Exploring Self-Embedded Knitting Programs with Twine

Amy Zhu
amyzhu@cs.washington.edu
University of Washington
USA

Adriana Schulz
adriana@cs.washington.edu
University of Washington
USA

Zachary Tatlock
ztatlock@cs.washington.edu
University of Washington
USA

Abstract

We examine how we might explicitly embed the intricate details of the fabrication process in the design of an object; the goal is for the programs that manufacture the object to also produce themselves within the object. We highlight how concretizing the design process of an object in the real object can help reconstruct items and remind us of the reality that all objects must be manufactured, incurring labour and environmental costs. By drawing inspiration from self-reproducing programs, we outline a new self-decoding language design centred around quines for knitting, a versatile technique in fabric construction, with both historical significance and recent advances in programmable whole-garment machines for their manufacture. We show some preliminary results of using this language design to create knitted quines, and discuss how this interesting question might be further advanced.

CCS Concepts: **• Software and its engineering → Domain specific languages**; **• Applied computing**;

Keywords: knitting, fabrication, embedded information, quines

ACM Reference Format:
Amy Zhu, Adriana Schulz, and Zachary Tatlock. 2023. Exploring Self-Embedded Knitting Programs with Twine. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design (FARM '23), September 8, 2023, Seattle, WA, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3609023.3609805

To err is human, to recreate, quine.

1 Introduction

Every fabricated object implicitly contains aspects of its fabrication program. Remnants of manufacturing processes are embedded in weave patterns, seams, joins, cut ends, and other artifacts. When experts in design and manufacturing analyze a manufactured object, they can often deduce many of these aspects, from the materials used, the manufacturing processes employed, and the assembly methods involved. However, this analysis requires substantial expertise and potentially dismantling the object to reverse engineer it and develop a manufacturing plan for replication. Moreover, a comprehensive understanding of the manufacturing process is often not possible because there can be multiple ways to produce a given design. Hence, we pose the question: can we leverage concepts from self-reproducing programs to generate designs encoding their manufacturing process?

Explicitly encoding the manufacturing process also reifies the making of objects. It calls attention to the production of the things we consume, providing quotidian reminders that these items did not spontaneously appear, but had to be created: the materials grown or synthesized, the processes engineered, the items crafted and transported. In our fast-paced consumerist society, we often disregard the origins of our purchases. By showcasing manufacturability within the objects themselves, we compel people to confront these considerations, suggesting a mindful approach towards consumption, prompting reflection on the labor and environmental implications associated with our things.

In this work, we identify an interesting challenge in embedding fabrication programs explicitly within knitted items, and provide preliminary steps in exploring the space of solutions alongside a short evaluation of the progress we have made thus far.

Our proposed question is rooted in the fundamental understanding that manufacturing plans can be viewed as programs. These programs consist of sequences of instructions that unveil the process of object creation. We deliberately selected knitting as the primary subject of our investigation, given its historical significance and its status as one of the earliest and most adaptable techniques for fabric construction. Moreover, the advent of programmable whole-garment knitting machines has facilitated the exploration of knitting from a programming language analysis standpoint, resulting in numerous studies within this field of research.

Furthermore, we insist that the fabrication instructions incorporate the means to replicate themselves, such that the new object derived from these instructions also contains the necessary fabrication instructions. From this perspective, a manufacturing process that exposes itself within the generated object is like a program that produces a replica of

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

FARM '23, September 8, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0295-2/23/09.
https://doi.org/10.1145/3609023.3609805

its own source code. This problem statement may recall a classic programming problem: that of writing a quine.

Interesting quines already present a challenge to write in general-purpose programming languages with general-purpose tools. Instead, we posit that for the purpose of producing specific quines easily and extensibly, this "self-printing information" must be embedded into the semantics of the language itself.

Our insights are twofold. First, we view the process of including the fabrication instructions for an object as *embedding the fabrication program within the fabrication program*, rather than including them as a layer on top or another processing step, enabling the worldview that these can be expressed as quines. Second, to enable the creation of many different knit quine programs, we should build a language that has self-production as a first-class citizen. We design such a language, called Twine, and use it to write and create some knitted quines. We identify key challenges within this problem space and describe what future work would be necessary to fully do justice to this interesting problem.

Our contributions serve as a prototype that begins to explore the problem space, not the final word on this subject. Rather than thinking of this work as solving the problem of self-embedded fabrication programs, we hope this paper opens avenues to more interesting and complete insights and implementations to come.

2 Related work

2.1 Fabricating embedded information

There has been a rich interest in fabricating items with embedded information, of which we provide a very brief assessment. Some lines of work find ways to either invisibly encode directions to external information on objects such as 3D-printed QR codes only visible to an IR camera [3], visibly render barcodes for 3D printing [8], or explore how to embed designs into 3D printed objects revealed through IR imaging or thermal conductivity [6]. Other, more subtle ways to embed information can also be powerful. [2] demonstrates the use of settings as "signatures" by changing slicing parameters to identify the manufacturing printer. [1] shows another watermarking mechanism through slight variations in layer thickness. We refer readers interested in a more in-depth exploration of the space to the papers cited above.

Perhaps most relevantly, [7] demonstrates how a DNA-inspired matrix of nano glass beads can be used as a material for 3D printing. The authors show how a 3D-printed Stanford bunny can be disassembled, sequenced, and used to replicate itself, doing this five times in total.

These works often aim to embed arbitrary information, which may be far larger than the object itself, or redirect the user to another resource. In our work, we aim to embed only the program producing the fabrication object directly into the object. Importantly, we do this within the domain of knitting.

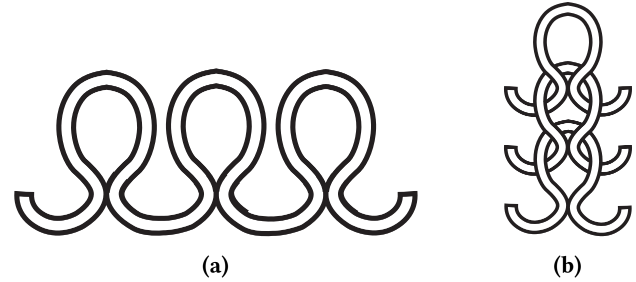

Figure 1. An illustration of knits. (A) A row of unsupported loops. These loops would be considered neighbors. (B) A wale of loops, showing the stability of loops pulled through loops. Notice how the legs of the yarn in the middle loop can no longer be pulled flat.

2.2 Knitting languages

Computational knitting is a rich field with advances from graphics, HCI, and programming languages. There are several notable examples of domain-specific languages developed to describe knitting. KnitSpeak [5] is a language to describe knitting patterns, particularly textures, which draws inspiration from how knitters typically write patterns. Knitout, first described in [9], is an abstract assembly language that encodes knitting machine operations.

3 Knitting background

Knitted fabric is composed, fundamentally, by loops on a connected yarn. A loop on its own is not stable, but the process of knitting pulls a loop through another loop, thus stabilizing it (Figure 1). We describe such a relationship as parentage: the old loop that was pulled through is the parent of the new loop. Following the link from parent to child along the knit object produces *wales*. Loops that are neighbors, i.e. next to each other on the yarn, are typically part of the same *course* (unless they are also parent and child).

The process of generating knitted items is regular, with instructions for what operations to perform and which loops to perform them on, similar to an assembly language.

4 Desiderata and Definitions

The design space of such self-decoding languages is vast, and one could imagine many different solutions with different properties. With an eye towards designing languages usable for adoption in all knitted objects, we have identified several properties as desiderata, some in tension with one another:

1. **Compression.** The act of reconstructing the program should be less tedious than transcribing every individual stitch.

2. **Robustness.** Imagine that some part of the fabric is destroyed. Will we still be able to recover the instructions? How many instructions will be affected? For example, extremely compressed programs might make losing a piece of the knit object catastrophic.
3. **Locality.** An instruction should somehow be displayed and recoverable close to the piece of the object it creates. If we are decoding a sweater with its embedded fabrication instructions, having the sleeve contain all of its fabrication instructions and the body contain its fabrication instructions makes tasks like editing or remixing patterns more straightforward. We also believe there is some amount of elegance in the idea that an instruction can be encoded in the result of its instruction.
4. **Decodability.** The language should offer a way to decode a pattern from a physical object that is easier and clearer than reconstructing the program through expert knowledge. For example, identifying where a pattern begins when it has been knit in-the-round can be done by close examination and the information that knitting is helical, but this is a tedious and uncertain process. The language should avoid being abstruse, so that checking that a decoding matches a specific object is possible, and potentially even enable the error-correction of any "flipped bits".
5. **Expressiveness.** The fact that the object must be a quine should not restrict the space of things we are able to knit, and users should be allowed freedom in making a wide array of design choices.

Very importantly, all quined knitting programs, using the following definitions,

$$\texttt{(Encode)}\ \ [\![\ ]\!] : \mathsf{Prog} \rightarrow \mathsf{Fabric}$$
$$\texttt{(Decode)}\ \ (\!|\ |\!) : \mathsf{Fabric} \rightarrow \mathsf{Prog}$$

must fulfill this property:

$$\forall p \in \mathsf{Prog}. p = (\!|[\![p]\!]|\!)$$

which says that all programs encoded in the fabric can be decoded from the fabric into the same program.

5 A Prototype

We propose a prototype language, Twine, that makes it simple to design knitted quines. In Twine, the progam is encoded within the fabric, and can be decoded from the fabric. A Twine program is then interpreted, where the interpreter represents the process of compiling the program to knitting instructions and knitting it. The program is then re-extracted from the knitted item by observing its colour pattern, which can then be re-interpreted to produce another knitted item.

We choose to use colours as the front-end for the language, reasoning that it is relatively easy to distinguish them. Each stitch has a colour; for example, a red stitch represents one instruction. To bootstrap the program, a user can begin from any valid knitted object, and generate a quine from it by assigning the correct colours to each stitch, preserving its initial validity.

One idea we originally had for creating self-replicating knits is to try and embed the proprietary low-level visual programming language from Shima Seiki, Knit Paint, as colourwork onto a knit object such that the Knit Paint program, when executed, fabricated that object. However, there is an difficulty in that in Knit Paint programs, changing colours means adding a new row to the program, thus expanding the program beyond any hope of making it self-contained. Instead, we decided that these colours shhould be parsed and translated into a quine-embedding language, which is then compiled into knitting instructions.

One language design idea could be, then, for each colour to describe its own method of manufacture: e.g. a red stitch might mean "purl with red", and a blue stitch might mean "knit with blue". However, because counting each individual stitch is cumbersome, we choose to incorporate a more idiomatic "repeat" strategy instead, which describes how many times to repeat a stitch. Each course begins with the specification for how to knit all the stitches in that course, at which point the next course begins and new colour instructions are rendered. When reading the program from the fabric, the end of each course is additionally signalled by a "switch direction" instruction, which hints to the user to start reading in the other direction in the course below. Currently, our program only supports repeating simple knit stitches, but in the future it will be important to support arbitrary repeating patterns of stitches and a variety of operations.

We present the syntax of the Twine frontend in Figure 3, translation from frontend to Twine IR in Figure 5, syntax of Twine IR in Figure 4, and semantics of Twine IR in Figure 6.

The target language of the Twine compiler is Knitout [9], which is an abstract assembly language over knitting machines. Each knitout statement is a command for the knitting machine to perform a single meaningful operation, such as knit, drop from needle, transfer from one bed to another at certain needle, or move a yarn carrier to a specific location. Here, we focus on the "knit" command, `knit(dir, needle, yarn)`, which takes a knitting direction, needle, and yarn as parameters, and makes one loop at that needle in that direction.

5.1 Discussion

Our language makes the following trade-offs in the desiderata landscape, as described in section 4. It is very local, distributing the fabrication instructions throughout the object, and keeping each course's fabrication instructions independent from the next, at the expense of further compression. A more compressed representation could, for example, specify when whole courses are repeated. We believe our design also strikes reasonable levels of robustness and expressiveness.

With instructions at only the beginning of each course, we are able to knit a large variety of shapes, though some design choices are constrained by the row-to-row decoding. On the other hand, a densely-encoded area that contains e.g. the whole item's worth of instructions in the beginning of the knit means the reconstruction fails catastrophically if that area is ruined; of course, such an encoding could allow even more design freedom in the rest of the garment. With respect to decodability, we chose colours as obvious and striking features that are easy to read, and each stitch as a unit of colour demarcates each instruction simply. However, the reader still needs to be able to distinguish individual stitches and their courses.

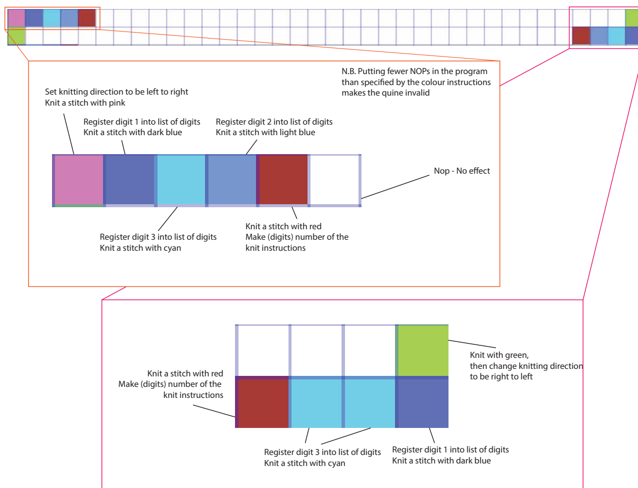

Figure 2. A visual decoding of a knit quine in the Twine frontend.

```
Program ::= Instr*
Instr ::= SetDirL2RKnit
        | SwitchDir
        | Base4Num0
        | Base4Num1
        | Base4Num2
        | Base4Num3
        | Repeat
        | Nop
```

Figure 3. Syntax of Twine frontend.

We believe that it should be possible to generate quines for knitting programs that use knits, purls, and short rows. Note that each such knitting program has only one Twine program that represents it, and each Twine program represents a unique knitted item, removing all ambiguity for decodability purposes. Each knitted item has a distinctive look imparted by the Twine self-decoding system.

```
IR ::= Op*
Op ::= l \leftarrow v
     |  knit(Colour, $\mathbb{N}$)
l ::= dir | digits
v ::= + | - | dir | -v
    | 0 | 1 | 2 | 3 | empty
Colour ::= 1 | 2 | 3 | 4 | 5
         | 6 | 7 | 8 | 9 | 10
```

Figure 4. Syntax of Twine IR.

```
lower : prog → IR
lower = flatmap lowerInstr
lowerInstr : Instr → IR

lowerInstr(SetDirL2RKnit) =
    dir ← +; knit(1, 1)
lowerInstr(SwitchDir) =
    dir ← - dir; knit(2, 1)
lowerInstr(Base4Num0) =
    digits ← 0; knit(3, 1)
lowerInstr(Base4Num1) =
    digits ← 1; knit(4, 1)
lowerInstr(Base4Num2) =
    digits ← 2; knit(5, 1)
lowerInstr(Base4Num3) =
    digits ← 3; knit(6, 1)
lowerInstr(ExecuteRepeat) =
    knit(7, 1); knit(8, to_int(digits));
    digits ← empty
lowerInstr(Nop) = ;
```

Figure 5. Lowering step from Twine frontend to Twine IR.

We considered supporting increases and decreases, but encountered increased constraints. Users should not be asked to to read colours off stacked stitches, as the stitch colour and order are obscured, complicating the process of reconstructing the program. We could also constrain the instructions to only knit stitches, which could be future work, but this accommodation makes feasible knits difficult to characterize. Further work is needed to expand the quine language and space of knitted quines, possibly using another encoding.

6 Compiler pipeline

We have implemented a Twine language interpreter in Python. Users can construct Twine programs (as one flat array of colours) and interpret them to get a list of knitting instructions.

As metadata, we insert two rows of each colour at the beginning of each knit piece. These should not be included

$$\textsc{SetDirLeft}\ \frac{\sigma' = \sigma[dir \mapsto -]}{\langle dir \leftarrow + :: P, \sigma, F\rangle \longrightarrow \langle P, \sigma', F\rangle}$$

$$\textsc{SwitchDirLeft}\ \frac{\sigma(dir) = - \qquad \sigma' = \sigma[dir \mapsto +]}{\langle (dir \leftarrow -dir) :: P, \sigma, F\rangle \longrightarrow \langle P, \sigma', F\rangle}$$

$$\textsc{SwitchDirRight}\ \frac{\sigma(dir) = + \qquad \sigma' = \sigma[dir \mapsto -]}{\langle (dir \leftarrow -dir) :: P, \sigma, F\rangle \longrightarrow \langle P, \sigma', F\rangle}$$

$$\textsc{Knit}\ \frac{\begin{array}{cc} d = \sigma(dir) & n = \sigma(needle) \\ F' = F :: knit_{ko}(n, d, c) & \sigma' = \sigma[needle \mapsto d(n,1)] \end{array}}{\langle knit(c, 0) :: P, \sigma, F\rangle \longrightarrow \langle P, \sigma, F'\rangle}$$

$$\textsc{KnitMultiple}\ \frac{\begin{array}{cc} \multicolumn{2}{c}{F' = F :: knit_{ko}(n, d, c)} \\ n = \sigma(needle) & \sigma' = \sigma[needle \mapsto d(n,1) \\ r' = r - 1 & P' = knit(c, r') :: P \end{array}}{\langle knit(c, r) :: P, \sigma, F\rangle \longrightarrow \langle P', \sigma, F'\rangle}$$

Figure 6. Twine semantics. Here, P is the Twine program being executed, σ is the environment wherein our machine state is held, and F is the fabric being produced (i.e. the Knitout knitting program. $knit_{ko}$ represents the Knitout instruction `knit` being emitted rather than the Twine IR terminal.

```
to_int(digits) =
            digits.reduce(
             lambda acc, digit, idx:
              acc + (digit *
               pow(4, len(digits) - 1 - idx)),
             0)
```

Figure 7. Helper function for Twine semantics, which converts a list of base four digits into a (base 10) integer.

in the program reconstruction, and they help users identify which colour represents which command and to mitigate potential fabrication pitfalls.

We compile to doubleknit jacquard, which we have empirically found to have several advantages. Doubleknit jacquard makes it possible to easily and stably fabricate colourwork changes like the ones we present here, and also enables the possibility to have the quine be on the backside of a knitted item (as both sides may have different colour patterns). Items knit with this colourwork style are also naturally quite flat. in the reverse (have the instructions on the inside), naturally flat. This choice unfortunately does lead to the produced knitted items sometimes having different properties from their non-jacquard counterparts, as seen in section 7.

Items were then compiled to the Knitout assembly language as described in [9], then KnitPaint's .dat format, then to Shima Seiki's .000 machine code, and finally knit on a 7-gauge Shima Seiki SWG091N2.

7 Evaluation

As a first illustration, we created a sample knitted square (Figure 8) through the quine language. The square is 35 stitches wide and starts at the top left, where the small yellow stitch, `SetDirL2RKnit`, sets the knitting direction. Note that to reconstruct the program, the knitting direction is also the reading direction, which is necessary to ensure the quine behaviour of the program. The first few stitches after (values `Base4Num1, Base4Num3, Base4Num2`) then define how many stitches to knit when `ExecuteRepeat` is encountered (30). The orange stitch denotes `ExecuteRepeat` and the blue stitches are `Nop`. Then, at the right side of that course, bright red (`KnitSwitchDir`) knits and sets the machine direction to right-to-left. We continue to read right-to-left now, starting from directly below the bright red stitch. This square example demonstrates how Twine works for a very simple shape and accordingly simple program.

Figure 8. The final knitted square. The first rows are the metadata stripes, setting the order of the colours used. To illustrate, because yellow is first, we see that it is the first instruction in the Twine frontend, `SetDirL2RKnit` and because blue is last, we see that it is the last instruction, `Nop`. After the 8 bands of colour, the first yellow stitch on the left is `SetDirL2RKnit` instruction.

Second, we wanted to demonstrate our language embedded in a functional object. As our tool supports short-row shaping, we adapted a pattern for a short-row hat [4] into our quine language format. The hat pattern was generated using the pseudocode in Figure 10, then translated into our

Figure 9. The Twine pattern for the square in Figure 8, as explained in Figure 2.

```
cast on 76
for repeat in 0..5 {
    for (let i = 72; i >= 40; i -= 1) {
        knit from 0 to i
        knit from i to 0
    endfor
endfor
```

Figure 10. The basic knitting instructions for the short-row hat.

Figure 11. A shapely version of the knitted hat, formed into a cone.

quine language. Unfortunately, we found that here doubleknit jacquard caused the fabric to become too thick to properly shape into a hat as the original pattern intended, as in Figure 11. We were also forced to shrink the hat pattern by over half, as the knitting pattern generated from the original pattern in Figure 10 proved too large for the knitting machine memory.

8 Limitations, Discussion, and Future Work

In this work, we present a study of what self-embedded fabrication programs for knitting might look like. Truly understanding the scope of the problem domain, and discovering a fully comprehensive solution would require overcoming some key challenges we have identified.

First, the space of all knit programs expressible and the space of all knit quines, and their relationship, still wants for a formal treatment. We would like to be able to understand what kinds of knit programs cannot be transformed into self-embedded quines, and what strategies could be employed to make it possible. Currently, Twine uses only the colour channel of knit objects to convey information, leaving the texture channel and shaping channel freely manipulable. Could we achieve richer information by incorporating semantic texture changes as well? Or perhaps in some cases it would be more visually appealing to reserve the colour channel and embed the information elsewhere.

Figure 12. On the right in blue are several versions of the knitted hat, where we had modified the pattern in an attempt to make the final product more hat-like. On the left in yellow is the expected full-size hat oracle. Note that the blue patterns typically have much less curvature from shaping.

The colourwork method of doubleknit jacquard has many benefits as described in section 6; however, its use resulted in undesirable metric changes in our evaluation, due to the thickness of the fabric. We found that patterns often had to be drastically re-developed from the original design. Figure 12 shows the process of finding a reasonably suitable set of parameters for a smaller hat pattern. Being able to update these patterns parameters with informed guidance, or being able to ensure the validity of a quine translation, could be one solution to this problem.

Alternatively, we could find a new colourwork method to stably produce individual stitches of colour across the knit object without greatly affecting the overall knit properties, which would facilitate wide applicability. One example could

be duplicate stitch — stitching over the existing loop with a new colour — but this is not scalable.

Also, as discussed in section 5, Twine does not yet support any operations beyond short rows. Further work is needed to support increases and decreases, after which a more thorough evaluation will be needed. Another target would be supporting stitch patterns of varying lengths with different stitch types. For example, if a user wanted to write a rib pattern, they would want to specify that the pattern (knit, purl) is repeated however many times.

Related to both of the latter challenges, another boon would be using some type of simulation or rigorous experiments to understand the effect that certain stitch types or colourwork strategies have on the overall visual saliency of knit stitch colours. For example, this data could be used to predict whether or not a three-stitch decrease can be reasonably decoded. Such feedback would be useful for tightening the iteration loop for both the language design and pattern design (such as in the hat case described above).

Finally, program decoding is currently reliant on human extraction of the program from knit objects. Though we believe this to be acceptable, and possibly the most accurate method at the moment, any of widespread adoption, more subtle encoding types, or very tedious encodings would necessitate some kind of machine-extractable system.

9 Conclusion

We suggest a language inspired by programming quines, Twine, for knitting programs that enables embedding the fabrication instructions within the fabrication instructions, thereby encoding them in the final knitted object. Our insight for making this possible is that the language itself should feature self-embedding as a first class citizen, which we implement. We identify several key desiderata that any such language should be evaluated against. Our work is a first step towards fully enabling embedded fabrication programs within knitted objects, which is itself a first step towards the idea that all objects will contain their fabrication instructions, and that they might do so within the fabrication program itself. One day, we may be able to recreate arbitrary objects-in-the-wild, and that each will be a reminder to us of the physical making of these objects.

Acknowledgments

We would like to thank Chandrakana Nandi for help iterating on the deign of Twine, Anjali Pal and James Yoo for critique on early and late drafts, and other members of the UW PLSE and GRAIL labs for in-depth discussions. We are very appreciative of the thoughtful, thorough, and insightful feedback from the anonymous reviewers. Finally, we would like to thank John Leo for his enthusiastic encouragement. This work was funded by NSF 2017927.

References

[1] Arnaud Delmotte, Kenichiro Tanaka, Hiroyuki Kubo, Takuya Funatomi, and Yasuhiro Mukaigawa. 2020. Blind Watermarking for 3-D Printed Objects by Locally Modifying Layer Thickness. *IEEE Transactions on Multimedia* 22, 11 (2020), 2780–2791. https://doi.org/10.1109/TMM.2019.2962306

[2] Mustafa Doga Dogan, Faraz Faruqi, Andrew Day Churchill, Kenneth Friedman, Leon Cheng, Sriram Subramanian, and Stefanie Mueller. 2020. G-ID: Identifying 3D Prints Using Slicing Parameters. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376202

[3] Mustafa Doga Dogan, Ahmad Taka, Michael Lu, Yunyi Zhu, Akshat Kumar, Aakar Gupta, and Stefanie Mueller. 2022. InfraredTags: Embedding Invisible AR Markers and Barcodes Using Low-Cost, Infrared-Based 3D Printing and Imaging Tools. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 269, 12 pages. https://doi.org/10.1145/3491102.3501951

[4] Brooke T Higgins. 2005. Tychus. https://knitty.com/ISSUEsummer05/PATTtychus.html

[5] Megan Hofmann, Lea Albaugh, Ticha Sethapakadi, Jessica Hodgins, Scott E. Hudson, James McCann, and Jennifer Mankoff. 2019. KnitPicking Textures: Programming and Modifying Complex Knitted Textures for Machine and Hand Knitting. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) *(UIST '19)*. Association for Computing Machinery, New York, NY, USA, 5–16. https://doi.org/10.1145/3332165.3347886

[6] Weiwei Jiang, Chaofan Wang, Zhanna Sarsenbayeva, Andrew Irlitti, Jarrod Knibbe, Tilman Dingler, Jorge Gonçalves, and Vassilis Kostakos. 2021. InfoPrint: Embedding Information into 3D Printed Objects. *ArXiv* abs/2112.00189 (2021).

[7] Julian Koch, Silvan Gantenbein, Kunal Masania, Wendelin J. Stark, Yaniv Erlich, and Robert N. Grass. 2020. A DNA-of-things storage architecture to create materials with embedded memory. *Nature Biotechnology* 38, 1 (01 Jan 2020), 39–43. https://doi.org/10.1038/s41587-019-0356-z

[8] Henrique Teles Maia, Dingzeyu Li, Yuan Yang, and Changxi Zheng. 2019. LayerCode: Optical Barcodes for 3D Printed Shapes. *ACM Trans. Graph.* 38, 4, Article 112 (jul 2019), 14 pages. https://doi.org/10.1145/3306346.3322960

[9] 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, Article 49 (jul 2016), 11 pages. https://doi.org/10.1145/2897824.2925940

Received 2023-06-01; accepted 2023-07-01