KnitPicking Textures: Programming and Modifying Complex Knitted Textures for Machine and Hand Knitting

Megan Hofmann
Carnegie Mellon University
Pittsburgh, PA

Lea Albaugh
Carnegie Mellon University
Pittsburgh, PA

Ticha Sethapakadi
Massachusetts Institute of Technology Cambridge, MA

Jessica Hodgins
Carnegie Mellon University
Pittsburgh, PA

Scott E. Hudson
Carnegie Mellon University
Pittsburgh, PA

James McCann
Carnegie Mellon University
Pittsburgh, PA

Jennifer Mankoff
University of Washington
Seattle, WA

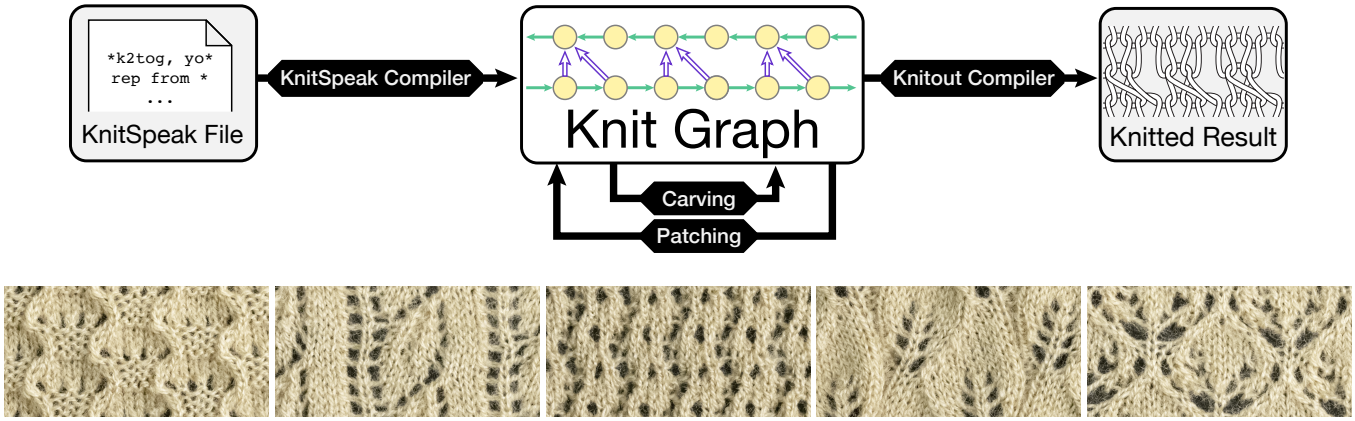


Figure 1: KnitPick converts KnitSpeak into KnitGraphs which can be *carved*, *patched*, and output to knitted results.

ABSTRACT

Knitting creates complex, soft fabrics with unique texture properties that can be used to create interactive objects.However, little work addresses the challenges of designing and using knitted textures computationally. We present KnitPick: a pipeline for interpreting hand-knitting texture patterns into KnitGraphs which can be output to machine and hand-knitting instructions. Using KnitPick, we contribute a measured and photographed data set of 300 knitted textures. Based on findings from this data set, we contribute two algorithms for manipulating KnitGraphs. KnitCarving shapes a graph while respecting a texture, and KnitPatching combines graphs with disparate textures while maintaining a consistent shape. KnitPick is the first system to bridge the gap between hand- and machine-knitting when creating complex knitted textures.

UIST '19, October 20-23, 2019, New Orleans, LA, USA.

ACM ISBN 978-1-4503-6816-2/19/10 ...$15.00.
http://dx.doi.org/10.1145/3332165.3347886

CCS Concepts

•Human-Centered Computing → HCI design and evaluation methods;

Author Keywords

machine knitting; fabrication; knitting; soft fabrication; compiler; knitspeak

INTRODUCTION

Digital fabrication of textiles is crucial to the production of interactive physical objects [10, 4, 3]. Knitted fabric, in particular, can create complex, seamless shapes with diverse structural properties (*e.g.,* stiffness, curl, stretch, opacity). Hand-knitters have also developed, documented, and curated a massive amount of practical knowledge on knitted textures [29, 26, 19, 16]. Industrially, automatic knitting machines are robust digital fabrication devices; however, machine knitting design interfaces require training to use and cannot directly leverage the widely available knowledge of hand-knitters.

Ultimately, the basis of a computer aided knitting design pipeline would enable the combination and manipulation of knitted textures to create complex textured fabric objects. We present KnitPick, a design pipeline that weaves together the

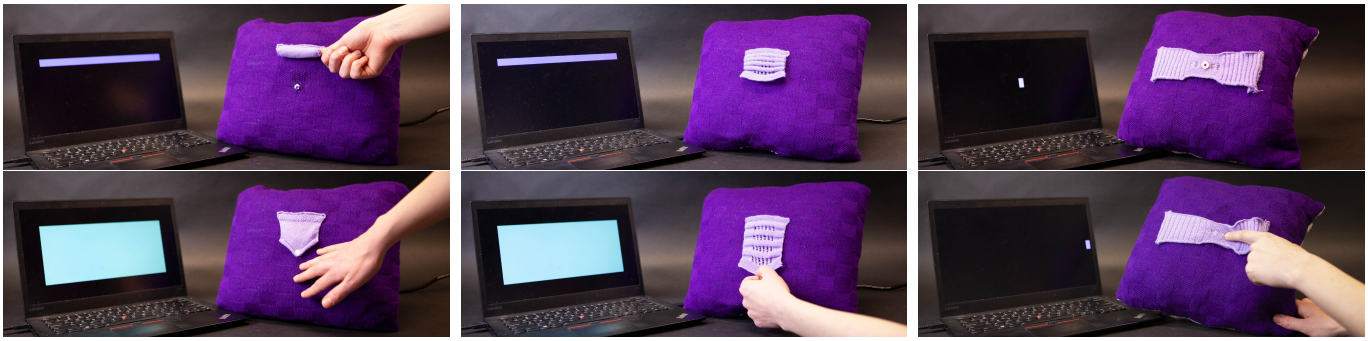


Figure 2: Three interactive knitted-texture pillows that support rolling (left), tugging (middle) and sliding (right) interactions.

knitting texture expertise of hand-knitters and the speed and reliability of machine knitting. KnitPick is a programming pipeline for interpreting and modifying hand-knitting patterns to create textured knitted objects that can be machine- or hand-knitted. There are three main contributions: a compiler for parsing and processing a large range of hand-knitting patterns into KnitGraphs; a large, measured data set of knit textures; and algorithms that can be used to modify (*KnitCarve*) and combine (*KnitPatch*) textures to create knit objects (Figure 1).

The KnitSpeak compiler interprets a pseudo-natural language often used in hand-knitting patterns – KnitSpeak – into a *KnitGraph* data structure. The KnitSpeak compiler is not the first to handle this task [6, 7], however it is the first language demonstrated over a large number of patterns and to support both machine knitting and hand knitting output. The KnitGraph data structure builds on the structure presented by Narayanan *et al.* [23], which automatically generates machine knitting instructions to fit arbitrary 3D meshes. Our KnitGraph is generated by the KnitSpeak compiler, which can verify four properties related to hand- and machine-knittablity.

Using the compiler, we contribute a large, diverse, measured data set of knitted textures and their properties. We compiled and machine-knitted 300 hand-knitting patterns. We photographed and measured the gauge (loops per inch of width and height) of each texture, with and without a loading force. Using the photographs, we measured the opacity of each texture. This data then serves as the basis for heuristics used in our core algorithms: KnitCarving and KnitPatching.

The relationship between shape and texture is tangled and fraught; it raises two challenges. First, textures are repeated patterns with the connections across repetition borders that are critical to the structure of the whole knitted object. That is, simply cutting a repeat to fit a desired size will likely cause the entire knitted object to unravel. The KnitPick pipeline addresses this with a dynamic programming approach to scaling textures, KnitCarving, based on a classic image-scaling algorithm, Seam Carving [5]. Second, combining many knitted textures into a single object often causes the properties of these textures to clash, stretching and distorting the shape of the object. The KnitPick pipeline addresses this with a heuristic-based optimization algorithm, KnitPatching, which joins knitted textures such that their boundaries remain flat.

Scenario

Imagine a designer creating a set of interactive machine-knitted pillows. She finds a tutorial on interacting with photocells and decides to control the pillows by laying opaque knitted fabric over the sensors. She browses our data set of knitted textures looking for variance in stretch and opacity, selecting four unique textures: stockinette that rolls up on itself, welts that spring vertically, ribbing that springs horizontally, and lace windows to let light shine onto the sensors. To create this design, she will need to: (1) convert hand-knitting texture patterns into knitting machine instructions, (2) combine the textures so that the lace window lays flat on the pillow case, and (3) adjust the textures to fit stitch-counts that align on the flat-case. This design process is cumbersome, requiring a machine-knitting expert to carefully define each stitch in the patterns. KnitPick provides a pipeline that automates each of these steps, laying the groundwork for computer aided knitting design that supports the creation of unique, interactive textured knitted objects. First she compiles her hand-knitted textures using the KnitSpeak compiler. Second, she lays out rectangular patches of the textures as she designs each pillow and uses KnitPatching to join the textures together to create one whole KnitGraph. In order to do this, KnitPick must adjust the number of repetitions and the sizes of each texture with KnitCarving. The KnitGraph is output to machine instructions so that she can manufacture and assemble her interactive pillows (Figure 2).

RELATED WORKS

There are a wide range of tools for computer-aided knitting. Commercial tools often target either hand or machine knitters, not both. Knitting machines use proprietary low-level chart-based design systems [27, 31, 28] that support customization of a few common patterns through a "wizard" interface. However, the user's options are extremely limited, particularly with regards to texture. For hand-knitters, there are numerous online pattern repositories [16, 26, 7], some of which guide users to make limited stitch-level changes. Ultimately, texture is reserved for knitting experts.

Researchers have approached sizing as a challenge of manipulating 3D meshes that are hand [22, 33] and machine [25, 23, 24] knittable. The addition of texture, when supported [24], may change the shape in unpredictable ways. Simulation of

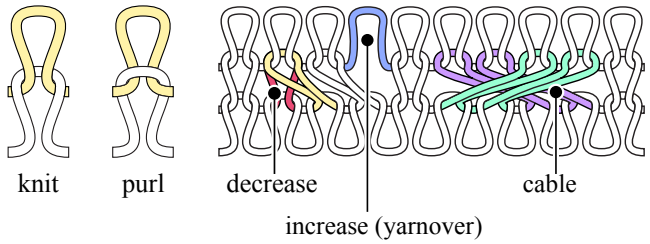


Figure 3: Loop-to-loop structures of the most common stitches: knits, purls, decreases, increases, and cables

knitted fabric has shown promising results [21, 12, 8], including supporting interactive design of small texture patches [17], but still requires case-specific hand-tuning to provide results that resemble a given yarn.

There are few tools that support interpreting, manipulating, and manufacturing machine knit objects. McCann *et al.* describes a base machine-knitting language [20] to support shaping un-textured objects made of sheets and tubes, and provide a transfer planning algorithm for converting these primitive shapes into machine code. McCann *et al.* stops short of supporting textures, neither expressing them or manipulating them. Narayanan *et al.* introduced knit graphs, representing stitches as nodes, with yarn and loop connections as edges [23]. This work did not support texture. Among six properties of machine knittable graphs, the properties of consistent orientation and limited node-degree are restrictive [23].

In later work, Narayanan *et al.* updated their system to use an annotated mesh structure in order to support arbitrary textures [24]. However their system includes only a limited number of pre-programmed textures and does not automatically account for how the shape changes because of the texture.Concurrently to KnitPick, Kaspar *et al.* approached texturing parameterized garments using a custom domain specific language, recognizing the same repeating, programmatic nature of knitted textures that KnitPick is based on [13].

BACKGROUND ON KNITTED STRUCTURES

A knitted structure is composed of a series of loops of yarn pulled through other loops; each loop stabilizes the loop it was pulled through [20]. A loop through a loop is called a *stitch*. A *course* is series of stitches side-by-side; these may also be called *rows* in a flat sheet of knitting, or *rounds* in a tube.

Texture derives from interconnected stitches. There are three composable properties of a stitch that change its effect on the texture: (1) the direction a loop is pulled through another loop, (2) how many loops it connects to, (3) how many other loops it crosses over. Hand knitters have developed a large set of named stitch-types that cover a variety of the most useful combinations. To demonstrate, we describe three groups: knits and purls, increases and decreases, twists and cables.

Knit/Purl: Loops can be pulled through other loops in either of two directions: from the back of the fabric to the front or, conversely, from the front to the back. The most basic stitch is a *knit*—a loop pulled from the back of the fabric, through another loop, to the front (Figure 3). A *purl* is the opposite: a loop pulled front-to-back through another loop (Figure 3). A single stitch in isolation cannot meaningfully be labeled as one or the other: a purl is simply a knit viewed from the back.

Decrease/Increase: More than one loop can be pulled through another loop (an *increase*), and a loop can be pulled through multiple other loops (a *decrease*). For textures, decreases and increases are locally paired (canceling each other out) to produce lace patterns. For example, a `yarn-over (yo)` leaves a small hole (*i.e.* an eyelet) in the fabric when paired with a decrease. Special types of increases and decreases are used create the first loops in a knitted object, and to stabilize the last loops in the object. Cast-ons are increases that increase the number of loops on the first course so that they can be pulled through subsequent courses. Bind-offs are decreases on the last course that decrease loops on the same course until only one loop is available which is knotted off by pulling the tail of the yarn through it.

Twist/cable: Finally, a stitch can cross over neighboring stitches. *Cables* are formed by transposing adjacent sets of stitches in the same course. A cable with just two stitches involved may additionally be called a *twist*. Cables tend to stiffen the fabric by creating additional tension on the loops as they are stretched across other loops. Cables give the appearance of a column of stitches winding across the fabric, colloquially known as a "traveling stitch".

KNITGRAPH REPRESENTATION

We define a KnitGraph as a directed graph where nodes represent loops[1] with yarn-wise and loop-to-loop edges. A KnitGraph is: an ordered set of loops on a yarn, $l \in Y$; a set of yarn-wise edges, $e(u \rightarrow v) \in E_Y$ between loops in the order they are constructed; and a set of loop-to-loop edges, $e(u \uparrow v) \in E_L$ representing how loops are pulled through other loops. Each loop-to-loop edge is labeled with an orientation: a loop pulled back-to-front or pulled front-to-back. By convention, the first yarn edge is directed from right-to-left when the cloth is viewed from the front. We explain our notation in Table 1 and diagrams in Figure 4.

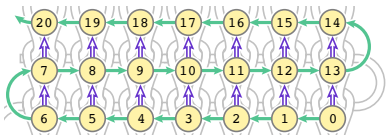
Notation	Interpretation
$t(l)$	The time loop l was constructed
$u < v$	Loop u was constructed before v
$e(u \rightarrow v)$	Loop u comes just before loop v on the yarn
$e(u \uparrow v)$	Loop v is pulled through loop u
$d(u \uparrow v)$	The edge from u to v has depth of d

Table 1: Summary of KnitGraph notation.

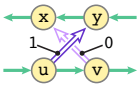
A knittable KnitGraph has the following four properties[2] These properties are stated for complete courses (*i.e.* no slip

[1] [23] use stitches as nodes. This difference is primarily for convenience with respect to our core algorithms.

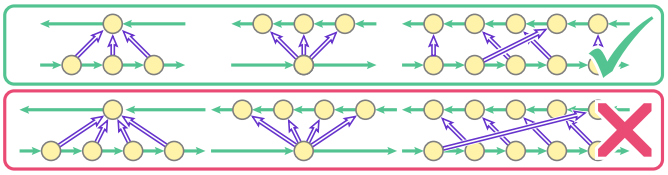
[2] Properties 2 and 4 are adopted from [23].



(a) Property 1 (Loop-to-Loop Stability) and Property 2 (Time Aligned Loops)



(b) Property 3 (Explicit Edge Depths)



(c) Property 4 (Limited Loop Distance)

Figure 4: Each loop (yellow circle) is constructed on a yarn (green arrows) which is pulled through (purple arrows) another loop. Loop-to-loop edges can cross over one another as long as the order they are crossed is defined (b). We limit the width of decreases, increases, and cables to prevent the yarn from tearing (c).

stitches [2] or short rows [1]). Our system uses similar properties to ensure knittability of graphs with partial courses.

Property 1: Loop-to-Loop Stability

The primary constraint of a knitted object is that each loop must have at least one other loop pulled through it. Any KnitGraph that satisfies this property will not unravel. For every loop, p, there exists a loop, l, that is pulled through p.

$$\forall p \in Y \; \exists l \in Y : e(p \uparrow l) \in E_L \tag{1}$$

Property 2: Time Aligned Loops

During knitting, yarn-wise edges establish the relative horizontal position of neighboring loops and the time that they are constructed. If a child loop, c, is pulled through a parent loop, p, p must be constructed before c.

$$\forall e(p \uparrow c) \in E_L : p < c \tag{2}$$

Property 3: Explicit Edge Depths

Decreases and cables *lean* to the left or right. In a cable, if the front-most parent loop is right of the other parent loops, it will make a left leaning cable. If the front-most parent is left of the other parents, it will create a right leaning cable. Similarly, loops stack on top of each other to create decreases. If the left most loop is stacked at the front, the increase and decrease will lean to the right. Conversely, stacking the rightmost loop at the front will create a left leaning decrease.

Cables are created when loop-to-loop edges cross. There is a cable between any two loop-to-loop edges $e(u \uparrow y)$ and $e(v \uparrow x)$ if the child loops are constructed in the order y then x and either: (1) the parent loops are constructed in the order u then v in rows or (2) v then u in rounds (Figure 4b).

$$cable(u \uparrow y, v \uparrow x) \equiv \begin{cases} e(u \uparrow y) \in E_L & \text{(3a)} \\ e(v \uparrow x) \in E_L & \text{(3b)} \\ y < x & \text{(3c)} \\ \begin{cases} u < v \text{ in Rows} \\ v < u \text{ in Rounds} \end{cases} & \text{(3d)} \end{cases}$$

Increases are created when there are more than one loop-to-loop edges from a common parent loop, p, to many children in the set $C \subset Y$. Similarly, decreases are created when there are many loop-to-loop edges from a set of parent loops, $P \subset Y$, into child loop c.

$$inc(p \uparrow C) \equiv \forall c \in C, \exists e(p \uparrow c) \in E_L \tag{4a}$$
$$dec(P \uparrow c) \equiv \forall p \in P, \exists e(p \uparrow c) \in E_L \tag{4b}$$

In a cable or decrease, we need to know how edges cross (cable) and stack (decrease): we must know each edge's *depth*. In a cable, if the loop-to-loop edge between u and y crosses the loop-to-loop edge between v and x, the depth of these edges cannot be equal. In a decrease, each loop-to-loop edge must have a unique stacking-order (*i.e.* each edge's *depth*).

$$cable(u \uparrow y, v \uparrow x) \iff d(u \uparrow y) \neq d(v \uparrow x) \tag{5}$$

$$\begin{aligned} &dec(P \uparrow c) \iff \\ &\quad \exists p' \in P : d(p' \uparrow c) \wedge \forall p \in P : d(p \uparrow c) \implies p' = p \end{aligned} \tag{6}$$

Property 4: Limited Loop Distance

If a loop is stretched to be pulled through another loop that is far away, the yarn is likely to tear; additionally, on a knitting machine, a needle may not be able to pull a new loop through very many loops stacked together. We allow loops to be pulled through loops up to 4 loops away (Figure 4c). For increases and decreases, the size of the set of child, C, or parent, P, loops must be less than or equal to 4. For a cable, both the distance between the parents (u, v) and the distance between the children (x, y) must be less than 4.

$$inc(p \uparrow C) \iff |C| \leq 4 \tag{7a}$$
$$dec(P \uparrow c) \iff |P| \leq 4 \tag{7b}$$
$$cable(u \uparrow y, v \uparrow x) \iff \begin{cases} |t(v) - t(u)| \leq 4 \\ |t(y) - t(x)| \leq 4 \end{cases} \tag{7c}$$

KNITSPEAK COMPILER

Knitting patterns are complex and vary sufficiently that they cannot easily be parsed; knitted *textures*, however, are often described with a consistent notation. Within curated sets of

knitted textures [30, 7], this notation is strictly enforced. It follows a consistent pattern for describing stitch-level instructions across repeated courses of texture. Colloquially, this notation is called KnitSpeak. Similar to programming, KnitSpeak loops through stitch instructions. Hand-knitters interpret these instructions like a computer interpreting machine-code.

At a high level, KnitSpeak is set of instructions for creating textures made up of *tiles* of repeatable patterns with a fixed number of loops and courses (Fig.5a). Tiles may be surrounded by bordering patterns on the left and right edge. Within a course, a KnitSpeak pattern defines a set of stitches that are repeated once, a set of stitches that are repeated in each width-wise tile, and another set of stitches that are repeated once. Each stitch is described with a keyword (*e.g.,* `k` for knit, `p` for purl, `yo` for yarn-over), or a keyword with associated variables (*i.e.* `k2tog` for knit two stitches together). Each course generally reads like a do-while loop (*e.g.,* `k *k,p* to last st, k` means "knit, then do knit and purl while there is one loop left on the last course, then knit"). Sets of courses are repeated to create vertical tiles. Courses are defined on each line of code and have a declared index (*e.g.,* `1st row`; `every odd row`).

Compiling KnitSpeak

Despite the programmatic structure, KnitSpeak is not directly transferable to machine instructions. We contribute the KnitSpeak Compiler, which translates the KnitSpeak used by Stanfield and Griffiths [30] and Stitch-Maps [7] into KnitGraphs. We compile KnitSpeak in three phases that ensure knittability.

Phase 1: Parsing KnitSpeak

We implement a parser using the Grammar-Kit parser generator [11] and JFlex lexer generator [15]. Grammar-Kit translates KnitSpeak into an abstract syntax tree based on a context-free grammar that covers both the Stanfield and Griffith and Stitch-Maps variants. We do not support knitting keywords that describe operations that are not strictly knitting (*e.g.,* wrapping yarn around loops). The grammar denotes the loop construction order with the keywords `row` and `round`. We exclude rounds if the KnitGraph will be constructed on our machine.

KnitSpeak is composed of stitch-tokens corresponding to canonical stitch structures (*e.g.,* `k,p,yo, k2tog, yo, t2l, c3b`). Our compiler interprets each stitch-token as instructions for creating and connecting loops. It optionally rejects stitches wider than the allowable loop-distance on the machine (Property 4). Tokens related to twists (*e.g.,* `t2l, t2r`), cables (*e.g.,* `c3b, c3f`), decrease (*e.g.,* `k2tog, skpo`) imply the direction they lean, from which the crossing depth of stitches can be derived (Property 3).

Phase 2: Semantic Analysis

Semantic analysis determines the execution order of stitch-token instructions, creating repeatable structures in a KnitGraph. This phase ensures the construction order of loops (Property 2) and connects all loops to a child (Property 1).

Each course instruction is stored in a symbol table with indices indicating the order that they will be executed. As the symbol table is filled in, variables for the tile and border width are updated based on the calculated loop counts from each instruction. If these values are not equal across courses, then one course could produce more loops than the following course consumes, resulting in a violation of the loop-to-loop constraint (Property 1). Mismatches in a loop count result in an error. Each course in the symbol table relies on the loops created on the course below it. As long as the loop counts match up between courses, the stitch-token instructions will be able consume the child loops of the last course and create new loops for the next course. The new loops are guaranteed to be created after the parents, satisfying Property 2.

```
1st and 5th rows k.
2nd and 4th rows p.
3rd row k2, [yo, k2tog, k2]
    to last 3 sts,
    yo, k2tog, k1.
6th row p3, [p2tog, yo, p2]
    to last 2 sts,
    p2.
```

(a) KnitSpeak

(b) Machine knitted sample

Figure 5: Our system converts KnitSpeak (a) to KnitGraph that is machine knittable (b).

Phase 3: KnitGraph Instantiation

Given a complete symbol table, we instantiate a KnitGraph with a specified number of repeated tiles (width and height). First, the compiler creates a cast-on course with the user specified loop count. Next, the compiler traverses all of the course instructions in the KnitSpeak pattern. At each instruction it creates the specified stitches, consuming available loops left on the last course. First it creates the stitches in the starting border, next it consumes available loops to create the width-wise tiles until the required number of loops for the ending border are left available. Once all of the course instructions have been actualized the system may repeat the process, to lengthen the graph by a specified number of height-wise tiles.

Generating Knitting Instructions

We output instructions for both hand knitting and automatic machine knitting. Knitting machines use rows ("beds") of hook-shaped needles; for more details, see [20, 4]. Unlike in hand knitting, in which the most recent course of loops is free to slide along the single long needle, each column of machine-made stitches is held at the top loop by its own separate needle. Thus, each loop must be allocated a specific needle at the time of its construction, and its parent must be located there at that time to receive it. Combining loops onto needles for decreases, creating spaces for increases on empty needles between loops, and using the front and back beds for knitting and purling, can all require re-arranging loops between courses of knitting.

By convention, the first course is allocated right-to-left, with each loop assigned a needle directly leftward of the one before it. For subsequent courses, two variables are maintained: a *cursor*, corresponding to the loop's position in the course, and a *slide* variable. While iterating over the loops from left to right (which may be the opposite of the order they will be constructed, in the case of a right-to-left course), *cursor* is incremented and *slide* is updated per loop: if a loop has one parent, *slide* remains the same; if a loop has more than one parent, *slide* is decreased for each parent; if a loop has no

Figure 6: Data set measurement setup, including camera, scale, and stretching rig.

Figure 7: Sample swatches from data set against a black background.

parents or its parents have other children, *slide* is increased. Note that for a swatch with only local increases and decreases (that is, no net loop count change), *slide* will be zero at the end of the course.

Each loop is allocated to a needle at position = [cursor] + [slide]. The allocated needles are then used to determine the amount that each parent will be offset to support the new course of loops. These re-arrangements are accomplished via needle transfers determined by Lin *et al.* 's "schoolbus+sliders" transfer solver [18].

To support hand knitters, we decompile KnitGraphs into KnitSpeak. Information about repetition is lost in the KnitGraph. We iterate over all loops in the yarn and determine which keyword corresponds to the set of edges entering connecting the loop. Each stitch is written out in the order it is found on the yarn. When we encounter a loop that is dependent on loops in the current course, a new course is created, starting with this loop.

KNITPICK TEXTURE DATA SET

Thousands of knitting patterns are available online. Even among textures such as those supported by our KnitSpeak compiler, the possibilities available number at least in the thousands. By using the KnitSpeak compiler on these patterns, we can better understand its capabilities and limitations. Thus, our next contribution focuses on creating a curated set of real-world textures. Using the KnitSpeak compiler we created a data set of 166 samples from Stanfield and Griffiths [30] and 306 samples from Stitch-Maps [7]. Given these compiled textures, as of publication we machine knit and measured 300 textures.

Sampling Strategy

Stanfield and Griffiths curated 300 knitted textures. Of these, the KnitSpeak compiler interpreted 166 (55.3%) into KnitGraphs which we machine-knitted. All textures in the book section "Bobbles and Leaves" were excluded for using wide increases, decreases, and cables that violate Property 4. All other excluded patterns included annotations that described actions that are not machine-knittable or included multiple yarns.

We collected 1454 of the most recent and popular KnitSpeak samples from the 5979 patterns on Stitch-Maps . We excluded 393 (27.0%) samples that were written in the round and 755 (51.9%) because they violated Properties 1 or 2. We observed that these 755 samples were not textures but full patterns (*e.g.,* sweaters) which are beyond the scope of this study. Ultimately, we compiled and knit 306 (21.0%) textures from Stitch-maps.

Swatch Construction and Measuring

We constructed and measured textures as follows: (1) machine knit a 60 loop by 60 course swatch with an additional border including eyelets for alignment; (2) weigh the swatch; (3) lay the swatch on fine-grained sandpaper to prevent curling; (4) photograph the swatch in the un-stretched state; (5) measure the un-stretched swatch across the center axes; (6) connect the swatch to a stretching rig and load it with a constant mass; (7) photograph the stretched swatch; (8) measure the stretched swatch. Our measurement station is shown in Figure 6.

We knit our swatches on an Shima Seiki SWG91N2 15-gauge v-bed knitting machine using Tamm Petit, a 2/30NM (8,147 yards per pound) acrylic yarn with moderate twist. We used our machine's digital stitch control system to regulate yarn tension and our stitch size was 40 with leading set 25. We tiled each texture to fit a 60 loop by 60 course swatch, then added knit-stitches to the borders to fill in the gaps. The texture is surrounded by a 12-stitch-wide border of a checkered knit and purl texture to stabilize the swatch edges and normalize their connection to the rig. We placed an eyelet at the center and ends of each edge of the swatch. To stretch a swatch, we hooked each eyelet to rods that can roll freely in one direction, pulling the swatch linearly along its width and height. These

Measurement	Min	Max	Mean	STD
Loops per Tile	1	60	14	17
Courses per Tile	1	60	11	9
Loops per Inch	5	25	12	3
Courses per Inch	9	27	17	2

Table 2: Texture repetition and gauge data

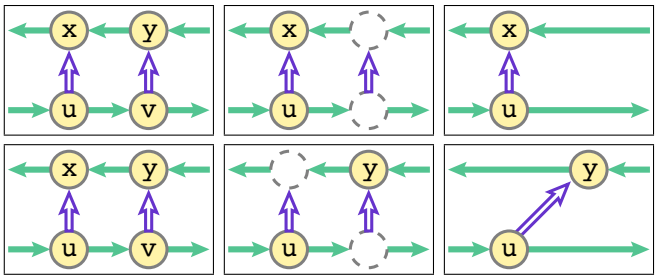


Figure 8: The top case shows the removal of connected loops in a path. The bottom case shows the removal of neighboring loops in a graph.

rods were attached to 608g weights. We gently dropped the weights off the edge of the table.

We collected four measurements to derive gauge: stretched width (1) and height (2); un-stretched width (3) and height (4). Gauge is the number of loops per unit width and height in a texture. Gauge decreases as a texture is stretched. We calculated opacity as the count of black pixels (matching the sand paper background) shown through the knitted texture and appearing per photograph.

Summary of Knitted Textures

We saw a wide range of gauges and tile sizes; we summarize these statistics in Table 2 and show swatch samples in Figure 7. The smallest horizontal gauge, 25 loops per inch, was five times as tight as the widest. This variation underpins a challenge of using textures on knitted objects: it is difficult to match desired measurements using discrete tiles, and the variation in gauges can compound the problem when multiple textures are integrated into a design.

KNITCARVING: RESIZING KNITTED TEXTURES

A knitted object is typically defined with a specific size, which is in turn refined into a specific loop and course count based on the gauge (loops per inch horizontally and vertically). Indeed, modifications to loop count is one of the first challenges a knitter encounters when changing a pattern. If a new texture (or a new yarn, or even new needles) is applied to an existing design, it is likely to change the gauge. A more complicated concern is that textures are discrete tile units. Thus, the knitting designer must not only adjust gauge, but also ensure that the size of the texture divides evenly into the number of loops and courses of each tile.

A naïve solution is to stop knitting mid-tile. In the best case, this will create an obvious line where the tile is stopped. However if cables, increases, and decreases are present in the pattern the result may violate Property 1. For example, a tile may have a decrease, but if that decreases' child loop is removed, the parent loops will not have a child loop stabilizing them.

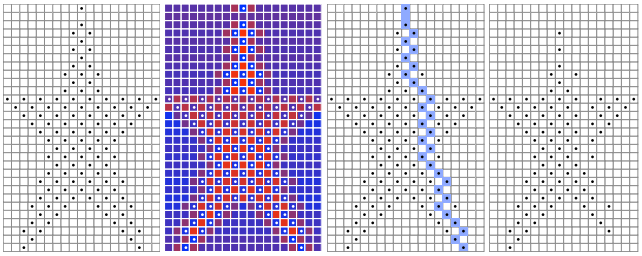


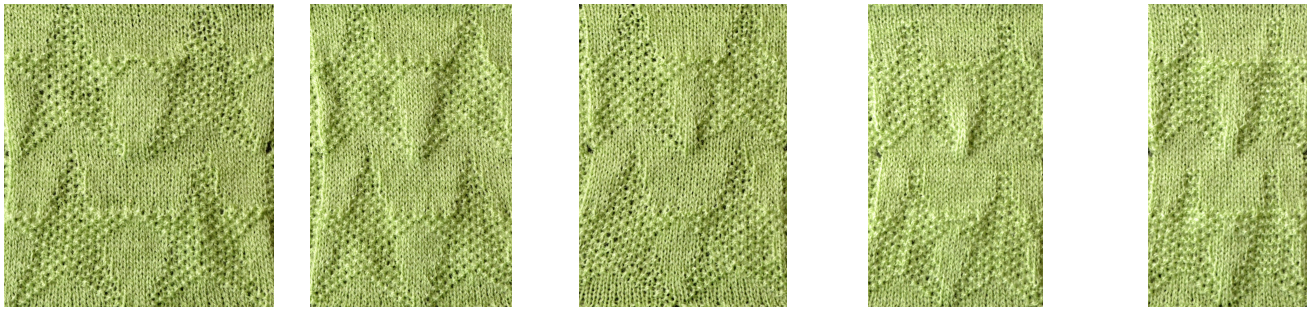
Figure 9: A "Star" knit/purl pattern is shown with knits represented as empty squares and purls represented as squares containing dots (a). The cost to remove each loop is represented by a heat-map (b); minimum-cost stitches along a path (c) can be removed to narrow the pattern (d).

KnitCarving is an alternate approach that maintains the stability of a KnitGraph, while removing a continuous path of loops that narrows [3] the graph with minimal changes to the texture. This is based on the "Seam Carving" technique for scaling images [5]. KnitCarving is a dynamic programming optimization that removes loops from the path with the least-significance (*i.e.* lowest cost) to the knittability and aesthetics of the KnitGraph. To ensure that a path is continuous, we require that loops which are removed in a given course be directly above the removed loop in the previous course, or directly above one of its yarn-wise neighbors. To maintain continuity across repetitions, the user can set an option which will remove repetitions before KnitCarving, or may carve the texture, maintaining the placement of repetitions. Additionally, rather than removing one path, they can remove a set of least valuable paths all at once.

A loop's removal cost is calculated locally. The path with the minimum removal cost is the path of loops with the lowest sum of each loop's removal cost. This path is found in a dynamic programming fashion. First, for each loop we calculate the local cost of removing the loop. Next, we use Dijkstra's shortest path algorithm [9] to find the path from a loop on the cast-on course to a loop on the bind-off course with the minimum removal cost.

Once a path has been selected, the loops in the path are removed from the KnitGraph, leaving behind loop-to-loop edges that are missing either a parent or a child. Figure 8 shows the two possible cases that can occur. In the simplest case, a loop v, its direct child y, and the edge between them are removed. The more complex case is when a loop v is removed, but its yarn-wise neighbor's (u's) child x is lower cost than its own child y. In this case, the edge from v to y most be removed *and* the edge from u to x most be reconnected from u to y to repair the KnitGraph and maintain Property 1.

[3] This same approach can be used to shorten a graph, but for brevity we describe the algorithm with respect to width.



(a) Original swatch (b) 6 columns removed (c) 9 columns removed (d) 12 columns removed (e) 15 columns removed

Figure 10: The above images show a progression from the original Star texture to the same texture with 15 columns removed by texture carving. These photographs were shown to crowd-workers who rated their similarity. Even with a whole repetition width removed from the Stars, the pattern remains a recognizable star pattern.

Group	All Textures	Knit-Purl Texture	Twist Texture	Cable Texture	Lace Texture	Wide Repeats	Narrow Repeats
Knit Carving: Mean (Std.)	51.73 (13.66)	12.53 (3.84)	12.7 (4.04)	13.54 (3.68)	13.19 (3.74)	26.43 (7.02)	25.30 (7.12)
Control Mean	41.38 (11.57)	10.39 (3.30)	10.78 (3.23)	10.20 (3.42)	10.40 (3.53)	20.46 (6.20)	20.92 (5.90)
Significance	**T=6.8 (p<.0001)**	**T=4.25 (p<.0001)**	**T=3.71 (p<.0001)**	**T=6.65 (p<.0001)**	**T=5.42 (p<.0001)**	**T=4.12 (p<.0001)**	**T=2.56 (p<.0001)**

Table 3: Summary statistics of a independent-sample t-test comparing the sum of similarity scores for KnitCarving and control conditions. The degrees of freedom across all tests was 199.

Although loops used for knits, purls, twists, and cables are all candidates for removal, loops used in decreases and increases cannot be removed as easily. For example, recall that a decrease involves many parent loops pulled through one child loop. If this child loop is removed, it is possible the repaired graph will violate either Properties1 or 4 by leaving a parent loop without a child or by creating a wide decrease. To forbid this, we assign an infinite cost to these loops.

For all other loops, we calculate a ratio representing that loop's rarity in the graph. We assess the value of the remaining removable loops based on the rarity of the stitch they are involved in. If a loop's relationship to other loops is rare in the graph, it is more significant. Loops are equivalent to other loops (*i.e.* $u \equiv v$) if all of their incoming and outgoing edges have the same orientation.

$$cost(l) = \frac{|Y| - |\{u \in Y : l \equiv u\}|}{|Y|} \tag{8}$$

Crowd-Sourced Evaluation

We evaluated KnitCarving in a study with 200 crowd workers. Each worker rated the similarity of eight sets of two images using a scale from 1 to 10 (10 being the very similar). The first image was a photograph of a texture swatch and the second image was a photograph of swatch where the texture had some number of paths removed. We asked workers to compare the images based on a list of features (*i.e.* skew, size, number of whole repetitions, stretch, opacity) with simple descriptions of how they apply to knitted textures.

We divided workers into two groups: 100 workers compared swatches that were narrowed with KnitCarving and 100 workers compared swatches that were narrowed with a control algorithm that removes the right most loop from each course in a swatch. The control algorithm may violate our Knittability properties, so some control swatches visibly unraveled (Figure 11). Workers were further grouped based on what portion of the repetition was removed.

Eight textures were selected from our texture data set: two knit/purl patterns, two twist patterns, two cable patterns, and two lace patterns. All eight textures were carved five times by one fifth of their repetition width. Within each texture category, one texture had a large repetition (in the third quartile), and the other had a small repetition (at least five, in the first quartile). Within these constraints, we randomly selected the textures.

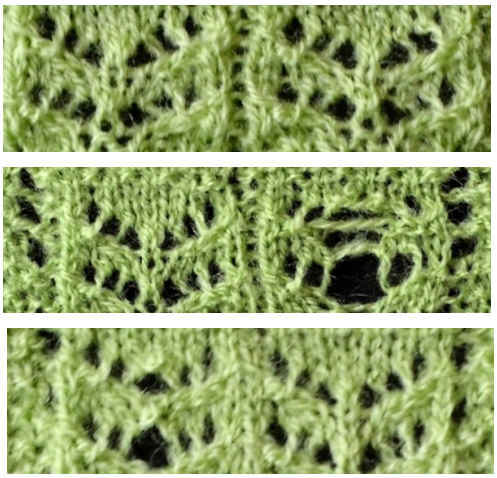
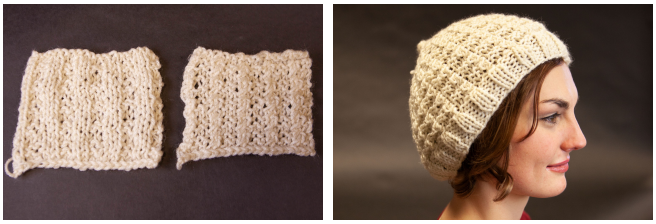


Figure 11: KnitCarving is effective on a variety of textures, including lace patterns with dependencies across repetitions of the pattern. Top: two repetitions of the lace texture "Little Branches." Middle: four columns naïvely removed from lace texture, causing it to unravel. Bottom: Four KnitCarves removed from texture, leaving behind a similar texture.



(a) Original texture from Ravelry (left) and a new "Twisted" texture (right)

(b) The hand-knit Hat created by KnitCarving the Twisted Texture

Figure 12: A simple change (replacing knits with cables) (a) causes significant gauge variances. As a result, the entire hat pattern must be adjusted given the new gauge. KnitCarving until only a few stitches remains will produce the crown of the hat (b).

We conducted seven independent-samples t-tests to compare the sum-total similarity score across (1) all textures per worker, (2-5) individual texture types, (6) wide repeat textures, and (7) narrow repeat textures. Across all tests, KnitCarving performed significantly better than the control algorithm. We summarize the results in Table 3.

Demonstration: Hand-Knitted Custom Hat

We created a hand-knitting pattern for a KnitCarved hat based on a free Ravelry pattern [32]. Knitting patterns specify the yarn type and needle size because they dramatically effects the texture's gauge. The pattern author selectively placed decreases on each course; essentially doing the work of KnitCarving. But when we change the texture, her work is lost.

Consider a hat; when flattened out it is essentially a triangle, with a wide base that narrows down to a few stitches that are sewn together at the tip of hat. We extracted KnitSpeak from the pattern and modified it to include twists. Figure 12a shows the difference between the basic texture and our twisted texture. To create the hat, we carved the pattern, decreasing the number of loops in each course over each height repetition until only a few loops remain that can be sewn together at the tip of the head. We generated KnitSpeak instructions for the resulting KnitGraph and hand knitted the hat (Figure 12b).

KNITPATCHING: COMBINING KNITTED TEXTURES

Many knit objects are made up of multiple textures; however, a beginning knitter may not have the expertise to properly combine them. KnitPatching lets knitters use multiple textures to create unique aesthetic and functional effects. Using multiple textures is non-trivial because the interactions between textures may change the fabric's shape in unintended ways, particularly when the textures have disparate gauges (Figure 13). We model different knitted textures as patches on a sheet of knitted fabric. Using heuristics based on common hand-knitting practice, we optimize the number of loops in each patch to minimize gauge variance and produce a flat, rectangular sheet of fabric.

Knitted Sheets made of Patches

Our knit patching solution focuses on flat (unshaped) KnitGraphs. We support combining an arbitrary layout of rectangular, textured *patches*. Each patch has a position, an assigned texture, a width, w_p, and height, h_p, in inches[4]. A sheet is completely covered in non-overlapping patches.

To create a KnitGraph for a given sheet, the textured patches must be assigned to loops in the KnitGraph so that the sheet is knittable. A tile of knitted texture, with width t_w in loops and height t_h in courses, is unlikely to exactly match a patch's width and height; tiles are typically just big enough to uniquely define the textural effect. When a region or patch is textured, tiles are typically repeated to fill it.

A texture is defined in units of loops and courses. A texture's *gauge* converts from loop/course to inches. A texture is defined by two measures of gauge. The texture's width gauge, g_{t_w}, is the number of loops per inch. Its height gauge, g_{t_h}, is the number of courses per inch. Gauge was calculated in our texture data set and can also easily be hand-calculated from a small sample of a provided texture.

Given a gauge and a patch size, it is trivial to determine how many repetitions of the texture fit in the patch in each direction: the patch size multiplied by the gauge and divided by the tile size. When a patch doesn't fit perfectly this introduces a **sizing error** (*i.e.* $(p_w \cdot g_{t_w}) \bmod t_w \neq 0$ or $(p_h \cdot g_{t_h}) \bmod t_h \neq 0$). KnitCarving is used to ensure that the texture fits into the number of loops that satisfy the patch size, but this introduces a **texture error**.

A third **joining error** occurs when the number of loops and courses of adjacent patches do not match up. If each patch were given the number of loops and courses dictated by its gauge, no patch would line up because their gauges are different. If not corrected, this can lead to knittability violations – since loops or yarn edges will not be able to connect 1-1 between patches (Properties 1 and 2).

[4]Any real world measurement unit could be used.

Figure 13: A naively joined texture, left, is distorted by variance in gauge and limited by repetitions of its constituent textures. The KnitPatched texture, right, lies flat and better matches the target size.

Our solution to this problem is *KnitPatching*. KnitPatching forces neighboring patches to have the same loop/course counts while minimizing the changes made to the patches' textures and sizes. This is done by judiciously deciding to increase or decrease the number of loops in a patch. We approach KnitPatching as an optimization problem.

KnitPatching Objective Function

We calculate error at the sheet level in terms of the three error metrics introduced above: error in the size of each patch, error in the texture of each patch, and error introduced by using decreases or increases to ensure correct loop counts at the borders between patches. Formally, the error, E, of a KnitPatch sheet, S, is a weighted sum of these three errors (Equation (9)). α, β, σ denote the weights of each error type. We describe each type of error in more detail below.

$$E(S) = \sum_{p \in S} \alpha E_s(p) + \beta E_t(p) + \sigma E_j(p) \tag{9}$$

Sizing error, E_s, describes how much a patch varies from the user-specified size. For a patch, p, this is the proportion of the absolute difference in inches between the desired patch size and the size predicted given the patch texture's gauge and the assigned loop and course counts, divided by the patch's size to capture the importance of scale. An inch difference in size is significant on a 4 inch patch, but insignificant over 100 inches.

$$E_s(p) = \frac{|w_p - l_p \cdot g_{t_w}|}{w_p} + \frac{|h_p - c_p \cdot g_{t_h}|}{h_p} \tag{10}$$

Texture error, E_t, is introduced by carving a texture to fit the assigned loop/course count. KnitCarving will remove Δ_w loops and Δ_h courses from a patch. Texture error is the fraction of total loops/course removed from the patch. Carving a path from a patch with 4 loops is more significant than carving a patch that is 100 loops. Note, that at this point no KnitGraph has been constructed, so we do not know the actual cost of carving the texture based on the objective function of KnitCarving (Equation 8).

$$E_t(p) = \frac{\Delta_{w_p}}{l_p} + \frac{\Delta_{h_p}}{c_p} \tag{11}$$

Joining error, E_j, describes the use of increases and decreases at patch borders to align loop counts. This has a similar effect to KnitCarving, but instead creates loop-to-loop edges at patch borders as needed between misaligned loops. This technique can only be used to align the top and bottom edges of a patch. The loop count of the top, $l_{\uparrow p}$, or bottom, $l_{\downarrow p}$, neighbors of a patch, p, is compared to the loop count for the patch, l_p, to determine the number of new edges needed on each border. The joining error, E_j, is the number of these new edges divided by the original number of loops in p. Again, adding a new edge among 4 connected loops is more significant than adding that edge among 100 connected loops.

Figure 14: A 4x4 inch panel with cable (left) and lace (right) patterns bordered by stretchy garter and ribbing textures

$$E_j(p) = \frac{|l_{\uparrow p} - l_p|}{l_p} + \frac{|l_{\downarrow p} - l_p|}{l_p} \tag{12}$$

Heuristic-Based Patch Sizing

This objective function reveals trade-offs between properly sized patches and variations in the texture.

Consider the trade off between sizing and texture errors. To minimize sizing error, we would ideally create loops for each patch based on its width w_p and texture gauge g_t (*i.e.* $l_p = w_p \cdot g_{t_w}$). However, it is unlikely that these counts will be evenly divide by the tile size, l_t, so we will need to carve it down, which will increase the texture error. The search space for minimizing the sum of E_s and E_t is small, bounded by the size of a tile. We search for the minimum weighted sum of these errors by iterating over possible loop counts, $\hat{l_p}$ (*i.e.* $l_p - l_t \leq \hat{l_p} \leq l_p + l_t$). We use the same approach to determine course counts.

Given these improved loop/course counts on each patch we must force these values to align across neighboring patches. Some patches will increase their sizing and texture errors to accommodate their neighbor. We determine which patches will increase their error by assigning each patch a significance value. This value can be set by a user or determined heuristically. We use the following heuristics to assign significance:

- *Tile Size*: The larger the tile size of a texture, the more texture error will likely be introduced by changing the loop/course count. The larger the tile size, the more significant the patch.
- *Stretch*: Stretchier textures are more likely to stretch to match the size of their neighbors, reducing the actual error in sizing. Thus stretchier textures have lower patch significance.
- *Carve-ability*: Patterns with many increases and decreases cannot be KnitCarved as effectively. Higher numbers of increases/decreases increase the significance.

Given significance values, we propagate the loop/course counts from the most significant patches to the least significant patches in a depth-first traversal across the borders between patches. Starting from the most significant patch, p, we assign its ideal loop count, l_p, to each of p's neighbors, starting with

the most significant neighbor, n. This changes the neighbor's loop count from l_n to l'_n. Note that we only assign the loops of l_p proportional to the overlapping width between p and n. n then propagates its new loop count, l_n, to its most significant neighbors. We traverse the edges between neighbors until the edge of the sheet is reached.

We repeat the process in parallel to assign course counts. At this point the course counts have been optimized. Put another way, the objective function for course optimization does not include joining. This is because joining changes the number of loops using increases and decreases, but we do not support an equivalent for courses (this would violate Property 1).

The final step of our algorithm is to optimize loop counts to minimized the sum of sizing and texture error. At this point, patches have a propagated loop count, l'_p and a loop count that minimizes the sum of sizing and texture errors, l_p. If these counts differ, we can correct this difference by creating new loop-to-loop edges to the misaligned loops. There is a trade-off between creating edges and KnitCarving. If the propagated loop count is less than the ideal loop count we can either: (1) create a new edge for each additional propagated loop (increasing joining error), or (2) carve out loops from the ideal loop count (increasing texture error).To find the minimum sum of the texture and joining error we iterate over the possible combinations of KnitCarving and additional edges (*i.e.* $l'_p \leq \hat{l_p} \leq l_p - l'_p$)

This heuristic based approach does not guarantee a globally optimal solution, but as shown in Figure 14, it generally results in flat knitted sheets. This method can accommodate a variety of knitted textures including lace and cable patterns. It is particularly well suited to the gauge variances in knit-purl patterns such as ribbing and garter stitch.

LIMITATIONS

KnitPick introduces a new technique for describing, using, and manufacturing knitted textures. However, it is primarily suited for creating shaped sheets of knitted textures rather than shaped objects. Clever hand-knitters construct complex custom-fit garments using the techniques we have automated. However, a simplified, graphical, interface is a necessary next step to making knitted textures widely accessible. Further, KnitPick's algorithmic core, KnitCarving and KnitPatching, may benefit from objective functions rooted in physics rather than hand-knitting practices. Little is codified about how textures shape an object. Accurate simulation of this interaction between shaping and texture requires an approach based in physics, not just visually plausibility [12]. It would be interesting to see if the measured properties of our data set could be used to predict the properties of the Kaspar *et al.* data set [14] and/or be combined in other ways to benefit research and knitting communities.

CONCLUSION

In this paper we presented the KnitPick programming pipeline for describing, shaping, combining, and manufacturing textured knitted objects. Central to this pipeline is a KnitGraph structure which maintains four properties: (1) each loop is stabilized by having another loop pulled through it; (2) each loop is pulled through loop[s] that were previously constructed; (3) cables have an explicit crossing-depth; and (4) all loops are pulled a limited distance, preventing yarn tears. Our KnitSpeak compiler translates a pre-existing pseudo-natural language created by hand-knitters to describe textures into KnitGraphs and verifies that they are hand- or machine-knittable. Using this compiler, we machine-knit, measured, and photographed 300 textures. Based on these textures we developed a KnitCarving algorithm which selectively scales a KnitGraph in width and/or height while maintaining the texture's aesthetic properties. Finally we contribute KnitPatching, which joins patches of knitted textures while accommodating disparate gauges to create flat sheets of fabric. Using the KnitPick pipeline, we have created three knitted interactions: roll, tug, and slide.

This work contributes to the body of literature that is helping to soften the hard corners of plastic and metal fabrication work. Knitting is certainly an important option if we want to create comfortable, wearable, or soft solutions to complex problems. In future work, we hope to untangle the relationship between texture and more complex shapes as represented in hand knitting patterns that describe whole objects. We also hope to develop new methods for pattern design by inexperienced knitters and knitters who are not experienced with programming. Finally, our work has the potential to inform simulation of knitting.

ACKNOWLEDGMENTS

This work was funded by: National Science Foundation Grants IIS-1718651 and IIS-1907337.

REFERENCES

[1] 2009. Short rows: method. (2009). http://techknitting.blogspot.com/2009/10/short-rows-method.html

[2] 2016. How to Knit: Slipping Stitches Purlwise and Knitwise | Lion Brand Yarn. (2016). http://www.lionbrand.com/how-to-knit-slipping-stitches

[3] Sean Ahlquist, Wes McGee, and Shahida Sharmin. 2017. PneumaKnit: Actuated Architectures Through Wale-and Course-Wise Tubular Knit-Constrained Pneumatic Systems. (2017).

[4] Lea Albaugh, Scott Hudson, and Lining Yao. 2019. Digital Fabrication of Soft Actuated Objects by Machine Knitting. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. DOI: http://dx.doi.org/10.1145/3290605.3300414

[5] Shai Avidan and Ariel Shamir. 2007. Seam Carving for Content-aware Image Resizing. In *ACM SIGGRAPH 2007 Papers (SIGGRAPH '07)*. ACM, New York, NY, USA. DOI:http://dx.doi.org/10.1145/1275808.1276390

[6] Chelsea Battell. 2016. Domain Specific Language for Modular Knitting Pattern Definitions: Purl. *CoRR* abs/1606.08708 (2016). http://arxiv.org/abs/1606.08708

[7] JC Briar. 2013. Stitch Maps. (2013). https://stitch-maps.com/

[8] Gabriel Cirio, Jorge Lopez-Moreno, David Miraut, and Miguel A. Otaduy. 2014. Yarn-level Simulation of Woven Cloth. *ACM Trans. Graph.* 33, 6 (Nov. 2014), 207:1–207:11. DOI: http://dx.doi.org/10.1145/2661229.2661279

[9] E. W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1, 1 (01 Dec 1959), 269–271. DOI: http://dx.doi.org/10.1007/BF01386390

[10] Scott E. Hudson. 2014. Printing Teddy Bears: A Technique for 3D Printing of Soft Interactive Objects. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 459–468. DOI: http://dx.doi.org/10.1145/2556288.2557338

[11] JetBrains. 2017. Grammar-Kit: Grammar files support & parser/PSI generation for IntelliJ IDEA. (Dec. 2017). https://github.com/JetBrains/Grammar-Kit original-date: 2011-08-04T12:28:11Z.

[12] Jonathan M. Kaldor, Doug L. James, and Steve Marschner. 2008. Simulating Knitted Cloth at the Yarn Level. In *ACM SIGGRAPH 2008 Papers (SIGGRAPH '08)*. ACM, New York, NY, USA, 65:1–65:9. DOI: http://dx.doi.org/10.1145/1399504.1360664

[13] Alexandre Kaspar, Liane Makatura, and Wojciech Matusik. 2019a. Knitting Skeletons: A Computer-Aided Design Tool for Shaping and Patterning of Knitted Garments. In *Proceedings of 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. ACM, New Orleans, LA.

[14] Alexandre Kaspar, Tae-Hyun Oh, Liane Makatura, Petr Kellnhofer, Jacqueline Aslarus, and Wojciech Matusik. 2019b. Neural Inverse Knitting: From Images to Manufacturing Instructions. *CoRR* abs/1902.02752 (2019). http://arxiv.org/abs/1902.02752

[15] Gerwin Klein, Steve Rowe, and Régis Décamps. 1999. JFlex - JFlex The Fast Scanner Generator for Java. (1999). http://jflex.de/

[16] Knit It Now LLC. 2013. KnitItNow Pattern Library. [Online]. Available from: https://www.knititnow.com/knit/catalog.cfm. (2013).

[17] Jonathan Leaf, Rundong Wu, Eston Schweickart, Doug L. James, and Steve Marschner. 2018. Interactive Design of Periodic Yarn-level Cloth Patterns. *ACM Trans. Graph.* 37, 6, Article 202 (Dec. 2018), 15 pages. DOI:http://dx.doi.org/10.1145/3272127.3275105

[18] Jenny Lin, Vidya Narayanan, and James McCann. 2018. Efficient Transfer Planning for Flat Knitting. In *Proceedings of the 2Nd ACM Symposium on Computational Fabrication (SCF '18)*. ACM, New York, NY, USA, Article 1, 7 pages. DOI: http://dx.doi.org/10.1145/3213512.3213515

[19] Knitty Magazine. 2017. Knitty Index : Knitty.com - Winter 2017. (2017). http://knitty.com/ISSUEw17/index.php

[20] James McCann, Lea Albaugh, Vidya Narayanan, April Grow, Wojciech Matusik, Jennifer Mankoff, and Jessica Hodgins. 2016. A Compiler for 3D Machine Knitting. *ACM Trans. Graph.* 35, 4 (July 2016), 49:1–49:11. DOI: http://dx.doi.org/10.1145/2897824.2925940

[21] Michael Meißner and Bernd Eberhardt. 1998. The art of knitted fabrics, realistic & physically based modelling of knitted patterns. In *Computer Graphics Forum*, Vol. 17. Wiley Online Library, 355–362.

[22] Yuki Mori and Takeo Igarashi. 2007. Plushie: An Interactive Design System for Plush Toys. In *ACM SIGGRAPH 2007 Papers (SIGGRAPH '07)*. ACM, New York, NY, USA. DOI: http://dx.doi.org/10.1145/1275808.1276433

[23] Vidya Narayanan, Lea Albaugh, Jessica Hodgins, Stelian Coros, and James McCann. 2018. Automatic Machine Knitting of 3D Meshes. *ACM Trans. Graph.* 37, 3, Article 35 (Aug. 2018), 15 pages. DOI: http://dx.doi.org/10.1145/3186265

[24] Vidya Narayanan, Kui Wu, Cem Yuksel, and James McCann. 2019. Visual Knit Programming. *ACM Trans. Graph.* 38, 4 (July 2019).

[25] Mariana Popescu, Matthias Rippmann, Tom Van Mele, and Philippe Block. 2018. Automated generation of knit patterns for non-developable surfaces. In *Humanizing Digital Reality*. Springer, 271–284.

[26] Ravelry 2017. Ravelry: Home. (2017). https://www.ravelry.com/

[27] Shima Seiki. 2011. SDS-ONE Apex3. [Online]. Available from: http://www.shimaseiki.com/product/design/sdsone_apex/flat/. (2011).

[28] Soft Byte LTD. 2012. DesignaKnit 8. [Online]. Available from: https://softbyte.co.uk/designaknit.htm. (2012).

[29] David J Spencer. 2001. *Knitting technology: a comprehensive handbook and practical guide*. Vol. 16. Crc Press.

[30] Lesley Stanfield and Melody Griffiths. 2010. *The Essential Stitch Collection*. The Reader's Digest Association, Inc.

[31] Stoll. 2011. M1Plus pattern software. [Online]. Available from: http://www.stoll.com/stoll_software_solutions_en_4/pattern_software_m1plus/3_1. (2011).

[32] Linda Suda. 2013. Bulky Waffle Hat pattern by Linda Suda. (Nov. 2013). https://www.ravelry.com/patterns/library/bulky-waffle-hat

[33] Kui Wu, Hannah Swan, and Cem Yuksel. 2019. Knittable Stitch Meshes. *ACM Trans. Graph.* 38, 1, Article 10 (Jan. 2019), 13 pages. DOI: http://dx.doi.org/10.1145/3292481