A Compiler for 3D Machine Knitting

James McCann[1] Lea Albaugh[1] Vidya Narayanan[1] April Grow[1,2]
Wojciech Matusik[3] Jen Mankoff[1,4] Jessica Hodgins[1]

[1]Disney Research [2]UC Santa Cruz [3]Massacusetts Institute of Technology [4]Carnegie Mellon University

Figure 1: *Our compiler processes high-level primitives into low-level instructions for production on industrial knitting machines.*

Abstract

Industrial knitting machines can produce finely detailed, seamless, 3D surfaces quickly and without human intervention. However, the tools used to program them require detailed manipulation and understanding of low-level knitting operations. We present a compiler that can automatically turn assemblies of high-level shape primitives (tubes, sheets) into low-level machine instructions. These high-level shape primitives allow knit objects to be scheduled, scaled, and otherwise shaped in ways that require thousands of edits to low-level instructions. At the core of our compiler is a heuristic transfer planning algorithm for knit cycles, which we prove is both sound and complete. This algorithm enables the translation of high-level shaping and scheduling operations into needle-level operations. We show a wide range of examples produced with our compiler and demonstrate a basic visual design interface that uses our compiler as a backend.

Keywords: knitting, fabrication, transfer planning, knitting machine

Concepts: •**Computing methodologies** → *Planning and scheduling; Graphics systems and interfaces;* •**Applied computing** → *Computer-aided manufacturing;* •**Software and its engineering** → Compilers;

SIGGRAPH '16 Technical Paper,, July 24 - 28, 2016, Anaheim, CA,
ISBN: 978-1-4503-4279-7/16/07

1 Introduction

One of the long-standing goals of computer graphics research has been to help people express themselves, and one of the ways we do this is by building tools and APIs that allow people to easily interact with output devices that they would otherwise have trouble controlling. OpenGL and Postscript help people control display hardware, CAD tools help folks program machine shops, and, recently, a myriad of tools have sprung up to help users control 3D printers.

Successful tools and APIs are built around high-level primitives that map well both to the hardware being controlled and to the tasks the user is likely to wish the hardware to perform. In this paper we introduce such primitives for the domain of machine knitting and show how they can be compiled to low-level machine operations.

Machine knitting is a mature fabrication technology, used to create items ranging from gardening gloves to fashionable sweaters. Knitting machines are programmable, general-purpose devices; however, they are used almost exclusively to manufacture a fixed palette of pre-programmed objects, occasionally with some customization of color patterns. No knit shop today approaches the flexibility common to CNC-on-demand machine shop operations.

This lack of flexibility is a consequence of current knit design tools. The industry standard tools for machine knitting [Shima Seiki 2011; Stoll 2011] provide high-level templates for a few standard objects, but otherwise leave the user to manipulate needle-level control instructions in a way that fails to divorce machine-specific details from actual fabrication operations. This situation is similar to requiring all CNC machine users to write toolpath G-code by hand, or all computer programmers to work in assembly.

In an attempt to change this landscape, we have developed a compiler that allows knitted objects to be specified in terms of high-level

DOI: http://dx.doi.org/10.1145/2897824.2925940

shape primitives, rather than detailed stitch descriptions. These primitives are knittable *by construction* – that is, they can be automatically transformed into stitch level instructions. With our primitives, users can create and edit designs at a high level, and easily change knitting order, needle location, shape, and scale.

For example, when knitting a design in a new yarn or on a new machine, a user may wish to scale it up or down to account for the different stitch size and aspect ratio. From an aesthetic standpoint, high-level operations decrease iteration time – adjusting the bend angle of a stuffed toy's arm is much easier when the bend is specified as a single primitive than when it is specified as a collection of hundreds of individual stitch commands. Also, because our input format provides *scheduling* (knitting order and location) as well as shaping control, users can see which needles on the machine will be in use at a given time and adjust their design to avoid any conflicts; without changing its shape.

The main contributions of our work are

- A knitting design representation consisting of generalized tubes and sheets, with gluing instructions at their boundaries, which allows high-level schedule and structure manipulation.
- A knitting assembly language that formalizes the low-level operations used by industrial knitting machines to construct knitted objects.
- A compiler that can transform the former representation into the latter, at whose core is a complete transfer-planning heuristic for cycles (with associated correctness proof).

2 Related Work

Our work sets out to offer knit designers and programmers a better choice of primitives to use for controlling their output device. We take inspiration from work in rendering, where the primitives have been tailored to output modality (e.g., Reyes [Cook et al. 1987] for offline rendering and OpenGL for real-time rendering), and from ongoing work in 3D printing, where the community is actively developing and refining primitives (e.g., [Vidimče et al. 2013]). Our knitting primitives have orthogonal scheduling and shaping degrees of freedom, inspired by the Halide system [Ragan-Kelley et al. 2012], in which algorithms are treated as having separate definitions and schedules.

Most prior work surrounding textile design assumes a "cut-and-sew" approach, where garments are made from flat sheets of fabric, cut by humans or machines, and sewn by humans. This area is well-covered [Liu et al. 2010], with commercial systems widely available [CLO Virtual Fashion Inc. 2010], and active research supporting sketched input [Mori and Igarashi 2007], situated interaction [Wibowo et al. 2012], and advanced simulation during interaction [Umetani et al. 2011].

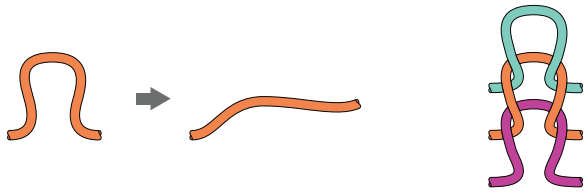
One of the great advantages of knit fabric, however, is that it need not be locally flat. This flexibility presents new specification challenges, which are not well-addressed by current tools. Knitting design systems from machine manufacturers [Stoll 2011; Shima Seiki 2011] provide detailed machine-level control languages (SINTRAL and KnitPaint, respectively) and some macro features that can be used to ease repetitive tasks (e.g., in hand-creating a library of shaped parts [Underwood 2009]), but little in the way of general high-level primitives. Third-party commercial design tools are limited to texture and color design on flat panels [Soft Byte Ltd. 1999]. In the research sphere, Knitty [Igarashi 2008] provides sketch-based design with tube primitives for hand knitting; it would be interesting, though nontrival, given the limitations of knitting machines, to retarget that system's output to our machine-knitting backend.

Recent advances have made knit simulation both tractable and predictive [Kaldor et al. 2008; Kaldor et al. 2010; Cirio et al. 2014; Cirio et al. 2015]. However, setting up initial yarn paths can be tedious. One option is to use visually reasonable (but not feasibly knittable) paths [Yuksel et al. 2012]. Our knitting assembly language provides another option: one could create an interpreter to run the language and output virtual yarn paths for a simulation system. Such an interpreter would be able to create simulation descriptions for literally anything a knitting machine could make, using the same instructions as the machine.

3 An Abstract Knitting Machine

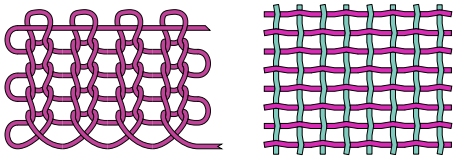
Our compiler targets a *knitting assembly language* which captures the capabilities common to industrial knitting machines, while abstracting mechanical details that may change between them. We define this language in terms of the actions of an abstract knitting machine.

Knitting machines build their output by manipulating loops of yarn. Consider these loops of yarn:



unravels stable

The orange loop on the left is not *stable* – pulling on either end of it would unravel it into a straight piece of yarn. However, the orange loop on the right is stable because it passes through and around other loops.

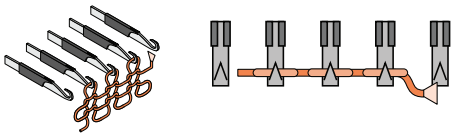


knit woven

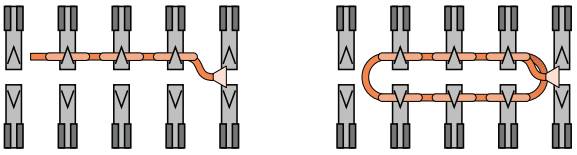
This "loops through loops" architecture is the basis of knit items. Notice how different the structure is from typical woven items, which use a "yarn over/under yarn" architecture.

3.1 Machine

Knitting machines hold loops on hooks called *needles*. These needles are arranged into rows called *beds*. Here we show an isometric and top view of a bed of five needles:



V-bed knitting machines have two beds whose needles face each-other. These are referred to as the *back bed* and *front bed*. Having two beds allows the machine to hold tubes as well as sheets.



Yarn enters the machine from a cone, passing through a tensioning device and a *yarn carrier* on its way into the knit object. Yarn carriers move laterally between the beds, positioning new yarn where it is needed. There is one yarn carrier for every yarn in use on the machine. In our diagrams, we draw yarn carriers as small triangles between the beds.

Machines create knit objects by manipulating the loops held on their needles. Needles can perform four basic operations: *tuck* adds a loop to those the needle is holding; *knit* pulls a loop through all the loops the needle is holding while releasing them; *transfer* hands all the loops a needle holds to another needle; and *split* is a combination of knit and transfer that passes a new loop through all the loops a needle is holding while moving them to another needle.

3.2 Knitting Assembly Language

We formalize the above operations as a knitting assembly language, which our compiler targets. A backend then further translates these instructions into a machine-specific format.

We begin by defining identifiers for each needle:

$$\forall i \in \mathbb{Z} : \begin{cases} b_i\text{: back bed needle} \\ f_i\text{: front bed needle} \end{cases} \tag{1}$$

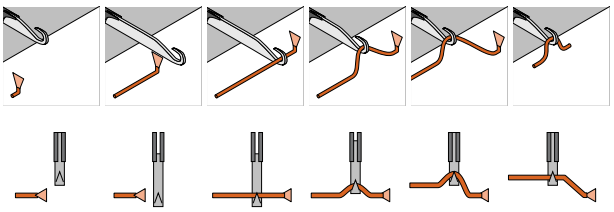
Needle indices run left-to-right along a bed, and are aligned front-to-back. So f_{-2} is aligned with b_{-2}, which is three needles to the left of b_1. Although needles often hold only one loop, they can hold several at once; thus, when talking about needle operations, we will write $n_i = [l_1, \ldots, l_t]$ to indicate that loops $l_1, \ldots, l_t$ are held by needle n_i, with l_t being closest to the tip of the needle. We will also occasionally (for notational convenience) conflate needle locations with their integer indices, writing such phrases as $f_2 - b_0 = 2$.

We endow our abstract machine with a set of active yarns $\mathcal{Y}$, which starts empty, and limit it with a maximum racking (lateral bed offset) value of R.

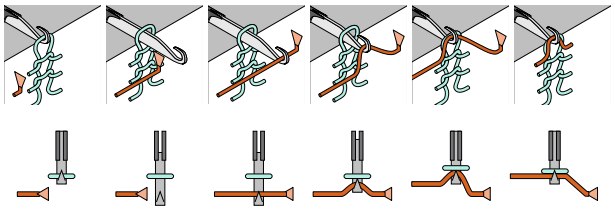
We abstract the motion of the yarn carrier by introducing a primitive to create loops: Let $\text{loop}(y, d, n)$ where $y \in \mathcal{Y}, d \in \{+, -\}, n \in \{f_i, b_i\}$ return a new loop created by passing yarn y in direction d over needle n.

Here we illustrate and define each of the four operations for a standard knitting machine:

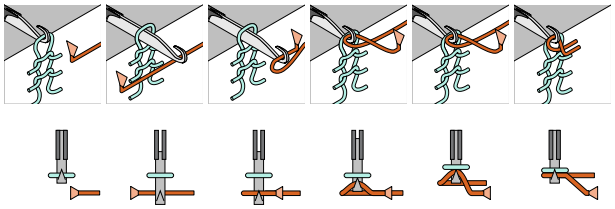
Tuck. The tuck operation adds a new loop of yarn l_{n+1} in front of the loops $[l_0 ... l_n]$ already held on a needle. Mechanically, the needle reaches forward, the yarn carrier moves to the right over the needle, and the needle retracts, now holding a new loop. We illustrate this in isometric and top views:



Tucking a needle already holding a loop stacks a new loop in front of the old loop:



The yarn carrier moved to the right over the needle in the example above, so this was a "tuck right." Tucks can be formed to the left or right, regardless of where the yarn was previously used. Here, the yarn was last used to the right of the needle, but the carrier can still be moved to the left, and then the needle tucked right:



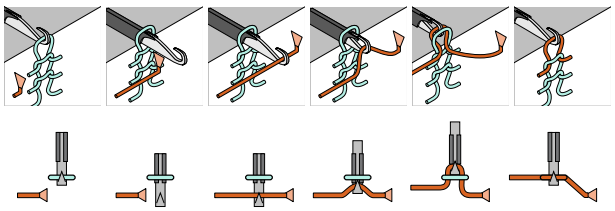
Mathematically, we define tuck as follows:

`tuck` y, d, n

> **Given:** $y \in \mathcal{Y}, d \in \{+, -\}, n \in \{f_i\} \cup \{b_i\}$
> $n \leftarrow \text{cat}(n, [\text{loop}(y, d, n)])$

Where "cat" is a function that concatenates lists.

Knit. Knitting a needle pulls a new loop of yarn through the *all of the loops* currently held by that needle. Mechanically, the needle reaches forward, the yarn carrier moves over it, and the needle retracts, using a secondary mechanical action to lift the loops that it was holding up and over the new loop and off of its tip.



Knit, like tuck, has a direction. The above example is a "knit right" because the yarn carrier moves to the right when supplying the yarn for the new loop.

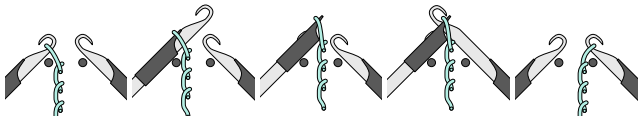
`knit` y, d, n

> **Given:** $y \in \mathcal{Y}, d \in \{+, -\}, n \in \{f_i\} \cup \{b_i\}, n \neq []$
> $l \leftarrow \text{loop}(y, d, n)$
> $\text{pull}(l, \text{reverse}(n))$
> $n \leftarrow [l]$

Where "pull" means to pull a loop through a list of other loops; and "reverse" reverses the order of a list.

Transfer. The transfer operation moves all the loops on a needle to the needle across from it. That is, it moves loops from the front bed

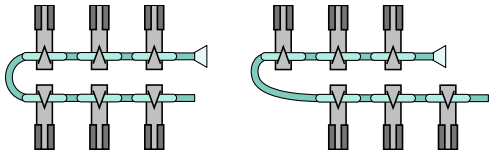
to the back bed or visa versa.



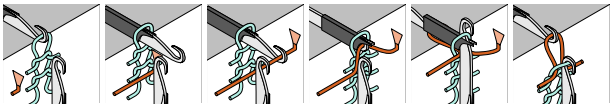
`xfer` n, n'

> **Given:** $(n, n') \in \{(f_i, b_j), (b_i, f_j)\}, |i - j| \leq R$
> $n' \leftarrow \text{cat}(n', \text{reverse}(n))$
> $n \leftarrow []$

This restriction of only moving between aligned needles may seem severe, but machines can *rack* (laterally move) the beds to change which needles are aligned. By convention, we take racking values as the offset of the back bed:



Split. The *split* operation combines knit and transfer into one operation. Split is useful because it allows the machine to knit through a loop without losing the ability to access the loop in the future:



Like knit, split has a direction. The above is a split right.

`split` y, d, n, n'

> **Given:** $y \in \mathcal{Y}, d \in \{+, -\}, n \neq []$,
> $(n, n') \in \{(f_i, b_j), (b_i, f_j)\}, |i - j| \leq R$,
> $l \leftarrow \text{loop}(y, d, n)$
> $\text{pull}(l, \text{reverse}(n))$
> $n' \leftarrow \text{cat}(n', \text{reverse}(n))$
> $n \leftarrow [l]$

Finally, we introduce three utility instructions that are important for yarn management and finishing:

Drop. The instruction `drop` causes a needle to drop the loops it is carrying. Mechanically, this is `knit` with no yarn:

`drop` n

> **Given:** $n \in f_i, b_i, f_i^h, b_i^h, n \neq []$
> $n \leftarrow []$

In, Out. The instructions `in` and `out` add and remove active yarns. When a yarn is removed, the connection between it and its last stitch is broken.

`in` y

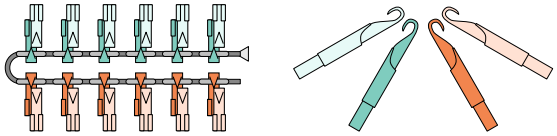
> **Given:** $y \notin \mathcal{Y}$, y is a yarn
> $\mathcal{Y} \leftarrow \mathcal{Y} \cup \{y\}$

`out` y

> **Given:** $y \in \mathcal{Y}$
> $\mathcal{Y} \leftarrow \mathcal{Y} \setminus \{y\}$

3.3 A Slight Extension

For convenience, we describe our algorithms in terms of a more advanced form of v-bed knitting machine called an x-bed machine. This machine adds an extra mechanical element (called a *holding hook*) above every needle in the front and back beds. These holding hooks can hold loops, but cannot knit or tuck. If any loops are held on the holding hook associated with a needle, that needle cannot be used to perform an operation, as the held loops will block it.



For an x-bed machine, we need to add two new types of hooks to the array of needles:

$$\forall i \in \mathbb{Z} : \begin{cases} b_i & \text{: back needle} \\ b_i^h & \text{: back holding hook} \\ f_i^h & \text{: front holding hook} \\ f_i & \text{: front needle} \end{cases} \tag{2}$$

Finally, because the use of the holding hook associated with a needle limits the use of that needle, we need to make it a condition of any knit, tuck, transfer, or split operation that all of the needles involved are *clear*. We say a needle n is clear if it is a holding hook or if its associated holding hook is empty:

$$\text{clear}(n) \equiv \exists i : \begin{array}{ll} & n = f_i^h \vee n = b_i^h \\ \vee & (n = f_i \wedge f_i^h = []) \\ \vee & (n = b_i \wedge b_i^h = []) \end{array} \tag{3}$$

3.4 What Knitting Machines Can Make

We now show how the operations described in the previous section can be used to make and deform 3D shapes. These techniques are the core of the generalized shape primitives supported by our compiler.

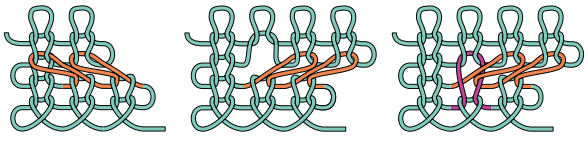


Figure 2: *Increase-decrease shaping can change primitive width.*

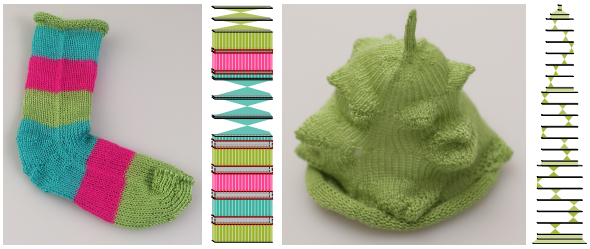


Figure 3: *Partially-knit rows can be used to bend shapes, as in the heel of this sock, and create bulges, as in this whimsical hat.*

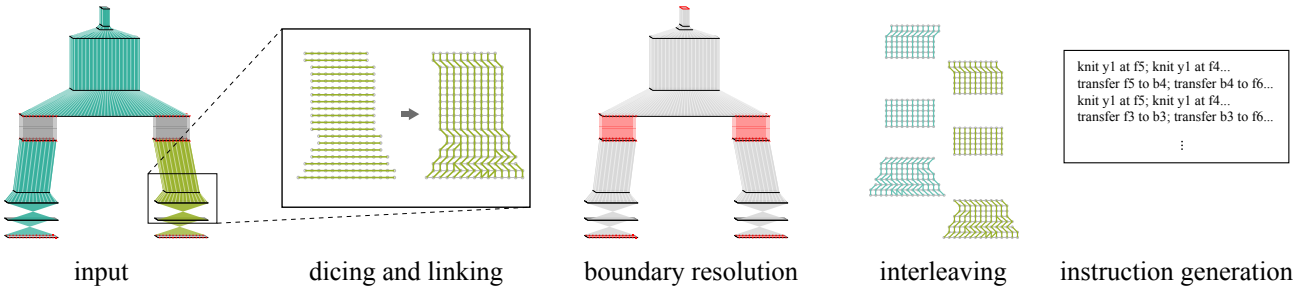


Figure 4: *Our compiler pipeline. Our compiler first dices each of its input primitives into courses, assigns short rows to be knit between their adjacent courses, and decides how to link stitches in adjacent courses together. Next, during the boundary resolution stage, it decides how to start and end each primitive. Finally, it interleaves the knitting and linking steps required for all the primitives into a final ordering, and generates knitting assembly language instructions for them.*

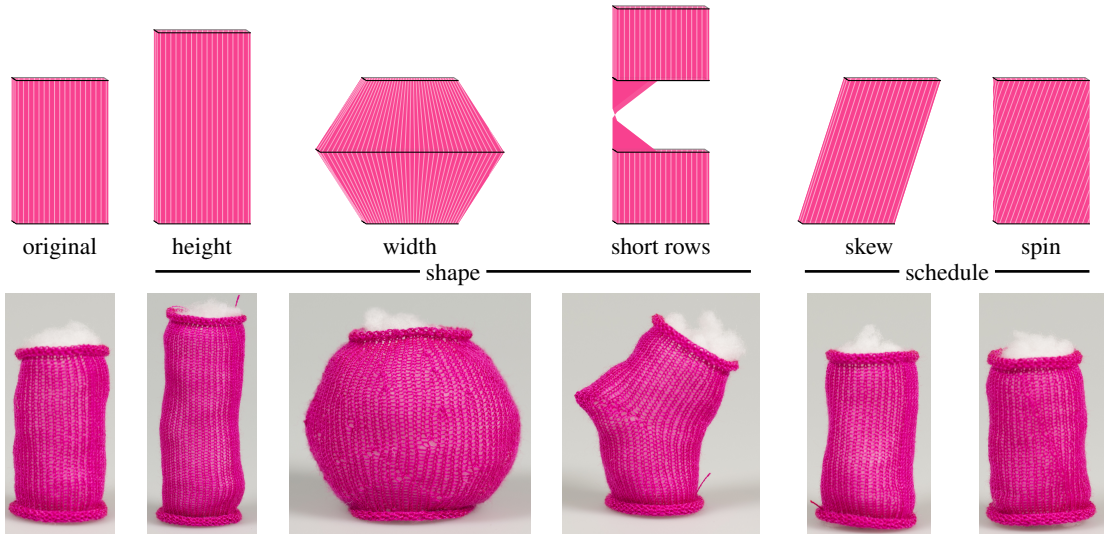


Figure 5: *The degrees of freedom of a tube in our input format. Schedule parameters do not change the final shape.*

Sheets can be created by knitting back and forth over a needle range; e.g., knitting right on $b_{-5}, \ldots, b_5$, knitting left on $b_5, \ldots, b_{-5}$, and repeating five times will create a sheet 11 stitches wide and 10 *courses* (rows of knitting) tall. Tubes can be created by knitting in a consistent direction around a circle of needles; e.g., knitting right on $b_1, \ldots, b_{10}$, knitting left on $f_{10}, \ldots, f_1$, and repeating 12 times will create a tube 20 stitches in circumference and 12 courses tall. These shapes can be further modified with *shaping* operations.

Increases and decreases (Figure 2) can be used to adjust the number of stitches in a course, allowing tubes (sheets) of continuously varying circumference (width). The width of a course of stitches can be *decreased* by moving a stitch onto the same needle as its neighbor, and knitting both of them with a single stitch in the next row. Conversely, the width of a course can be *increased* by moving stitches apart and filling this gap with a tuck in the next course. A visually more pleasing increase can be performed by using a split instead of a tuck.

Shapes can also be modified by knitting partial courses, called *short rows*. These short rows have the effect of pushing their adjacent courses apart, creating a bend or bulge in the fabric (Figure 3). An example is the traditional shaping of a sock, where extra rows are added to create the rounded part of the heel.

4 Compiler Pipeline

Our compiler transforms high-level primitives into knitting assembly language commands through a series of stages illustrated in Figure 4. Given a group of input primitives to knit, our compiler first dices each primitive into courses (assigning short rows to be knit at the same time as their adjacent course), and then determines how they should be linked using stretch-limited dynamic time-warping. Next, during boundary resolution, the compiler selects operations to perform at the start and end of each primitive, based on definitions specified in the input and the needle occupancy of the diced primitives. Finally, linking and knitting steps are interleaved based on their construction time, with stash/unstash blocks added as needed to ensure that no yarn is stressed during linking, and knitting assembly language instructions are emitted.

4.1 Input

Most objects created on a knitting machine, such as gloves, sweaters, stuffed toys, and socks, are assemblies of tubes and sheets, with varying radii and bends. Motivated by this observation, we designed our compiler to take as input a list of tube and sheet primitives, positioned according to their construction time and location on the knitting machine bed (e.g., Figure 1 and 3). This layout makes it easy to ensure that no two primitives require the same nee-

Figure 6: *In our compiler's input, each primitive has a start and end boundary definition that indicates how to stabilize loops. These can result in primitives that are open, closed, or attached together in various ways. Not shown are front, front (closed), and front-to-back gluing, which are defined analogously to their back variants.*

dles at the same time, because this would result in an overlap in the input.

Primitives have both shaping parameters, which influence their final appearance, and scheduling parameters, which change which needles are used to construct them (Figure 5). For tubes, the shaping parameters are: **height**, the number of courses; **circumference**, the number of stitches in each course; and **short rows**, partial courses that will cause the tube to bend. The scheduling parameters map each course to the needle bed: **time** indicates when the first course will be knit; **skew** gives the horizontal position of each course; and **spin** rotates courses on the bed. In our compiler's input file format, all parameters except for **height** and **time** are linearly interpolated along the height of the tube.

Sheets are specified relative to a "supporting tube" and have one additional shape parameter: the **percent** of the supporting tube's circumference occupied by the sheet. For example, a sheet at 100% is a tube with a slit in it (useful as a thumb slit when making a hand warmer, Figure 14).

All dimensions are specified in (possibly fractional) numbers of stitches, that, given a yarn, can be converted to real-world lengths. Thus, porting a design from one yarn to another requires only that one multiply all dimensions by a constant factor determined by the yarns' relative stitch size.

4.2 Dicing

After reading input primitives, our compiler breaks them into horizontal slices (courses). Computing these needle lists from the supplied input parameters is straightforward rasterization. Each course has an abstract "knitting time" value, a list (in counterclockwise order) of needles where loops will be formed, and a parameter value for every needle (running from 0 to 1 counterclockwise around the cycle) that will be used in linking. Short rows are treated as part of their closest course.

4.3 Linking

Once a primitive has been transformed into courses, links between adjacent courses are made. These links will be used to generate transfer instructions later. Linking is accomplished by selecting an optimal combination of 1-1, 2-1 (decrease), and 1-2 (increase) links between stitches in adjacent courses, in a process akin to stretch-limited dynamic time warping. The objective function to be minimized is the sum of absolute differences between the parameter value at each stitch and the parameter value at its assigned location, plus a large additional penalty for every increase or decrease. This additional penalty prevents the optimization from arbitrarily introducing paired increases and decreases.

Using a parameter value, rather than simply linking stitches that are closest on the bed, is the key to separating scheduling (needle location) and shape (stitch connectivity). Without using parameter values to accomplish linking, it would be impossible to spin or skew primitives on the bed without also distorting them.

Figure 7: *Primitive scheduling (particularly, spin – the orientation on the bed) is important at boundaries. This tube has closed boundaries at the top and bottom, but has had its "spin" scheduling parameter adjusted at the top boundary.*

4.4 Boundary Resolution

Once all the primitives have been diced and linked, their first and last courses are compared in a stage we call boundary resolution. As we have belabored, knit items are created by pulling loops through loops. Any loop not pulled through another loop could unravel, causing the final item to fall apart. Thus the starting and ending loops of each primitive must be made stable by being pulled through or pulling through another loop. This stabilization can be accomplished by attaching the loops to loops from another primitive, resulting in the primitives being attached ("glued"); alternatively, local techniques can be used to stably create ("cast on") or stabilize then drop ("bind off") loops.

Each input primitive includes a *boundary definition* that describes which of these actions to take at its start and end. Our compiler supports various binding and gluing styles, illustrated in Figure 6. Boundaries are the one place in the primitive definition where the scheduling parameters, specifically, spin, can influence structure, as shown in Figure 7.

4.5 Interleaving and Instruction Generation

Once the boundaries have been marked, courses and links between courses are sorted based on their knitting time. The knitting time for links is set to halfway between their adjacent courses. The compiler walks through this sorted list, tracking which needles are currently in use and emitting instructions.

For links, the compiler calls on our transfer planning algorithm (Section 5) to generate a series of transfer instructions to enact the link. If the generated plan involves any large racking, it may stretch or break *other* primitives currently held on the bed (Figure 8). To avoid this, the compiler will prefix the transfer instructions from the link with what we call a *stash* operation, which moves every primitive currently attached to two beds onto one bed, using the holding hooks.

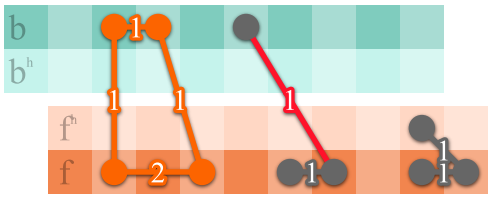


Figure 8: *As part of moving a target cycle (left), our transfer planning algorithm may generate rackings that stress other primitives (center). In this case, our compiler will "stash" the other primitives on one bed by using the holding hooks (right).*

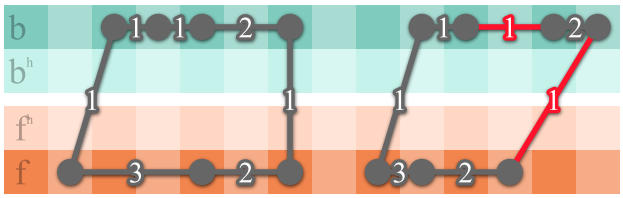


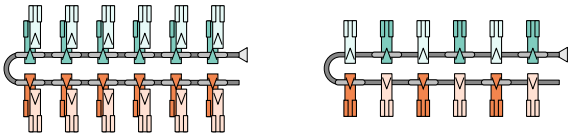
Figure 9: *Cycles and slack. When drawing a cycle, we typeset slack as labels on edges. Both cycles above have the same slack, but the one on the left respects that slack, while the one on the right places some stitches too far apart (red edges).*

For courses, the compiler first emits transfer instructions to unstash any required stitches that were previously stored on holding hooks, and then walks through the course, emitting a knit instruction for every needle in the course. In the case of needles that do not currently contain a loop because of the previous link, our compiler follows the common knitting practice of using a split or tuck instruction to fill the gap. If the course is a boundary, instructions are emitted to enact the requested binding or glue operations.

4.6 Backend

We have developed a translator from our knit assembly language to the '.dat' format used by Shima Seiki's knit specification system, KnitPaint. We use KnitPaint to translate these files into Shima Seiki's proprietary machine control format.

Though our compiler assumes an x-bed machine, we only have access to a v-bed machine (no holding hooks); thus, as part of this translation process, our compiler transforms x-bed instructions into *half-hauge* v-bed instructions, where alternate needles are used to emulate holding hooks:



Our translator also handles low-level tasks such as inferring yarn carrier motion from knitting instructions, unifying instructions into "passes", generating machine-specific boilerplate for yarn insertion and removal, setting racking flags appropriately, setting yarn tensions, and performing appropriate cast-on and bind-off stitches.

Given that knitting assembly language operations all correspond to generally-available machine capabilities, there is no technical reason that a similar program could not be developed to output, e.g., Stoll's SINTRAL machine control language.

5 Transfer Planning

In order to support general links between courses, our compiler needs a way of generating transfer instructions to move collections of stitches around on the needle beds. During knitting, our primitives occupy needles arranged in *cycles* (counterclockwise loops), constrained to obey a given *slack* (distance between stitches).

Definition 1 (Cycle)**.** Define a *cycle* to be a collection of needles $\mathbf{c} = [c_1, \ldots, c_n]$ that, when connected in index order, form a non-self-intersecting loop on the needle bed. When dealing with cycles, we will use cyclic indexing (so we define $c_{i+xn} \equiv c_i$ for integers x).

In our case, cycles are commonly the ends of tubes that have been previously knit. It is important not to excessively stretch the yarn between adjacent stitches in the cycle, so we introduce the notion of *slack* (Figure 9):

Definition 2 (Slack)**.** Define *slack* to be a list of non-negative integers $\mathbf{s} = [s_1, \ldots, s_n]$ that indicates the greatest allowed length between subsequent locations in a (cyclic) list of needles. We say a list of needles $\mathbf{a}$ *respects* slack $\mathbf{s}$ if $\forall i : |a_{i+1} - a_i| \leq s_i$ (notice that, by cyclic indexing convention, s_n is the slack between the first and last needle).

With these definitions in hand, we may finally define the transfer planning problem:

Definition 3. An instance of the **transfer planning problem** consists of a slack $\mathbf{s}$, counterclockwise cycles of needles $\mathbf{n}$ (the source) and $\mathbf{n}'$ (the target), a maximum racking $R \geq 1$, and a free range $[min, max]$ such that $|\mathbf{s}| = |\mathbf{n}| = |\mathbf{n}'|$, both $\mathbf{n}$ and $\mathbf{n}'$ respect slack $\mathbf{s}$, no needles are split (i.e. $n_i = n_j \implies n'_i = n'_j$), slack is at least one between all needles, and all needles in the source and target are contained in $[min + 1, max - 1]$.

A list of transfers is a **solution to this instance** if it transforms $\mathbf{n}$ to $\mathbf{n}'$, requires no racking greater than R, and all intermediate poses lie in free range $[min, max]$ and respect slack $\mathbf{s}$.

Our compiler is based on the first – to our knowledge – general solution to this problem. Our solution is heuristic but complete; that is, it can solve all instances of the problem, but it may use more transfers than necessary. We define the algorithm here and sketch a proof of completeness, but a full treatment is left to the supplementary material.

The outline of our algorithm is as follows: we define a penalty function that takes positive integer values for all bed configurations except the target configuration $\mathbf{n}'$ (at which it is zero) along with a type of transformation (collapse/expand); our transfer planning algorithm will compute two locally optimal collapse/expand transformations at each step and choose whichever decreases the penalty the most.

In the supplementary material, we provide a proof that for any counterclockwise cycle that respects $\mathbf{s}$, there exists collapse and expand phases that could be chosen by our algorithm that will reduce its associated penalty function. Given that the penalty function is integer-valued, and that our algorithm always choose a collapse and expand with maximum penalty reduction, this argument suffices to prove completeness.

5.1 The Roll-Goal Penalty

For transfer planning, we need a penalty function that measures the distance from the current configuration to $\mathbf{n}$, does not have any local minima, and can be computed as a sum over penalties at each

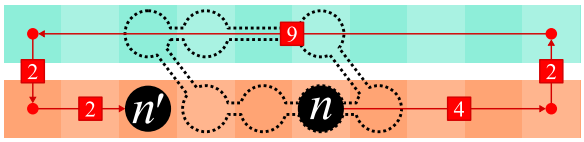
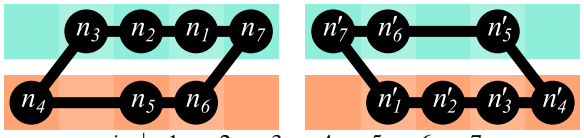


Figure 10: *Visualizing the penalty* $\mathrm{p}(n, 2, n') = 19$ *computed for stitch* n *with goal* n' *and roll number* $+2$*. The penalty is computed by walking around free needle range in the direction indicated by the roll number, charging* 1 *for every needle traversed and* 2 *for every change of bed.*



i	1	2	3	4	5	6	7
δ_i	0	0	1	0	0	1	0
δ'_i	0	0	0	1	0	0	1
w_i	1	1	1	0	1	1	0
r_i	-1	-1	-1	0	1	1	0

Figure 11: *Winding* (w_i) *and roll* (r_i) *numbers are determined by where cycles* $\mathbf{n}$ *and* $\mathbf{n}'$ *cross between the front and back bed.*

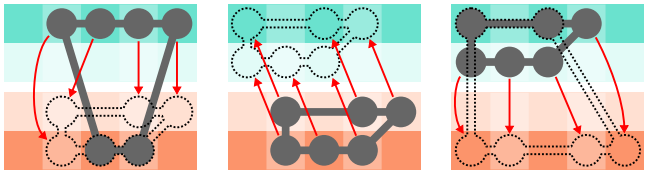


Figure 12: *During a collapse-expand transform, back-bed stitches are* collapsed *to the front bed, all stitches are moved to the back bed, then the cycle is* expanded *by moving some stitches from the back bed to the front bed.*

needle. (This last condition makes computing the locally optimal behavior with dynamic programming feasible.)

We decided to use a penalty involving the distance to the goal needle measured *around* the bed. That is, for each stitch n_i in the cycle, we assign a roll number r_i indicating the number of times that stitch should switch beds to the left (if $r_i < 0$) or right (if $r_i > 0$) of the other stitches.

Given an assignment of roll numbers, it is straightforward to define a penalty

$$\mathrm{penalty}(\mathbf{n}, \mathbf{r}, \mathbf{n}') \equiv \sum_i \mathrm{p}(n_i, r_i, n'_i) \tag{4}$$

where

$$\mathrm{p}(n, r, n') \equiv \begin{cases} |n' - n| & \text{if } r = 0 \\ 2 + (n - min) + \mathrm{p}(min, -(r+1), n') & \text{if } r < 0 \\ 2 + (max - n) + \mathrm{p}(max, -(r-1), n') & \text{if } r > 0 \end{cases} \tag{5}$$

We illustrate the individual-stitch penalty p, which measures distance "around" the free needle range, in Figure 10.

It remains to assign roll numbers. We will define these in terms of winding numbers w_i, whose values will indicate how many times a stitch should be rolled counterclockwise.

Defining δ_i to be the number of times the cycle switches beds in counterclockwise order between n_i and n_{i+1}, and δ'_i analogously for $\mathbf{n}'$, it must be the case that

$$w_i - \delta_i + \delta'_i = w_{i+1} \tag{6}$$

This equation determines w_i, up to global addition of an even number. A proof is included in the supplemental material.

The conversion between roll and winding numbers is straightforward:

$$r_i = \begin{cases} -w_i & \text{if } n_i \text{ is on the back bed} \\ w_i & \text{if } n_i \text{ is on the front bed} \end{cases} \tag{7}$$

An illustration of winding number assignment is shown in Figure 11.

5.2 Collapse-Expand

Our transfer planning algorithm uses a sequence of collapse-expand transformations. Each collapse-expand transformation (Figure 12) collapses the cycle onto one bed, moves it to the other bed, and expands it back to both beds, with each motion providing an opportunity to reposition stitches in order to reduce the overall penalty.

In the following description and accompanying figures, we show a collapse-expand from the back bed to the front bed; however, it is equally reasonable to switch the roles of the beds, and our overall planning algorithm considers both options.

The collapse phase will, at each step, either transfer the leftmost or rightmost stitch remaining on the back bed to the hooks or needles of the front bed. This ordering ensures that there are at most two places where the slack between adjacent stitches must be respected.

After the collapse phase, the cycle is moved from the front bed to the back bed to allow the expand phase to access the front stitches. The algorithm will shift the whole cycle by a constant offset if doing so decreases the penalty function while not moving any stitches outside the free needle range.

The expand phase moves the stitches on the holding hooks of the back bed to the front bed. Expand phases used by our algorithm can start with any stitch, but proceed by transferring only stitches adjacent to those already transferred. This constraint again ensures that there are at most two places that must be checked for valid slack.

An optimal collapse or expand for a given state may be computed with a memoized recursive function in $O(|\mathbf{n}|^2 R^2)$ time. This complexity bound arises from the fact that the actions available during a collapse or expand depend only on the indices of the next-to-be-transferred stitches and the locations of the previous stitches.

Our planner proceeds by repeatedly choosing an optimal collapse followed by an optimal expand; one such pair is shown in Figure 13. It is worth noting that this is not the same as an optimal collapse-expand *pair*; however, choosing separately avoids the prohibitively high complexity of a full search, and it is sufficient for correctness (see supplemental material).

5.3 Final Algorithm

Our final transfer planning algorithm (Algorithm 1) first assigns roll values to every stitch and then iteratively makes a greedy choice of collapsing/expanding to the back or front bed.

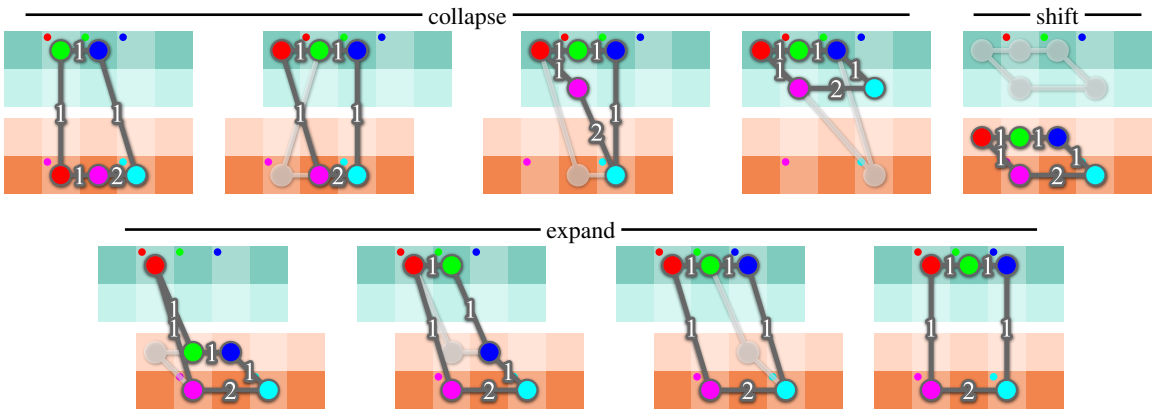


Figure 13: *A transfer plan generated by our algorithm. Solid colored circles indicate goals. First,* **top***, the collapse phase moves stitches to the needles and hooks of the back bed. Next,* **top right***, the collapsed cycle is moved to the front bed. Finally,* **bottom***, the expand phase moves stitches from the front bed back to the back bed.*

Algorithm 1 Transfer planning outer loop

procedure TRANSFERPLAN($\mathbf{n}, \mathbf{n}', \mathbf{s}$)
  $\mathbf{c} \leftarrow \mathbf{n}$
  $\mathbf{r} \leftarrow \text{AssignRoll}(\mathbf{n}, \mathbf{n}')$
  $T \leftarrow []$
  while $\text{penalty}(\mathbf{c}, \mathbf{r}, \mathbf{n}) > 0$ **do**
    $(T_\text{f}, \mathbf{c}'_\text{f}, \mathbf{r}'_\text{f}) \leftarrow \text{CollapseExpand}(\textbf{Front}, \mathbf{c}, \mathbf{r}, \mathbf{n}', \mathbf{s})$
    $(T_\text{b}, \mathbf{c}'_\text{b}, \mathbf{r}'_\text{b}) \leftarrow \text{CollapseExpand}(\textbf{Back}, \mathbf{c}, \mathbf{r}, \mathbf{n}', \mathbf{s})$
    if $\text{penalty}(\mathbf{c}'_\text{f}, \mathbf{r}'_\text{f}, \mathbf{n}') \leq \text{penalty}(\mathbf{c}'_\text{b}, \mathbf{r}'_\text{b}, \mathbf{n}')$ **then**
      $(\mathbf{c}, \mathbf{r}) \leftarrow (\mathbf{c}'_\text{f}, \mathbf{r}'_\text{f})$
      $T \leftarrow \text{cat}(T, T_\text{f})$
    else
      $(\mathbf{c}, \mathbf{r}) \leftarrow (\mathbf{c}'_\text{b}, \mathbf{r}'_\text{b})$
      $T \leftarrow \text{cat}(T, T_\text{b})$
    end if
  end while
  return T
end procedure

6 Results

All knit objects in the paper were created with our compiler and knit on a Shima Seiki SWG091N2 15-gauge v-bed knitting machine. This is a two-bed machine with 15 needles per inch and a 91cm long needle bed. It has a maximum racking value of eight needles and includes ten yarn carriers. We knit our designs in half-gauge (Section 4.6), as this machine is not an x-bed machine. Knitting at half gauge results in a somewhat loose knit, because the 15-gauge needles cannot hold yarns thick enough to produce a dense 7-gauge knit.

The machine is able to use a wide variety of yarns, though for this paper our examples were knit in acrylic ("Supersheen 1-ply" from Yeoman Yarns) and merino wool ("Polo 1-ply" from Yeoman Yarns). Note that Yeoman uses "ply" in the UK sense: as a size designation that does not reflect the number of component yarns plied together to form these yarns. On our machine at half gauge, these yarns both produce stitches that are approximately 3.04mm wide by 1.58mm tall; values we obtained by knitting a tube, stuffing it, and computing the circumference and height of a known stitch-count portion.

Compile time is generally trivial compared to knitting time, with the most expensive steps of compilation being the linking (time warping) and transfer planning stages. This observation is unsurprising; both are worst-case cubic in the number of stitches.

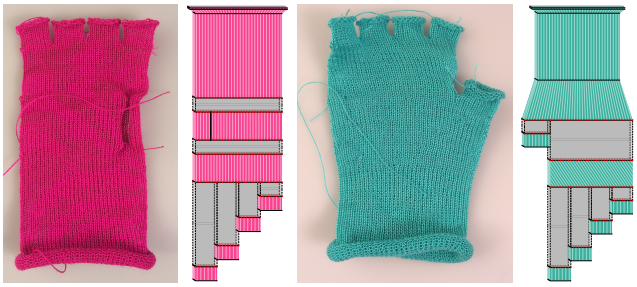


Figure 14: *Two hand-warmers designed in our system. One uses a sheet to create a slit for the thumb, while the other uses another tube for the thumb then decreases the width of the main tube to fit the wrist.*

We used our system to create clothing objects for plush toys (Figure 1) and people (Figure 14). Having primitives with easy-to-adjust size values is important in both cases. Our sock example (Figure 3) is far too small to be worn, but does demonstrate how gluing together tubes knit with different yarns can make colorful stripes.

We also created a number of knit toys. Our snake toy (Figure 15) shows how easy it is to build higher level primitives – its body is the result of writing a script to translate a helix into a tube with short rows. The Hilbert curve (Figure 17) was generated by a similar script. Our collection of robots (Figure 16) demonstrates the benefit of high-level primitives for rapid iteration. Finally, the teapot example (Figure 18) makes heavy use of skew operations to schedule its spout and handle; indeed, some of the stitch motions in this example are so long that they do not consistently knit; in the version we show, several dropped stitches were manually corrected.

6.1 Graphical Editing

We developed a graphical interface to edit our compiler's input format. The user interface (Figure 19) consists of linked preview and bed views. The preview view shows a rough approximation of the final 3D shape of the model, computed in an exceedingly ad-hoc way using as-rigid-as-possible alignment of tube primitives. The

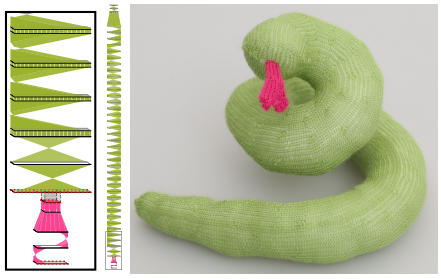


Figure 15: *The helical shape of this snake is the result of many sets of short rows.*

Figure 16: *These plush robots are all variations on a design, created rapidly by editing high-level primitives to be smaller, with a chunkier torso and claws, and in a seated posture.*

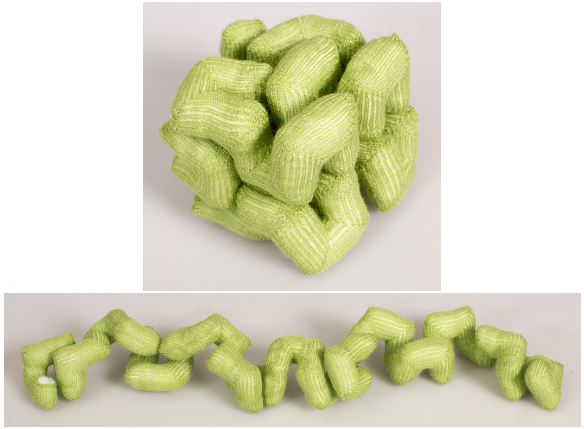


Figure 17: *Two views of a 3D Hilbert curve of order two, generated by using a small script to write an input file for our compiler.*

Figure 18: *This teapot makes extensive use of the "skew" scheduling primitive.*

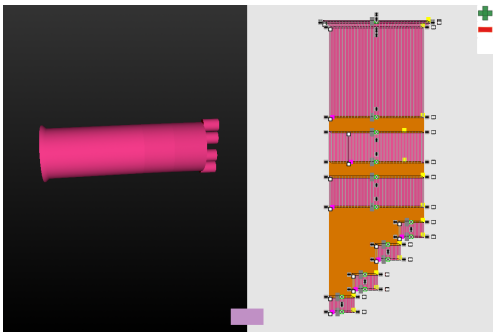


Figure 19: *The 3D preview (left) and 2D bed view (right) of our interface. The displayed object is the left hand-warmer from Figure 14.*

bed view shows the input to the compiler, and provides basic vector-graphics-style editing capabilities for primitive positioning and sizing, along with some special tools to handle the spin and skew degrees of freedom, adjust the width of sheets, and set boundary definitions. As a convenience, the interface also loads a description of the color information of our yarns, and can use this to set the color of the shape primitives.

7 Discussion

In this paper, we introduced a compiler which can translate high-level shape primitives into low-level knitting instructions. The core of our complier is a transfer planning algorithm for knit cycles, which is *provably correct* on all inputs, though it may not produce time-optimal plans. In addition, we presented a formal, general treatment of the basic operations of a knitting machine; this "knitting assembly language" could be further refined into a general file format for describing knitting – both for knitting machines, and perhaps even for rendering and simulation algorithms.

The focus of our compiler is to make it easier to create knit *structures*; however, many knit objects also use various combinations of stitches to create interesting surface *textures*. In the future, we plan to extend our compiler to support surface texturing, by adding local texturing programs akin to fragment shaders. These shaders will be constrained to not change global structure, a process which is complicated by the prevalence in knitting of mid-scale structures (e.g., "cables") which use the local movement of small blocks of stitches to create texture.

We believe that the knit fabrication community can benefit from a uniform control language akin to our knitting assembly language, and it is part of our ongoing research to port the language to any knitting machines that we are able to access. Such a language also allows for uniform structural debugging tools and error checking (e.g., "you stretched that loop out a long way; it might break"). It would also be interesting to create a knitting assembly language backend that could set up yarns for simulation (e.g., by creating a stitch mesh [Yuksel et al. 2012]); this backend would allow physical and virtual garments to be created using the same code.

Our pipeline presents many opportunities for small refinements: during dicing, it would be interesting to try more sophisticated rasterization techniques (possibly involving "hinting" akin to that used in font rendering); linking could be user-controlled for interesting shaping effects; transfer plans could be optimized further, possibly by using limited lookahead in the collapse-expand planning space; and, with some reverse engineering work, the backend could write machine control files directly.

Our transfer planning algorithm cannot work with general (i.e., non-cyclic) slack constraints. Indeed, transfer plans do not always *exist* in the presence of general yarn constraints – consider a cycle with an additional edge linking the front and back beds at its center; this cycle cannot be rotated 180 degrees even though both the starting and ending positions are valid. It would be interesting to attempt to characterize the space of feasible transfer planning algorithms.

Our compiler uses unoptimized implementations of transfer planning and time-warping, both of which can approach $O(s^3)$ for pessimal inputs (time-warping runs an $O(s^2)$ alignment at $O(s)$ possible offsets; transfer planning takes $O(s^2)$ per step, and may require $O(s)$ steps to complete a plan). Both of these could likely be sped up by an order of magnitude by using early-out checks (e.g., the collapse phase could terminate as soon as it knows it does not have space to place all remaining stitches; the time warping phase could not consider paths that cannot possibly align based on the slope constraints), and, further, are independent per-course tasks, so could be distributed across threads relatively easily.

The knitting machine we used for output, like many machines, actuates its needles using a cam system that slides along the beds. This design means that, in practice, it takes basically the same amount of time for the machine to operate on *any* number of needles, as long as those operations can all be performed during one *pass* of the cam system. (In addition, some operations can only be performed when the pass is being made in a specific direction.) A production-level system would almost certainly want to carefully track machine state for the current target machine and attempt to use production time when breaking ties between otherwise equivalent actions.

Right now, consumer-level knitting machines lack the sophisticated transfer capabilities of industrial machines – indeed, the basic home knitting machine has not changed in hardware capability since the 1980s. It might be interesting to come up with optimization passes for our compiler that could make it as easy as possible to construct objects on these severely restricted machines. There is also a vibrant community developing new technology for these machines [OpenKnit 2014; Guljajeva and Canet 2012; All Yarns Are Beautiful 2014], so perhaps more automated home knitting is only a few Kickstarters away. Until that time, having a common instruction format (e.g., our knitting assembly language) could make it much easier to send knit jobs to a central location for industrial machine processing.

We believe that 3D machine knitting should join 3D printing in the pantheon of end-user-accessible additive fabrication, and that getting it there will require new tools, algorithms, and data exchange formats, of which our compiler, transfer planning algorithm, and knitting assembly languages are first examples.

References

ALL YARNS ARE BEAUTIFUL, 2014. Ayab - all yarns are beautiful. [Online]. Available from: http://ayab-knitting.com/index_en.html#features.

CIRIO, G., LOPEZ-MORENO, J., MIRAUT, D., AND OTADUY, M. A. 2014. Yarn-level simulation of woven cloth. *ACM Trans. Graph. 33*, 6 (Nov.), 207:1–207:11.

CIRIO, G., LOPEZ-MORENO, J., AND OTADUY, M. A. 2015. Efficient simulation of knitted cloth using persistent contacts. In *Proceedings of the 14th ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 55–61.

CLO VIRTUAL FASHION INC., 2010. Marvelous designer. http://marvelousdesigner.com.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The reyes image rendering architecture. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, 95–102.

GULJAJEVA, V., AND CANET, M., 2012. Knitic open hardware knitting machine. [Online]. Available from: http://www.knitic.com.

IGARASHI, Y. 2008. Knitty: 3d modeling of knitted animals with a production assistant interface. In *Eurographics 2008 Annex to the Conference Proceedings*.

KALDOR, J. M., JAMES, D. L., AND MARSCHNER, S. 2008. Simulating knitted cloth at the yarn level. *ACM Trans. Graph. 27*, 3 (Aug.), 65:1–65:9.

KALDOR, J. M., JAMES, D. L., AND MARSCHNER, S. 2010. Efficient yarn-based cloth with adaptive contact linearization. *ACM Trans. Graph. 29*, 4 (July), 105:1–105:10.

LIU, Y.-J., ZHANG, D.-L., AND YUEN, M. M.-F. 2010. A survey on CAD methods in 3D garment design. *Computers in Industry 61*, 6, 576–593.

MORI, Y., AND IGARASHI, T. 2007. Plushie: An interactive design system for plush toys. *ACM Trans. Graph. 26*, 3 (July).

OPENKNIT, 2014. Openknit: open source digital knitting. [Online]. Available from: http://www.openknit.org.

RAGAN-KELLEY, J., ADAMS, A., PARIS, S., LEVOY, M., AMARASINGHE, S., AND DURAND, F. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph. 31*, 4 (July), 32:1–32:12.

SHIMA SEIKI, 2011. Sds-one apex3. [Online]. Available from: http://www.shimaseiki.com/product/design/sdsone_apex/flat/.

SOFT BYTE LTD., 1999. Designaknit. [Online]. Available from: https://www.softbyte.co.uk/designaknit.htm.

STOLL, 2011. M1plus pattern software. [Online]. Available from: http://www.stoll.com/stoll_software_solutions_en_4/pattern_software_m1plus/3_1.

UMETANI, N., KAUFMAN, D. M., IGARASHI, T., AND GRINSPUN, E. 2011. Sensitive couture for interactive garment modeling and editing. *ACM Trans. Graph. 30*, 4 (July), 90:1–90:12.

UNDERWOOD, J. 2009. *The design of 3D shape knitted preforms*. PhD thesis, Fashion and Textiles, RMIT University.

VIDIMČE, K., WANG, S.-P., RAGAN-KELLEY, J., AND MATUSIK, W. 2013. Openfab: A programmable pipeline for multi-material fabrication. *ACM Trans. Graph. 32*, 4 (July), 136:1–136:12.

WIBOWO, A., SAKAMOTO, D., MITANI, J., AND IGARASHI, T. 2012. Dressup: A 3d interface for clothing design with a physical mannequin. In *The 6th International Conference on Tangible, Embedded and Embodied Interaction (TEI 2012)*, 99–102.

YUKSEL, C., KALDOR, J. M., JAMES, D. L., AND MARSCHNER, S. 2012. Stitch meshes for modeling knitted clothing with yarn-level detail. *ACM Trans. Graph. 31*, 4 (July), 37:1–37:12.