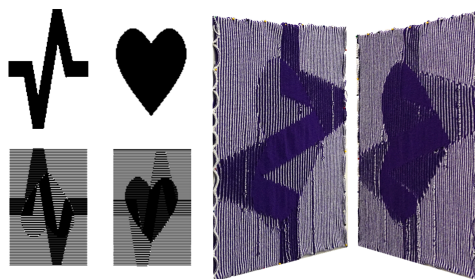


Figure 4.11: A single-view head-on illusion and its inputs. Viewed from the front, the dragon is visible, but when viewed at an angle, the larger circle illusion engulfs it, and only a circle is visible. A tile variation makes it possible to produce this other type of knit illusion.



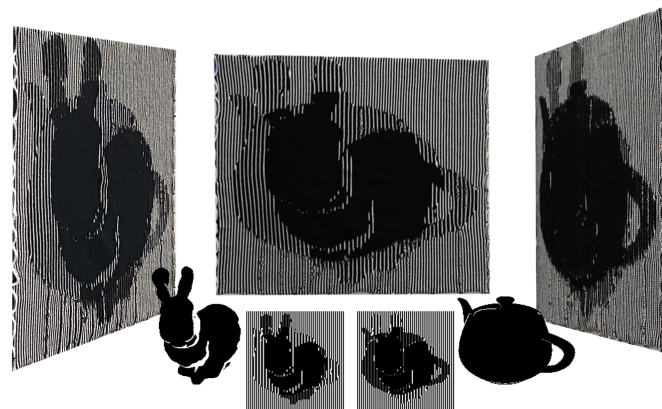
(a) ECG pulse and heart



(b) the text ILLUSION and KNITTING

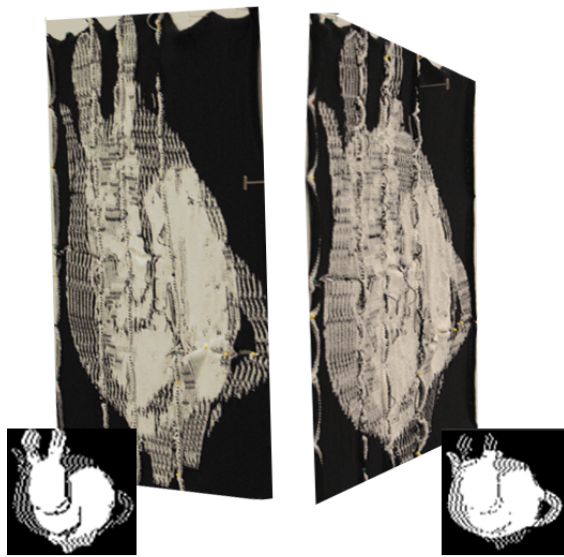


(c) SIGGRAPH and SIGGRAPH Asia Logos

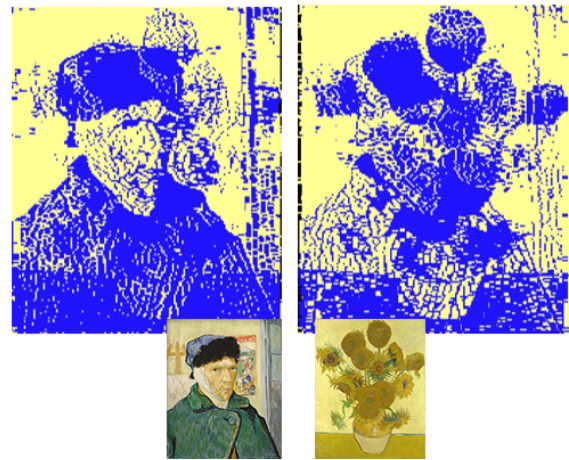


(d) Utah teapot and Stanford bunny

Figure 4.12: Double-view examples, pictured with their inputs, and, in (a) and (d), the rendered prediction of their patterns. In each example, the image from the left and right viewing angle differs, as depicted. In (d), we also show the head-on view.



(a) The result of using our MaxSAT formulation to solve for a pattern that satisfies the most constraints. Notice how the areas where the two images differ still have a lot of visual interference. This pattern took 10 hours to knit.



(b) The result (large images) of running optimization using our differentiable renderer and MSE loss against the target inputs (small pictures) to get a double-view illusion design. Although the predicted result looks reasonable, this design would have taken 30 hours to knit.

Figure 4.13: Examples of knit illusions produced by other design tools. Regardless of the quality of these automated approaches, a human-in-the-loop tool is still necessary so designers have recourse to correct or modify parts of the design.

We also used our differentiable renderer to perform gradient descent, using MSE for the loss. Figure 4.13 shows results of both.

These patterns are slow to knit: 12 hours for the MaxSAT-generated bunny and teapot vs 2 hours for our tiled solution, and an estimated 30 hours of knitting for the optimized Van Gogh vs 10 hours for our tiled solution (which is a conservative comparison, as our evaluation example is a larger piece). Additionally, we found the illusions from the automated techniques less obvious than our tiled approach. Other automated methods may improve clarity and lower fabrication times, but those can also be incorporated into our framework. In our experience, a designer-in-the-loop tool is highly effective for optimizing the ultimate objective of human-perceptible knit illusions.

4.5.4 Fabrication

We implemented a knitting pattern to machine instruction pipeline, which lowers a logical design to a physical pattern, then compiles it to a knitout program. We use the KnitPaint program to generate machine code and knit the objects on a 7-gauge (i.e., 7 needles per inch) SWG091N2, which is a Shima Seiki WholeGarment knitting machine. Designs took between 50 minutes (ECG pulse and heart, 200 x 200 stitches) to ten hours (Van Gogh, 660 x 400 stitches). The fabrication time for single-view patterns is determined by the size and in double-view by pattern complexity. In addition to the illusion results, we show in Figure 4.14 what happens when each key feature of our fabrication algorithm (i.e., float avoidance, path planning, and catching) is not applied, demonstrating their importance. In Figure 4.15, we specifically highlight a full piece that has been knit without path planning.



Figure 4.14: Knitting with (top row) and without (bottom row) each contribution. Without float avoidance (left), floats appear on the front of the work. Without path planning (middle), the piece systematically dropped stitches. Without catching (right), the two pieces knit with different colors are disconnected at the vertical edges, leading to the blue sides of the plus sign to curl inwards.



Figure 4.15: An attempted illusion knit without path planning. Note that the failure manifests not as mere visual artifacts on the knit object, but in the fact that the knit is structurally not a surface at all on the right side, and nothing is viewable.

4.6 Limitations and Future Work

Our work presents the first framework for users to explore the space of illusion knitting where more than one image can be hidden. We focus on how user-in-the-loop systems that systematically relax constraints with automated assistance can help users express their desired designs, and hope to inspire more work in this methodology.

In the future, the fabrication strategies we propose can be applied in situations beyond the mixed colorwork and texture needed for illusion knitting. Though the development of the yarn anchoring strategy was motivated by our inability to use the back bed at will in illusion knitting, this advancement could enable simpler intarsia scheduling for stockinette garments since plating is simple to reason about. Our yarn path planning strategy might also be used for intarsia scheduling as well as more broadly when considering how to schedule complex objects.

Our contributions also open several other avenues for future work. First, we considered the space of illusions achieved only through standard knits and purls on a knitting machine. More complex fabrication techniques expand the possibilities. For example, knitting a color change into a purl bump can partially achieve different colors on each side of the bump. Alternatively, expert machine programmers might attempt to enhance the height difference between bump and flat microgeometry by adding extra yarn to bumps via additional tucks, or by mimicking e-wraps, or another similar strategy. New design tools would be essential to understanding this more complex design space and to make design decisions that trade off different aspects, e.g., physical limitations of yarns, fabrication methods, human perception, and visual complexity.

Second, there may be other techniques for machine knitting that we could have used to further improve fabrication. For example, we did not consider using sliders independently from their needles, which can hold loops temporarily and could catch yarn for intarsia and for anchoring. This solution would call for more complex scheduling algorithms, perhaps when the aesthetics of

the knitted piece is a larger consideration.

Our knits tend to have small areas of dropped stitches. We believe this is due to the extra transfers incurred, each adding a chance that loops will fall off the needle. Investigation into reducing transfers would be valuable for reducing the number of dropped stitches.

It is clear that illusions present interesting challenges in computational knitting and provide a rich case study for novel methods in pattern design, program scheduling, and interaction. Our efforts serve as a framework for working with more general illusion knitting, and we intend to continue expanding its functionality.

4.7 Conclusion

We developed a theoretical understanding of illusion knitting by leveraging the key insight that both the microgeometry that underpins illusions and the principles behind illusions can be expressed as constraints. Combined with practices from artists in the craft, we built a human-in-the-loop design framework, focused around tiles, for creating illusion knitting patterns that reduce illusion design to familiar image editing tasks. To fabricate these complex designs combining colorwork and texture, we introduced novel machine knitting scheduling techniques. We further developed completely new kinds of illusion designs and so establish a foundation for the community to further explore illusion knitting.

Chapter 5

Scheduling Knitting

In the previous chapter, we took the first step of creating a design representation so we could ergonomically explore the space of illusion patterns for an input geometry. However, illusion knitting uses only a small, specific set of fabrication operations. Though we did have to devise new machine knitting algorithms to make these patterns knittable, all of that is abstracted away from the user level, as the design freedom is limited to placing bump stitches and flat stitches of various colours. All designs were rectangular and planar, and only involved one assumed viewing angle, one assumed row height, and so on.

Although we could extend the system design to accommodate changes such as these, each one expands the list of potential illusions for a given set of input images. If we go beyond illusion knitting and look at the cache of techniques in knitting in general, then the list of ways to construct a given shape balloons to the point where it isn't feasible to display all of them to the user anymore.

Thus, to proceed, we need some way to specify the user's intended, potentially partial, design that isn't generating all of them and letting the user choose. Instead, we look to supplying construction information with a combination of geometry and a program detailing the construction. This program can be arbitrarily complex or simple, and we know how to edit it to get the result



Figure 5.1: The Martha Stewart Cozy Mitten [Martha Stewart Living(2013)]. Knit as a single flat panel (a) and assembled by seaming along path A–D, producing the contrast seam and vertical stitch direction that define the finished object (b). No knitting compiler would choose this plan; PLANEL lets users write it without resorting to machine code.

we want. Thinking back to the thesis statement, to create a knitting system that enables the specification of many different ways to fabricate a single shape, we still first need language to capture domain-specific fabrication abstractions. Then, we need to use partial geometry in conjunction with construction programs as user input, and ensure the users can get feedback and iterate.

5.1 Introduction

Industrial V-bed knitting machines produce nearly every commercial knit textile today: sweaters, socks, athletic apparel, shoe uppers, and industrial mesh. These programmable machines execute low-level machine code that controls vector-like bulk operations over two banks of hundreds

of needles. Each instruction is a *carriage pass* that sweeps across the needle banks and applies one of a handful of operations (knit, transfer, drop, tuck) per needle. A single strand of yarn threads through every operation, so the needle beds (“register files”) are not random-access and execution is strictly in-order. A Shima Seiki SWG091N2, for example, has 250 needles per bed, runs roughly 60 carriage passes per minute, and is programmed through the vendor’s proprietary visual language, KnitPaint.

A decade of work in computational fabrication has produced knitting compilers [McCann et al.(2016), Narayanan et al.(2018a), Narayanan et al.(2019), Jones et al.(2022), Nader et al.(2021), Kaspar et al.(2019), Wu et al.(2019), Igarashi et al.(2008), Kaspar et al.(2021), Mitra et al.(2023), Mitra et al.(2024), Lin et al.(2018), Hofmann et al.(2023), Hofmann et al.(2019), Hofmann et al.(2020)] that translate shape specifications into machine code. These tools generally assume the garment is one continuous object, knit start to finish, and expose limited control of the underlying hardware. Real garments routinely break this assumption, with panels, seams, and post-machine assembly. Today, designers who want to explore these constructions are forced to manually iterate at the machine code level.

Architects designing chips tradeoff between power, performance, and area (PPA). Designers using a knitting machine must navigate *knitting PPA* tradeoffs: performance (fabrication time), physical constraints (racking limits, loops per needle, transfer reliability), and aesthetics. Fabrication time on a V-bed machine is dominated by carriage-pass count, not the work done per pass. Widening a pass from 50 to 60 stitches barely changes its time; going from 50 to 60 passes takes 20% longer. A scarf 200 stitches long and 10 stitches wide can be knit the long way (200 passes) or the short way (10): a 20× difference for the same finished object.

Physical constraints narrow the space of valid plans further. Racking the beds too far (laterally shifting one bed relative to the other) can stretch the yarn to breaking, so schedules must keep loop movements within a small lateral range. Two knit panels can only be joined on the machine when their stitch directions align at the boundary. Panels knit in perpendicular directions cannot

be joined this way; they must be assembled off the machine afterward. The three axes are coupled: meeting a physical constraint may force an aesthetic compromise, and a plan that saves fabrication time often changes how the garment looks, because stitch direction and texture follow the plan.

Scheduling languages were invented for these kinds of problems: many valid plans per specification, large tradeoffs between them, no single optimum. Halide [Ragan-Kelley et al.(2013)] separates an image pipeline from its schedule so users can explore implementations without rewriting the pipeline. Later systems [Chen et al.(2018), Hagedorn et al.(2020a), Ikarashi et al.(2022), Hu et al.(2019), Hagedorn et al.(2020b)] extend this approach to vector units, GPU kernels, and ML accelerators.

Knitting has vendor assembly languages like KnitPaint, which provide full control but require working at the machine code level. A scheduling language would make alternatives cheap to write, compare, and iterate on. Scheduling generally ensures the output is invariant under valid schedules. In Halide, that holds: valid loop reorderings yield the same output pixels. In knitting, it is more subtle: a mitten that is knit flat and then seamed up and a mitten knit in-the-round (cylindrically) look and even behave* very differently. Scheduling knitting requires a different notion of what the schedule preserves.

Consider the Martha Stewart mitten in Figure 5.1: its distinctive appearance comes entirely from the fabrication plan. No source-level compiler will find this approach. No low-level analyzer can discover it; the two mittens are not equivalent at the stitch level. Aesthetics do not reduce to a number, so even a geometric design-level optimizer cannot stand in for the user. The designer has to write the plan.

We present PLANEL, a scheduling DSL for machine knitting embedded in TypeScript. A PLANEL program specifies a knit object as a graph of connected 2D panels, then schedules its fabrication with a small set of operators. Panels can be split and recombined freely, so users can explore how

*Knitting direction affects how fabric “drapes”, i.e., which directions it tends to fold or fall in.

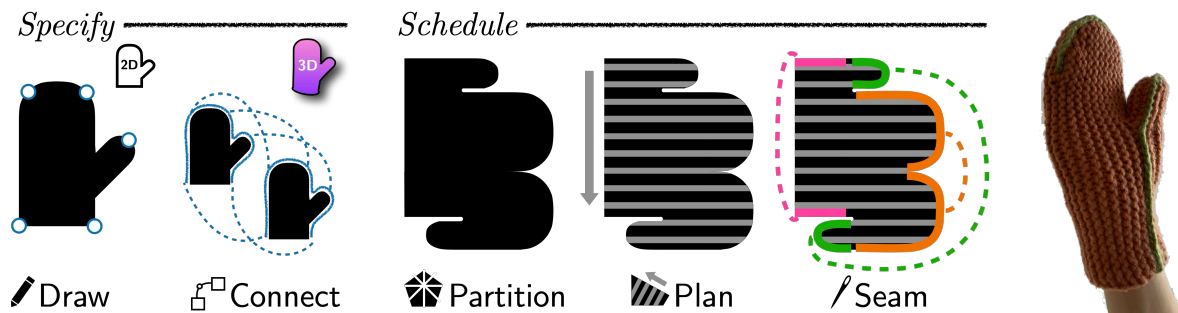


Figure 5.2: PLANEL workflow for specifying and scheduling a knit object. *Left:* The user specifies a 3D knitted object by connecting a set of 2D panels to form the desired shape. *Right:* The user then defines an explicit fabrication plan by (1) *partitioning* the shape into smaller components, (2) assigning knitting directions and stitch textures via *plan* operations, and (3) optionally inserting *seams* to re-join selected components by stitching in post-processing. Any components which are not explicitly seamed are automatically joined and knit together on the machine. This separation between shape and schedule enables modular construction, flexible trade-offs, and reuse across related designs.

to decompose a shape into components, how each component should be fabricated (direction, texture, shaping), and how the components are joined back together. PLANEL guarantees that *the original shape is always preserved* and checks that the plan is physically realizable.

Real knitting workflows include post-machine steps like seaming and assembly. PLANEL introduces Seam to bring these into the schedule, a capability no prior scheduling language offers. Figure 5.2 shows the PLANEL workflow on the Martha Stewart mitten: four panels drawn and connected, partitioned along A–D, planned, and seamed. §5.2 develops the full walkthrough; §5.3 details the operator set.

To evaluate PLANEL, we select five challenge garments designed for hand knitting [Martha Stewart Living(2013), Higgins(2005a), Linneman(2005), Zimmermann(1989), Swansen(2016)], where construction is far more flexible than on a machine, and show that PLANEL can implement them on a Shima Seiki industrial knitting machine. These constructions are within the machine’s capabilities, but no existing automatic compiler will generate code to fabricate the garment as designed. Without PLANEL,

users must write machine code by hand. Beyond these five, we express twelve mitten construction classes that together capture 365 of the most popular mitten designs on Ravelry [Ravelry(2025)]. Case studies show that the shape and the fabrication plan can be edited independently: changing one does not require changing the other. Across nine schedule variants spanning three designs, fabrication time varied by up to 10× for the same finished shape; the Sideways Mystery Mitten [Zimmermann(1989)] alone fabricates in 20 min on the default plan and 2 min on a plan `PLANEL` expresses. This paper contributes:

- `PLANEL`, the first high-level scheduling DSL for machine knitting. Users specify fabrication plans directly and iterate between alternatives without rewriting. `PLANEL` includes `Seam`, an off-machine scheduling operator that brings post-machine actions into the schedule.
- A graph-of-2D-panels shape representation closed under decomposition; `PLANEL` scheduling preserves shape and supports correctness and physical-feasibility checking.
- A three-stage compilation backend that discretizes continuous panel shapes to stitch grids, resolves panel interfaces under physical constraints, and generates correct machine-code instruction sequences via graph tracing.
- An evaluation covering 5 challenge garments, 12 mitten classes representing 365 Ravelry designs, and schedule variations with up to 10× fabrication-time differences on the same finished shape (20 min versus 2 min on the Sideways Mystery Mitten [Zimmermann(1989)]).

5.2 `PLANEL` System Overview

`PLANEL` separates the 3D shape of a knitted object from its fabrication plan, giving users far more control over the knitting machine than existing compilers and thereby explicit control over

```

1  /* Specify shape for mitten (m) and thumb (t) */
2  m_back = Draw(...)
3  m_front = Draw(...)
4  t_back = Draw(...)
5  t_front = Draw(...)
6
7  /* Connect panels to
8   * construct 3D shape spec */
9  c0 = Connect(m_back.●, m_front.●)
10 c1 = Connect(t_back.●, t_front.●)
11 c2 = Connect(m_front.●, t_back.●)
12 c3 = Connect(m_front.●, t_front.●)
13
14 /* Schedule fabrication */
15 [m_left, m_right] = Partition(m_front.●)
16 for x in m_back, m_left, m_right, t_front, t_back:
17     Plan(x, Texture.Garter, KnitDir.Left_to_Right)
18
19 Seam(m_left.●, m_right.●) /* A to B */
20 Seam(t_back.●, t_front.●) /* B to C */
21 Seam(m_left.●, m_back.●) /* C to D */
22
23 /* Generate code */
24 c3.compile()

```

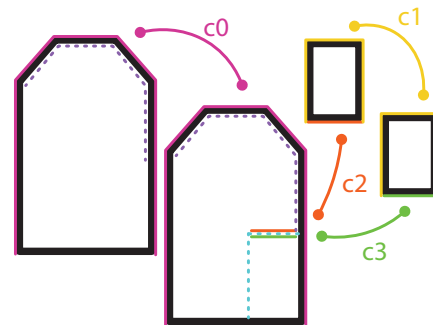


Figure 5.3: The Martha Stewart Cozy Mitten in PLANEL. The shape is specified by *drawing* four flat panels and *connecting* them to construct the desired 3D shape. The fabrication plan (1) *partitions* the shape to enable the panel to be unfolded, (2) *plans* the knitting direction and texture, and (3) *seams* the partitioned edges so the panel is knitted flat.

how their design is constructed. Chapter 3 provides a machine knitting primer, but the key design insights in PLANEL can be understood without it. We identify three desiderata for a machine knitting scheduling language: **(1)** the design intent must remain stable while the fabrication plan changes; **(2)** users must be able to explore and compare alternative fabrication strategies cheaply; **(3)** the system must detect infeasible plans and report precise errors to guide the user.

PLANEL achieves these by decomposing designs into compositions of 2D panels and their connectivity. Users schedule fabrication on individual components, and the system recomposes them into a fully scheduled version of the original shape. A key operator, *Seam*, marks components for off-machine assembly rather than on-machine merging. *Seam* captures real construction methods that actual garment designs rely on, and it also acts as a pressure-relief valve: when on-machine constraints would make a plan infeasible, converting a merge to a *Seam* relaxes those constraints and opens up new fabrication strategies. We illustrate with the Martha Stewart Cozy Mitten (Figure 5.3).

The mitten’s shape is specified by drawing four 2D panels (hand front, hand back, thumb front, thumb back; lines 2–5 of Figure 5.3) and connecting them with *Connect* (lines 9–12) to form a complete mitten. This graph of connected panels is the design invariant: it defines what the user wants to make, independent of how it will be fabricated.

To schedule the mitten’s fabrication, we use *Partition* to subdivide the front panel along the intended seam line (line 15). The two resulting sub-panels can now be knit at different times. We then use *Plan* to set the knitting direction (left to right) and texture (garter stitch) for each panel (lines 16–17). By default, the compiler merges all partitioned sub-parts back together, ensuring that the schedule does not alter the shape.

But we do not want all these panels merged on the machine. The Martha Stewart mitten’s distinctive look comes from being knit flat and seamed by hand. We apply *Seam* (lines 19–21) to mark the partition edges for off-machine assembly, which prevents automatic merging and

produces the unfolded, flat-knitting plan. Without Seam, this construction is inexpressible in any prior tool.

If the user specifies an incompatible schedule (e.g., perpendicular knitting directions on panels that must be merged), the PLANEL compiler detects the conflict and reports a precise error identifying which panels are incompatible and why. The user can then adjust the shape, try a different schedule, apply Adapt to resolve the mismatch locally, or convert the merge to a Seam.

This walkthrough illustrates all three desiderata. The shape remained stable throughout scheduling: every operator preserves the original panel graph. We could try a different partition, a different texture, or knitting in the round instead of flat, all by editing the schedule without touching the shape. And the compiler catches any infeasible plan before it reaches the machine. The 2D-panel representation makes this possible: it is coarse enough to preserve shape under rescheduling and fine enough to expose the fabrication decisions that matter.

Compiling PLANEL schedules to machine code is itself a non-trivial problem: the backend must discretize continuous 2D shapes to stitch grids, resolve panel interfaces under physical constraints, and generate correct instruction sequences for the target hardware. We compile in three stages: planned panel graph, per-panel knit-graphs with merge resolution, and executable Knitout. §5.4.1 details the backend techniques.

5.3 PLANEL Embedded DSL Design

PLANEL is designed as an embedded DSL to specify knit shapes and explicitly schedule their fabrication plans. As an embedded DSL, PLANEL leverages expressive features of its host language like variable binding, control flow, data structures, functions, and integration with tooling to enable parameterization, reuse, and modularization of complex shape specifications and knitting schedules.

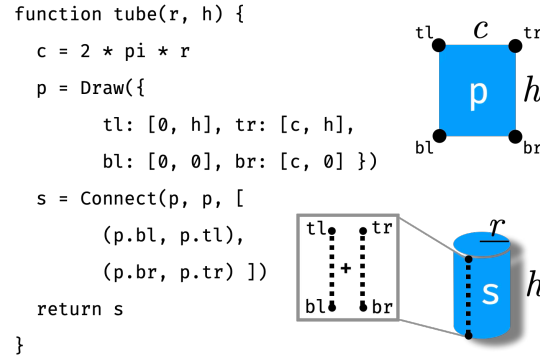


Figure 5.4: **PLANEL Shape Construction.** Users define panels with `Draw` and compose them via `Connect` by joining boundary segments. References to named points and edges (e.g., `tl`, `tr`) persist across composition, enabling downstream scheduling. Here, a rectangular panel composes into a simple cylindrical tube by connecting two edges.

Figure 5.5 shows the core types and operators in `PLANEL`. Users define individual panels with `Draw`, specifying stable references to key boundary points and edges for later use in shape composition and scheduling. Shapes are then assembled via `Connect`, using these references to align panels. Subsequently, explicit scheduling operators—such as `Partition` for dividing shapes, `Plan` for assigning fabrication parameters, `Seam` for off-machine assembly, `Merge` for implicit (automatic) joins, and `Adapt` for joins requiring small adjustments—explicitly capture essential fabrication decisions. Some scheduling operations (e.g., `Partition`) enforce *linear*** usage to prevent ambiguous or conflicting fabrication decisions, ensuring that every component gets fully scheduled to a feasible and physically realizable knitting plan.

5.3.1 Specifying Shapes

In `PLANEL`, knitted shapes are specified as compositions of 2D panels. Panels provide a simple yet expressive abstraction, mapping to knitting workflows that produce complex 3D objects by assembling flat pieces. Crucially, *restricting panels to 2D is one of the most important design decisions*

**Here *linear* usage means the operation consumes its argument, preventing further reuse—analogous to file handles, which become invalid once closed.

Types

Line Directed boundary segment between two named points
 Ref Reference to a line, used in composition & scheduling
 Panel Closed 2D shape with named boundary elements, Shape subclass
 Shape Graph of connected Shapes

Operators

Draw	:	[Line]	->	Panel	Construct a new 2D panel
Connect	:	(Ref, Ref)	->	Shape	Join shapes along shared edge
Partition	:	(Ref)	->	[Shape]	Partition a shape into subcomponents
Plan	:	(Shape, ...)	->	()	Assign knitting direction, texture, etc.
Seam	:	(Ref, Ref)	->	()	Join adjacent shapes off-machine
Adapt	:	(Ref, Ref)	->	()	Resolve panel interface mismatch
Merge	:	(Ref, Ref)	->	()	Join adjacent unseamed shapes

Figure 5.5: **Core types and operators in PLANEL.** These minimal, compositional operators explicitly represent key knitting decisions—panel creation, shape assembly, decomposition, fabrication parameters, and joins—to ensure correctness and feasibility. The PLANEL backend automates Merge to simplify common cases.

in *PLANEL* — it **radically** simplifies both the user-facing programming model and the PLANEL implementation by sidestepping many subtle complexities arising from 3D geometry.

Each panel represents a flat piece of fabric defined using the Draw operator, which constructs a closed 2D shape with programmatic access to its boundary points and edges. Panels can be imported from 2D vector formats like SVG, or generated procedurally within the host language. The boundary points and edges of each panel are assigned stable identifiers (e.g., named corners like `tl`, `tr`, or edges like `top`, `side`) that can be referenced precisely later during scheduling, allowing users to define exactly how segments should be partitioned, fabricated, or joined.

Multiple panels can be composed into a larger shape using the Connect operator, which joins matching boundary segments across panels. The result of a Connect operation is a new shape composed of multiple panels with shared structure. These compositions form a *shape graph*, where panels are nodes and shared boundary segments define edges, which encodes the intended 3D topology of the knitted object, e.g., attaching a thumb panel to the side of a mitten body, and serve as the input to scheduling.

At this stage, shapes are purely geometric and topological; no fabrication plan has yet been attached. While each constituent panel is locally flat, a composed knitting shape can have implicit curvature and nontrivial topology. For example, the simple tube shape in Figure 5.4 is created by defining a rectangular panel and connecting its left and right edges — illustrating how a simple 2D panel composition can express a nontrivial 3D knitted shape.

5.3.2 Scheduling Shapes to Fabrication Plans

Once a shape has been specified, users explicitly schedule its fabrication by applying scheduling operators to the shape and its components. These operators partition shapes into manageable subcomponents, annotate them with detailed fabrication parameters, and optionally specify off-machine seams. Scheduling operations enforce logical constraints—such as linear consumption of partitioned shapes (Partition) and leaf-only scheduling of fabrication parameters (Plan)—to ensure that schedules remain unambiguous, feasible, and directly correspond to physically realizable knitting steps. Any interfaces left unseamed by the user are implicitly rejoined by the backend using Merge. The final result is a fully scheduled shape ready to be compiled into knitting machine instructions. We detail each scheduling operator with illustrative examples below.

Partition: Decomposing Shapes. The Partition operator divides a shape into named subcomponents by splitting it along a user-defined line. Partitioning is essential when different parts of a shape require distinct fabrication choices, such as stitch direction, texture, or seams.

Importantly, Partition is a linear operation: once a shape is partitioned, it is consumed and cannot be directly reused in subsequent scheduling. Users must proceed by scheduling each of the resulting subcomponents. This linear discipline ensures schedules remain unambiguous so that each subcomponent receives precisely one consistent scheduling decision. For convenience, point and line references from a partitioned shape can still be used in subsequent operations.

When a shape is partitioned, the new boundary and resulting subcomponents are recorded. If users do not explicitly apply a Seam rejoining subcomponents at this boundary, the backend attempts to automatically rejoin the adjacent subcomponents later using Merge.

Plan: Assigning Fabrication Parameters. The Plan operator assigns knitting *direction* and *texture* to a panel.[†] A direction is a 2D vector defining the panel’s ultimate orientation on the knitting machine. This orientation strongly influences the shaping, stretch, and drape of the resulting fabric. Texture specifies a stitch pattern directly determining the performance, visual appearance, and mechanical properties of the knitted fabric: e.g., *stockinette* is fast to knit, but tends to curl at edges; *garter* is horizontally stretchy and lays flat but requires frequent loop transfers between needle beds; *ribbing* is vertically stretchy and lays flat but requires simultaneous use of both needle beds.

Plan may only be applied to *leaf panels* — they cannot not be further partitioned or composed. In our prototype, we enforce this using an internal “dirty bit,” analogous to the linear consumption tracking used for Partition. Every leaf panel must be planned to complete a valid schedule.

Seam: Declaring Off-Machine Joins. The Seam operator explicitly declares that two partitioned panels will be re-joined off-machine, e.g., by hand with Kitchener or mattress stitch or by sewing machine, linker, or serger. Seam enables fabrication steps beyond industrial knitting machines’ capabilities like attaching sleeves, closing panels, or integrating fasteners. Critically, seams drop the constraints that would be imposed by automatic (on-machine) merges, significantly increasing the expressivity of schedules in PLANEL. With explicit seams, users can specify knitting plans that unfold complex 3D shapes into machine-knitable 2D layouts that are assembled afterward. This capability is essential for non-machine-knitable designs (e.g., Möbius strips) and opens up a broader range of appearance and performance tradeoffs (e.g., on the Martha Stewart Cozy Mitten).

[†]Other fabrication parameters like colorwork should be straightforward to incorporate; we leave them to future work.

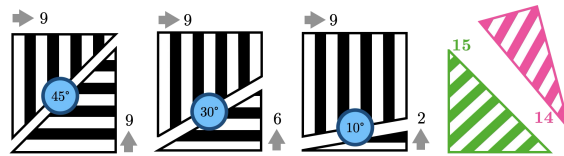


Figure 5.6: **Discretization Mismatch.** Each pair of panels is joined along a diagonal interface of equal geometric length. Due to differing knitting directions, the number of courses varies, causing Merge to fail. Adapt enables more-tolerant seamless joins by locally injecting short rows or increases.

Unlike implicit merges, seams produce no machine instructions; instead, they explicitly mark boundaries requiring manual intervention after machine knitting. Seams thus maintain structural correctness during compilation and validation while clearly documenting fabrication intentions for post-processing assembly steps.

Merge: Implicit Rejoining of Planned Panels. Merge operations are implicitly carried out by the backend to automatically and *seamlessly* rejoin adjacent panels that were previously partitioned but were not explicitly marked for off-machine assembly via Seam. The panels must be fully scheduled (via Plan) with compatible parameters.

Automatic merges are not always possible: panels scheduled with incompatible fabrication parameters like significantly differing knitting directions (e.g., perpendicular) or mismatched stitch discretizations cannot be directly merged.

PLANEL’s backend generates an error highlighting which merge failed and why, guiding users toward schedule fixes. Critically, merge failures illustrate a fundamental challenge in knitting: *not all geometrically valid assemblies of panels are physically realizable as seamless knitted objects*. Fabrication constraints, such as the requirement for continuous courses and wales, consistent stitch discretization, and compatible knitting directions, impose significant limits on feasible schedules. PLANEL checks these constraints when scheduling to guarantee fabrication plans are knittable.



Merge Failure. Merges must be course-to-course or wale-to-wale to seamlessly rejoin panels. PLANEL will report an error which the user can address by changing the schedule to add a seam or change direction.

PLANEL’s automated Merge mechanism ensures that shapes are completely reassembled before compilation while offering users flexibility to leave common joins implicit. When a merge failure occurs, users can resolve the issue explicitly (e.g., by changing knitting direction or introducing a seam), or they can apply the next operator, Adapt, to automatically handle small incompatibilities. Adapt: *Tolerant Joins for Interface Mismatches*. Users can apply the Adapt operator to instruct the backend to reconcile small discrepancies in stitch counts or directions when re-joining partitioned panels. Unlike Merge, which requires exactly matching stitch discretizations (course-to-course or wale-to-wale), Adapt tolerates small inconsistencies by inserting local adjustments.

For mismatched course-to-course joins, the backend inserts *short rows* on one side: partial courses that add extra rows. For mismatched wale-to-wale joins, the backend inserts *increases*: stitches that add columns. Adapt accepts parameters controlling which side receives these extra stitches, how the adjustments are evenly distributed across a panel, and a maximum allowed adjustment size. If the required adjustment exceeds this limit, an error is raised. These adjustments, common in hand knitting, allow PLANEL to resolve minor mismatches without resorting to off-machine seams.

Adapt is the only operator that can introduce a logical discrepancy between the shape specification and its scheduled fabrication plan. While setting the error limit very high could allow large deviations, in practice we choose limits that yield discrepancies comparable to those resulting from stitch-level discretization. These small shape adjustments rarely impact the overall form or function of the resulting object due to the flexibility of knit fabric.

Another use case for Adapt arises in rare scenarios when a merge would be valid as either wale-to-wale or course-to-course. These ambiguities are almost resolved using a common-sense “quad heuristic” (described in §5.4). However, if the heuristic chooses incorrectly—observed only once in our PLANEL experiments—users can explicitly apply Adapt to override the automatic merge

decision, forcing a specific wale-to-wale or course-to-course alignment.

Adapt is essential when knitting directions or slight variations in panel angles cause stitch counts along joined edges to differ, as shown bottom right in Figure 5.6. Large mismatches still require explicit seams, but careful use of Adapt significantly expands the space of feasible knitting plans, accommodating real-world variations in stitch density and enabling smoother integration of rotated or shaped components.

5.3.3 Tracking Merges through Scheduling Operations

To account for the transformations that the initial geometry undergoes, we first maintain a Piece Table, which tracks the internal state of each piece as scheduling operators are applied. The Piece Table enforces linearity, as each new Panel created is assigned an id and tracked. When Panels are cut, they are split into two new pieces with different ids (but retaining the ids of any parents they came from for reference purposes). Pieces also store Plan information.

Then, to keep track of what pieces will be rejoined, we log all the pending merges that will be done in something called a Merge Table. This table logs the merges between ids of pieces in the Piece Table. To complicate tracking, what these merges are might change depending on subsequent cuts. Merges are atomic, meaning that each merge is between a single atomic panel. So, if a further cut happens on a panel, the existing merges on it may have to be divided between the new subcomponents, and if any straddle both, they must be split. Additionally, if a cut happens across an existing merge, we must decompose each of those merges into atomic merges as well. Thus, we implement some functions over the merge table that perform this accounting.

The Cut operation only divides the lines involved with each merge, but doesn't do any accounting to make sure each merge involves two "leaf" pieces (that is, pieces that are not subdivided). Instead, `DistributeMerges` is called after each Cut, which restores the invariant that

Algorithm 4 Cut every panel carrying id along line ℓ . N.B. In this step, Merge Table entries can contain the ids of the original pieces before the cuts (in this case, potentially $m.a$ and $m.b$, as either might have been cut by l). As seen in lines 8 and 9, the old ids are retained so we can find all pieces cut from a given panel id. After the cutting step, we call `DistributeMerges` on the table, which will restore merges so they are between leaf panels.

```

1: function CUT(mergeTable, pieceTable, id,  $\ell$ )
2:   newIds  $\leftarrow$  [ ]
3:   newShapes  $\leftarrow$  [ ]
4:   for all  $p \in$  pieceTable do
5:     if  $p.ids.CONTAINS(id)$  or  $p.id = id$  then
6:        $(o_1, o_2) \leftarrow$  CUTSVG( $p.shape, \ell$ )
7:       pieceTable.REMOVE( $p.id$ )
8:        $id_1 \leftarrow$  pieceTable.SET(NEWID(),  $(o_1, p.ids \cup \{id\})$ )
9:        $id_2 \leftarrow$  pieceTable.SET(NEWID(),  $(o_2, p.ids \cup \{id\})$ )
10:      newIds.PUSH( $id_1, id_2$ )
11:      newShapes.PUSH( $o_1, o_2$ )
12:   end for
13:   for all  $m \in$  mergeTable do
14:     if  $m.edgeA.OVERLAPS(\ell)$  and  $m.a = id$  then
15:        $(l_1^A, l_2^A) \leftarrow$  SPLITLINE( $m.edgeA, \ell$ )
16:        $(t_1, t_2) \leftarrow$  CALCULATETIMES( $m.edgeA, [l_1^A, l_2^A]$ )
17:        $(l_1^B, l_2^B) \leftarrow$  SPLITINPROPORTION( $m.edgeB, t_1, t_2$ )
18:       mergeTable.REMOVE( $m.id$ )
19:       mergeTable.SET( $m.a, m.b, l_1^A, l_1^B$ )
20:       mergeTable.SET( $m.a, m.b, l_2^A, l_2^B$ )
21:   end for
22:   return newShapes, newIds
23: end function

```

each merge is leaf panel-to-leaf panel.

For `DistributeMerges`, there are a few common cases to consider. For example, a cut may have split one panel entirely in half. In that case, every merge attached to that panel should be checked to see which half the merge falls on now. Or, a cut may have gone across an existing merge, in which case both sides of the merge need to be checked. Depending on the placement of the cut, many new merges might have to be created.

Algorithm 5 Restore the panel-to-panel invariant after cuts

```

1: procedure DISTRIBUTE_MERGES(mergeTable, pieceTable)
2:   for all  $m \in \text{mergeTable}$  do
3:     if pieceTable.HAS( $m.a$ ) and pieceTable.HAS( $m.b$ ) then
4:       continue
5:     else
6:        $aCuts \leftarrow \text{pieceTable.CollectPiecesWithId}(m.a)$ 
7:        $bCuts \leftarrow \text{pieceTable.CollectPiecesWithId}(m.b)$ 
8:        $\triangleright$  break the edges where each piece starts and ends
9:        $aTimes \leftarrow \text{GETTIMEONLINE}(aCuts, m.\text{edgeA})$ 
10:       $bTimes \leftarrow \text{GETTIMEONLINE}(bCuts, m.\text{edgeB})$ 
11:       $allTimes \leftarrow \text{SORT}(aTimes \cup bTimes)$   $\triangleright$  each time remembers its panel
12:      for all  $t \in allTimes$  do
13:         $newEdgeA \leftarrow m.\text{edgeA}.GETAT(t)$ 
14:         $newEdgeB \leftarrow m.\text{edgeB}.GETAT(t)$ 
15:         $a \leftarrow t.aPanel$ 
16:         $b \leftarrow t.bPanel$ 
17:        mergeTable.SET(NEWID(),  $a$ ,  $b$ ,  $newEdgeA$ ,  $newEdgeB$ )
18:      end for
19:      mergeTable.REMOVE( $m.id$ )
20:   end for
21: end procedure

```

On the other hand, `seam` removes merges from the table. Here, we need to detect what part of the seam overlaps with a merge, then create new entries that join the leftover edges in the merge together.

Finally, to lower to the correct set of planned panels, we first rotate each panel so that the

Algorithm 6 A seam deletes the overlapping part of a merge

```

1: procedure REMOVMERGES(mergeTable, pieceTable, id, line)
2:   for all m  $\in$  mergeTable do
3:     if m.edgeA.OVERLAPS(line) and m.a = id then
4:       [ startToMidA, midToEndA ]  $\leftarrow$  m.edgeA.SPLITAROUND(line)
5:       [ startToMidB, midToEndB ]  $\leftarrow$  m.edgeB.SPLITAROUND(line)
6:       mergeTable.SET(NEWID(), m.a, m.b, startToMidA, startToMidB)
7:       mergeTable.SET(NEWID(), m.a, m.b, midToEndA, midToEndB)
8:       mergeTable.REMOVE(m.id)
9:   end for
10: end procedure

```

starting edge lies on the $x = 0$ axis (and the knitting directions are all parallel).

Once we have finished modifying the Merge Table, we can extract a *plan graph*, which connects all Pieces in the Piece Table by edges that represent merges tracked in the Merge Table.

5.3.4 Language semantics

Now that we have discussed Merge Tables, we present the semantics of the front-end language in terms of operations on them in Figure 5.7.

A store $\sigma = (\rho, \mu, \pi)$ consists of an environment ρ mapping variables to values, a merge table μ , and a panel table π . Judgments have the form $\langle \sigma, e \rangle \Downarrow \langle \sigma', v \rangle$ for expressions, evaluating term e under store σ to a value v and an updated store σ' , and $\langle \sigma, s \rangle \Downarrow \sigma'$ for statements, which produce no value. We write $[[\cdot]]$ to denote a geometric operation, $v.id$ for the identifier of a panel value v , and $\perp$ for an undefined result.

5.4 PLANEL Backend: Codegen

We showed how a PLANEL program specifies a shape and its fabrication schedule using a small set of core operators (Partition, Plan, Seam, Adapt) embedded in a general-purpose host language.

$$\begin{array}{c}
\text{VARIABLE} \frac{\rho[x] = v}{\langle \rho, \mu, \pi, x \rangle \Downarrow \langle \rho, \mu, \pi, v \rangle} \qquad \text{ASSN} \frac{}{\langle \rho, \mu, \pi, x = e \rangle \Downarrow \langle \rho[x \mapsto e], \mu, \pi \rangle} \\
\\
\text{PANEL} \frac{p = \text{new Panel}(s, R, \perp, \perp)}{\langle \rho, \mu, \pi, \text{Panel}(s, R) \rangle \Downarrow \langle \rho, \mu, \pi, p \rangle} \qquad \text{INDEX} \frac{\rho[x] = \vec{v} \quad \vec{v}[i] = v}{\langle \rho, \mu, \pi, \text{index}(x, i) \rangle \Downarrow \langle \rho, \mu, \pi, v \rangle} \\
\\
\text{SEQ} \frac{\langle \rho, \mu, \pi, s_1 \rangle \Downarrow \langle \rho', \mu', \pi' \rangle \quad \langle \rho', \mu', \pi', s_2 \rangle \Downarrow \langle \rho'', \mu'', \pi'' \rangle}{\langle \rho, \mu, \pi, s_1; s_2 \rangle \Downarrow \langle \rho'', \mu'', \pi'' \rangle} \qquad \text{MAKEREf} \frac{\rho[x] = v \quad \pi[v.\text{id}] = p \quad p' = p[R][n \mapsto t] \quad \pi[v.\text{id} \mapsto p'] = \pi'}{\langle \rho, \mu, \pi, \text{makeRef}(x, n, t) \rangle \Downarrow \langle \rho, \mu, \pi', v \rangle} \\
\\
\text{GETREF} \frac{\rho[x] = v \quad \pi[v.\text{id}] = p \quad \text{ref} = p[R][n] \quad l = \llbracket \text{clip} \rrbracket(\text{ref})}{\langle \rho, \mu, \pi, \text{getRef}(x, n) \rangle \Downarrow \langle \rho, \mu, \pi, (v, l) \rangle} \\
\\
\text{GETREFLEFT} \frac{\rho[x] = \text{Panel}(c_1, c_2) \quad \langle \rho, \mu, \pi, \text{getRef}(c_1, n) \rangle \Downarrow \langle \rho', \mu', \pi', (v_1, l_1) \rangle \quad v_1 \neq \perp}{\langle \rho, \mu, \pi, \text{getRef}(x, n) \rangle \Downarrow \langle \rho', \mu', \pi', (v_1, l_1) \rangle} \\
\\
\text{GETREFRIGHT} \frac{\rho[x] = \text{Panel}(c_1, c_2) \quad \langle \rho, \mu, \pi, \text{getRef}(c_2, n) \rangle \Downarrow \langle \rho', \mu', \pi', (v_2, l_2) \rangle \quad v_2 \neq \perp}{\langle \rho, \mu, \pi, \text{getRef}(x, n) \rangle \Downarrow \langle \rho', \mu', \pi', (v_2, l_2) \rangle} \\
\\
\text{MERGE} \frac{\langle \rho, \mu, \pi, r_1 \rangle \Downarrow \langle \rho', \mu', \pi', (v_1, l_1) \rangle \quad \langle \rho', \mu', \pi', r_2 \rangle \Downarrow \langle \rho'', \mu'', \pi'', (v_2, l_2) \rangle \quad v = \text{MergePanel}(v_1, v_2) \quad m = [v_1.\text{id}, v_2.\text{id}, l_1, l_2] \quad \mu''[v.\text{id} \mapsto m] = \mu'''}{\langle \rho, \mu, \pi, \text{merge}(r_1, r_2) \rangle \Downarrow \langle \rho'', \mu''', \pi'', v \rangle} \\
\\
\text{CUT} \frac{\langle \rho, \mu, \pi, r \rangle \Downarrow \langle \rho', \mu', \pi', (\text{Panel}(s, R, j), l) \rangle \quad \llbracket \text{cut} \rrbracket(s, l) = o_1, o_2 \quad v_1 = \text{Panel}(o_1, R) \quad v_2 = \text{Panel}(o_2, R) \quad \pi'[j \mapsto \perp, v_1.\text{id} \mapsto v_1, v_2.\text{id} \mapsto v_2] = \pi''}{\langle \rho, \mu, \pi, \text{cut}(r) \rangle \Downarrow \langle \rho', \mu', \pi'', (v_1, v_2) \rangle} \\
\\
\text{SEAM} \frac{\langle \rho, \mu, \pi, r_1 \rangle \Downarrow \langle \rho', \mu', \pi', (v_1, l_1) \rangle \quad \langle \rho', \mu', \pi', r_2 \rangle \Downarrow \langle \rho'', \mu'', \pi'', (v_2, l_2) \rangle \quad \llbracket \text{removeMerge} \rrbracket(\mu'', v_1.\text{id}, l_1, v_2.\text{id}, l_2) = \mu'''}{\langle \rho, \mu, \pi, \text{seam}(r_1, r_2) \rangle \Downarrow \langle \rho'', \mu''', \pi'' \rangle}
\end{array}$$

Figure 5.7: Scheduling language semantics.

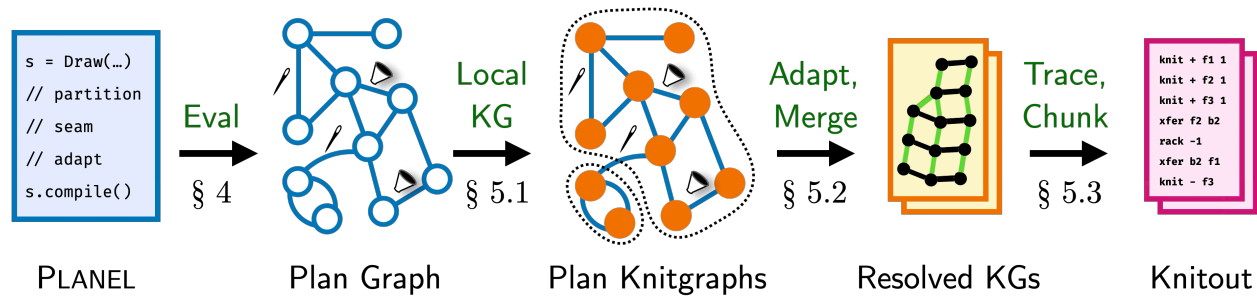


Figure 5.8: **PLANEL Compilation Pipeline.** A PLANEL program specifies a shape and its fabrication schedule using embedded DSL constructs (left). This program evaluates to a *plan graph* (white nodes) where each node is a planned panel and edges represent joins between them, including implicit merges, explicit seams, or adaptive joins. The compiler generates a per-panel *knitgraph* (orange nodes) for each planned panel, representing stitch structure, direction, and texture. These knitgraphs are joined across edges: merges are validated for stitch compatibility, seams are deferred to post-processing, and Adapt operators locally adjust misaligned joins. The resulting global knitgraph is lowered to Knitout by *tracing*, and then compiled to KnitPaint (not shown).

When the program is evaluated, PLANEL uses the Merge Table to track the resulting *plan graph*: a collection of planned panels connected by joins that define how the final object should be constructed, and captures all the information needed to generate machine-executable knitting code.

The PLANEL backend processes this graph in three stages (Figure 5.8). First, it generates a local *knitgraph* for each panel, using the scheduled direction and texture to determine stitch layout and shaping. Second, it assembles these component knitgraphs into a global knitgraph by joining adjacent panels. Implicit Merge operations are applied when stitch-level compatibility allows; Seam joins are skipped (deferred to off-machine), and Adapt joins trigger backend strategies such as short-row or increase insertion to make interfaces compatible. The backend validates that all joins are feasible; the final graph faithfully implements the user-specified shape via the scheduled plan (modulo small discretization errors or Adapt adjustments). Third, the resulting global knitgraph is lowered to Knitout, a hardware-agnostic intermediate format for knitting machines by a *tracing* procedure. Our prototype then uses existing tools to compile Knitout to

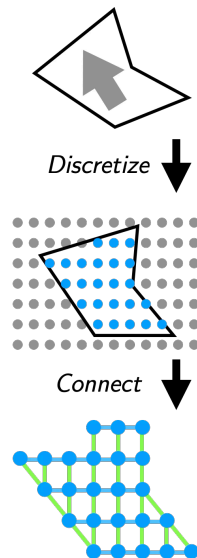
KnitPaint, Shima Seiki’s visual assembly language.

5.4.1 Generating KnitGraphs from Planned Panels

The first stage of compilation lowers each planned panel into a *knitgraph*, a stitch-level representation of loops that encodes both neighbor and parent–child relationships. Each panel is a closed 2D shape annotated with fabrication metadata, e.g., knitting direction, texture, from its associated Plan.

To generate a knitgraph, the compiler orients the panel according to its scheduled knitting direction. The oriented panel is discretized onto a regular grid of the user-specified knitting gauge (stitch width and height). Each grid intersection within the panel’s 2D shape becomes a *loop node*, labeled with a unique identifier of the form `panelID:loopNumber`.

Adjacent nodes horizontally across the grid are linked as *neighbors* within the same course (row), while vertical links represent parent–child relationships within the same wale (column). This discretization yields a structured graph aligned with the original geometry and intended knitting orientation. The knitgraph’s texture annotation is realized by marking each loop to be either a knit or a purl. Optionally, the panel’s Plan may *densify* the knitgraph, inserting additional links near panel boundaries. Densifying ensures that every loop, including boundary loops, has internal vertical links (parent–child connections), allowing local shaping operations such as increases and decreases. The generated knitgraph retains stable references to the original panel edges, allowing direct querying of loops positioned along these edges. These edge references become crucial in the subsequent compilation stage, where multiple knitgraphs are joined to form a global knitgraph.



5.4.2 Joining KnitGraphs

After generating a knitgraph for each planned panel, the backend assembles these individual knitgraphs into a global knitgraph by joining adjacent panels along shared boundary edges. These joins follow the connectivity defined in the shape graph produced by Partition operations from the schedule, which specifies whether planned panel interfaces should be re-joined implicitly via Merge, explicitly via Seam, or adjusted via Adapt.

Each boundary edge defined in the shape graph is processed sequentially. The compiler first checks for an explicit Seam or Adapt; if neither is present, it attempts an implicit merge. Successful merges connect panel knitgraphs directly, while unsuccessful merges without adaptation result in immediate errors. This design ensures the global knitgraph remains correct-by-construction, faithfully reflecting the user’s declared schedule, and produces a connected, machine-level representation of the final knitted object.

Merges. By default, adjacent panels are joined via an implicit Merge operation. A merge succeeds only if the panels’ boundary edges align exactly in stitch orientation (wale-to-wale or course-to-course, never mixed) and discretize to the same number of loops. If these conditions hold, the compiler seamlessly splices the panel knitgraphs, aligning loop IDs and updating dependencies.

Sometimes seamless joins from Merge or Adapt may be valid as either course-to-course or wale-to-wale connections, meaning both horizontal and vertical alignments match equally well. To resolve such ambiguities, the backend employs a surprisingly robust heuristic:[‡] it interprets each panel as a logical quadrilateral and only permits implicit merges between corresponding top-to-bottom (wale-to-wale) or side-to-side (course-to-course) edges. Specifically, the panel’s lowest edge along the knitting direction is designated as the *cast-on* (bottom) edge,[§] the edge

[‡]This heuristic “chose wrong” in only one of our experiments, but users can override it explicitly with an Adapt operation.

[§]A single point can serve as a degenerate cast-on edge.

opposite and farthest from it is the *cast-off* (top) edge, and all remaining edges connecting these two are considered *side edges*.

Seams. When users explicitly Seam a boundary, the compiler disables merging at that interface, excluding links across that edge from the global knitgraph. Seams correspond directly to real-world off-machine assembly techniques (e.g., manual Kitchener stitch, mattress stitch, or machine sewing), eliminating the need for exact loop-count or stitch orientation alignment. This flexibility enables many novel fabrication strategies, such as the Martha Stewart Cozy Mitten knit in garter stitch and highlighted in Figure 5.1.

Adapts. When merges fail, typically due to minor loop-count mismatches or slightly skewed orientations, users may explicitly apply Adapt to reconcile interface discrepancies. Adapt strategies employ short-row insertions for course-to-course mismatches and loop increases or decreases for wale-to-wale mismatches. Adapt succeeds provided the adjustments remain local (e.g., width changes must remain smaller than the total height of the panel), and short-row insertions must occur in multiples of two to maintain knitability.

Users can also specify how these adjustments are distributed across an edge: short rows can be concentrated at the top, bottom, evenly distributed, or split between both ends. Similarly, increases and decreases can be distributed on the left, right, evenly, or split between sides. If users attempt impossible proportional adjustments, the compiler emits compile-time errors, guiding users to resolve these discrepancies explicitly through changes to geometry, knitting direction, or using Seam.

5.4.3 Tracing and Chunking

Once the global knitgraph has been assembled, the compiler extracts an ordered sequence of Knitout machine instructions. This process involves *tracing* [Narayanan et al.(2018b)] a valid knitting path through the global knitgraph which requires determining *chunks* — regions knitted contiguously with a single yarn.

The global knitgraph encodes a partial order on loops derived from two core knitting constraints: (1) each loop must be knit *after* its parent loop has been formed, and (2) loops within the same course are knit in a consistent direction (left-to-right or right-to-left), requiring that each loop is knit *before* its immediate neighbor. Tracing transforms this partial order into a total ordering of loops suitable for sequential machine execution. If the knitgraph contains cycles (e.g., loops indirectly depending on themselves), the tracing fails as the graph is not physically knittable.

Tracing proceeds row-wise, beginning from a user-specified loop or defaulting to the first loop on the lowest course. The algorithm moves horizontally, loop-by-loop, across the current course following horizontal neighbor links. At course boundaries, tracing continues vertically to the next course up, moving through parent–child relationships.

This naive traversal can lead to excessively long yarn *floats* (gaps between stitches with the same yarn), especially in shapes with concavities or disconnected regions (e.g., a U-shaped panel). To prevent such undesirable yarn paths, tracing ensures that neighbor relationships cannot cross gaps beyond a user-specified limit or discontinuities in the panel layout. When such discontinuities are encountered, the knitting path is broken into separate *chunks*.

Chunks are maximal continuous sequences of loops that can be knit using a single strand of yarn without resetting or breaking the yarn. Each chunk represents one contiguous yarn path through the knitgraph. In practice, chunk boundaries occur at discontinuities or explicitly marked seams, ensuring that each chunk corresponds directly to a feasible yarn path on the machine.

Chunking does not explicitly depend on the original scheduling metadata beyond the fact that scheduling influences panel shapes and orientations, indirectly affecting the tracing result.

The tracing algorithm guarantees that every loop is scheduled exactly once, respecting knitting constraints and ensuring feasibility. Tracing does not optimize for minimal chunk count or size, but it produces reliable sequences of contiguous loops for flat panels and tubular structures, handling typical cases robustly in practice. Given a valid trace, knitout instructions are generated for each stitch in trace order.

5.4.4 Knittability

Once the global knitgraph and final Knitout program have been assembled and validated, the backend performs a final set of checks to ensure that the program can be executed on the specific target machine (e.g., Shima Seiki). Knittability checks are applied at the level of the stitch graph and include:

- **Loop stack depth:** Ensure no needle is required to hold more loops than physically possible.
- **Transfer reachability:** Verify transfers (e.g., for shaping) are executable within racking limits.
- **Stitch tension:** Detect overstretched or overly compressed stitch columns.
- **Unscheduled regions:** Confirm all loops are included within scheduled panel regions.

The novel Seam and Adapt operators expand expressiveness by sidestepping or resolving constraints that would otherwise make merge-based joins unworkable. Together, these mechanisms ensure that the final machine code satisfy not just the user’s shape and fabrication plan, but also the physical limits of the machine. Importantly, all user schedules are validated for feasibility: the compiler does not guess how to make unknittable plans work.

5.5 Evaluation

We show how explicitly scheduling fabrication enables users to create and refine knitted objects while navigate practical tradeoffs. Specifically, we focus on the following research questions for evaluating PLANEL.

RQ1 Can PLANEL be used to implement challenging objects originally designed for hand knitting?

RQ2 Can PLANEL express schedules for a large set of diverse real-world objects?

RQ3 Does PLANEL provide control over tradeoffs in fabrication, like knitting speed, reliability, assembly complexity, and aesthetic quality?

Additional experiments. Appendix 5.6 presents four case studies on iterative editing. Because PLANEL separates shape from schedule, users can resize a garment without touching the fabrication plan, simplify a schedule for performance without changing the shape, transfer a schedule wholesale to a new shape, and swap textures where the hardware permits. Each edit is local; PLANEL re-checks feasibility automatically.

5.5.1 RQ1. Supporting Challenging Objects

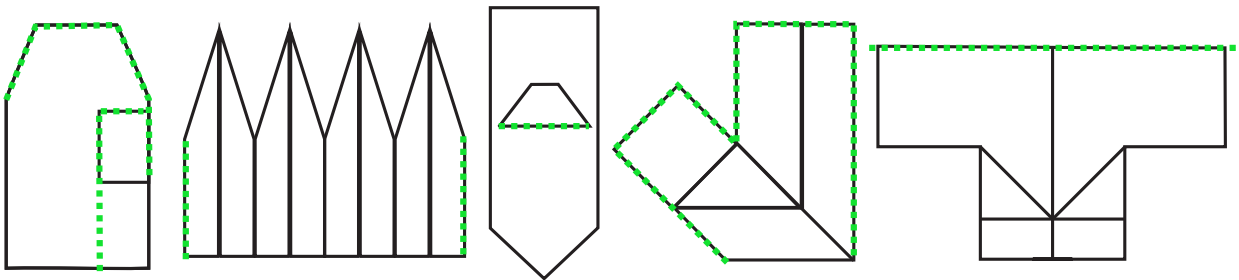
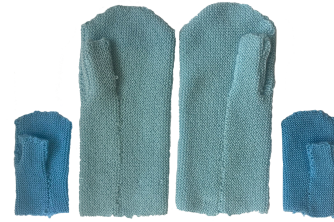


Figure 5.9: Five complex knitted constructions. From left to right: Martha Stewart mitten, short-row hat, afterthought heel sock, sideways mystery mitten, and baby jacket. Diagrams illustrate key construction details, not shape specifications; green lines indicate seams.

We show how PLANEL can be used to implement 5 real-world objects, each selected to represent distinct and challenging construction methods (see Figure 5.9). These were originally designed for hand knitting and no existing compilers prior to PLANEL can generate these fabrication plans.

Martha Stewart Cozy Mitten

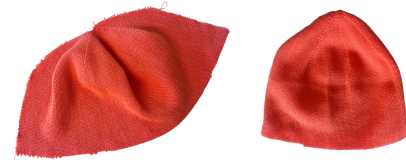
The Martha Stewart Cozy Mitten [Martha Stewart Living(2013)], introduced in Section 5.2, showcases how a seemingly complex shape can be concisely specified in PLANEL. We also created multiple size variations by scaling the shape proportions and maintaining the schedule (right) resulting in a child-sized mitten (7 minutes to fabricate, averaged over two samples) and an adult-sized mitten (11 minutes, averaged over four samples).



Tychus Hat The Tychus Hat [Higgins(2005a), Higgins(2005b)] is knit *flat* (though it does not lie flat), wrapped laterally around the head, using short rows that progressively shorten toward the crown to create 3D curvature without explicit decreases. In PLANEL, we define one panel, duplicate it eight times, and connect the segments. The hat knit in 31 minutes (garter) or 6 minutes (stockinette) at 0.4 m/s.

Afterthought Heel Sock

The Afterthought Heel Sock [Linneman(2005)] separates the sock body (a simple tube with an opening) from the heel, which is knit separately and attached off-machine via Seam (Figure 5.10), allowing simple machine production of the main tube and subsequent customization of the heel.



At 0.1 m/s (slowed for shaping reliability), the sock body took 40 minutes, the heel 8 minutes.



Figure 5.10: Afterthought heel sock fabrication steps. **Left:** Sock body and heel knit separately with live edges. **Center:** Heel inserted and sewn. **Right:** Folded, finished sock.

Sideways Mystery Mitten The Sideways Mystery

Mitten [Zimmermann(1987a), Zimmermann(1987b)] show-

cases how changes in knitting direction can create dis-

tinctive aesthetic effects. Inspired by this design, our

mitten plan explicitly schedules these direction changes

around the palm and thumb by partitioning the shape

as indicated in Figure 5.9. The mitten is knitted from

the outer contour inward. This example highlights the need for Adapt. The original SVG shape

specification did not reflect the knitted reality—the thumb’s measured drawing edge suggested

more rows than the adjacent palm triangle, even though physically both edges require the same

row count to align properly. We resolved this mismatch using Adapt to proportionately rescale

the number of rows in the thumb. The final mitten (Figure 5.11) was knit at the machine’s slower

speed to ensure accuracy, averaging 20 minutes per side (averaged over three examples).



Figure 5.11: Left: Flat-laid mitten panel, illustrating knitting-direction changes between panels. Right: Completed mitten after seaming.



Figure 5.12: Top left: “Flat” knit form of the surprise jacket. Notice inherent shaping prevents laying flat. Top right: Partial assembly by folding the left side inward to align seams; right side unfolded. Bottom left: Fully assembled jacket with shoulder and sleeve seams sewn. Bottom right: back view.

Surprise Baby Jacket The Surprise Baby Jacket [Swansen(2016)] highlights another unusual construction method (Figure 5.12). Knitted as one connected piece, the jacket is shaped with clever increases and decreases, resulting in a three-dimensional form. We only sew along the shoulders and upper sleeves. This intricate geometry is challenging to visualize or manually derive from standard flat diagrams, due to the inherent three-dimensionality that emerges during knitting.

PLANEL naturally supports such intricate constructions by clearly composing 2D panel shapes into complex 3D forms. We first defined its overall geometry, merging along side seams and sleeve undersides, and partitioned the resulting panel at visible direction changes (Figure 5.13). Each subpanel created by the partitions was assigned explicit knitting directions, textures, and increases or decreases as required. We designated the upper shoulder-and-sleeve edges explicitly as seams. We easily refined the design by adding a proportional front band. Knit at

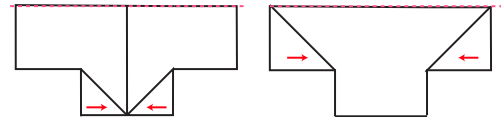


Figure 5.13: Partitions, knitting directions, & seam locations.

id	Category name	Tally	LOC
a [tincanknits(2016)]	Thumb detached, vertical	147	34
b [Reeves(2008)]	Thumb detached, horizontal	68	40
c [Patons([n. d.])]	Triangle gusset	14	46
d [Soho(2015)]	Arched gusset	23	46
e [Heusser(2015)]	Selbu	53	37
f [Together(2018)]	Baby mitten	27	22
g [Apodaca(2008)]	East-west	3	36
h [Winslow(2018)]	Convertible	6	36
i [Väätäinen(2016)]	Direction play	6	54
j [Scott(2018)]	Geometric variation	8	47
k [by Sofie Hågensen(2020)]	Cuff variation	7	36
l [Lange(2015)]	Layered mitten	3	43

Table 5.1: Mitten patterns represented in PLANEL. Reference is to the representative example.

the machine’s slower speed for reliability, our small-scale

prototype—suitable for a doll—required approximately 36 minutes to fabricate.

5.5.2 RQ2. Large-scale evaluation

We took 365 mitten designs from the prominent Ravelry database and show how PLANEL’s abstractions can represent the diverse range of real-world knit construction techniques.

We collected all unique mitten patterns from the first five pages of Ravelry’s “Most Popular” and “Most Projects” rankings, yielding 365 patterns used in 211,657 individual projects. We categorized these into 12 categories based on geometric construction features, each with at least three examples. We successfully expressed representative patterns from all 12 categories in PLANEL. Table 5.1 summarizes each category and Figure 5.14 shows them visually.

Same shape, different schedules. In PLANEL, we can achieve a broad range of designs for the same shape, simply by applying different schedules. Mittens (a), (h), (i), (k) and (l) in Figure 5.14 showcase the results that are achieved by applying different schedules to the same shape.

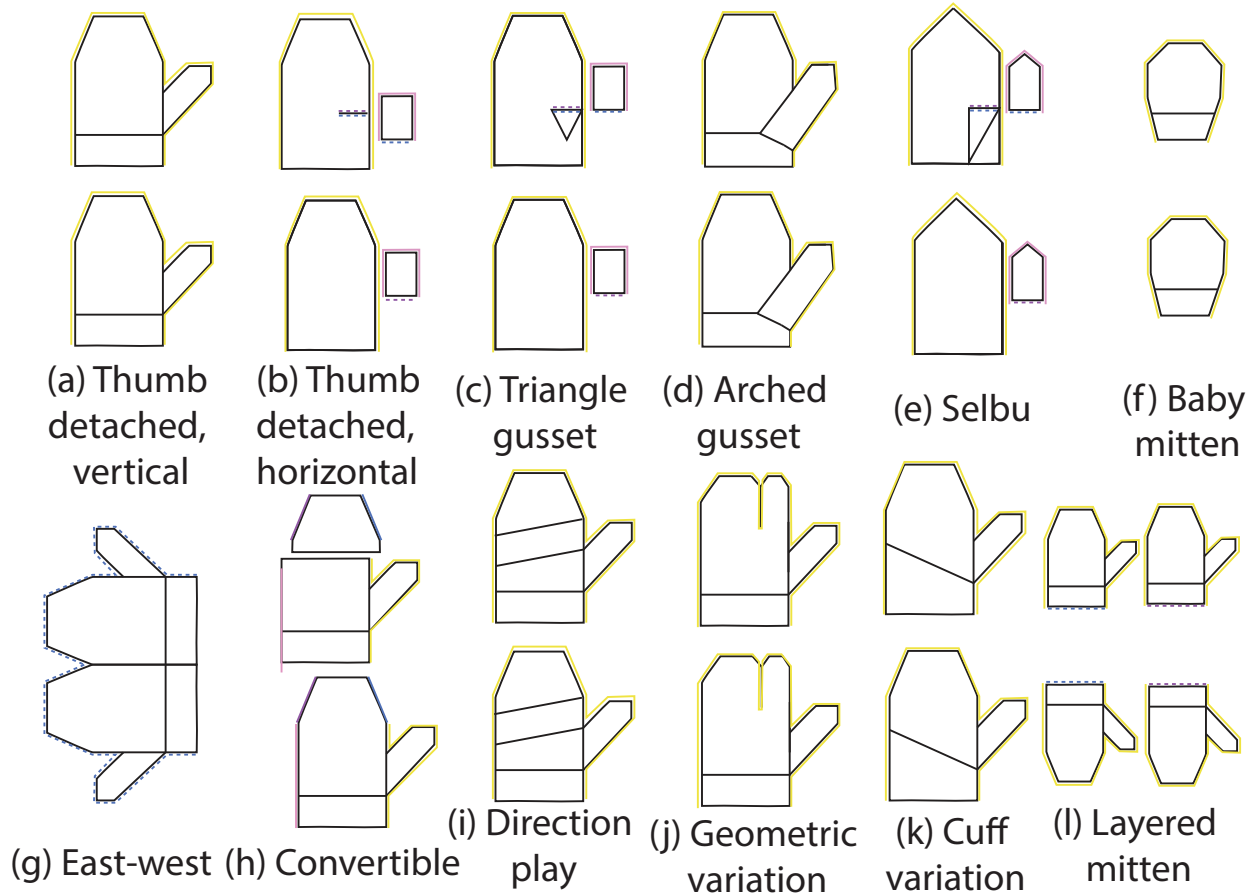


Figure 5.14: 12 mitten examples, corresponding to table Table 5.1. Pairs of coloured lines indicate a merge. Black lines indicate partitions. Dotted lines indicate seams.

- (a) is characterized by simple edits to cuff texture and knitting direction in the thumb
- (h) is partitioned at the fingers to create a convertible fingerless mitten
- (i) has a partitioned strip with shifted knitting direction
- (k) is partitioned diagonally to create a unique cuff
- (l) utilizes the same basic shape in the form of two smaller and two larger panels that are composed so that the smaller ones can line the inside of the larger ones.

We found only one mitten that could not be expressed in our system. It featured a special piping texture that is machine-knittable but not currently supported by our scheduler. We discuss this limitation in §5.8.

5.5.3 RQ3. Controlling Performance Tradeoffs

We showcase three examples of scheduling tradeoffs for three designs in PLANEL, showing: (1) $10\times$ speedup in fabrication time by sacrificing aesthetics (mitten A vs C), (2) reducing sewing by sacrificing reliability (sock A vs B), (3) maximizing aesthetics and minimizing off-machine sewing time by sacrificing reliability (sweater B vs C).[¶]

Crucially, without PLANEL, any of these programs would be difficult to create, especially the more complex programs with particular aesthetic choices, and that our system enables specifying their construction at all.

Mittens. We compare three variants of the Sideways Mystery Mitten [Zimmermann(1987b)] by their manufacturing time and design complexity (Figure 5.15 row 1): (A) modified mitten with a single direction change created through shortrows, (B) modified mitten with no direction changes, (C) original example from Figure 5.11 which has two direction changes achieved by increases and decreases.

PLANEL supports both increase-decrease shaping and short-row shaping for a shape. Increases and decreases have strong visual lines, but require transfer operations that are slower and more error-prone on a machine. Short rows are more machine-friendly, though they produce a different aesthetic.

Measuring just one side per mitten, we knit mitten C in 20 minutes (the increases and decreases requires many transfers and slower knitting to reduce failure rates). Mitten B required 4 minutes,

[¶]Sewing times are illustrative. In industrial manufacturing, a *linker* automates seaming.

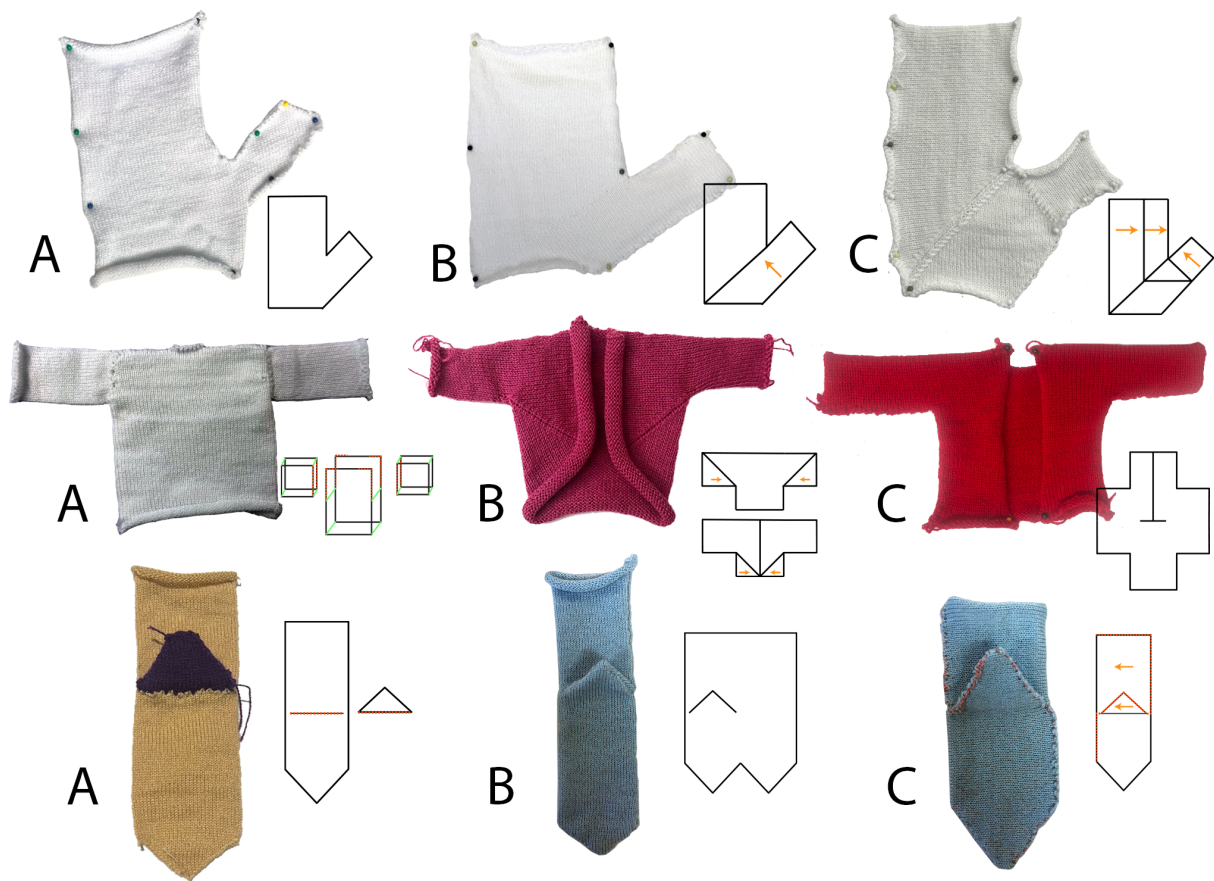


Figure 5.15: Various fabrication plan options for different types of garments, trading off fabrication time, robustness, and aesthetics.

as short rows are much quicker to knit, and don't require transfers or reducing knitting speed. The directionless mitten A needed just two minutes, as it had no detailing whatsoever. Each mitten has the same amount of sewing to do off-machine.

Sweaters We compare three variants of a baby jacket (Figure 5.15 row 2). **A:** a simple sweater from one partial tube (body) and two complete tubes (sleeves); 6 min knit, 30 min sew. **B:** the Surprise Baby Jacket from Figure 5.13, knit as one shaped piece; 36 min knit, 10 min sew. **C:** fully flat, unfolded along the shoulder seam, no direction changes; 2.5 min knit, 20 min sew. A is robust and fast to knit but needs the most sewing. B minimizes sewing but requires complex shaping. C is fastest to knit and supports full-gauge texturing (since it avoids knitting in the round), at the cost of more assembly.

Socks. For socks, we compare the Afterthought Heel sock (Figure 5.15 row 3) from Figure 5.10 (A), which has two separately-knitted in-the-round pieces that are sewn together, a sock where the heel is shaped on the machine (B), and an unfolded sock (C), which is knit fully in the flat and rotated, similarly to the Martha Stewart mitten. (A), as described, needed to be knit slowly to improve robustness, and so took 48 minutes on the machine, but only needed about 5 minutes to seam the heel to the sock body. (B) took 1 minute 45 seconds and needed about 20 minutes of seaming, just sewing up the side seam. (C) took about 2 minutes 45 seconds to knit, and required roughly 40 minutes to sew up.

Rapid Feedback. Users should be able to quickly get high level feedback and refine a design before actual knitting. The Tychus hat is our largest example, totaling 38,640 loops of yarn. Over 20 runs, the hat averaged 12.6 seconds for end-to-end compilation. Compilation time typically scales according to the number of loops that make up a design and the number of merge operations performed.

5.6 Editing and Iteration Case Studies

We demonstrate how our Scheduling DSL enables users to explore the design space more fluidly. We highlight how design decisions can be edited, reused, or simplified—and how those changes influence tradeoffs in fabrication performance, aesthetics, and usability.

5.6.1 Modify Shape, Maintain Schedule

Consider a novel construction like the Martha Stewart mitten. Suppose we want to size it down for a child or elongate the thumb (Figure 5.16). Traditionally, such tweaks would require updating multiple parts of the program to maintain correctness. In our system, users can directly modify the shape geometry, and the system automatically regenerates all relevant side lengths, preserving the original fabrication plan.

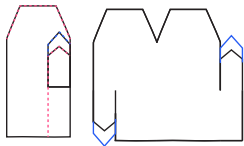


Figure 5.16: Editing the thumb to be longer (new shape in blue) will automatically produce the correct corresponding unwrapped shape. Red dashed lines indicate seams.

5.6.2 Simplify schedule

Some fabrication plans are intentionally complex—to explore unconventional techniques, showcase texture or construction novelty, or offer a more engaging hand-knitting experience. However, in many practical scenarios, such as prototyping, mass production, or user accessibility, we may want to simplify the schedule without significantly altering the final shape.

As an example, we revisit a design inspired by Debbie New’s *Unexpected Knitting*, which features a space-filling curve construction for a sweater insert (Figure 5.17). The original design

is intricate and playful: panels are arranged to trace out a continuous labyrinth-like path, with many seams and turns. This makes the piece visually striking and engaging to knit by hand—but from a machine knitting standpoint, it introduces unnecessary overhead in both fabrication and assembly.

Using our Scheduling DSL, we can represent this original plan faithfully, and then iterate on the schedule to produce a more efficient version. By replacing seams with merges, reducing panel fragmentation, and maintaining the same overall silhouette, we simplify the construction. This revised plan preserves the shape but dramatically reduces the number of sewing operations, resulting in a faster and more robust fabrication process.

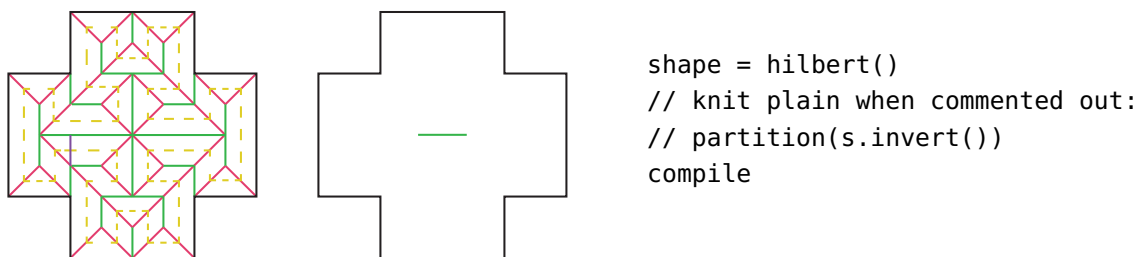


Figure 5.17: Left: original sweater pattern with curves (green lines are seams, red lines are partitions, dotted yellow line represents the knitting direction and how it changes). Right: all curves collapsed into a simple shape.

5.6.3 Transfer schedule

Schedules in our system can be reused independently of the geometry they were originally designed for. This modularity—enabled by the separation between shape and schedule—is a key strength of our PLANEL. A schedule is interpretable, self-contained, and portable: once authored, it can be applied to other shapes with similar topological structure.

This facet allows for rapid experimentation and reuse. For example, consider a mitten with a fabrication plan that merges the top edge and seams the side edges. We can take this exact

schedule—without modification—and apply it to an entirely different object, such as a simple hat with ears (Figure 5.18). As long as the underlying shape supports the same structural relationships, the transferred schedule will produce a coherent fabrication plan.

This flexibility enables reusable design workflows. A user might develop and refine a fabrication strategy for one object, and then explore how it generalizes to others. It also supports variations and personalization: once a schedule has been validated for robustness or performance, it can serve as a reusable template for new shapes that share its topology.

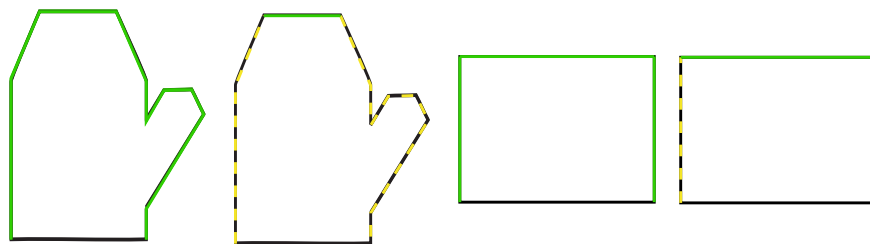


Figure 5.18: Applying existing schedule directly to new shape.

Editing Texture

While this work does not focus heavily on texture handling, it is part of the fabrication plan and can influence more than just surface appearance. For instance, garter stitch takes significantly longer to knit but lays flat and makes the object reversible. It also compresses more vertically—using the same number of stitches results in a shorter object. In PLANEL, switching between stockinette and garter stitch requires changing only the texture parameter.

5.7 Related Work

We compare PLANEL to higher-level knitting representations, knitting compilers, and user-schedulable languages, focusing on the control each gives users over the fabrication plan.

Higher-Level Knitting Representations. Prior higher-level representations compile to machine knitting code [McCann et al.(2016), Narayanan et al.(2018a), Narayanan et al.(2019), Jones et al.(2022), Nader et al.(2021)], often via remeshing algorithms for 3D shapes [Mitra et al.(2023), Mitra et al.(2024)]. They produce a single fabrication plan, offer little control over construction, and overwhelmingly require 3D mesh inputs; PLANEL operates on 2D panels, which are easier to specify and edit. Knitting Skeletons [Kaspar et al.(2021)] provides a CAD-like interface over assembly language but couples designs tightly to their schedules. Knit Sketching [Kaspar et al.(2021)] accepts 2D SVGs but is not schedulable: each program implements exactly one plan, and changing it requires a full rewrite. QUILT [Hester et al.(2025)] composes 2D patches but does not handle shaping or 3D objects. PLANEL differs from all of these: it separates shape from fabrication plan, uses 2D panels as a schedulable representation, and lets users explore alternatives without rewriting.

Knitting Compilers. Other tools optimize specific aspects of knitting programs: transfer planning [Lin et al.(2018)], program reasoning [Hofmann et al.(2023)], and Knitout optimization [Hofmann(2024)]. Task-specific tools address texture [Hofmann et al.(2019), Hofmann et al.(2020)], spacer fabrics [Albaugh et al.(2021)], solid 3D objects [Hirose et al.(2024)], knitted circuits [Luo et al.(2021)], actuators [Albaugh et al.(2019), Luo et al.(2022)], and artistic exploration [Zhu et al.(2024), Albaugh et al.(2023)]. Each defines a single fabrication plan; PLANEL exposes the plan as a user-editable schedule.

User-Schedulable Languages. Such languages separate what to compute from how to compute it [Ragan-Kelley et al.(2013), Chen et al.(2018), Hagedorn et al.(2020a), Kjolstad et al.(2017), Ikarashi et al.(2022), Hu et al.(2019), Hagedorn et al.(2020b)]. Halide [Ragan-Kelley et al.(2013)] does this for image pipelines; TACO [Kjolstad et al.(2017)] for sparse tensors. PLANEL brings scheduling to knitting fabrication, where the output is not invariant to the schedule.

5.8 Limitations and Future Work

While PLANEL significantly advances the expressivity and control of machine knitting, several limitations and exciting areas for future exploration remain:

Richer 3D Shape Support. Our current implementation emphasizes specifying shapes via compositions of flat 2D panels. However, certain knitted objects—such as highly contoured 3D forms like plush toys—are difficult to capture succinctly using only flat panel primitives. While fine-grained mesh modeling is overly complex and laborious, other intermediate primitives (e.g., knitting skeletons or parameterized cylindrical patches) may strike a more productive balance for specifying intricate 3D geometry.

Texture and Colorwork. PLANEL’s current focus has not extensively explored detailed textures or sophisticated colorwork. Though PLANEL supports basic stitch textures (e.g., stockinette, garter, ribbing), it lacks intuitive constructs specifically designed for more intricate patterns or multi-colored knitting. Future extensions could introduce higher-level texture operators or structured colorwork annotations to expand creative possibilities without sacrificing usability.

Fabricable yet Unsupported Designs. Despite its expressivity, our scheduling language still does not capture all theoretically machine-knittable objects. For example, among the 365 popular mitten patterns analyzed, we encountered one (the "Chimera" mitten) that cannot be expressed using the current operators. This design requires a "piping" detail on the cuff, holding one bed stationary while knitting on the other bed to form a horizontal tube—a construct theoretically supported by industrial knitting machines but not yet representable in PLANEL. We believe a relatively straightforward extension—allowing knitgraphs to manage children loops across multiple layers—would enable such complex details.

Formalization. While we have extensive empirical studies using PLANET, we currently lack a formal semantics or any sort of rigorous completeness proofs for PLANET’s expressivity. Future work could provide rigorous theoretical characterization, clarifying exactly what classes of machine-knitable shapes PLANET can or cannot represent and what construction methods are possible with PLANET’s operators.

5.9 Conclusion and Future Work

Industrial knitting machines are capable of far more than existing compilers let users achieve. Before PLANET, designers who wanted constructions beyond the compiler’s defaults had to write machine code by hand. PLANET is a scheduling DSL for machine knitting that separates shape from fabrication plan, letting users explore alternative constructions freely. Its panel-graph representation preserves shape under rescheduling; its off-machine operator Seam brings post-machine actions into the schedule; and its backend compiles schedules to machine code through discretization, interface resolution, and instruction tracing. Our evaluation shows that PLANET can implement challenging hand-knitting patterns on industrial machines, cover a broad landscape of real designs, and expose fabrication-time tradeoffs of up to 10×.

Looking ahead, formalizing PLANET’s semantics and the knitting machine’s operational model would strengthen correctness guarantees and enable verification. More broadly, PLANET suggests that scheduling can be productive even in domains where the output is not invariant to the schedule, given the right choice of invariant. Identifying such invariants in other fabrication domains remains an open question.

Chapter 6

Flip displays

The previous chapter showed a design representation for knitting that formalized fabrication domains, used a combination of geometry and fabrication programs as input, and provided affordances for direct editing. We now turn to other non-fabrication physical domains to see if these principles are broadly applicable. Just as with knitting, designers for these boards “care about *how* a task is accomplished”, especially because their criteria involves taste, aesthetics, and perception. Given the infinite complexities of physical objects, it will be essential to abstractions that can actually capture the salient features of display behaviour. Thus, as a transfer case, we examine how these ideas can be used in designing content for electromechanical displays: displays where, as the content is dynamically updated, its elements move into place.

6.1 Introduction

As the world goes increasingly digital and cities move to create huge, architectural-scale screens, there is respite in analog installations. Designers, architects, and artists are turning to the electromechanical displays of yesteryear to create dynamic, immersive, interactive installations. Flip



Figure 6.1: Left: Flip-dot display on a bus [fli([n. d.])] Right: Split-flap display showing airplane departures timetable [spl([n. d.])].

dots and split flaps convey information, unveil animations, or interact with viewers while providing no eye-strain nor light pollution. They additionally ripple small, ambient noises that draw people in without disrupting conversation and atmosphere, as a screen with speaker might.

These types of displays have been installed everywhere, from cruise ships to the Coca-Cola headquarters. Oftentimes, unfortunately content not specifically designed for these displays look sub-par. Importing a flat design and trying to convert it directly has many pitfalls; it's not true that any original 2D content can be imported and displayed directly on these screens.

First, everything is extremely low resolution, so inputs must be downsampled. Downsampling images is well-understood, but we also must downsample in time, as frame rates are low. Smaller, faster movements are difficult to translate. Also, the motion caused by the display updating can be distracting. For flip dot displays, the subset of high-energy, flashing, dots is attention-grabbing and more prominent than just the statically-visible colour layout, especially when the display isn't wall-scale. It can even cause visual interference: for example, in transitioning between states, flip dot units are briefly visible on their side, and reveal the black background behind them. Conventional motion-graphics and animation tools, even those built for low-resolution pixel art,

therefore cannot help, because they neither reason about movement of this kind nor offer any way to program the hardware.

A huge part of programming these systems is programming for this motion. And so, much like the performance tradeoffs across fabrication plans, we see that different sequences of motion each have perceptual tradeoffs. Flipping in a continuous cascade might suggest riffling cards or papers, while flipping slowly suggests more of a water drip. Either of these may reinforce or disrupt the narrative being presented. Rather than simply being incidental, I observe that how we move from one state to the next is a coherent, self-contained description of some “effect”, and, like the construction information within fabrication processes, it should be lifted to its own first-class abstraction. Crucially, like digital fabrication machines, these hardware are controlled by low-level programs, so higher-level abstractions capturing how to control these boards are naturally programmatic as well.

Images, specifically keyframes, are a natural starting point for animations. Our goal is not to make them targets, importantly; inverse optimization for the display of these frames would clobber the between-frame perceptual effects we are striving to exhibit. The input serves as a starting point for authoring the “construction plan”, and is an interface onto which we can apply effects, and which also influence the effects usable. Thus, these frames form our partial geometric input.

Finally, we need a user-in-the-loop system where designers can experiment with strategies and visualize the result. This iterative process is driven by designers through direct editing.

Thus, we see an opportunity to use our three thesis principles to make designing content for these displays in a way that is expressive, composable, and designs for this motion, enabling artistic and narrative experimentation.

We want to build a system that allows for expressible, composable design that will encapsulate the perception effects of motion. We highlight the reusability of expert knowledge, because

companies like Oat Foundry and Breakfast Studio have been pushing the boundaries artistically, but their works are bespoke, hand-crafted, and done in consultation with designers and engineers to deliver a set of effects. Thus, they are incredible, but painstaking, and not broadly accessible.

We create a user-in-the-loop system where the user imports short keyframe animations as the partial geometric specification, and applies Effects, which abstract over machine code, to specify how the display flips from one frame to the next. These Effects are modular, encapsulates expert and perceptual knowledge, and defined in a way that provides compiler extensibility. Based on the observation that variation in *how units flip* between states is a function of how frequently a unit flips and when a unit starts flipping, we factor effects into two reusable kinds of component. An Order dictates the sequence in which units begin (or finish) flipping; a Transition dictates what each unit does once it starts — how many times it acts and how spaced out those actions are. One Transition can be performed in many Orders, and one Order can be applied to various Transitions, with each computed independently. Thus, a small vocabulary yields a large space of effects. Users can write, remix, and experiment with these components to build their animations. Once we have determined the Effect for each set of frames, we have a schedule for how each dot will act, expanding the vocabulary of visuals that no other interface can capture. Finally, a hardware interface declares what a given board can actually do. Can its flip dots units wiggle without completely flipping, for example? This interface lets the compiler check that a desired schedule is in fact realizable during the design phase.

The evaluation here will not be fully comprehensive, but reflect our attempt to establish this pipeline. To test the validity of this transfer case, we will prepare two case studies. In the first, we take an input animation and prepare as diverse schedules as possible, showing the range of effects achievable and how it can enhance the original input, while still displaying the original input. In the second, we take an input animation, a set of grid orderings, and a set of transitions, and generate as many variations as possible via reuse.

We will not be attempting to cross-evaluate hardware types (i.e. animations developed for one hardware type will not be tested on other types). We also have a third simulation type, Brixels, but we will not develop any special schedules for that, nor will we add any additional types of simulation. Also, these case studies will be performed on planar hardware layouts. This work is framed by three research questions.

RQ1. Can using and remixing these components produce effects with different visual impressions, that users might want to use?

RQ2. Does the system enable us to create various effects that actually capture “emergent behaviour” easily?

RQ3. Can we extend the compiler with new hardware and reuse components?

For the thesis, I will address points RQ1 and RQ2.

6.2 Background on display hardware examples

This work aims to support a broad and flexible definition of “display”, from analog to mechanical to electronic. Almost anything can become a display: a marching band of members alternately standing and sitting, a crowd raising and lowering coloured signs, a pinscreen display that is manually updated by experts, a sisyphus table with a magnetized marble drawing in sand. People have also developed novel display technology prototypes at home and in studio settings, like a “multiplexed” flip-dot display using turning balls as the flaps, or the now-defunct Pixel Track from berg*. In particular, we are concerned with displays that contain inherent motion when updating, as we want to take advantage of that motion.

For evaluating this idea, we focus on two types of hardware, which are fairly well-known technologies designed for display, and are programmable.

*<https://web.archive.org/web/20151119061742/http://blog.bergcloud.com/2014/05/20/pixel-track>

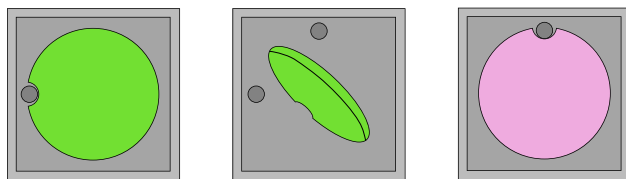


Figure 6.2: An example of a flip-dot unit, where the stems are placed at the left and top sides.

6.2.1 Flipdots

Flip dots or flip disks are matrices of circular disks that can flip quickly between its front and back states. Mechanically, the flip dot has an shaft running inside it, allowing for the dot to flip back and forth. It has two magnets embedded inside of it, one positively charged and one negatively, and it rests on two stands that are electromagnetically charged. To activate a flip, the controller sends a short pulse that changes the polarity of the magnets on the stands, driving the lightweight disk to flip. This mechanism allows for flip dots to maintain a stable image when powered off. The surface of the flip dot is then stickered, painted, or otherwise decorated the desired colour or pattern. Typically, a flip dot unit is in the range of half an inch to an inch in diameter.

6.2.2 Split flaps

Split flaps displays are similar to flip dots, but instead of flipping between two display options, it flips between a preset reel of options (usually sets of characters, but now even sets of colours like in Oat Foundry’s Chromaflap). The flaps of the reel are attached to a wheel. The top flap of the reel is held back, usually by the frame of the unit. When advancing, the stepper motor turns the wheel slightly just until the top flap falls down and reveals the flap behind it with the top half of the next character. The backside of the top flap contains the bottom half of the character to be displayed.

The interesting constraint with split flaps is that characters are no longer “random access”.

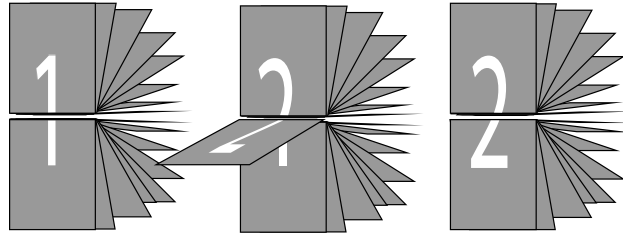


Figure 6.3: Split flap as it transitions through one flip

Whereas we could control, in totality, what a flipdot display looked like pretty much at any given time, the split flap display must transition through every intermediate state. This transition takes time, and are necessarily visible.

6.3 Design Overview

The user controls the animation by supplying an input set of frames that are used to generate a specification. Each set of frames is parsed into its constituent entities that can act independently. These entities are differentiated by colour. The term “Object” refers to one entity across the set of keyframes that is drawn in a specific colour, and the term “Target” refers to the state of that Object in a specific keyframe.

To complete the user specification, the user writes down how each frame transitions to the next. These are specified as Effects. So in the end, we have separate subgraphs, one for each Object, where the edges between each Target node is an Effect, and the graphs are given layering orders.

From these graphs, the system compiles the program to be run on the hardware. First, it ensures that each subgraph contains a Target for each frame, interpolating if needed. lowers the graph to a sequential list of $\text{GroupAction}(t, a, us)$, each of which stores a timestamp t , an abstract action a , and the units acting at that time us . Next, it uses the hardware interface

chosen by the user (and possibly implemented by the user) to specialize the list of `GroupActions` into machine-specific instructions. In doing so, it also checks that the plan is feasible. First, the actions must all actually be possible on the chosen hardware. Second, it must be possible that units can refresh quickly enough for the generated plan (as a counterexample, if it takes 2 seconds to complete a flip, a unit can't activate on $t=0$ and $t=1$).

6.3.1 Effects

Our focus is compiler extensibility and reuse. The idea is to codify specific types of Effects so that experts can share what they create easily (much like brushes in drawing software). It takes skill and study to recognize that a specific sequence of flips creates an emergent or auxiliary effect. But at the same time, we recognize that a lot of Effects might have shared details. Thus, we want to break down the implementation of Effects into reusable components.

To give a specific example, we could imagine two different Effects: `FlipReveal` flips each required unit the right number of times and stops. `KeepFlipping` continuously flips each unit just until the duration of the transition is up, then flips all the required units to their correct ending states. Now, let's imagine we want to make variations on these Effects. Like, rather than flipping every unit to its end state at the same time, maybe I want to make a "wipe" (similar to a Powerpoint Wipe effect), revealing the image. So then, we write an Effect called `WipeTopDown`, which would flip the first row of units, then the second row, then the third row, and so on. Then, it also seems natural to want an Effect called `WipeBottomUp`. But then, what if I wanted to `WipeTopDown`, but build up anticipation by flipping every unit for the duration of the effect first? Then, I would have to basically duplicate the logic of `KeepFlipping` for both of `WipeBottomUp` and `WipeTopDown` to make `KeepFlippingWipeBottomUp` and `KeepFlippingWipeTopDown`, and so on. Then what if I wanted to add a wipe that started from the bottom-left corner, or went left-to-right,

or right-to-left? Or if I wanted an Effect that would perform KeepFlipping, but time the flips to match a specific beat (for the bottom to top, left-to-right, right-to-left, bottom-left corner out directions)? We would have to continue to re-program each separate behaviour.

Consider another example of this distinction for split flaps in particular. To transition from an input state I to an output state O , we need to flip every unit u n times, where there are n steps between I_u and O_u in the split flap reel. This value n varies over the units. Thus, we have a choice when transitioning: should we allow the image to settle in bits and pieces by flipping each unit the minimum number of times? Or should we control how the image is revealed, like ensuring the flaps are settled left-to-right? Then, as soon as we introduce that control, it seems clear we'd like to control the direction that flaps are settled as a separate plane.

The key observation is that these two types of variation are actually orthogonal. One is figuring out how we traverse the board: if we're moving right-to-left, top-to-bottom, etc. as we "perform" (which might be units activating, or units settling into their correct positions, or even something else). The other is deciding, for a given unit, when it should flip. For FlipReveal, we only need one flip that will be synchronized across all units. For KeepFlipping, that would be scheduling back-to-back flips. For Wipes, that would be only flipping once the unit is allowed to activate, or for WipeKeepFlipping, that would be flipping until the unit is allowed to stop.

We capture the first set of concerns with Orders, and the second set with Transitions. The basic idea is that Transitions capture the temporal: they dictate how many times and what times a unit flips, like adding events onto a timeline. Orders capture the spatial: they dictate the order that units activate across the board, showing how an effect progresses and is "drawn".

So then, breaking down the above examples, we could have Orders AllAtOnce, TopDown, BottomUp, LeftToRight, RightToLeft, BottomLeftOut, and so on. Each of these produces a grid matching the size of the board with an activation time for each unit, where the first units to act start at 0. (If some units don't act, they don't receive times.) Then, we would have Transi-

tions `KeepFlipping`, `FlipOnce`, and `BeatFlip`. `FlipReveal` can be expressed as `FlipOnce(AllAtOnce)`. `WipeTopDown`, etc. are just `FlipOnce` with the appropriate `Ordering`. The original `KeepFlipping` is `KeepFlipping(AllAtOnce)`, where the variations that flip and then reveal in order are `KeepFlipping(TopDown)`, etc.

Finally, we compile each of these `Effects` to a set of abstract assembly instructions. Then, we create an interface called `Hardware` that allows us to reuse all of these `Effects` across different hardware backends. We can write one backend-specific code generator and instantly be able to use compatible `Effects`. That is not to say that every `Effect` will work equally well on each display; in fact, many are custom-designed for specific displays. However, the reusable components are still useful across backends.

L1 DSL

$$Duration = \mathbb{R} \qquad Ordering = \mathbb{N} \times \mathbb{N} \rightarrow Target \rightarrow Time[][]$$

$$Transition = Ordering \times Target \times Target \times Duration \times Hardware \rightarrow GroupAction[]$$

L0 DSL

$$Time = \mathbb{R} \qquad Action = Flip \cup Set \cup Wiggle \cup \dots$$

$$UnitId = \mathbb{N} \qquad StateId = \mathbb{N}$$

$$GroupAction = Time \times Action \times UnitId[]$$

Hardware Specification

$$Time = \mathbb{R} \qquad Hardware = Element[] \times Constraint \times Action \rightarrow UnitId[] \rightarrow Time \rightarrow void$$

$$HUnit = Action[] \times (Action \times StateId[])[] \times UnitId$$

$$Constraint = Action \rightarrow UnitId[] \rightarrow Time \rightarrow (UnitId[] \times Time)[]$$

$$compile_{Hardware} = (Hardware \times HUnit \times State)[] \times GroupAction \rightarrow (HUnit \times State)[]$$

6.3.2 Hardware specification

Each user-implemented hardware specification requires three components. The first is a list of Units with a coordinate map from index to Unit. Each Unit stores 1) its current state (for a flipdot, whether it's the front or back; for a splitflap, what is shown on the reel). 2) a list of possible actions (perhaps some specific hardware can control flipdots so they wiggle without completely flipping). 3) a transition system for actions and States (e.g. for a split flap reel, flipping advances the reel by one, so if the Unit started at "A", flipping might move it to "B".)

Secondly, the hardware stores constraints describing the most expressive program possible. Imagine that, as an extreme example, your display system is a Sisyphus table. You're only able to make updates at the head of the ball where the drawing is occurring. A flipdot, for example, is able to flip as many of its unit as it wants at a time, and the only requirement is that if the unit was previously flipped at time t , the current time $t_c \geq t + \text{flipLength}$. Programs that don't respect this will be rejected.

Thirdly, the hardware should implement a code generation method to take abstract actions to the concrete, machine-specific actions that can activate the machine. This step should fail if actions aren't available on the units or the units aren't able to activate.

The full compilation process runs as follows: to check if we can run these group actions on the hardware, we invoke the hardware constraints. (If at some point either we cannot flip the units we need, or we cannot start at the time required, we fail.) Then, we use the hardware-specific code generation to translate group actions into individual hardware actions.

We have implemented hardware backends for Brixel simulations, split flap simulations, flip dot simulations, and flip dot physical hardware.

6.4 Implementation details

Listing 6.1: Example program code for a flipdot display depicting a wave.

```

1 const targetsPerFrame = frames.map(f => extractTarget(f));
2 const tr = () => new FlipOnceTransition(new InterpolationOrder(new LeftToRight())
  );
3 for (let i = 0; i < targetsPerFrame.length - 1; i++)
4   targetsPerFrame[i].effect = new Effect(frames[i], frames[i + 1], tr,
    EffectType.Complete);
5 const timing = Array.from({ length: 10 }, (_, i) => (i + 1) * 2);
6 return animationToGroupAction([[frames[0]]], timing, hw);

```

This snippet (Listing 6.1) is an example of how to program an animation with a single target. It generates the Targets from a list of frames, wires each frame to the next with an Effect parameterised by a Transition and Order, supplies a duration for each interval, and hands the resulting graph to `animationToGroupAction` along with the hardware. The output is a flat schedule of timed unit flips that the hardware backend can consume directly.

The remainder of this section walks through each construct in turn: how Targets describe the state of each component at a keyframe, how Effects link them into a per-Target chain, how Transitions and Orders make up Effects, and how `animationToGroupAction` traverses the graph and merges the per-Target schedules into a single timeline.

6.4.1 Lowering the front-end graph

The front end describes animation as a collection of subgraphs of Targets connected by Effect edges. Each Target carries an optional field `effect : Effect ∪ {⊥}`, so following `effect.to` from one target to the next produces a per-Target linked list of keyframes $T_0 \xrightarrow{e_0} T_1 \xrightarrow{e_1} \dots \xrightarrow{e_{n-1}} T_n$. Each Object need not be specified for each keyframe. During the lowering process, we ensure that each subgraph has one Target entry per frame by interpolating if required. An Effect bridges

two adjacent keyframes by exposing a single lowering hook:

`Effect.generateGroupActions : time × flips → hw → [GroupAction]`

That is, each Effect node knows how to compile itself into a schedule, which is done by delegating to a specific set of Transitions and Orders. The Effect returns a closure over the hardware *hw*, so that instructions specific to the hardware may be generated later, separately.

The global lowering pass is done in `animationToGroupAction(targets, t, hw)`. It walks each object’s Effect chain, requests a set of `GroupActions` from each Effect, time-shifts that set by the cumulative time so far, and concatenates them. After every subgraph for each Object has been traversed, it takes all the `GroupActions` and merges the unit lists across the same object and time, turning multiple subgraphs into a flat, time-sorted `[GroupAction]` that any `hw.compile` can consume.

6.4.2 Transitions

A Transition encapsulates the strategy for how individual units change state between two frames. Given a source and destination Target, a duration, and a HardwareInterface, it produces a sequence of timestamped GroupAction Targets:

```

1 interface Transition {
2     generateGroupActions: (o1: Target, o2: Target, t: Duration,
3                           h: HardwareInterface) => GroupAction[];
4 }

```

All transitions begin by calling `diffIndices`, identifies units that change between `o1` and `o2`—cells present in one frame but not the other. The result is a set of unit indices that must be actuated.

Algorithm 7 Lowering the Target/Effect graph to a global schedule

```

1: function LOWER(targets, frameTime, hw)
2:   actions  $\leftarrow$  []
3:   for all target  $\in$  targets do
4:      $\triangleright$  flip to the first target from a blank keyframe
5:     head  $\leftarrow$  target.firstTarget
6:     actions += SNAPTRANSITION.lower( $\perp$ , head, 0, hw)
7:     cursor  $\leftarrow$  head; time  $\leftarrow$  frameTime[head.frameId]
8:     while cursor.effect  $\neq \perp$  do
9:       next  $\leftarrow$  cursor.effect.to
10:      span  $\leftarrow$  frameTime[next.frameId] - time
11:      window  $\leftarrow$  cursor.effect.genGroupActions(span, span)(hw)
12:      SHIFTBY(window, time)  $\triangleright$  place on the global timeline
13:      actions += window
14:      cursor  $\leftarrow$  next; time  $\leftarrow$  time + span

15:    $\triangleright$  merge across targets: bucket by time, union unit ids per action
16:   buckets  $\leftarrow$  {}
17:   for all groupAction  $\in$  actions do
18:     for all (action, units)  $\in$  groupAction.actions do
19:       t  $\leftarrow$  groupAction.tPlus
20:       buckets[t][action]  $\leftarrow$  buckets[t][action]  $\cup$  units
21:   return [ GROUPACTION(t, buckets[t]) | t  $\in$  SORT(KEYS(buckets)) ]
22: end function

```

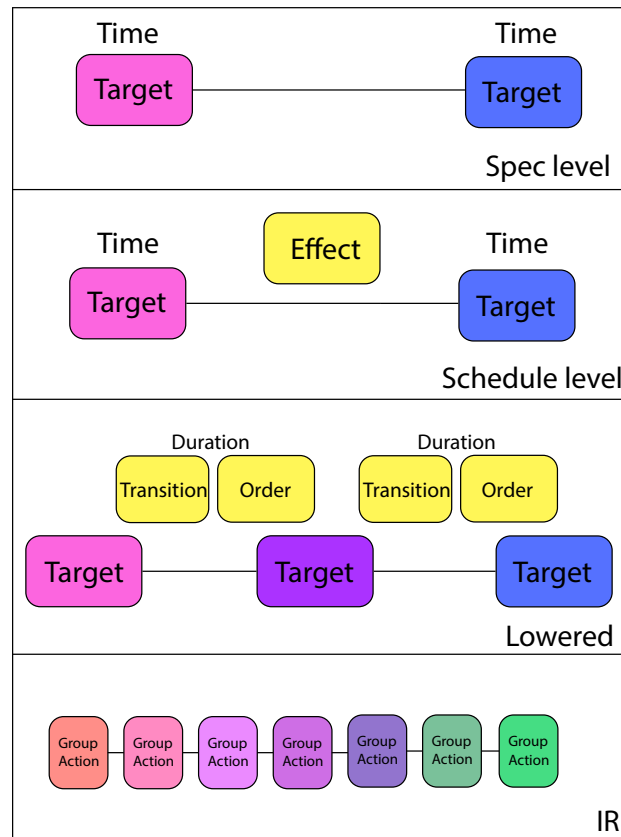


Figure 6.4: Illustrating the lowering process. The top level is the initial user input of the images and the timing. Next, the user creates a schedule by assigning Effects to each Object-specific graph. Then, these Effects are lowered and their constituent Transitions and Orders are used to generate the abstract assembly GroupAction instructions.

Transitions differ in *how* they schedule those actuations over time. For example:

- **SnapTransition** emits a single `GroupAction` that flips all differing units simultaneously at time t .
- **KeepFlippingTransition** keeps all units in `o2` flipping until the duration is up (at which point they are ensured to be flipped “on”).
- **OffsetFlipTransition** separates “on” pixels (those present in `o2`) from “off” pixels, flipping them on a half-period offset to create a staggered transition.
- **BeatFlipTransition** is a continuous-motion transition, like `KeepFlipping`, but spaces the flips out according to various beats. For example, the time between flips might alternate 1, 2, 1, 2 and form a heartbeat pattern.

6.4.3 Orders

An `Order` defines a spatial sequencing pattern independent of any particular display content. It produces a 2D *time grid*, where each cell has an activation time:

```
1 abstract class Order {  
2     abstract generateGrid(width: number, height: number): OrderedGrid;  
3     applyMask(shape: boolean[][]): [OrderedGrid, number[]];  
4 }
```

`applyMask` customizes the full grid for the shape’s active pixels. This process usually just refers to masking the grid by the shape, but some `Orders` are shape-aware and generate custom activation times.

The library provides `Order` implementations spanning a range of visual vocabularies:

- **Linear sweeps:** `BottomUp`, `TopDown`, `LeftToRight`, which assign activation time equal to row or column index.
- **Radial/concentric:** `OutFromCentre`, `CentrePulse`, which radiate outward from the geometric center.
- **Organic patterns:** `OrganicRipple` adds Perlin-like noise to a ripple effect; `MatrixDown` assigns random per-column start offsets with sequential column fill, evoking a falling-characters effect.
- **Shape-aware:** `SpiralOrder` and `GrowAlongContour` trace contour paths extracted via connected-component analysis, so the activation sequence follows the outline of the rendered shape.

Most Orders generate an activation grid directly from the dimensions and the shape. A second class of Orders takes another Order as input and derives its own activation pattern from that one. This makes the Order layer compositionally. For example:

- **StutterOrder.** Given a base order, `StutterOrder` copies its activation grid and then randomly perturbs a small fraction of the entries, swapping their activation times with neighbouring cells. The shape of the underlying motion is preserved, but individual cells fire slightly early or late, producing an organic sparkle-like effect.
- **InterpolationOrder.** `InterpolationOrder` reads its input order as a directional sweep, identifies the earliest and latest active edges of the shape under that order, and interpolates between those two contours. When the input is `LeftToRight`, the earliest edge is the leftmost contour of the shape and the latest edge is its rightmost contour. When the input is `TopToBottom`, the same construction produces a sweep between the top and bottom

contours instead. A single combinator thus generates a different shape-aware animation for every base order it is given.

6.4.4 Hardware

The `HardwareInterface` is the terminal layer that translates abstract `GroupAction` sequences into device-specific commands:

```
1 interface HardwareInterface {  
2     units: Unit[];  
3     indexToCoord: Map<number, [number, number]>;  
4     actionDurations: Map<Action, Duration>;  
5     compile(groupActions: GroupAction[]): void;  
6 }
```

`indexToCoord` provides a mapping between pixel coordinates (used throughout `Effects` and `Transitions`) and hardware unit indices (used in `GroupAction` scheduling). `actionDurations` exposes per-action timing constants—e.g., the minimum time between flip pulses—which transitions use to pace their action sequences correctly.

6.4.5 Playground

The playground is a browser-based simulation environment for authoring and previewing animations on split-flap and flipdot displays. It makes the composition of orders and transitions explicit and immediately visualizes them: a designer selects any pairing and sees the result in a 3D-rendered simulation. Thus, the design space of physical display animation becomes simple to explore. Beyond the named repertoire, the playground exposes two escape hatches for cases that built-in primitives cannot express. A painter mode lets a designer hand-author an arbitrary

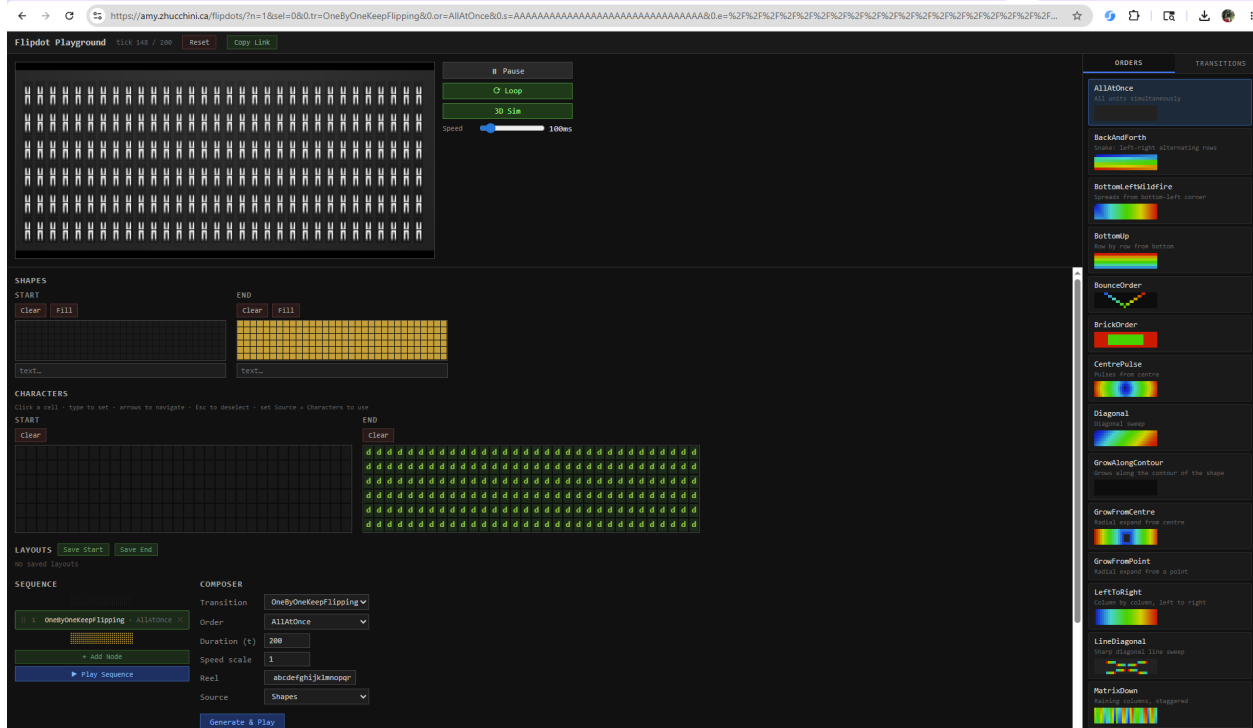


Figure 6.5: Interactive playground for prototyping combinations of Transitions and Orders. The user is able to set the start and end targets for a single Effect, or sequence together multiple Effects. For each Effect, they choose which Transition and which Order to apply, potentially setting any other parameters provided by either. They can also devise new custom components in a limited way; for Orders, they can draw the order onto a canvas, and for Transitions, they can specify the number of and spacing of flips. They can also customize the reel setup. After the user Each change is immediately reflected on the simulation board.

per-unit timing grid directly on the display canvas, producing a one-off `CustomOrder` that can be paired with any transition. A `CustomTransition` exposes a way to select various units and assign them individually configurable timing sequences, enabling bespoke behaviors without writing code. Animations can be composed into multi-step sequences, and the entire session state is serializable, so configurations can be shared or reproduced.

Because Orders and Transitions are composable building blocks, a designer can vary one axis while holding the other fixed. For example, comparing how a diagonal sweep interacts with synchronized-end flipping versus constant-rate flipping on the same message content.

6.5 Evaluation

For this evaluation, I will address the first two research questions laid out in the beginning of this chapter. To address RQ1, we will perform two case studies. In the first, we show that using effects at all is valuable. In the second, we remix different Orders and Transitions to create meaningfully varied visual impacts without any new code. For RQ2, we then discuss how to capture emergent effects and show how it can be done.

6.5.1 Effects Case Study

First, we want to show how adding specific effects can enhance the narrative of the content. That is, we want to show that controlling effects have value at all, before we even attempt to remix components or capture emergent behaviour.

Added Effects

Animation tells a story across multiple frames. Traditional pipelines start with keyframes and then use tweening strategies to fill in the intermediate frames that make a transition legible. On a

split-flap or flipdot display, the equivalent of tweening is the order in which units flip. The start and end states are fixed by the source frames, but the path between them is unspecified.

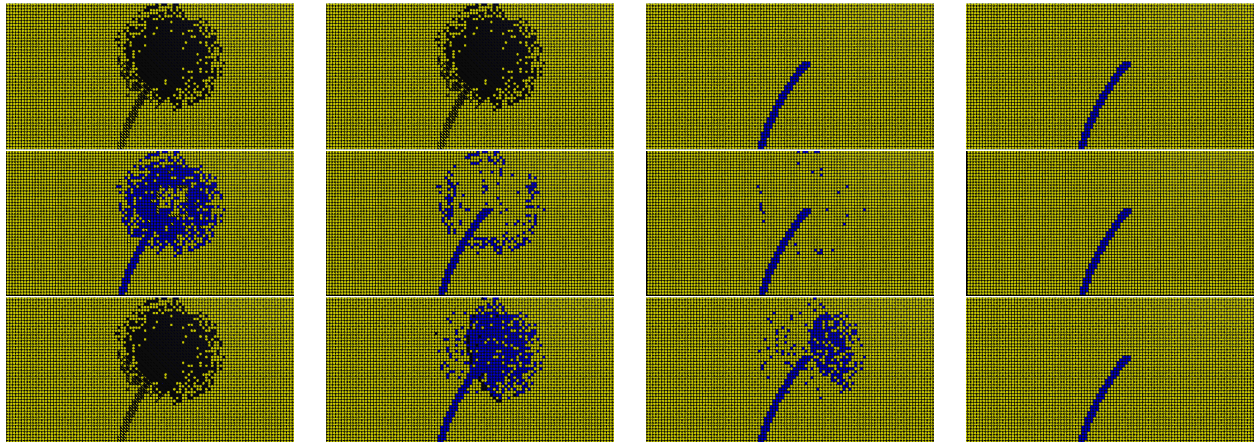


Figure 6.6: Three versions of a dandelion getting blown away. Top: FlipTransition + AllAtOnce produces an unnatural effect where the whole head disappears. Middle: FlipTransition + StutterOrder(GrowFromCentre) makes the dandelion gently dissolve. Bottom: FlipTransition + StutterOrder(CrescentOrder) makes the dandelion head directionally blown away.

First, a basic example showing why control over order matters. Here, a dandelion (Figure 6.6) is ready to be blown away. A direct snap from frame one to frame two reads as the dandelion blinking out of existence, which feels unnatural and almost comical. A FlipTransition with StutterOrder(GrowFromCentre) disperses the lit pixels outward over time, producing the impression of seeds carried off by wind. Using StutterOrder(CrescentOrder) instead makes it appear like someone is blowing the dandelion from one direction. The two end-frames are identical in either case; the order in which the units flip is what makes one read as a broken display, one as wind, and the third as a breath.

We could have produced these animations by adding extra tween frames manually or in animation software, but then we would still need to provide the remaining in-between behaviour. We could even have author every intermediate frame by hand and played them back as a fixed sequence. That works, and sometimes it is what you want. This system gives you the other option:

an end-to-end iteration loop where you play with orders, preview the result, adjust, and try again, without touching the source frames.

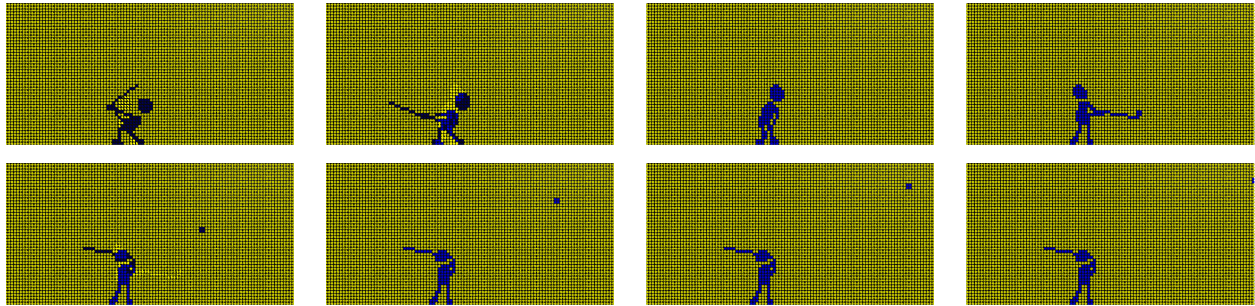


Figure 6.7: Unmodified golf animation.

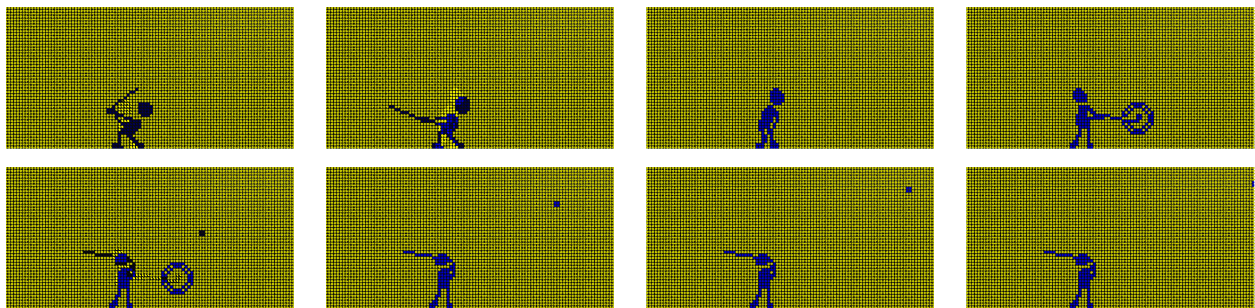


Figure 6.8: Golf animation with a highlight around the moment of collision.

The golf clip is a pre-composited animation with nine keyframes of a golfer swinging at a ball, initially programmed as seen in Listing 6.2. The main impact and the narrative core of the scene is the moment that the club connects with the ball, but because the ball is so small, and the frames are constant timing, there is no weight to that motion (Figure 6.7). The variations shown here layer effects on the input sequence using the language and try to reintroduce this emphasis. The eye tracks the ball along a traced arc of its path (Figure 6.9), expanding the visual weight across many frames. A collision burst rendered at the frame of impact brings focus back to the contact point (Figure 6.8). Each of these effects is declared onto the Objects in the clip (ball, golfstick, golfer); the source frames stay untouched.

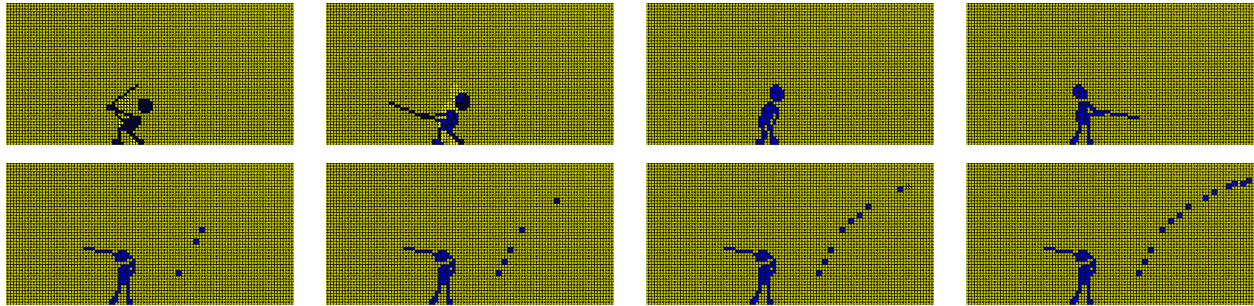


Figure 6.9: Golf animation reifying the path of the ball.

Listing 6.2: The embedded Typescript code that generates the unmodified golf animation.

```

1  const golfstick = makeTargets(golfRgb, matchColor([0, 0, 0 ]), 0, 8);
2  const golfer    = makeTargets(golfRgb, matchColor([95, 205, 228]), 0, 8);
3  const ball      = makeTargets(golfRgb, matchColor([91, 110, 225]), 4, 8);
4
5  chainSnapTransition(golfstick);
6  chainSnapTransition(golfer);
7  chainSnapTransition(ball);
8
9  const timing = [4, 8, 12, 16, 20, 24, 28, 32, 36];
10 return animationToGroupAction(
11     [golfstick[0], golfer[0], ball[0]],
12     timing,
13     hw,
14 );

```

Effects Enhance Narrative

To illustrate how transition choice shapes narration, in Figure 6.13 we animate a person thinking with split flaps, who gradually comes up with a thought. The destination text is identical and only the transition differs. With `FlipConstantSpeed`, every unit flips the minimum number of times at a constant speed and settles, so the message gradually and stochastically forms, evoking the slow crystallisation of an idea. With `FlipSyncEnd`, units calibrate their individual flip rates so that all arrive simultaneously; the thought snaps into existence as a sudden insight. With `FlipDirectional`, characters are committed left-to-right in a rolling front, as though the thought is buffered in, perhaps similar to dialogue in narrative games. The same destination state thus carries markedly different connotations depending on the path taken.

6.5.2 Effect remixing

We also want to show that the set of primitives provided by the system is enough to create a wide variety of effects out-of-the-box.

For this evaluation, we chose three types of Orders and three types of Transitions. The results are in Figure 6.14. The Transitions are `FlipOnce`, which simply flips units in the Order given, `Pulse`, which is similar, but also flips back units that were flipped $t - N$ flips ago (where N is 1 by default, but configurable), and `EvenOddRhythm`, which creates a staggered flip with an alternating flipping delay (i.e. long pause, short pause) that differs between Order numbers with even or odd parity. Each Transition ends with the units flipping to the end state.

The Orders are `BottomToTop`, which simply orders the bottom row first, `GrowAlongContours`, which traces along shapes, and `Random`, which simply assigns a random activation time to each cell. We also add `AllAtOnce`, which just flips everything at the same time, but is generally not aesthetically interesting.

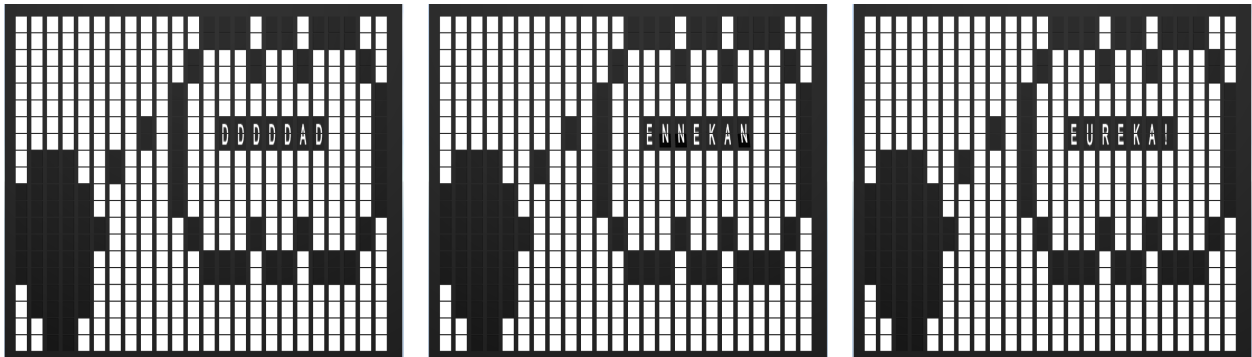


Figure 6.10: FlipConstantSpeed: Settling each letter as it comes. Note how the flaps are always simultaneous in their flipping, as the flipping rates don't vary.

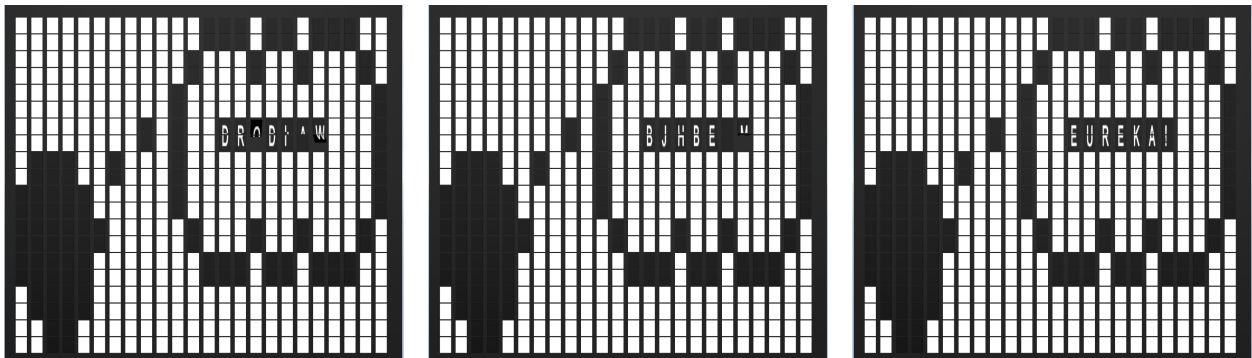


Figure 6.11: FlipSyncEnd: Syncing the text so the whole message is unveiled simultaneously; no letters are settled before the ending moment

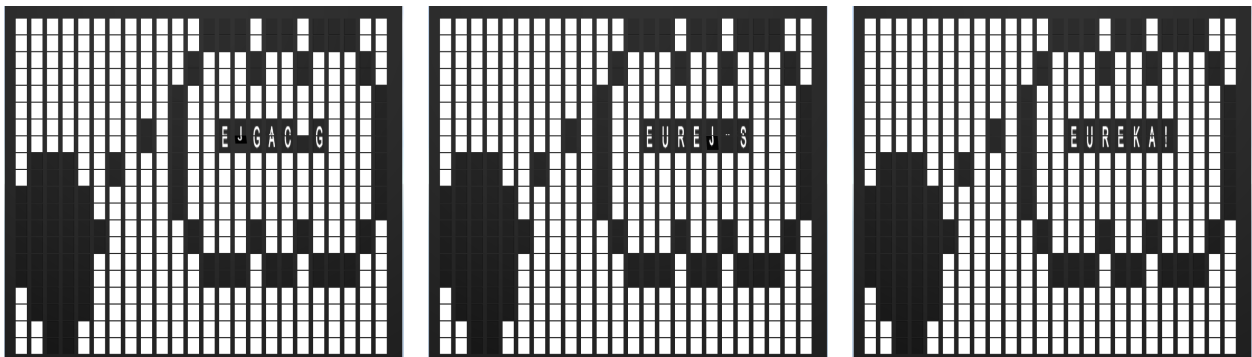


Figure 6.12: FlipDirectional: Settling the text left to right as though it's buffering. We could change the Order here to settle it in another direction (e.g. right to left).

Figure 6.13: Varying the Transition type produces different meanings.

Our goal is to create effects with different feelings and potentially used for different contexts. Some of these effects are more effective perhaps at conveying information, whereas others are less readable, but perhaps more intriguing and able to draw people in to figure out what they're looking at (like the Pulse-Contour example). Some serve to attract a lot of attention by employing constant motion, similar to how incandescent lights on old Vegas signs constantly flicker. This energy is especially apparent on the EvenOddRhythm Transition examples and the Random Order examples. Other examples have a more naturalistic motion, like the Pulse and FlipOnce examples with Contour Ordering.

6.5.3 Emergent effects

Some visual effects on these displays exist only emergently: they are artifacts of motion that the viewer's eye constructs from a sequence of frames, and they would not appear in any static rendering of the same content. Hardware interfaces expose no command that produces them directly. To make them controllable, they must be captured at the program level. Because of how these effects live “between states”, it may not be possible to find a comprehensive definition of these types of effects. However, I attempt to give a loose taxonomy of different ways these effects manifest:

1. **Frame differences.** The viewer perceives a shape moving across the display—for example, a box drifting from bottom-left to top-right—because they remember the previous frame. Any single frame on its own contains not even a part of the shape.
2. **Coherent images.** The eye assembles a recognizable image that exists at no single instant. For example, the image of a flashing arrow that first lights up the body of the arrow, then in the next frame, lights up the arrowhead. The full arrow is never present, yet the sequence of partial states resolves into one.

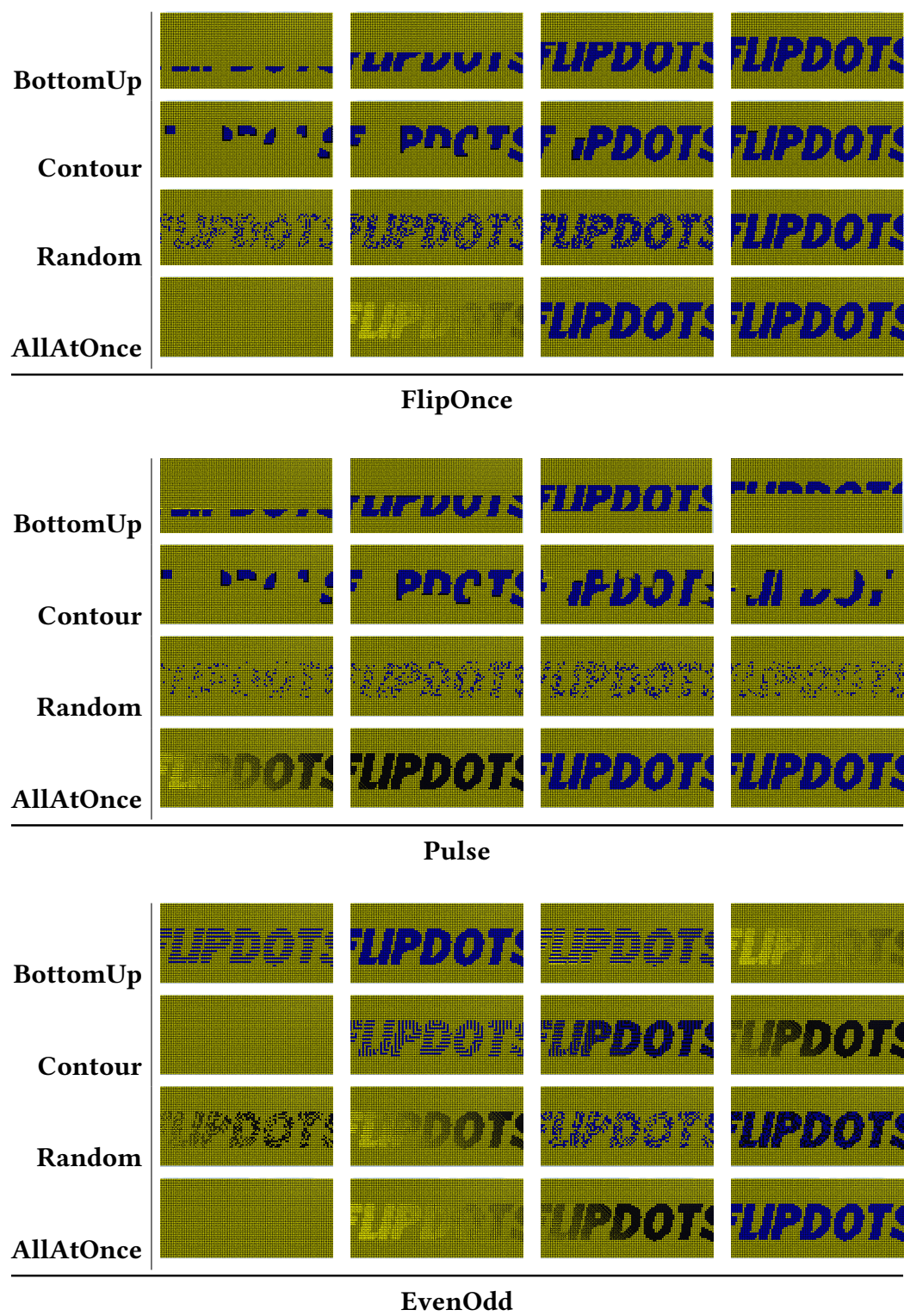


Figure 6.14: The same Orders evaluated under each Transition (FlipOnce, Pulse, EvenOdd).

3. **Differential motion.** Two regions of the display update at different rates. A foreground that flips at half the rate of its background separates it visually, even when both regions are flipping through the same units.
4. **Board-mechanical effects.** The physical construction of a particular display produces secondary visuals during a flip. On a flipdot panel with a black housing, the act of flipping briefly exposes the dark backing through the moving disc, producing a visible flicker that depends on the colour of the housing.
5. **Environmental effects.** The placement of the board in the world introduces another channel. Displays with reflective material produces brightness gradients that depend on the angle of incident light; UV-reactive paint applied to the dots is invisible until a UV lamp is switched on. These effects rely on properties outside the display itself.

Can our system capture these? Categories (1) and (2) are reachable through the language (e.g. `PulseTransition` could be used to achieve (1)) but their success relies on user-curated content: the effect is built into the specific frames the user supplies, so each instance is bespoke. Category (3) is fully codified inside a single `Transition`, since differential rates across the display are exactly what a transition can express; once defined, the `Transition` is reusable across content. Category (4) emerges from the combination of a `Transition` with a particular `Order`, and although the resulting effect depends on the physical board, as long as the board shares the characteristics that matter, we can reuse the effects for them. Category (5) is left outside the system. Environmental effects are site-specific and as varied as the real world, so the user must explore them by hand and without much system support.

To showcase how our system allows for flexible experimentation with emergent effects, we show an example of (4): the same inputs and same `Transition` create vastly different-appearing animations, depending on the `Order` that the units begin to flip.

The input graph starts with an empty canvas and transitions to a filled-in canvas, and the fixed Transition starts flipping units in the given order and continuously flips for the duration of the Transition.



Figure 6.15: Effect formed with an Order that grows in a spiral out from the centre.

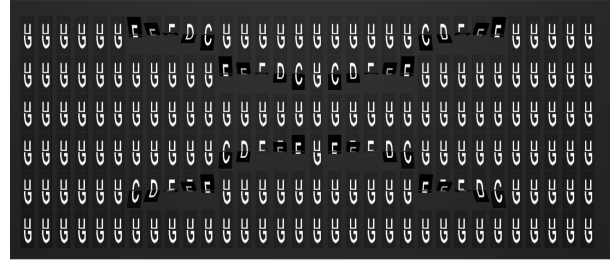


Figure 6.16: Effect formed with an Order that specifically tries to recreate an X.

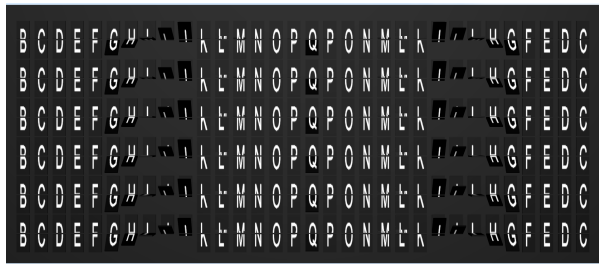


Figure 6.17: Effect formed by flipping each column together, from the middle outward.



Figure 6.18: Effect formed by going from right to left, but staggering each row to start later than the last.

Figure 6.19: Emergent effects that arise from a combination of the KeepFlipping transition and various orders. Each of these forms different figures in the darkness.

A visual difference emerges because as the flip falls, it casts a shadow at varying stages: the falling flap leans forward and blocks its own light source, then casts a shadow. This effect is most clearly visible in Figure 6.16. Therefore, by controlling the order that continuous flaps activate, we can create lines that the eye recognizes and follows.

We note that these effects lie *between* addressable states. Each unit passes through physical states that are transient; the actions of "falling" or "flipping" are not members of the explicit control set and these transient states cannot be directly accessed by any command in the machine code.

The explicit control set provided by hardware interfaces doesn't expose these states, even though interesting visual effects live there.

In contrast, by capturing this effect via a combination of Transitions and various Orders, generating each of these examples in Figure 6.19 requires just a few lines of code. In the playground, in fact, we need only click on different Orders to do so.

```
1 function keepFlipping(order: Order, duration = 50) {  
2   return (sfhw: SplitflapHardware, rect: RectangleTarget): GroupAction[] =>  
3     new KeepFlipping(order).generateGroupActions(  
4       emptyBoard(sfhw), rect, duration, sfhw);  
5 }
```

Figure 6.20: This function generates many variations on the emergent effect in Figure 6.19 by simply invoking it with different Orders.

6.6 Related work

6.6.1 Unconventional displays

There is a dearth of work on computational tools for analog/mechanical display technology. However, there is a long history of researchers building alternate types of display systems that support animation and dynamic information update.

Daniel Rozin's [smo([n. d.])] unconventional materials mirrors showcase the myriad ways to create dynamic displays that capture the viewer, from pom-poms of different colours, to reflective surfaces, to trash. Tilt display has interactive tilting tiles to enhance the 3D shapes being displayed. [Alexander et al.(2012)] Kinecam [Sethapakdi et al.(2022)] is a camera-and-printer technology that captures and creates three-frame animations that can be printed out and animated with a sliding transparency. Another genre of these displays use pins or rods that become raised and

lowered in an actuated 2.5D grid. [Leithinger and Ishii(2010), Follmer et al.(2013), Siu et al.(2018)] Most commonly, these are found in refreshable braille displays [Völkel et al.(2008), orb([n. d.))]

These displays also focus on tangible components whose state transitions are visible, like ours. Mostly, these are prototypes for research and not fabricable for the same scale as the displays we use.

Other display systems are less similar in that the final effect of the animation has a completely different modality, or the methods are extremely nontraditional and site-specific. [Alonso-Mora et al.(2012)] and follow-on work use a swarm of mobile robots, each representing a pixel, to create dynamic displays of various images. Even less related are physical volumetric displays ([Barnum et al.(2010), Ochiai et al.(2016), Bouzbib et al.(2025), Hirayama et al.(2019)]. Pixie dust [Ochiai et al.(2014)] describes how to use acoustic levitation to create imagery by precisely manipulating levitated objects. TuVe [Inoue et al.(2018)] proposes an experimental fluidic user interface that can control the shape of fluids in a tube to display characters. Thread Display [Ebisawa and Hashimoto(2025)] uses reflective threads, a laser projector, and wind to manipulate the 3D position of threads to form coherent silhouettes. A user would have to manufacture one of these displays themselves, rather than it something readily available on the consumer market.

Primrose [Dierk et al.(2022)] may be the most similar display technology, as it arranges tiles of non-emissive smart window materials that can be controlled to create varying levels of opacity and reflectiveness; these displays can be non-planar, non-rectilinear, and of varying sizes. Shutters [Coelho and Maes(2009)] similarly modifies how much light passes through each point of a curtain, displaying varying values across its surface and controlling permeability. We could imagine our techniques being adapted to displays like Primrose or Shutters, especially when they involve motion.

Integrating displays with the environment in a low-information, non-digital, shared, perception-focused fashion. [Wisneski et al.(1998)]. Our work isn't as radical, but I think we do consider the

benefits of delivering low-bandwidth information in ways that appeal to the human aesthetic.

Each of these technologies requires specialized pipelines to reason about how content will be displayed on them. Part of the motivation of our work in the transfer case is seeing the commonality that all of these have some kind of transient "transition" state as the display changes, and observing that perhaps users would want to control how this transition manifests.

6.6.2 Programmable motion graphics

There has been more work on languages for animating from sparse inputs. Mover [Ma and Agrawala(2025)] is probably the most similar and presents a DSL for describing motion and verifying its spatio-temporal properties. Others have explored generating programmatic representations of motion from video [Zhang et al.(2023)] to improve editability. Keyframer [Tseng et al.(2025)] is an LLM-based system for creating animations that uses a combination of direct editing and prompting and finds semantic categories that correspond to motion animation. We think that these ideas can be adapted to the low-level programs for split flaps and flip dots hardware.

6.6.3 Low-resolution rendering

Various works have described how to take a high-resolution image and attractively generate a lower-resolution, alternate rendering-method version, like pixel art [Gerstner et al.(2012)] or LEGO blocks [Zhou et al.(2023)]. In motion and animation, [Bénard et al.(2013)] surveys techniques for rendering stylized animations with sufficient temporal cohesion.

6.6.4 Scheduling

Section 5.7 contains a discussion of scheduling languages, most of which applies. Importantly, scheduling languages have not yet focused on image and geometry preservation, physical actuation,

or properties over time.

6.7 Conclusion

We successfully create a system that allows us to create animations for electromechanical displays. This system is focused on compiler extensibility and the ability to easily explore the space of potential animations, especially accounting for how these displays act in the physical world.

Thus, we created language-based abstractions of the domain-specific fabrication operators. These were the reusable components that made up Effects and controlled how units act. We used a combination of geometry in the form of keyframes, as well as construction details in the form of specifying what effects were used, as the design input. We showed how to enable user-in-the-loop iteration workflows to give instant feedback.

In creating this system, we thought of the construction plan for realizing the animation as a “schedule”, which said what order and when to activate units. Subsequently, we think of the “performance” of a particular construction plan or schedule to mean perception, or what appears to the viewer when the schedule is played.

We believe this shows that the three principles for creating these design representations we laid out for fabrication domains still holds in this novel domain, and think there is much promise in this transfer case.

Chapter 7

Conclusion

My goal has been to create design systems where users are able to control the appearance and behaviour of the object in conjunction with how that object is realized. The construction determines what manufacturing processes are used, the ease of creation, the cost, the speed, the efficiency, the sustainability, the accessibility, and the performance of the object.

We show that without geometry as partial input, the expressiveness of the input is limited; in fact, some form of geometric representation is essential to compactly describe the appearance and shape of the final object. We also show that this must be in conjunction with specifying its construction; the geometry should not simply be a “target”. (If it were, there would be some optimization step that the user does not directly specify, meaning the user would not necessarily be able to predict the output.)

We show that these construction details should be captured as language abstractions of physical, domain-specific processes. This decision is natural, as the steps of construction are essentially imperative commands, and digital fabrication machines are already programmed with low-level machine code. However, we want to elevate the reasoning beyond the machine code level, so that it is possible to describe the construction plan in a standalone, self-contained way, whether that is

“knitting something from left to right, unfolded, and in garter” or “synchronize so the final image is revealed simultaneously”. Thus, we design abstractions of the construction details at that level.

Finally, we show the necessity of building user-in-the-loop systems. While the specifics of building user interfaces are beyond the scope of this work, the representations used must support direct editing, as well as have previewing processes to shorten iteration lengths. The user should know exactly how to make a change in the input to effect the change they want in the output.

I hope that this recipe will serve anyone well who wants to build their own design tools for their own physical domains, however far-flung or experimental those domains might be, however strange those construction ideas, and however imaginative those designs.

Bibliography

- [smo([n. d.])] [n. d.]. Daniel Rozin Interactive Art — smoothware.com. <https://www.smoothware.com/danny/index.html>. [Accessed 28-05-2026].
- [fli([n. d.])] [n. d.]. File:Luminator MegaMax flip-dot destination sign in TriMet bus 2246, a 1998 New Flyer D40LF.jpg - Wikimedia Commons — commons.wikimedia.org. https://commons.wikimedia.org/wiki/File:Luminator_MegaMax_flip-dot_destination_sign_in_TriMet_bus_2246,_a_1998_New_Flyer_D40LF.jpg. [Accessed 28-05-2026].
- [spl([n. d.])] [n. d.]. Net migration turns negative as more people leave the country long term than arrive — interest.co.nz. <https://www.interest.co.nz/property/128658/sharp-increase-number-nz-citizens-leaving-country-long-term-has-pushed-country>. [Accessed 28-05-2026].
- [orb([n. d.])] [n. d.]. orbitresearch.com. <http://www.orbitresearch.com/product/graphiti/>. [Accessed 28-05-2026].
- [Albaugh et al.(2019)] Lea Albaugh, Scott Hudson, and Lining Yao. 2019. Digital Fabrication of Soft Actuated Objects by Machine Knitting. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, Glasgow Scotland Uk, 1–4. doi:10.1145/3290607.3313270
- [Albaugh et al.(2023)] Lea Albaugh, Scott E Hudson, and Lining Yao. 2023. Physically Situated Tools for Exploring a Grain Space in Computational Machine Knitting. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. ACM, Hamburg Germany, 1–14. doi:10.1145/3544548.3581434
- [Albaugh et al.(2021)] Lea Albaugh, James McCann, Scott E. Hudson, and Lining Yao. 2021. Engineering Multifunctional Spacer Fabrics Through Machine Knitting. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–12. doi:10.1145/3411764.3445564

- [Alexa and Matusik(2011)] Marc Alexa and Wojciech Matusik. 2011. Images from Self-Occlusion. In *Proceedings of the International Symposium on Computational Aesthetics in Graphics, Visualization, and Imaging* (Vancouver, British Columbia, Canada) (CAE '11). Association for Computing Machinery, New York, NY, USA, 17–24. doi:10.1145/2030441.2030445
- [Alexander et al.(2012)] Jason Alexander, Andrés Lucero, and Sriram Subramanian. 2012. Tilt displays: designing display surfaces with multi-axis tilting and actuation. In *Proceedings of the 14th International Conference on Human-Computer Interaction with Mobile Devices and Services* (San Francisco, California, USA) (MobileHCI '12). Association for Computing Machinery, New York, NY, USA, 161–170. doi:10.1145/2371574.2371600
- [Alonso-Mora et al.(2012)] Javier Alonso-Mora, Andreas Breitenmoser, Martin Rufli, Roland Siegwart, and Paul Beardsley. 2012. Image and animation display with multiple mobile robots. *Int. J. Rob. Res.* 31, 6 (May 2012), 753–773. doi:10.1177/0278364912442095
- [Apodaca(2008)] Mariella Apodaca. 2008. *Kidlett Mittenz*. Ravelry. <https://www.ravelry.com/patterns/library/kidlett-mittenz> Knitting pattern page. Accessed 2026-04-16.
- [Barnum et al.(2010)] Peter C. Barnum, Srinivasa G. Narasimhan, and Takeo Kanade. 2010. A multi-layered display with water drops. In *ACM SIGGRAPH 2010 Papers* (Los Angeles, California) (SIGGRAPH '10). Association for Computing Machinery, New York, NY, USA, Article 76, 7 pages. doi:10.1145/1833349.1778813
- [Bénard et al.(2013)] Pierre Bénard, Forrester Cole, Michael Kass, Igor Mordatch, James Hegarty, Martin Sebastian Senn, Kurt Fleischer, Davide Pesare, and Katherine Breeden. 2013. Stylizing animation by example. *ACM Trans. Graph.* 32, 4, Article 119 (July 2013), 12 pages. doi:10.1145/2461912.2461929
- [Bermano et al.(2012)] Amit Bermano, Ilya Baran, Marc Alexa, and Wojciech Matusk. 2012. ShadowPix: Multiple Images from Self Shadowing. *Computer Graphics Forum* 31, 2pt3 (May 2012), 593–602. doi:10.1111/j.1467-8659.2012.03038.x
- [Bouzbib et al.(2025)] Elodie Bouzbib, Iosune Sarasate, Unai Javier Fernández, Ivan Fernández, Manuel Lopez-Amo, Iñigo Ezcurdia, and Asier Marzo. 2025. FlexiVol: a Volumetric Display with an Elastic Diffuser to Enable Reach-Through Interaction. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, Article 977, 16 pages. doi:10.1145/3706598.3714315
- [by Sofie Hågensen(2020)] Strikkesoffe by Sofie Hågensen. 2020. *Bedre Sammen Votter / Better Together Mittens*. Ravelry. <https://www.ravelry.com/patterns/library/bedre-sammen-votter--better-together-mittens> Knitting pattern page. Accessed 2026-04-16.

- [Chen et al.(2018)] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR* abs/1802.04799 (2018). arXiv:1802.04799 <http://arxiv.org/abs/1802.04799>
- [Coelho and Maes(2009)] Marcelo Coelho and Pattie Maes. 2009. Shutters: a permeable surface for environmental control and communication. In *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction* (Cambridge, United Kingdom) (*TEI '09*). Association for Computing Machinery, New York, NY, USA, 13–18. doi:10.1145/1517664.1517671
- [Dierk et al.(2022)] Christine Dierk, TJ Rhodes, and Gavin Miller. 2022. Project Primrose: Reflective Light-Diffuser Modules for Non-Emissive Flexible Display Systems. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Bend, OR, USA) (*UIST '22*). Association for Computing Machinery, New York, NY, USA, Article 22, 14 pages. doi:10.1145/3526113.3545625
- [Ebisawa and Hashimoto(2025)] Tsukasa Ebisawa and Naoki Hashimoto. 2025. *Thread Display: A Novel Wind-Driven Spatial Display Technique*. Association for Computing Machinery, New York, NY, USA. <https://doi.org/10.1145/3757374.3771485>
- [Follmer et al.(2013)] Sean Follmer, Daniel Leithinger, Alex Olwal, Akimitsu Hogge, and Hiroshi Ishii. 2013. inFORM: dynamic physical affordances and constraints through shape and object actuation. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology* (St. Andrews, Scotland, United Kingdom) (*UIST '13*). Association for Computing Machinery, New York, NY, USA, 417–426. doi:10.1145/2501988.2502032
- [Gerstner et al.(2012)] Timothy Gerstner, Doug DeCarlo, Marc Alexa, Adam Finkelstein, Yotam Gingold, and Andrew Nealen. 2012. Pixelated image abstraction. In *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering* (Annecy, France) (*NPAR '12*). Eurographics Association, Goslar, DEU, 29–36.
- [Hagedorn et al.(2020a)] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodík, and Vinod Grover. 2020a. Fireiron: A Scheduling Language for High-Performance Linear Algebra on GPUs. *CoRR* abs/2003.06324 (2020). arXiv:2003.06324 <https://arxiv.org/abs/2003.06324>
- [Hagedorn et al.(2020b)] Bastian Hagedorn, Johannes Lenfers, Thomas Köhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020b. Achieving High-Performance the Functional Way: A Functional Pearl on Expressing High-Performance Optimizations as Rewrite Strategies. *Proceedings of the ACM on Programming Languages* 4, ICFP, Article 92 (2020), 29 pages. doi:10.1145/3408974

- [Hester et al.(2025)] Jack Hester, Sebastian Law, and Megan Hofmann. 2025. QUILT: Supporting Modular Design of Machine-Knitting Programs. In *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology (UIST '25)*. Association for Computing Machinery, New York, NY, USA, Article 101, 26 pages. doi:10.1145/3746059.3747608
- [Heusser(2015)] Erica Heusser. 2015. *Songbird Mittens*. Ravelry. <https://www.ravelry.com/patterns/library/songbird-mittens> Knitting pattern page. Accessed 2026-04-16.
- [Higgins(2005a)] Brooke T. Higgins. 2005a. *Tychus*. <https://knitty.com/ISSUEsummer05/PATTtychus.html> Knitting pattern, Summer 2005 issue. Accessed 2026-04-15.
- [Higgins(2005b)] Brooke T. Higgins. 2005b. *Tychus*. Ravelry. <https://www.ravelry.com/patterns/library/tychus> Knitting pattern page. Accessed 2026-04-15.
- [Hirayama et al.(2019)] Ryuji Hirayama, Diego Martinez Plasencia, Nobuyuki Masuda, and Sriram Subramanian. 2019. A volumetric display for visual, tactile and audio presentation using acoustic trapping. *Nature* 575 (2019), 320 – 323. <https://api.semanticscholar.org/CorpusID:207986492>
- [Hirose et al.(2024)] Yuichi Hirose, Mark Gillespie, Angelica M. Bonilla Fominaya, and James McCann. 2024. Solid Knitting. *ACM Trans. Graph.* 43, 4, Article 88 (July 2024), 15 pages. doi:10.1145/3658123
- [Hofmann(2024)] Megan Hofmann. 2024. KODA: Knit-program Optimization by Dependency Analysis. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (Pittsburgh, PA, USA) (*UIST '24*). Association for Computing Machinery, New York, NY, USA, Article 64, 15 pages. doi:10.1145/3654777.3676405
- [Hofmann et al.(2019)] Megan Hofmann, Lea Albaugh, Ticha Sethapakadi, Jessica Hodgins, Scott E. Hudson, James McCann, and Jennifer Mankoff. 2019. KnitPicking Textures: Programming and Modifying Complex Knitted Textures for Machine and Hand Knitting. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) (*UIST '19*). Association for Computing Machinery, New York, NY, USA, 5–16. doi:10.1145/3332165.3347886
- [Hofmann et al.(2023)] Megan Hofmann, Lea Albaugh, Tongyan Wang, Jennifer Mankoff, and Scott E Hudson. 2023. KnitScript: A Domain-Specific Scripting Language for Advanced Machine Knitting. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (<conf-loc>, <city>San Francisco</city>, <state>CA</state>, <country>USA</country>, </conf-loc>) (*UIST '23*). Association for Computing Machinery, New York, NY, USA, Article 21, 21 pages. doi:10.1145/3586183.3606789

- [Hofmann et al.(2020)] Megan Hofmann, Jennifer Mankoff, and Scott E. Hudson. 2020. KnitGIST: A Programming Synthesis Toolkit for Generating Functional Machine-Knitting Textures. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. ACM, Virtual Event USA, 1234–1247. doi:10.1145/3379337.3415590
- [Hu et al.(2019)] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Transactions on Graphics* 38, 6, Article 201 (2019), 16 pages. doi:10.1145/3355089.3356506
- [Igarashi et al.(2008)] Yuki Igarashi, Takeo Igarashi, and Hiromasa Suzuki. 2008. Knitty: 3D Modeling of Knitted Animals with a Production Assistant Interface. In *Eurographics 2008 - Short Papers*, Katerina Mania and Eric Reinhard (Eds.). The Eurographics Association. doi:10.2312/egs.20081011
- [Ikarashi et al.(2022)] Yuka Ikarashi, Gilbert Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for Productive Programming of Hardware Accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*. ACM, 703–718. doi:10.1145/3519939.3523446
- [Inoue et al.(2018)] Yuki Inoue, Yuichi Itoh, and Takao Onoye. 2018. TuVe: a flexible display with a tube. In *SIGGRAPH Asia 2018 Emerging Technologies* (Tokyo, Japan) (SA '18). Association for Computing Machinery, New York, NY, USA, Article 16, 2 pages. doi:10.1145/3275476.3275487
- [Jones et al.(2022)] Benjamin Jones, Yuxuan Mei, Haisen Zhao, Taylor Gotfrid, Jennifer Mankoff, and Adriana Schulz. 2022. Computational Design of Knit Templates. *ACM Transactions on Graphics* 41, 2 (April 2022), 1–16. doi:10.1145/3488006
- [Kaspar et al.(2019)] Alexandre Kaspar, Liane Makatura, and Wojciech Matusik. 2019. Knitting Skeletons: A Computer-Aided Design Tool for Shaping and Patterning of Knitted Garments. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) (UIST '19). Association for Computing Machinery, New York, NY, USA, 53–65. doi:10.1145/3332165.3347879
- [Kaspar et al.(2021)] Alexandre Kaspar, Kui Wu, Yiyue Luo, Liane Makatura, and Wojciech Matusik. 2021. Knit sketching: from cut & sew patterns to machine-knit garments. *ACM Transactions on Graphics* 40, 4 (Aug. 2021), 1–15. doi:10.1145/3476576.3476614
- [Kjolstad et al.(2017)] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. doi:10.1145/3133901

- [Lange(2015)] Leeanne Lange. 2015. *Winter Walk Mittens*. Ravelry. <https://www.ravelry.com/patterns/library/winter-walk-mittens> Knitting pattern page. Accessed 2026-04-16.
- [Leithinger and Ishii(2010)] Daniel Leithinger and Hiroshi Ishii. 2010. Relief: a scalable actuated shape display. In *Proceedings of the Fourth International Conference on Tangible, Embedded, and Embodied Interaction* (Cambridge, Massachusetts, USA) (*TEI '10*). Association for Computing Machinery, New York, NY, USA, 221–222. doi:10.1145/1709886.1709928
- [Lin and McCann(2021)] Jenny Lin and James McCann. 2021. An Artin Braid Group Representation of Knitting Machine State with Applications to Validation and Optimization of Fabrication Plans. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, Xi'an, China, 1147–1153. doi:10.1109/ICRA48506.2021.9562113
- [Lin et al.(2023)] Jenny Lin, Vidya Narayanan, Yuka Ikarashi, Jonathan Ragan-Kelley, Gilbert Bernstein, and James McCann. 2023. Semantics and Scheduling for Machine Knitting Compilers. *ACM Trans. Graph.* 42, 4 (Aug. 2023), 17. doi:10.1145/3592449 In Press.
- [Lin et al.(2018)] Jenny Lin, Vidya Narayanan, and James McCann. 2018. Efficient transfer planning for flat knitting. In *Proceedings of the 2nd ACM Symposium on Computational Fabrication*. ACM, Cambridge Massachusetts, 1–7. doi:10.1145/3213512.3213515
- [Linneman(2005)] Laura Linneman. 2005. *Afterthought Heel Socks*. Ravelry. <https://www.ravelry.com/patterns/library/afterthought-heel-socks> Knitting pattern page. First published January 2005. Accessed 2026-04-15.
- [Luo et al.(2021)] Yiyue Luo, Kui Wu, Tomás Palacios, and Wojciech Matusik. 2021. KnitUI: Fabricating Interactive and Sensing Textiles with Machine Knitting. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–12. doi:10.1145/3411764.3445780
- [Luo et al.(2022)] Yiyue Luo, Kui Wu, Andrew Spielberg, Michael Foshey, Daniela Rus, Tomás Palacios, and Wojciech Matusik. 2022. Digital Fabrication of Pneumatic Actuators with Integrated Sensing by Machine Knitting. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) (*CHI '22*). Association for Computing Machinery, New York, NY, USA, Article 175, 13 pages. doi:10.1145/3491102.3517577
- [Ma and Agrawala(2025)] Jiaju Ma and Maneesh Agrawala. 2025. MoVer: Motion Verification for Motion Graphics Animations. *ACM Trans. Graph.* 44, 4, Article 33 (July 2025), 17 pages. doi:10.1145/3731209
- [Martha Stewart Living(2013)] Martha Stewart Living. 2013. *Cozy Mittens*. 165 pages. Martha Stewart Living, December 2013.

- [McCann et al.(2016)] James McCann, Lea Albaugh, Vidya Narayanan, April Grow, Wojciech Matusik, Jennifer Mankoff, and Jessica Hodgins. 2016. A compiler for 3D machine knitting. *ACM Transactions on Graphics* 35, 4 (July 2016), 1–11. doi:10.1145/2897824.2925940
- [Mitra et al.(2024)] Rahul Mitra, Erick Jimenez Berumen, Megan Hofmann, and Edward Chien. 2024. Singular Foliations for Knit Graph Design. In *ACM SIGGRAPH 2024 Conference Proceedings*. 1–9.
- [Mitra et al.(2023)] Rahul Mitra, Liane Makatura, Emily Whiting, and Edward Chien. 2023. Helix-Free Stripes for Knit Graph Design. In *ACM SIGGRAPH 2023 Conference Proceedings*. 1–9.
- [Nader et al.(2021)] Georges Nader, Yu Han Quek, Pei Zhi Chia, Oliver Weeger, and Sai-Kit Yeung. 2021. KnitKit: a flexible system for machine knitting of customizable textiles. *ACM Transactions on Graphics* 40, 4 (Aug. 2021), 1–16. doi:10.1145/3450626.3459790
- [Narayanan et al.(2018a)] Vidya Narayanan, Lea Albaugh, Jessica Hodgins, Stelian Coros, and James Mccann. 2018a. Automatic Machine Knitting of 3D Meshes. *ACM Transactions on Graphics* 37, 3 (June 2018), 1–15. doi:10.1145/3186265
- [Narayanan et al.(2018b)] Vidya Narayanan, Lea Albaugh, Jessica Hodgins, Stelian Coros, and James Mccann. 2018b. Automatic Machine Knitting of 3D Meshes. *ACM Trans. Graph.* 37, 3, Article 35 (Aug. 2018), 15 pages. doi:10.1145/3186265
- [Narayanan et al.(2019)] Vidya Narayanan, Kui Wu, Cem Yuksel, and James McCann. 2019. Visual knitting machine programming. *ACM Transactions on Graphics* 38, 4 (Aug. 2019), 1–13. doi:10.1145/3306346.3322995
- [Ochiai et al.(2014)] Yoichi Ochiai, Takayuki Hoshi, and Jun Rekimoto. 2014. Pixie dust: graphics generated by levitated and animated objects in computational acoustic-potential field. *ACM Trans. Graph.* 33, 4, Article 85 (July 2014), 13 pages. doi:10.1145/2601097.2601118
- [Ochiai et al.(2016)] Yoichi Ochiai, Kota Kumagai, Takayuki Hoshi, Jun Rekimoto, Satoshi Hasegawa, and Yoshio Hayasaki. 2016. Fairy Lights in Femtoseconds: Aerial and Volumetric Graphics Rendered by Focused Femtosecond Laser Combined with Computational Holographic Fields. *ACM Trans. Graph.* 35, 2, Article 17 (Feb. 2016), 14 pages. doi:10.1145/2850414
- [Patons([n. d.])] Patons. [n. d.]. *Mitts knitted on 4 needles*. Ravelry. <https://www.ravelry.com/patterns/library/mitts-knitted-on-4-needles> Knitting pattern page. Published in *Patons Beehive 140, Socks, Mitts and Gloves for Children*. Accessed 2026-04-16.
- [Ragan-Kelley et al.(2013)] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for

- optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.* 48, 6 (June 2013), 519–530. doi:10.1145/2499370.2462176
- [Ravelry(2025)] Ravelry. 2025. Ravelry: A knit and crochet community. <https://www.ravelry.com/>. Accessed: 2025-03-20.
- [Reeves(2008)] Nadine Reeves. 2008. *Newfie Mittens*. Ravelry. <https://www.ravelry.com/patterns/library/newfie-mittens> Knitting pattern page. Accessed 2026-04-16.
- [Scott(2018)] Shirley A. Scott. 2018. *Spring Ice*. Ravelry. <https://www.ravelry.com/patterns/library/spring-ice-2> Knitting pattern page. Published in *Saltwater Mittens*. Accessed 2026-04-16.
- [Sethapakdi et al.(2022)] Ticha Sethapakdi, Mackenzie Leake, Catalina Monsalve Rodriguez, Miranda J. Cai, and Stefanie Mueller. 2022. KineCAM: An Instant Camera for Animated Photographs. *Proc. ACM Comput. Graph. Interact. Tech.* 5, 4, Article 46 (Sept. 2022), 9 pages. doi:10.1145/3533613
- [Shen et al.(2023)] Pengfei Shen, Ruizeng Li, Beibei Wang, and Ligang Liu. 2023. Scratch-based Reflection Art via Differentiable Rendering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2023)* 42, 4 (2023), 1–12.
- [Siu et al.(2018)] Alexa F. Siu, Eric J. Gonzalez, Shenli Yuan, Jason B. Ginsberg, and Sean Follmer. 2018. shapeShift: 2D Spatial Manipulation and Self-Actuation of Tabletop Shape Displays for Tangible and Haptic Interaction. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) (*CHI '18*). Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3173574.3173865
- [Soho(2015)] Purl Soho. 2015. *Arched Gusset Mittens*. Ravelry. <https://www.ravelry.com/patterns/library/arched-gusset-mittens> Knitting pattern page. Accessed 2026-04-16.
- [Swansen(2016)] Cully Swansen. 2016. *The Complete Surprise: Knitting Elizabeth Zimmermann's Surprise Jacket*. Schoolhouse Press, Pittsville, WI. 104 pages. <https://www.schoolhousepress.com/the-complete-surprise-knitting-elizabeth-zimmermann-s-surprise-jacket.html> Softcover.
- [tincanknits(2016)] tincanknits. 2016. The World's Simplest Mittens. <https://www.ravelry.com/patterns/library/the-worlds-simplest-mittens> Knitting pattern page. Accessed 2026-04-16.
- [Together(2018)] We Knit Together. 2018. *Teeny-tiny Knit Mitts*. Ravelry. <https://www.ravelry.com/patterns/library/teeny-tiny-knit-mitts> Knitting pattern page. Accessed 2026-04-16.

- [Tseng et al.(2025)] Tiffany Tseng, Ruijia Cheng, and Jeffrey Nichols. 2025. Keyframer: Empowering Animation Design using Large Language Models. doi:10.48550/arXiv.2402.06071 arXiv:2402.06071.
- [Väätäinen(2016)] Tiina Väätäinen. 2016. *Kinos*. Ravelry. <https://www.ravelry.com/patterns/library/kinos> Knitting pattern page. Accessed 2026-04-16.
- [Völkel et al.(2008)] Thorsten Völkel, Gerhard Weber, and Ulrich Baumann. 2008. Tactile Graphics Revised: The Novel BrailleDis 9000 Pin-Matrix Device with Multitouch Input. In *Proceedings of the 11th International Conference on Computers Helping People with Special Needs (linz, Austria) (ICCHP '08)*. Springer-Verlag, Berlin, Heidelberg, 835–842. doi:10.1007/978-3-540-70540-6_124
- [Winslow(2018)] Alexis Winslow. 2018. *Cabled Dad Mittens*. Ravelry. <https://www.ravelry.com/patterns/library/cabled-dad-mittens>
- [Wisneski et al.(1998)] Craig Wisneski, Hiroshi Ishii, Andrew Dahley, Matthew G. Gorbet, Scott Brave, Brygg Ullmer, and Paul Yarin. 1998. Ambient Displays: Turning Architectural Space into an Interface between People and Digital Information. In *Proceedings of the First International Workshop on Cooperative Buildings, Integrating Information, Organization, and Architecture (CoBuild '98)*. Springer-Verlag, Berlin, Heidelberg, 22–32.
- [Wu et al.(2019)] Kui Wu, Hannah Swan, and Cem Yuksel. 2019. Knittable Stitch Meshes. *ACM Transactions on Graphics* 38, 1 (Feb. 2019), 1–13. doi:10.1145/3292481
- [Zhang et al.(2023)] Sharon Zhang, Jiaju Ma, Jiajun Wu, Daniel Ritchie, and Maneesh Agrawala. 2023. Editing Motion Graphics Video via Motion Vectorization and Transformation. *ACM Trans. Graph.* 42, 6, Article 229 (Dec. 2023), 13 pages. doi:10.1145/3618316
- [Zhou et al.(2023)] Mingjun Zhou, Jiahao Ge, Hao Xu, and Chi-Wing Fu. 2023. Computational Design of LEGO® Sketch Art. *ACM Trans. Graph.* 42, 6, Article 201 (Dec. 2023), 15 pages. doi:10.1145/3618306
- [Zhu et al.(2024)] Amy Zhu, Yuxuan Mei, Benjamin Jones, Zachary Tatlock, and Adriana Schulz. 2024. Computational Illusion Knitting. *ACM Trans. Graph.* 43, 4, Article 152 (July 2024), 13 pages. doi:10.1145/3658231
- [Zhu et al.(2023)] Amy Zhu, Adriana Schulz, and Zachary Tatlock. 2023. Exploring Self-Embedded Knitting Programs with Twine. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design (Seattle, WA, USA) (FARM 2023)*. Association for Computing Machinery, New York, NY, USA, 25–31. doi:10.1145/3609023.3609805
- [Zimmermann(1987a)] Elizabeth Zimmermann. 1987a. *Sideways Mystery Mittens*. Schoolhouse Press. <https://www.schoolhousepress.com/sideways-mystery-mittens-spp76.html> Schoolhouse Press Pattern #76. First published 1987. Accessed 2026-04-15.

- [Zimmermann(1987b)] Elizabeth Zimmermann. 1987b. *Sideways Mystery Mittens*. Ravelry. <https://www.ravelry.com/patterns/library/sideways-mystery-mittens> Knitting pattern page. First published January 1987. Accessed 2026-04-15.
- [Zimmermann(1989)] Elizabeth Zimmermann. 1989. *Knitting Around*. Schoolhouse Press, Pittsville, WI. 240 pages. <https://www.schoolhousepress.com/knitting-around-updated-preorder.html> Updated edition, softcover.