

Machine In The Stack: CNC Machine Motion Programming at Execution Time

Emilie Yu

emyu@ucsb.edu

University of California Santa Barbara
Santa Barbara, California, USA

Taneesha Panda

taneeshapanda@ucsb.edu

University of California Santa Barbara
Santa Barbara, California, USA

Benjamin F Ancho

benjaminancho@ucsb.edu

University of California Santa Barbara
Santa Barbara, California, USA

Jennifer Jacobs

jmjacobs@ucsb.edu

University of California Santa Barbara
Santa Barbara, California, USA

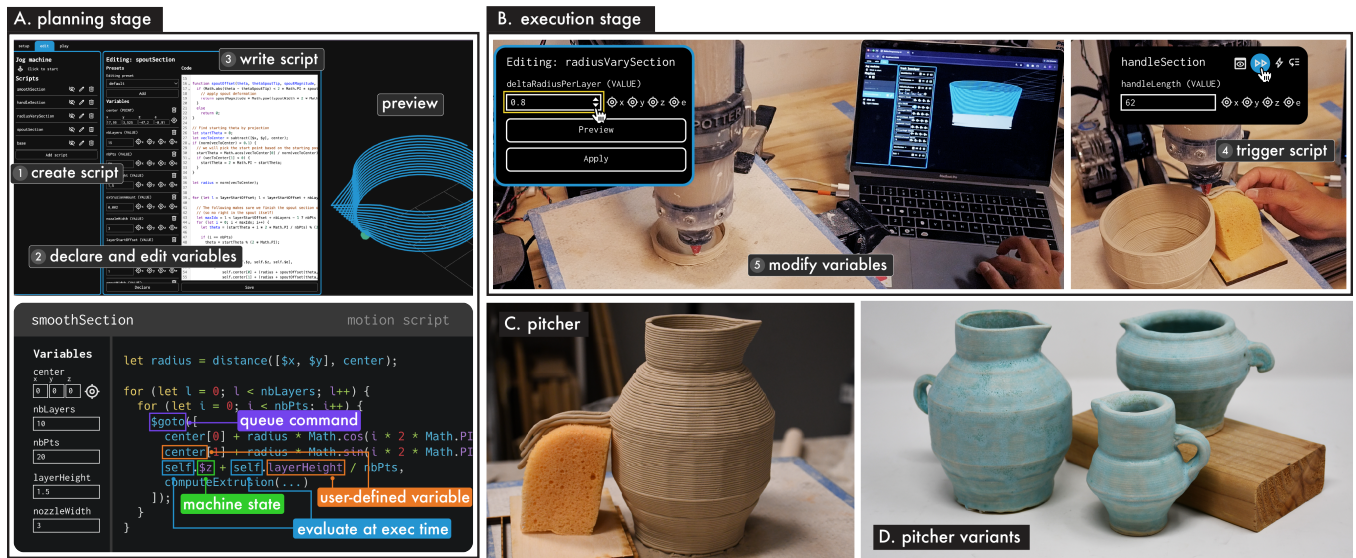


Figure 1: We developed MINTS, a system to support machine motion programming at the planning stage and at the execution stage. A) Users write motion scripts to plan motion as a queue of expressions, that can reference both user-defined variables and machine state. B) When executed, a motion script forms a queue of commands described by the expressions, and each expression is evaluated into linear move coordinates just before it is executed by the machine, allowing users to edit variables during execution. Motion scripts can be triggered at any point, either canceling or interrupting current motion. C, D) We used the system to implement six demonstrative workflows, including a program to fabricate pitchers with clay 3D printing.

Abstract

To establish new workflows, digital fabrication practitioners repeatedly iterate across machining parameters, material properties, and toolpath geometry. This process is hindered by computer numerical control (CNC) machine code, which consists of precompiled static instructions, uploaded in a single batch for automated execution. To support digital fabrication workflow development, we created MINTS, a programming tool that allows the operator to describe

machine instructions as dynamic expressions that can be modified during CNC execution. MINTS comprises a development environment for authoring dynamic machine behavior and a runtime to execute machine motion instructions while allowing modification of execution order and expression variables. We demonstrate how MINTS enables the planning and execution of dynamic CNC operations through six sample workflows using a pen plotter, a clay 3D printer, and a user-modified plastic printer. The resulting artifacts provide evidence for how structured real-time modification of automated machine operations can extend the digital fabrication design space.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

SCF '26, Tokyo, Japan

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2821-1/2026/12

<https://doi.org/10.1145/3828611.3845650>

CCS Concepts

• Human-centered computing → Interactive systems and tools.

Keywords

Digital Fabrication, Interactive Fabrication, End-User Programming

ACM Reference Format:

Emilie Yu, Benjamin F Ancho, Taneesha Panda, and Jennifer Jacobs. 2026. Machine In The Stack: CNC Machine Motion Programming at Execution Time. In *Proceedings of the 11th ACM Symposium on Computational Fabrication (SCF '26), December 06–08, 2026, Tokyo, Japan*. ACM, New York, NY, USA, 20 pages. <https://doi.org/10.1145/3828611.3845650>

1 Introduction

Digital fabrication workflows supported by CNC machines such as 3D printers, laser cutters, or milling machines enable people to plan a fabrication process entirely in the digital space, using computer-aided design (CAD) software to model the desired part, and computer-aided manufacturing (CAM) software to define machine instructions to fabricate the part. Current CAD and CAM software tools allow practitioners to leverage powerful computational methods at the *planning* phase, for instance to create intricate parametric patterns [Bourgault and Jacobs 2024; Bourgault et al. 2023; Magrisso et al. 2018], or plan complex machine toolpaths [Mahdavi-Amiri et al. 2020; Zhong et al. 2023a]. At the *execution* phase—when the machine physically moves and fabricates the design—interactive software tools are, for the most part, absent. The machine operator has little to no agency in the fabrication process once execution starts, beyond tuning global parameters like extrusion rate, or deciding to pause, or cancel the current job. The absence of meaningful operator control during fabrication is at odds with the nature of digital fabrication; the success of fabrication workflows heavily depends on understanding and accounting for specific constraints that arise from the physical machine, environment, and material [Hirsch et al. 2023]. Committing to a fabrication plan prior to execution often leads to failures that a practitioner can only observe after executing machine instructions partially or completely. Designers and operators must then address issues in future iterations by modifying the instructions, for instance by altering toolpath geometry, machining parameters, and even material properties—or often some combination of all three. This iterative process is essential for practitioners to develop innovative fabrication methods that leverage non-standard materials [Bell et al. 2025; Buechley et al. 2024], make use of novel machines [He and Adar 2020; Hirose et al. 2024; Moyer et al. 2024b] and unconventional making processes [Laput et al. 2015], or produce novel products [Hirsch et al. 2023]. We identify an opportunity to explore how computational descriptions of machine motion can extend user control *across planning and into the execution phase*, to support practitioners in authoring and adjusting machine motion programs while observing machine and material behavior during execution.

Conventional CNC toolpaths are specified as a sequence of instructions, such as G-code, which are compiled at the planning phase, then uploaded or streamed to the machine for execution. Rather than representing machine motion as static *data* compiled in batch prior to execution, we propose describing machine motion with dynamic *expressions* that are evaluated synchronously during machine execution. By “expressions” we mean computational statements that can refer to real-time machine state and application variables, allowing machine motion to adapt on-the-fly during

execution, taking into account any user modification of the variables. By “evaluated during machine execution”, we mean that each machine motion command is evaluated right before the machine performs that motion.

We developed MINTS (**Machine IN The Stack**), a software tool to program machine behavior as dynamic expressions at the planning phase, and to let the user modify motion commands right up until they are sent to the machine at the execution phase. Prior work in digital fabrication machine control has focused on providing fine-grain and parametric toolpath control through procedural representations [Bourgault et al. 2023; Subbaraman and Peek 2022; Twigg-Smith and Peek 2023] and allowing operators to modify pre-defined parameters such as machine speed and extrusion during execution [Fossdal et al. 2021; Subbaraman et al. 2025]. We build on this work to contribute:

- a *development environment* that supports defining machine motion commands as dynamic expressions rather than static data,
- and a *runtime* that exposes a dynamically-generated user interface to modify machine instructions during execution.

To support describing machine motion as dynamic expressions that can be modified at execution time, MINTS contains three technical features that bridge the gap between the digital programming space and the physical motion execution space. First, we develop a software architecture that allows communicating with a CNC machine while running all machine control logic on a desktop computer (Section 5.1), supporting interactive modification of machine behavior at execution time (Section 5.4). Second, we define new syntax building on Javascript to let end-users declare a queue of motion commands as expressions that refer to machine state and user-defined variables (Section 5.2). Third, we develop a way to explicitly manage a queue of motion commands, in order to expose control over execution order to the user, and to synchronize application logic such as previews and next command execution, with physical motion execution (Section 5.3). In combination these features support user definition of machine-agnostic high- and low-level parameterizable machine operations that can be seamlessly updated at run-time, in a manner not feasible with prior interactive or parametric CAM fabrication tools. We released the source code for MINTS¹, to scaffold future efforts at building motion programming systems across both planning and execution phases.

We show the capabilities of MINTS in supporting the development of novel and experimental fabrication workflows by implementing 6 demonstrative workflows across 3 CNC machines: an Axidraw pen-plotter (Section 4), a Potterbot Micro 10 clay 3D printer (Section 6.1, Section 6.2, and Section 6.3), and a modded Ender 3 with a syringe end-effector (Section 6.4). Our results show that describing machine motion as dynamic expressions evaluated at execution time can support a broad range of exploratory and idiosyncratic workflows that blend the advantages of machine precision with the flexibility of real-time operator control.

¹<https://github.com/em-yu/mints>

2 Related work

2.1 Machine motion programming systems

In the typical computer-aided design (CAD) workflow, the user provides a target geometry, and machine toolpaths are automatically generated by an algorithm. Automatic toolpathing algorithms have been developed in order to provide to the end-user an efficient and reliable fixed process from design specification to physical parts [Llach 2015]. Given a target 3D shape designed in CAD, prior work has proposed methods to generate machine toolpaths to approximate that shape in different fabrication techniques and materials including additive manufacturing [Chermain et al. 2025; Livesu et al. 2017; Zhong et al. 2023a], molding [Alderighi et al. 2022], multi-axis CNC milling [Ganeshan et al. 2025; Mahdavi-Amiri et al. 2020], hot-wire foam cutting [Duenser et al. 2020], and hybrid workflows [Zhong et al. 2023b]. Such computational systems excel in efficiently managing mass-manufacturing efforts [Noble 2017], but they are not well suited for developing novel digital fabrication techniques. Empirical studies of professional digital fabrication practice showed that practitioners develop idiosyncratic workflows and software tools that afford a high level of control over machine toolpath, in order to explore specific manufacturing goals that may be unsupported by standard toolpathing software [Hirsch et al. 2023; Twigg-Smith et al. 2021]. For instance, by intentionally specifying machine toolpaths for a CNC milling machine, designers can create intricate surface textures that emerge from individual milling paths [Bourgault and Jacobs 2024; Hirsch et al. 2023]. In additive manufacturing, researchers have developed novel path geometries and leveraged intentional under- or over-extrusions to develop new materials from plastic deposition [Forman et al. 2020; Laput et al. 2015; Takahashi and Miyashita 2017], improve shell wall thickness [Friedman-Gerlicz et al. 2024], program shape-changing behavior [Jourdan et al. 2023] or create surface appearance effects [Chermain et al. 2023]. Exposing methods for end-users to control machine behavior beyond defining a target 3D geometry is thus essential in both creative and experimental use cases of digital fabrication.

Prior work in human-computer interaction and digital fabrication research has explored how to support users in directly authoring machine toolpaths. Practitioners have developed software tools that allow direct manipulation of the toolpath geometry, for instance Blender add-on *Nozzleboss* [Claywoven 2026; Heinz-Loepmeier 2026] allows building and editing a toolpath by moving points or applying procedural effects. Direct manipulation interfaces [Hutchins et al. 1986] can support toolpath authoring: building a clay printing toolpath using the interaction of sketching individual layers [Frost et al. 2024], deforming layers through 3D manipulation [Passananti et al. 2026], or painting textural effects on the surface [Kaplan et al. 2025]. Such interfaces can facilitate expressive authoring for people who may be highly familiar with a craft or fabrication practice, but novices in machine toolpath programming [Frost et al. 2024; Jasper 2026], and can cast toolpath authoring as a skilled embodied practice [Passananti et al. 2026].

To explore a wider design space unrestricted by predefined system affordances, artists and designers have used symbolic programming or node-based programming. Pen-plotting artists commonly use programming to generate the design to be drawn, and the pen-plotting community has developed software tools and utilities to

facilitate artwork design and iteration [DesLauriers 2026; Reinder Nijhoff 2018; Twigg-Smith et al. 2021]. In *CoilCAM*, a node-based programming environment supports exploration of a variety of forms and textures in clay 3D printing [Bourgault et al. 2023], and in *Vespidae* node-based programming is used to orchestrate custom CAD/CAM workflows [Fossdal et al. 2023]. Twigg-Smith and Peek [2023] developed *Dynamic Toolchains*, a node-based programming environment to author parametric toolchains for digital fabrication workflows. They argued that providing a software infrastructure for modular, extendable development of data-flow programs could support people in exploring a wide range of digital workflows for machine motion control. General-purpose programming frameworks have also been proposed as means to generate machine motion commands, for instance by using Excel sheets to define parametric structures in 3D printing [Gleadall 2021], or a custom “turtle-graphics” Python library to easily define the geometry of an extruder path in additive manufacturing [Pezutti-Dyer and Buechley 2022]. With *Millipath*, Bourgault et al. [2023] generate CNC milling paths through the composition of visual components that affect the carving actions. To facilitate experimental development of plastic extrusion techniques, *P5.fab* adapts programming practices from creative coding that encourage iterative development and testing [Subbaraman and Peek 2022]. Unlike other approaches which focused on generating or modifying a G-code file that needs to be uploaded to a fabrication machine before execution, *P5.fab* implements direct communication with a Marlin controller [MarlinFirmware 2011] on a Creality Ender printer, enabling shorter feedback loops between motion programming and execution. Similarly, *Verso* is a programming framework based on a computational notebook capable of dispatching a batch of motion commands directly to a Jubilee CNC machine [Vasquez et al. 2020] over USB serial [Tran O’Leary et al. 2022]. Finally, by formalizing fabrication workflow development as a programming practice, researchers have proposed tools and techniques drawn from software development, such as documenting experimental practices through computational notebooks to facilitate learning and reproducibility [Tran O’Leary et al. 2023, 2024], or workflow sharing and remixing [Subbaraman and Peek 2026].

We take inspiration from prior work by extending a general purpose programming language (Javascript) to support people in programming machine motion. Prior approaches either let users generate complete toolpaths as static G-code, or enable them to stream a sequential queue of G-code commands, optionally adding new commands to the queue when the machine idles [Subbaraman and Peek 2022; Tran O’Leary et al. 2023, 2022]. By departing from G-code which represents motion commands as static data, we contribute a representation of motion commands as dynamic expressions evaluated synchronously at execution time. Whereas *P5.fab* and *Verso* allow users to generate and execute complete fabrication programs, enabling rapid iteration on an entire workflow between fabrication runs, our approach is lower-level targeting continuous manipulation of the toolpath itself. Specifically, we enable fine-grain-manipulation of the toolpath in real-time while the machine is engaged in executing that specific toolpath.

2.2 Real-time control over machine motion

Our work extends interactive fabrication: prior work showed that real-time human control of CNC machine motion can help people experiment quickly and learn from observing material behavior [Moyer et al. 2024b; Subbaraman et al. 2025], design physical form without the intermediary of a digital representation [Mueller et al. 2019; Willis et al. 2010], design in context of existing structures [Mitterberger et al. 2022; Peng et al. 2018], lower the barriers to using digital fabrication machines [Gui et al. 2025], augment manual craft operations [Rivers et al. 2012; Tian and Paulos 2021; Tian et al. 2019; Zoran and Paradiso 2013], and integrate more embodied interactions into the design and fabrication workflow [Bourgault et al. 2025; Kim et al. 2018].

Most systems for real-time operator control over a fabrication process are developed as static machines and tools that a person can use within a pre-defined interaction space, but not necessarily modify or extend towards ends unforeseen at design time. Such systems are very effective at supporting end-users in exploring a specific design space, but it is unclear how they could be generalized to support workflows outside of their prescribed affordances and behaviors. Recognizing the need to develop more flexible user interfaces for intervention during fabrication Kim et al. [2018] introduced a middleware to expose pathways for interaction designers to define a variety of ways that a user can interact with a 3D printer. They argued that a variety of existing interaction techniques such as sketching, gestures, and direct manipulation of physical objects, can be appropriate in the context of designing for specific 3D printing applications. Our approach extends their concept of “middleware”, by providing a machine-agnostic motion programming environment for end-users to develop machine behavior in, facilitating development and testing of novel interactive fabrication applications.

Subbaraman et al. [2025] similarly extended toolpath programming system *P5.fab* to let end-users define live mappings between MIDI controller data and some 3D printing toolpath parameters (speed, extrusion factor, and deposition height). *WORM* is a system for cobot motion programming that allowed people to define cobot motion by executing repetition loops of a pre-recorded motion with programmable rigid transformations. These systems showed that integrating programmatic control over machine motion with real-time user control is useful to combine the benefits of automation (precision, repeatability) with the qualities of human observation and intervention (real-time feedback loop, expressivity of human motion). While the *P5.fab* extension and *WORM* exposed prescribed parameters for the user to modify at interaction time (respectively printing parameters, and repetition parameters), our system lets the end-user decide which parameters will be available in the UI during motion execution. Closest to our work, Fossdal et al. [2021] built a system for programming machine toolpaths within a CAD environment that lets users execute machine motion, and monitor real-time machine state. They showed that rapid turn-taking between programming, parameter adjustments, and motion execution facilitates the development of idiosyncratic workflows. We take inspiration from their architecture allowing rapid execution and machine monitoring, and extend it to allow iterative adjustments *during* fabrication, as the machine is already busy executing

a program. The key technical differences supporting that extension is that we implemented an execution queue for commands that can be modified at any point, and that we allowed the user to program motion commands that are evaluated just before motion is executed, allowing machine state and user input to determine execution behavior.

Our implementation uses the modular machine development platform *Stepdance* to interface between our programming environment and CNC machines. We use a Stepdance machine controller module to generate step and direction signal for a variety of desktop CNC machines actuated with stepper and servo motors, and extend the Stepdance Software Library to set motion targets for each motor through a USB serial connection [Moyer et al. 2026]. Our support for execution-time modification of machine behavior is enabled by Stepdance’s relative, motion control paradigm hence our selection of it as a platform. Overall, however our contribution is orthogonal to that of Stepdance. While Stepdance contributed an embedded platform for rapid prototyping of CNC machines and interfaces that mix real time and pre-programmed behavior, we contribute a development environment for CNC machine behavior planning and execution. Our demonstrative examples highlight the difference in our software contribution by showcasing complex workflows that would be difficult or impossible to implement within the Stepdance Software Library without significant effort in writing additional state management code.

Prior work has shown that it is possible to control CNC machines directly from a desktop computer, managing machine state and performing all or most of machine motion planning logic in a high-level programming language. Such approaches to machine controller development can support building highly modular machine architectures composed of both actuators and input devices [Gershenfeld et al. 2022; Read et al. 2023a,b], and provide a useful open-ended abstraction layer to scaffold the rapid prototyping of CNC machines [Peek et al. 2017]. Similarly, our software system architecture relies on the concept of virtualizing machine state [Peek et al. 2017]: we maintain a virtual model of the machine on the computer, and generate per-motor commands at a fixed rate of 1kHz, which are sent over USB Serial to a Stepdance module capable of generating corresponding step and direction signals for the motors. In addition to shortening the iteration loop during development compared to working with compiled embedded programs, working with high-level interpreted languages (e.g. Javascript) allowed us to implement MINTS based on convenient existing features from the languages, such as managing code and motion execution asynchronicity with Promises and handling injection of variables at execution time with execution context management in Javascript.

Lastly, while we develop MINTS to rely on desktop GUIs for interaction (numerical inputs, buttons) due to their ease of implementation and their scalability to any number of desired inputs, prior work shows that diverse input techniques can be useful, such as gestures [Kim et al. 2018], extended reality [Tran O’Leary et al. 2023], embodied interaction [Bourgault et al. 2025], analog inputs [Moyer et al. 2026; Tian et al. 2019], and MIDI controllers [Subbaraman and Peek 2024].

2.3 End-user programming outside of digital fabrication

In MINTS, one of the main authoring interaction consists in writing code that generates a queue of motion commands. While we acknowledge the existing challenges in terms of accessibility and literacy when relying on programming as the main interaction for authoring, we take inspiration from approaches in other creative domains where domain specific programming languages and specialized programming environments have successfully supported novice and intermediate programmers in authoring complex outcomes such as data visualizations [Bostock et al. 2011; Pollock and Satyanarayan 2026], diagrams [Pollock et al. 2024; Ye et al. 2020], visual artworks [Jacobs et al. 2018; Processing Foundation 2001], and interactive 3D experiences [Unity Technologies 2022]. We thus draw from ideas developed in neighboring fields of end-user programming and malleable software design [Borowski et al. 2022; Klokmoose et al. 2015; Litt et al. 2025]. Prior work argues that designing malleable software can support people and communities who use a given software in adapting software tools to their idiosyncratic needs [Litt et al. 2025], or develop personal forms of automation [Li et al. 2021]. In practice, researchers have developed software tools that are editable and shareable like documents [Klokmoose et al. 2015], and developed data structures and application logic to facilitate authoring of such software. In this work, we took inspiration from both the overarching goal of malleable software and from specific technical choices: similar to *Varv*, we define motion scripts as declarative data structures, containing imperative code that can be triggered upon specific events, and application data (script variables) in a data store that can be accessed from the UI to apply user modifications.

Lastly, we designed MINTS by drawing from principles for developing creativity support tools, for instance the idea of developing systems with “wide walls” [Resnick et al. 2005], and supporting a wide range of explorations and outcomes. Additionally, Li et al. [2023] have argued that software tools for artistic practices should provide flexible computational abstractions, to empower users to switch between different levels of abstraction when using the software. We seek to apply these principles typically restricted to the digital domain to the scenario of controlling the motion of a physical machine.

3 Design goals

Our overarching goal is to support practitioners in developing novel and custom workflows with CNC machines, by exposing programmatic control over machine motion in both planning and execution stages. In order to do that, we seek to give end-users a way to specify motion commands as *dynamic expressions*, that encode a planned design intent, yet leave opportunities for modifications at execution time. Fabrication workflows could then be performed by first writing programs that declare sequences of motion commands, and then executing these programs. Specifically, we define the following design goals for a system that allows machine motion programming at both planning and execution stages:

- **G1. User-defined motion behavior:** the system should support end-users in authoring programs that generate a

sequence of motion commands. Additionally, we aim at abstracting programming from machine-specific concerns, so that users can write programs for different machine architectures.

- **G2. Flexible execution order:** the system should allow users to make decisions at execution time about when and in what order motion commands are triggered.
- **G3. Context-aware motion commands:** motion commands should be able to adapt to the context of machine state and user-defined variables. Additionally, the system should let users set the value of the variables at planning and execution time, to support decision-making both before and during motion execution.

Achieving these goals within existing general-purpose programming frameworks is challenging, because the divide between physical machine and digital program introduces non-trivial problems that a programmer would have to contend with before even getting to define application-specific logic.

First, code execution and motion command completion happen asynchronously: while the code that triggers a motion command executes instantly, the motion will be executed by the machine over a longer period of time (Fig. 6B), constrained by maximum acceleration that the stepper motors can produce, and that the physical mechanisms can handle without damage. The system must gracefully handle this asynchronicity, detecting when motion commands actually finish executing to trigger logic, in order to expose control over execution order (G2) and to trigger evaluation of the next motion command (G3).

Second, the physical state of the machine needs to be known by the digital system at all times, in order to use live machine state to adapt motion commands (G3) and to manage sequential command execution (G2). Doing so without loss of generality over consumer-level CNC machines that are not equipped with sensors or tracking apparatus requires modeling physical machine state in the digital application, and managing state synchronization between program and machine.

These challenges justify the need to develop a novel programming environment and runtime that can provide a structure for end-users to build from, enabling them to focus on application-specific concerns.

4 System overview

MINTS is an integrated development environment and runtime for machine motion programming, that enables end-user development of ad-hoc CNC behaviors through motion programming in both planning and execution stages. To give an overview of the system operation, we use an example workflow of plotting a generative artwork with inks.

4.1 Example scenario: generative artwork plotting with inks

Pen-plotters have drawn interest from a large community of artists, who have extensively used the machine as a new medium for computer-generated artwork. The pen-plotting artist community is particularly invested in developing idiosyncratic workflows around this fabrication machine [Twigg-Smith et al. 2021]. Developing a

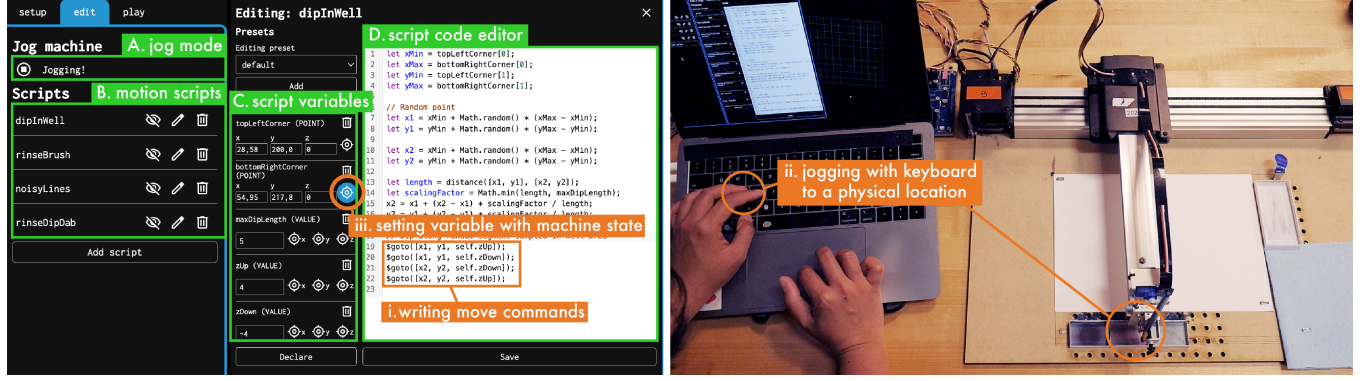


Figure 2: MINTS development environment for scripts that define machine motion. The system allows a user to write motion scripts (B), in a code editor panel (D). Machine motion is planned through calling the `$goto` function, to queue linear moves. Variables can be declared for a script (C) and set through numerical inputs. By jogging the machine (A), the user can bring the end-effector to a specific point in the physical environment (ii) and take a snapshot of that machine state to set a script variable (iii).

successful workflow for plotting with painting media that requires regular refills is non-trivial in terms of motion programming as it requires planning motion commands both to draw the design, and also to trigger refill actions (ie, go dip the brush in paint or ink).

The fictional artist in this scenario aims to develop a generative painted motif based on a Perlin noise field [Ken Perlin 2002; Patt Virra 2023]. They use the AxiDraw V3 plotter, and their painting material is composed of a paint brush, three ink wells holding a small quantity of red, blue, and green inks, a larger well to hold rinsing water, and paper towels to clean and dry the brush. They use a custom MDF board, magnets, and peg holes to hold the paper and wells secured in place during operation. Their goal in developing the generative ink painting is to achieve the following features:

- **Adaptability to physical workspace:** they want to be able to quickly change the workspace configuration to adapt to different paper sizes, ink well counts and locations.
- **Controllable generative design:** they want to retain control over some aspects of the design *during* painting execution.
- **Ink dipping based on live observation:** decisions over when the brush gets dipped, and which color it is dipped in should be taken by the artist during machine operation.

These goals are difficult to achieve with existing plotting machine control software (e.g. AxiDraw interpreter [Evil Mad Scientist 2019], *Dynamic Toolchains* [Twigg-Smith and Peek 2023]) because these systems require fully defining machine toolpaths for both dipping and drawing actions at the planning stage. These tools also require determining in advance how those paths should be interleaved. For instance, artists may cut the drawing path into segments at a fixed length interval to insert dipping actions [Amy Goodchild 2024; Licia He 2024].

In the following sections, we use this workflow as a running example to illustrate features of MINTS, describing how an artist would use the system to build a set of behaviors that satisfy the objectives listed above.

4.2 Workflow overview

With MINTS, the user proceeds in two stages: first in the planning stage they define a program relevant to their specific workflow and machine, then in the execution stage, they control program execution to fabricate an artifact (Fig. 3). In the first stage ①, the user starts by creating and editing motion scripts in MINTS programming interface (Fig. 2). A motion script is composed of imperative code with motion commands ③ (Section 4.3), as well as user-defined variables ② (Section 4.3.2). The system allows setting variables by taking snapshots of machine state (Section 4.3.3), and variables can be read at motion execution time (Section 4.3.4). In the second stage, the user opens the execution interface (Fig. 4), which allows executing motion scripts as tracks (Section 4.4), exposing different options to trigger execution with respect to the execution queue ④. The user can modify script variables prior to triggering execution, and also during track execution, affecting all subsequent commands ⑤ (Section 4.4.2).

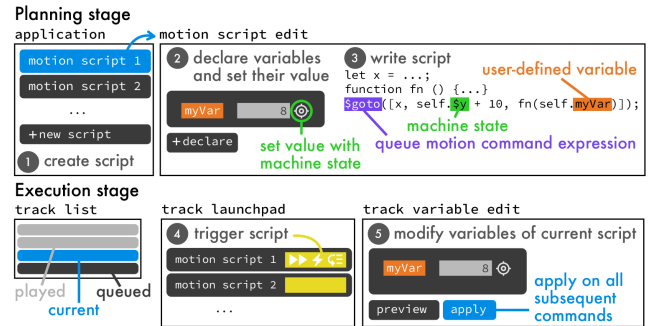


Figure 3: A fabrication workflow in MINTS is authored in two stages: the planning stage (top) and the execution stage (bottom)

4.3 Motion script authoring

4.3.1 Motion script programs. Motion scripts are blocks of imperative code written in Javascript (Fig. 2D). In MINTS, users can write multiple scripts that are saved in the system (Fig. 2B). In a script, one can programmatically queue linear move commands towards a target position (similar to G-code G1/G0 commands) by using the **\$goto(target)** function that can be called at any point within a script. Sequential **\$goto** commands are executed in order, one after the other. For instance, a simple motion script that draws a straight line across the page:

```
1 let startX = 0, endX = 200, y = 0, zUp = 6, zDown = -6;
2 $goto([startX, y, zUp]); // Bring brush to the first point
3 $goto([startX, y, zDown]); // Lower brush
4 $goto([endX, y, zDown]); // Draw line
5 $goto([endX, y, zUp]); // Lift brush
```

Note that we can describe motion target points as expressions referencing variables that exist within the script scope (in this case, constant values hard-coded in the script). The user can preview the effect of their motion script by hitting the "Save" button and seeing a preview of the 3D path that the machine end-effector would take if they were to execute the script (Fig. 1A and Fig. 8A).

4.3.2 Script variables. Each script has associated *variables*, which are declared by the user along the script, and populated with values through numeric input fields (Fig. 2C). Variable values are injected in the script at *code execution time*, making them available to logic and commands in the script. In addition to user-defined variables, scripts also have read access over machine state at code execution time through variables injected by the system (e.g. **\$x**, **\$y**, **\$z** for the pen-plotter and **\$x**, **\$y**, **\$z**, **\$e** for machines with an extrusion axis).

In the example workflow of ink painting, the artist uses script variables to create easily adaptable behaviors, parameterized by a small number of inputs. In Listing 1 we provide a simplified version of the code for the **rinseDipDab** motion script of that workflow, responsible for the commands to refill the brush with ink. Specifically, this script first rinses the brush by dragging it through the water well, then dabs a random point on a paper towel, then in an ink well, and on the towel again. Note how we use both user-defined variables such as **inkTopLeftCorner** and machine state variables **\$x** and **\$y**. The user-defined variables let the artist quickly redefine the location of physical elements if needed. The machine state at script execution time is used here to trigger the brush to be raised before the travel move (l. 16) and to command the machine to go back to the point where we paused once the dipping action is done, ready to resume the main drawing (l. 29-30).

```
1 // User-defined variables:
2 // towelTopLeftCorner, towelBottomRightCorner,
3 // inkTopLeftCorner, inkBottomRightCorner,
4 // zUp, zDown
5
6 function dipIn(ptA, ptB) {
7   // Random point
8   let x = ptA[0] + Math.random() * (ptB[0] - ptA[0]);
9   let y = ptA[1] + Math.random() * (ptB[1] - ptA[1]);
10  $goto([x, y, zUp]);
11  $goto([x, y, zDown]);
12  $goto([x, y, zUp]);
13 }
14
15 // First raise brush up at the initial point:
```

```
16 $goto([$x, $y, zUp]);
17
18 // RINSE
19 // ... drag brush through water well
20 // DAB TOWEL
21 dipIn(towelTopLeftCorner, towelBottomRightCorner);
22 // DIP INK
23 dipIn(inkTopLeftCorner, inkBottomRightCorner)
24 // DAB TOWEL
25 // ... same as above
26
27 // Finish by going back to the initial point
28 // (resuming where we paused):
29 $goto([$x, $y, zUp]);
30 $goto([$x, $y, $z]);
```

Listing 1: "Motion script **rinseDipDab** to refill with ink"

4.3.3 Jogging and setting variables. Whenever the machine is idle the user can enter a manual jogging mode, which allows them to freely move the machine in cartesian coordinates by pressing on keyboard keys (Fig. 2A). Jogging happens at interactive rates (< 1ms latency) sufficient for the operator to perform quick rough positioning of the end-effector. While manual jogging is implemented in most CNC machine controllers (e.g. [Duet3D 2026]), MINTS lets the user leverage it for more than setup and maintenance operations. They can use the current position of the machine to populate script variables by taking a snapshot of the current machine state (Fig. 2iii). The snapshot feature allows setting variables with respect to the physical machine and workspace state observed in real-time (for instance, saving the position of an ink well in variable **inkBottomRightCorner**, see Fig. 2ii).

In the example workflow of ink painting, being able to set variables by reading machine state facilitates adaptation of the motion script to a variety of different ink well locations. In the example workflow, the artist aims to make paintings on sheets of varying sizes (7x10in and 5x7in), moving ink wells around to keep them as close as possible to the page boundaries in order to minimize travel. Changing where the drawing area is, and where the ink wells are located, can be done without a single code change, through the jog and snapshot feature (Fig. 2ii and iii).

4.3.4 Motion-execution-time script variables. In order to support parameter changes during motion execution, we also allow **\$goto** commands to refer to variables at *motion execution time*, querying the variable value just at the time when an individual **\$goto** command starts, rather than upon code execution, when all commands are added to the execution queue but no motion from the script has actually been executed. Within **\$goto** commands, we allow the user to refer to variables at motion execution time by using the keyword **self**. Both user-defined variables and machine state variables can be referred to through **self**.

```
1 // User-defined variables:
2 // bottomRightCorner, topLeftCorner, zDown, zUp,
3 // nbLines, nbPtsPerLine,
4 // noiseFrequency, noiseScale, noiseOffset
5
6 let W = bottomRightCorner[0] - topLeftCorner[0];
7 let H = bottomRightCorner[1] - topLeftCorner[1];
8
9 function applyNoise(basePt, freq, scale, offset) {
10   let dispY = noise.noise(
11     freq * basePt[0] / W,
12     freq * basePt[1] / H,
13     offset);
```

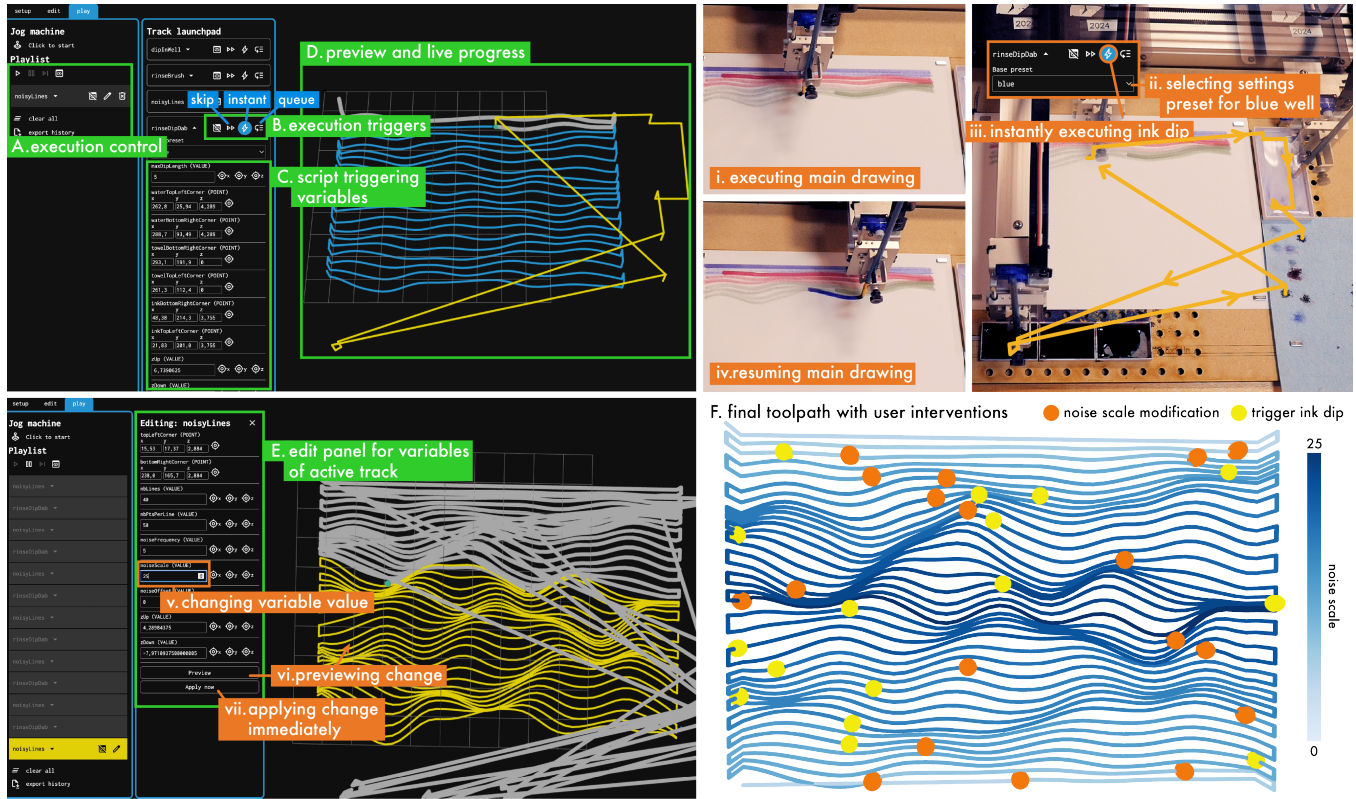


Figure 4: Runtime to trigger and control motion script execution. The system lets users execute scripts as motion tracks, that are queued in a playlist (A). Any motion script can be triggered, with different modes (B) defining how the track gets inserted in the playlist (replacing current track, inserting in current track, or adding to end of queue). The user can execute tracks (i), optionally they can modify script variables before executing (ii and iii) in a UI panel (C). When an inserted track like the ink dip action has completed, the previously active track can resume (iv). Users can also modify variables from an active track while it is being executed (E), for instance modifying the Perlin noise scale value (v) to introduce variations in the generative artwork across the page (F). The system gives a preview (vi), displaying the simulated future toolpath (yellow) if the change were to be applied. Once applied, a variable change (vii, and orange dots in F) will affect all future `$goto` commands.

```

14   return [basePt[0], basePt[1] + scale * dispY, basePt[2]];
15 }
16
17 for (let l = 0; l < nbLines; l++) {
18   let y = topLeftCorner[1] + (l + 0.5) * H / nbLines;
19   for (let i = 0; i < nbPtsPerLine; i++) {
20     let x = topLeftCorner[0] + i * W / nbPtsPerLine;
21     $goto(
22       applyNoise([x, y, zDown],
23         self.noiseFrequency, self.noiseScale, self.noiseOffset)
24     );
25     // ... handle pen up/down at line start and end
26   }
27 }

```

Listing 2: Motion script `noisyLines` to generate horizontal lines across the page displaced by an adjustable amount of Perlin noise

In Listing 2, we provide an abbreviated version of the motion script responsible for performing the main drawing of a noisy line field across the page. In this script, the artist used `self.noiseScale` in the `$goto` command in order to refer to the value of the `noiseScale`

variable at the time when the `$goto` command is executed. In Section 4.4.2, we explain how this can be used to trigger live modification of the noise scale value by the user, with changes to the amount of noise displacement taking effect immediately in subsequent queued commands. Additionally, in Listing 2 the artist uses the noise object from `three.js`², which the system makes available in script execution context too.

4.4 Motion track execution

MINTS provides a user interface to run scripts and manage execution flow and modify script variables during execution (Fig. 4). Because of the temporal nature of script execution, we refer to a motion script queued for execution as a *motion track*. The UI displays a “playlist” of played, current, and queued tracks, and exposes basic control to pause and resume execution of the playlist (Fig. 4A).

4.4.1 Track launchpad. The user interface for execution presents a launchpad to trigger any of the previously defined script as a track

²<https://threejs.org/docs/#ImprovedNoise>

(Fig. 4B), with options to add a track to the back of the queue, or to trigger a track to be executed instantly (interrupting any current motion). The “instant” trigger interrupts any active track but resumes it once the interrupting track is over, while the “skip” trigger interrupts any active track without resuming. The launchpad also exposes input fields to set the value of user-defined script variables prior to triggering track execution (Fig. 4C). To facilitate recalling commonly used sets of variables, we let users define presets: when selected, a preset automatically changes the value of multiple variables to saved values.

In the example workflow, the artist first executes the main drawing script (Listing 2) and uses the “instant” trigger mode to launch a track with the ink refill script (Listing 1) whenever they decide to refill on ink, or change color. They created presets for the ink dipping script for all three ink wells: red, green, and blue. This allows them to quickly command dipping in one of these pre-specified well locations. Over the course of the painting Fig. 4, the artist triggers 16 ink dipping actions (yellow dots in Fig. 4F).

4.4.2 Script variable modification during track execution. While a track is running, the user can open a UI panel that exposes input fields for the script variables (Fig. 4E). Using that panel, the user can change the value of script variables, affecting any `$goto` command that refers to motion-execution-time variables using the keyword `self` (e.g., Listing 2, l.23). The change in variable is applied instantly and will be taken into account for any not-yet-executed queued command for that track. This allows making live decisions about modifying the machine toolpath.

In the example workflow, the artist varies the scale of vertical displacement caused by Perlin noise by modifying the `noiseScale` variable while the track is running (Fig. 4F). Because the script in Listing 2 refers to `self.noiseScale`, any subsequent motion command will take into account the new scale value. In the demonstrative painting, the noise scale was modified 19 times (orange dots in Fig. 4F).

4.5 Visualization and utilities

The system provides previews for motion script debugging and for track execution in a 3D visualization panel where the resulting toolpath is visualized as a 3D polyline (Fig. 4D). In execution mode, the 3D scene also displays in grey the toolpath for any motion already executed in that session, to provide context. The current machine toolhead position is visualized as a green sphere.

The system provides utilities for saving a set of motion scripts and associated variables as a JSON file, and to load it into the system in another session. We also define a format to compactly describe new machine kinematics through per-axis transmission ratios in a JSON file that can be loaded by the Python backend. We created machine configurations for the Axidraw, Potterbot, and modified Ender.

5 Implementation

In this section, we first give details on the system architecture (Section 5.1), data structures and communication across the different components, then explain our three main implementation choices: how to define and execute motion command expressions (Section 5.2), how to manage a queue of commands and allow the

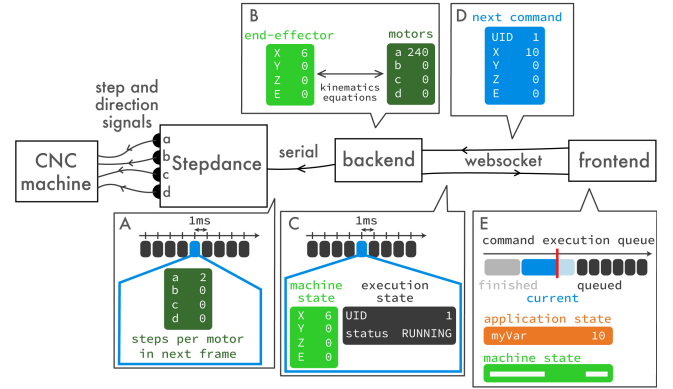


Figure 5: MINTS main data structures and communication

user to modify it at execution-time (Section 5.3), and how to compute and transmit motion commands to any CNC machine with 4 axis or less and a closed-form kinematic model (Section 5.4). Throughout this section, we refer to our design goals (Section 3) to justify implementation decisions.

5.1 System architecture

MINTS is composed of a frontend user interface, a backend server, and an embedded application running on a Stepdance machine controller board [Moyer et al. 2026].

5.1.1 Backend and machine controller. The backend server is implemented in Python, and holds machine state (Fig. 5B). It manages and stores both the machine end-effector position in world-space (in cartesian 3D or 4D coordinates for all the machines in our demonstrative examples), as well as the position of each individual motor in machine-space (in number of steps turned per motor). The backend server must implement the kinematics equation (forward and inverse kinematics) of the machine, in order for the system to reason in terms of both end-effector and motor positions, which we refer to as world space and machine space respectively. This can be provided by the end-user, or by system developers, through a set of standard kinematics models. Converting a target displacement from world space to machine space is useful to compute machine motor motion in terms of the amount of steps each motor should turn, and converting a target displacement from machine space to world space is useful to maintain an accurate digital simulation of end-effector position, and achieve our goal (G3) of letting users implement context-aware effects that take machine state into account. The backend program runs on a 1kHz loop, sending target positions in machine space for the next frame to the Stepdance machine controller over USB serial (Fig. 5A), and sending machine position, and current program status to the frontend over websocket (Fig. 5C).

5.1.2 Backend and frontend. The frontend is a Javascript application in the browser, that builds on the UI framework Svelte³ and modern Javascript features. The frontend maintains the execution queue (Fig. 5E), allowing the end-user to modify that queue during execution (G2). Motion command expressions are evaluated by the

³<https://svelte.dev/>

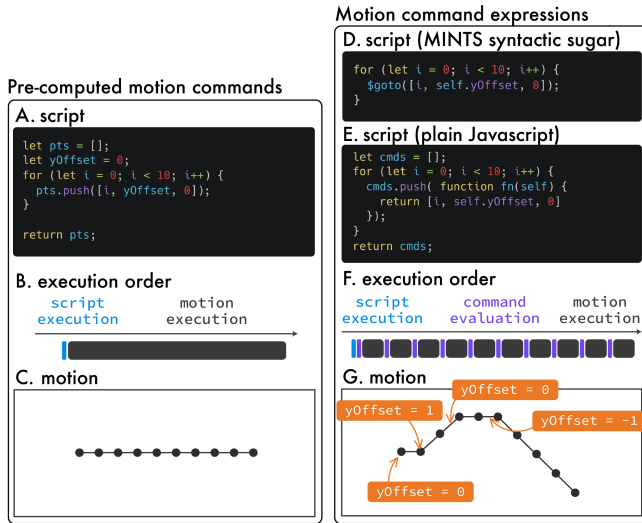


Figure 6: Pre-computed motion commands VS motion command expressions. A) With pre-computed commands, the code returns a static array of target coordinates. B) Code executes near-instantly, but motion execution of the entire list of target points is much longer. C) The variable `yoffset` cannot be changed once code execution is over. D) In MINTS motion command expressions can be defined concisely, they are then expanded into pure Javascript (E). F) Evaluation of each command takes place right before its execution, allowing changes to `yoffset` to affect the slope of the path (G).

frontend into compiled coordinates when they are triggered, and they are sent to the backend as target positions (Fig. 5D). The frontend stores a copy of live machine state and the application state—a dictionary of user-defined variables and their values (Fig. 5E). Access to application and machine state from motion command expressions is essential for context-aware motion commands (G3).

5.2 Motion command expressions

Motion command expressions let end-users describe a command with a programmatic expression (G1) that can refer to machine state and user-defined variables. The system defers the evaluation of that expression to happen right before the command is executed by the machine (Fig. 6F). This allows people to define context-aware motion commands (G3), for instance allowing changes to the variable `yoffset` to affect the motion of the machine along the Y axis (Fig. 6G).

In practice, we implemented motion command expressions as *function expressions* in Javascript⁴, that we declare and store in an array at script execution time (see syntax in Fig. 6E). Motion command expressions are evaluated right as they get dequeued from that array when the previous command in the queue completes (interleaved purple blocks in Fig. 6F). Therefore, updated values for user-defined variables like `yoffset` will be taken into account for any subsequent command.

We defined a special function syntax `$goto(...)` that acts as syntactic sugar (Fig. 6D) to encapsulate the longer form declaration of adding motion command expressions to a queue (Fig. 6E). This syntax makes scripts easier to read and write, without significantly reducing expressive power. In the frontend application, the user writes the motion script code in a rich text input field⁵ (see Fig. 2D), and once they click the Save button, the entire code text is parsed in order to expand the syntax into a pure Javascript program. Specifically, we search and replace `$goto(expression)` commands with the desugared syntax that appends a function returning the result of evaluating expression, given an input variable. We also add a line at the beginning (declaring the array of commands) and end (returning the array) of the code (Fig. 6E).

Within an expression, the user can refer to script variables at command evaluation time by using the keyword `self`, which after desugaring corresponds to the input argument for the function (Fig. 6E). At command evaluation time, the system populates this variable with the context of the motion script as a plain object: machine state is passed via the reserved keys `$x`, `$y`, `$z`, `$e`, and user-defined variables are assigned to properties of the object, with the key as the variable name, and the value as the variable value. Additionally, we also pass a snapshot of this context at script execution time (machine state and user-defined script variables), that can be referred to in the script body as well as in `$goto(...)` commands. These values are referred to simply as plain variables (e.g. `$x` in Listing 1, or `bottomRightCorner` in Listing 2). These values stay constant throughout evaluation of all commands, which is useful to define logic that should be consistent across the entire script. For instance, we want to make the machine go back to its initial state after the dipping action in the ink painting workflow (Fig. 4 iv), therefore we call `$goto($x, $y, $z)` in the last line of Listing 1, referring to machine state at script execution time.

5.3 User-control over the execution queue

Our high-level goal was to handle the asynchronicity of code execution and motion command completion, in order to expose control over execution order to the user, and to synchronize application logic with physical motion execution.

5.3.1 Asynchronous execution of script commands. When a motion script is executed, each command from its list of commands is wrapped by the system in a promise, and all the promises are chained such that they can be called sequentially, executing a motion command only once the previous promise resolves⁶.

First, when a command is triggered, the frontend sends to the backend the evaluated target position as well as a unique identifier associated with that command (Fig. 5D). The backend server then immediately starts executing a linear motion until the end-effector reaches the target point (see Section 5.4, and Appendix A). Once the frontend receives a websocket message from the backend indicating that the current command has finished running, we call the promise's resolve callback, to trigger the next command in the queue.

⁴<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function>

⁵<https://codemirror.net/>

⁶<https://javascript.info/promise-chaining>

Motion commands can be immediately interrupted, which allows pausing execution and instantaneously starting new commands even while another command is already executing (ie, as the machine is moving). When the UI interrupts a command during execution, a websocket message instructs the backend server to immediately stop sending target motion displacements associated with that motion track on its 1kHz loop (Fig. 5A).

This implementation based on promises allowed us to add not only motion commands to the queue but also general-purpose functions, as long as we wrap them in a promise. A function inserted in the queue will be called whenever the preceding promise in the chain is resolved. This feature is useful to schedule callbacks that handle UI-specific concerns, such as updating the toolpath 3D preview (Fig. 4D, grey stroke) when a command finishes executing, and notifying the system when a complete motion script is done executing.

5.3.2 Motion tracks. While the queue of commands supports executing a single motion script, our goal was to allow users to trigger many different scripts at any point during the execution phase (G2). In the execution UI, we thus exposed additional objects called “motion tracks”, that encapsulate the execution of a motion script (see Section 4.4).

The system maintains a “playlist” of tracks as an array, and keeps track of the index “play head”, corresponding to the currently playing track. When the playlist is played, the system executes the first track in the array as explained in Section 5.3, by chaining promises corresponding to the script’s motion commands. Right after, we add a general purpose function to the queue, scheduling the play head index to be incremented when the last command of the track is done, which will make the player execute the next track in the array. The UI displays the playlist, highlighting which track is currently playing, and allowing the user to open an edit panel on the current track or any queued tracks (Fig. 4E).

5.3.3 Modifying motion track playlist at any time. Equipped with the track playlist representation, the user can manipulate tracks within the playlist to modify execution order at any point. Track insertion at the end of the queue (“queue” trigger mode in Fig. 4B) is implemented by simply appending the new track to the end of the array. Skipping the current playing track at index i in the playlist and replacing it by a new track (“skip” trigger mode in Fig. 4B) is implemented by canceling all pending motion commands from the current playing track, inserting the new track in the queue at index $i + 1$ and incrementing the play bar index to start playing the new track. For the “instant” trigger mode (Fig. 4B), we proceed like for “skip”, but also re-insert the interrupted track in the playlist at index $i + 2$ (after the inserted track). We make track objects save the last executed command index, and start again from that index when they are interrupted.

5.4 Moving the machine

To allow for real-time interventions like interruptions (G2) and script variable modifications (G3), we implemented machine motion control via a Stepdance machine controller module and the

Stepdance Software Library. Stepdance embedded software provides a convenient API that abstracts the generation of synchronized high-frequency (25kHz frequency) relative motion streams across multiple axes of a CNC machine. In addition, the flexibility of Stepdance to work with many 3-to-4 axes CNC machines directly supported us in developing machine-agnostic logic (G1).

To enable users to program motion behavior without having to worry about machine-specific velocity constraints (G1), the backend server transforms user commands (move to a target world space position), into temporal series of smaller moves in machine space, over as many 1ms frames as necessary to achieve that command while attempting to match a prescribed velocity, and respecting per-axis velocity limits (see Appendix A).

To avoid execution delays due to serial communication, the desktop MINTS application sends target positions in machine space for each axis on the 1ms frame. These targets are sent through USB serial, then the embedded program splits them into linear segments on the 40us-frame. In the Stepdance Software Library we implemented this as a new Plugin.

To avoid having to send large data packets that could cause latency, we encode target motion for a single axis on one byte. We avoid quantization artifacts by normalizing the displacement values by a user-specified maximum expected number of full steps of the stepper motor per 1ms period (see Appendix A).

Jogging. We implemented jogging as a distinct command mode from the main one described above, in which the backend receives a target position from the frontend. Putting the machine in jogging mode in the UI is integrated in the execution queue by creating a command in the queue that will be resolved when the user clicks “Stop jog” or triggers another command. In jogging mode, the backend listens to the frontend for binary displacement values along each axis (key is pressed or not), for instance to jog in a diagonal XY line the frontend sends {"x": 1, "y": 1, "z": 0}. The backend then converts these relative motion targets from world-space to machine-space.

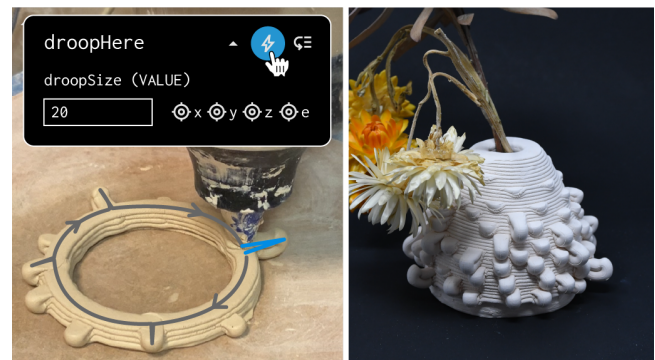


Figure 7: Vase with improvised drooping loops pattern (right). We launched a “droop here” track (left, inset) whenever we wished to add an outward segment to the toolpath (left, blue).

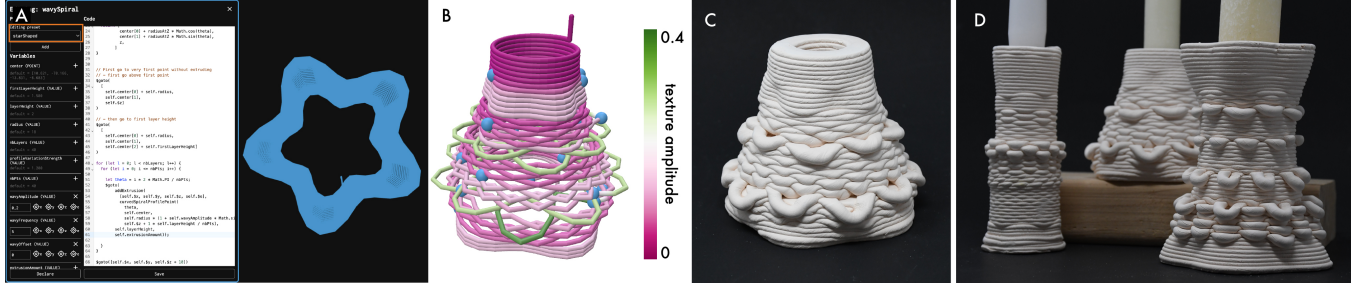


Figure 8: Program to create candle holders with a curved profile and textures. A) In the planning stage, we wrote the motion script and declared user-defined variables, using the preview screen on the right to test the code. We also created presets (orange) for the variables. B) In the execution stage, we selected different base presets and further tweaked the amplitude variable (blue spheres mark where we applied a change of variables). C, D) We created a set of three candle holders with different shapes and textures.

6 Demonstrations

We evaluated the applicability and expressive range of our system across different machines (G1) by implementing a series of 6 fabrication workflows across 3 machines: a pen-plotter, a clay 3D printer, and a slip deposition machine. These demonstrations constitute a form of evaluation of our system [Bourgault et al. 2023; Ledo et al. 2018; Twigg-Smith and Peek 2023]. The demonstrative workflows would be impossible or very challenging to implement with CAD-CAM software, existing systems for real-time machine control like Stepdance [Moyer et al. 2026], or pre-planned toolpath programming methods [Twigg-Smith and Peek 2023]. All combined, the workflows show 1) various examples of motion script programs to achieve different balances of automation and user control, 2) different approaches to real-time user intervention, including variable tweaking and execution flow modification, and 3) how MINTS can be used for a variety of off-the-shelf CNC machines and custom machine prototypes. In the following section, we describe all examples, analyze how MINTS supports our design goals, and discuss the impacts on the fabrication workflow.

6.1 Improvisational design in clay printing

Clay 3D printers are typically controlled in the same precompiled toolpath format as plastic 3D printers. Clay is a highly variable material that supports unique fabrication properties like the sagging of clay coils or the appearance of textural bumps from over-extrusion. These properties are difficult to simulate because the outcome is determined by the interplay of moisture content, clay body composition, and extrusion nozzle size, which in turn necessitates multiple iterations of toolpath programming and execution on the machine to achieve a desired outcome.

In this demonstration, we use MINTS to develop a program that generates a clay deposition toolpath on which we can add drooping in an ad-hoc manner by commanding the nozzle to move outwards by a certain distance at any point in the toolpath, before resuming the base shape. We first write a motion script generating a spiral toolpath with a curved profile, while moving the extrusion plunger down proportionally to toolpath segment length. We then write a second motion script that generates a displacement in the XY plane along the line passing through the radius. By using the “instant”

trigger mode, we can insert a “droop here” action at any point during execution of the main drawing, which lets us improvise on how the droops are spaced and scattered on the surface. We declare a variable in the motion script that controls the size of the displacement, which allows making larger or smaller droops by setting the variable value just before triggering. With this program, we fabricate a small vase with surface patterning (Fig. 7).

Setting the droop size on the fly allowed us to experiment with larger and smaller droops, and avoid making overly long droops (that would risk breaking the coil or blocking lower droops). We also chose droop sizes that gradually decreased towards the top to create a global effect. On-demand triggering of the droop program (G2) allowed us to see how a roughly evenly distributed pattern of droops would look like, with very simple logic that does not implement any algorithm for random sampling. We see such a workflow as a way to quickly test out design ideas in the physical space, before committing to more complex automated motion command generation algorithms. This approach is different from prior real-time texturization systems that rely on tuning predefined periodic texturization patterns [Moyer et al. 2024a; Subbaraman et al. 2025]. Our approach enables the insertion of an individual operator-specified feature at any arbitrary position on the toolpath during fabrication.

6.2 Mixing textural patterns

In addition to making real-time decisions to add new geometric features to a predefined toolpath, MINTS supports the paired exploration and definition of higher-level textural and geometric structures that can then be iteratively applied in a sequence determined by the operator as the piece is fabricated. To showcase this, we extended a demonstration from Stepdance (Wheelbot) [Moyer et al. 2026] by making a series of ceramics candle holders with varying shapes and textural patterns. In the planning stage, we defined one motion program that creates a spiral toolpath with a curved profile controlled via a sine function parameterized along the vertical Z axis. The textural effects are created with a sine wave parameterized along the point indices of the toolpath. This approach is common in CAM-based toolpath design [Bourgault et al. 2023]. We exposed profile wave amplitude and texture wave amplitude, frequency, and

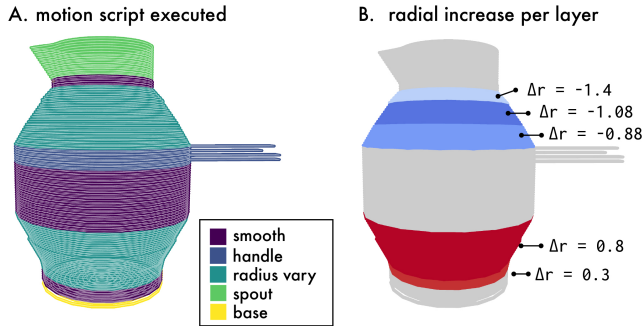


Figure 9: Pitcher program execution. A) We executed a sequence of motion scripts corresponding to the “parts” of the pitcher. Each part stacks onto the previous one. B) During the radiusVarySection script execution, we decided to change the variable `deltaRadiusPerLayer`, controlling the slope of the slanted section (red/blue colors, see labels).

offset as variables of the motion script. Prior to any fabrication, we explored the effect of different variable values by using the virtual preview. We saved combinations of variable values that we found interesting as presets, such as the star-shaped deformation (Fig. 8A). Once ready to start fabricating, we executed the motion script and opened the execution-time edit panel for the script (Fig. 4E). We chose and loaded preset values on-the-fly, previewed the effect, and either applied as is or adjusted variables further. We modified variable values such as texture wave amplitude throughout motion execution, a total of 19 times in this particular example (Fig. 8B). By executing the script multiple times and modifying the variable values while leaving the script itself unchanged, we created a set of three candle holders with varied shapes and surface pattern arrangements.

MINTS allowed us to specify a pre-defined overall form with a curved profile (G1), while enabling us to decide on-the-fly what textures to apply by modifying script variables (G3). Furthermore, the virtual toolpath preview supported us in exploring the parameter space *prior* to execution start, and the presets allowed us to easily recall parameter vectors that yield the most interesting outcomes. While the resulting artifacts are similar in look to those made with the Wheelbot [Moyer et al. 2026], the Wheelbot necessitated skilled and timely manipulation of physical UI elements to achieve complex outcomes, while the MINTS implementation allows us to rely on the planning phase (saving presets of variable combos, programming a global profile for the form) to reduce the risk of manipulation error when modifying the program in real-time during fabrication.

6.3 Pitcher as kit-of-parts

In our final clay 3D printing demonstrative workflow, we aimed to fabricate a series of pitchers, as they have a complex non-rotationally symmetric shape, and elements (spout, handle) that must be precisely placed for both functional and aesthetic purposes⁷. We programmed five motion scripts to describe different functional sections of the pitcher:

- **Base:** a spiral infill pattern to form a circular puck of compressed clay;
- **Smooth:** a cylindrical spiral path to form a straight vertical wall;
- **Varying radius:** a spiral path where the radius linearly varies along the vertical axis, forming a straight slanted wall (see Listing 3);
- **Handle:** a cylindrical spiral path with a small section that juts outwards from the main cylinder to form a long straight unsupported handle;
- **Spout:** similar to the handle, except the zone that juts out can be made wider and is formed into a triangular shape that gradually grows out from the cylindrical base over multiple layers, making it well supported.

We exposed design parameters as variables, such as the length of the handle, the width and sharpness of the spout, and the radius offset per layer for the varying radius section. In order to be able to stack sections together, the radius of a section and its first layer Z height cannot be hard-coded, as they need to match the last layer of the prior section. We thus compute the `radiusStart` variable at the beginning of each section script by reading the machine end-effector position [x , y] at script execution time (Listing 3, l. 10-11). We similarly match the angle at which we start the spiral motion, and the Z value. This allows us to seamlessly join different sections⁸. For each of the 3 pitchers, we launched tracks for the different sections in “skip” triggering mode (Fig. 1, step 4) by making decisions at execution time, based on our aesthetic and functional judgment. For instance, we placed each handle high enough to have space to curve it back onto the body and still grab it, and only triggered execution once a support material was manually placed (re: sponge) to hold the unsupported coils. For the medium-size pitcher (Fig. 1D), we decided to trigger the spout section after observing that the clay extruder tube was almost empty, in order to still form a functional (albeit shorter) pitcher. Beyond the order and timing of different sections, we also tweaked some script variables in order to change the slanted walls angle (increasing `deltaRadiusPerLayer` to make a more dramatic overhang Fig. 9B) and to vary the size of coils that form the handle (changing the handle length variable). In the large pitcher (Fig. 9), we triggered a total of 12 different sections, including changes of section and changes of the radial increase parameter within a varying radius section. After the print concluded, we formed the unsupported loose handle coils into curved handles following traditional ceramics techniques (Fig. 1C and Fig. 1D).

Dividing the program into multiple motion scripts by vertical sections of the pitcher that can be triggered sequentially (G2) made it possible to create all the functional parts with relatively simple encapsulated logic. Since motion scripts can adapt commands depending on machine state at script execution time (G3), we could program them such that they are easy to recombine in a modular fashion. This is particularly meaningful for clay 3D printing, which has tighter tolerances for overhanging geometry than plastic printing due to lack of infill and printed support material. Making structural determination during fabrication rather than in desktop CAD enables the operator to rapidly modify overall geometry to avoid part collapse while preserving the primary design intent.

⁷The pitcher as a demonstrative challenge for CNC machine motion programming also echoes the famous Utah teapot for rendering algorithms.

⁸Slight differences in radius caused by discretization appear.


```

1 // User-defined variables:
2 // center, nbLayers, nbPts, layerHeight, extrusionMultiplier,
  nozzleWidth, deltaRadiusPerLayer
3
4 // System methods:
5 // subtract(vec1, vec2): computes the vector difference
6 // norm(vec): computes the L2 norm of vec
7 // addExtrusion(startPt, endPt, layerHeight, extrusionMultiplier):
  computes the extrusion amount for a segment and returns a
  copy of the endPt vector, with extrusion value appended
8
9 // Find starting radius and theta
10 let vecToCenter = subtract([x, y], center);
11 let radiusStart = norm(vecToCenter);
12 let startTheta = 0;
13 if (norm(vecToCenter) > 0.1) {
14   startTheta = Math.acos(vecToCenter[0] / norm(vecToCenter));
15   if (vecToCenter[1] < 0) {
16     startTheta = 2 * Math.PI - startTheta;
17   }
18 }
19
20 let radius = radiusStart;
21
22 // This function computes the radius by incrementing the "radius"
  variable in script scope
23 function calculateRadiusAtPt(deltaRadiusPerLayer) {
24   let deltaRadiusPerPt = deltaRadiusPerLayer / nbPts;
25   radius += deltaRadiusPerPt;
26   return radius;
27 }
28
29 function spiralPoint(ptIdx, self) {
30   let theta = startTheta + ptIdx * 2 * Math.PI / nbPts;
31   let radiusAtPt = calculateRadiusAtPt(self.deltaRadiusPerLayer)
32   return [
33     self.center[0] + radiusAtPt * Math.cos(theta),
34     self.center[1] + radiusAtPt * Math.sin(theta),
35     self.z + self.layerHeight / nbPts,
36   ];
37 }
38
39 for (let l = 0; l < nbLayers; l++) {
40   for (let i = 0; i < nbPts; i++) {
41     $goto(addExtrusion(
42       [self.x, self.y, self.z, self.e],
43       spiralPoint(i, self),
44       self.layerHeight,
45       self.extrusionMultiplier
46     ));
47   }
48 }

```

Listing 3: Motion script radiusVarySection for slanted profile on the pitcher

6.4 Developing new workflows with a modified CNC machine

As previously mentioned, the design space of digital fabrication also includes the invention of new machines and fabrication techniques for which there are not yet established fabrication parameters or toolpathing strategies. We therefore investigated how MINTS supports the development of novel workflows for unconventional CNC machines. We developed two workflows for ceramics using a custom clay slip deposition machine created by Benjamin and Taneesha for a course project. We built the machine by re-using parts from an off-the-shelf Ender 3 plastic printer. We retained the Ender’s Cartesian motion platform with a bed actuated in Y and an XZ gantry that holds the end-effector. We removed the hot-end extruder and mounted an extruder axis (axis E) instead, with a custom



Figure 10: Developing ad-hoc features for the slip deposition machine. A) Making droplets of different sizes on an ad-hoc test plate by triggering extrusion moves of varying length (orange highlight). B) Commanding the syringe to move to the temporary refill station we defined.

linear motion stage that actuated a syringe piston. We replaced the original printing base plate with a temporary attachment system for a square pottery bat. The bat can also be attached to a pottery wheel to facilitate centering of objects, or to directly throw vessels on that will be centered by construction. We added a laser pointer at a known XY offset from the syringe nozzle `laserToNozzle`. The laser aids in previewing where the slip would fall if it were pushed out based on the same motion pattern but offset by `laserToNozzle`.

6.4.1 Motion scripts for testing and utilities. We started working with the slip machine by using MINTS to characterize how the syringe and slip material would perform to create controlled dots. In the process we developed a procedure to refill the syringe.

We defined the `dripHere` motion script to extrude a controlled amount of slip at the current XY position, and a specified `zFloor` height. To set the `zFloor` variable, we jogged the end-effector down towards the base plate until the nozzle was in contact, then took a snapshot of the machine’s Z value to set that variable (Section 4.3.3). At execution stage, we used the script to create a rough test grid characterizing how the droplet size varies depending on plunger motion length (Fig. 10A). We built the grid sequentially, assessing what values to test next by observing the outcome of previous tests. In subsequent programs, we used the test grid as a reference sheet when setting extrusion amounts.

Since the syringe we used in the current iteration of the machine could only carry 20mL of slip, we had to find a way to rapidly refill the syringe for workflows that required larger amounts of slip (Section 6.4.3). While future iterations of the slip machine could solve this problem by redesigning the mechanism to accommodate larger syringes, we opted to implement refilling by simply bringing the syringe to a point where we could submerge the tip in a slip container, then lifting the piston up to fill the syringe. We defined a one-line motion script `goToPoint`: `$goto(target[0], target[1], $z)`, for which we saved in a preset the point of the refill station as `target`. At execution time, we could then simply trigger this script with the saved preset, then jog the extruder axis up (Fig. 10B).

Since the system supports programming of any motion behavior (G1) without restrictions on the scope or function of the behavior, we were able to use the same motion programming paradigm to build small utility and testing scripts, and to later implement more complex scripts for fabricating polished artifacts. Since the effort required to implement small utility features can be very low (writing



Figure 11: Geometric slip trailing for ceramics decoration. We use the slip deposition machine to deposit droplets of colored slip onto the surface of a thrown plate. A) The machine with a thrown plate. B) The colored slip has to be deposited by bringing the syringe nozzle end close to the surface of the plate. C, D) The decorated plates after firing.

a couple lines of code), the system facilitates building ad-hoc, partial automation operations that function as simple scripts, early in the fabrication development process. The utility features could also be realized with pre-compiled G-code instructions [Fossdal et al. 2021; Subbaraman and Peek 2022]. We see this overlap in capabilities as a strength of our approach, allowing users to work flexibly across pre-compiled and real-time execution contexts.

6.4.2 Thrown vessel surface decoration. Slip trailing is a traditional ceramics decoration technique where a colored slip is deposited on top of a leather-hard clay vessel⁹. We created a program to do slip trailing of concentric patterns on thrown vessels, using a slip that contains cobalt carbonate, which turns blue once fired. We developed a motion script that makes the syringe move along a circular path, stopping at regularly spaced intervals along that path to extrude a droplet of slip. We declared a variable for the center of the circle, and made the circle radius and initial point dependent on machine state at script execution time. We used the laser as an in-situ probing and previewing tool since it is more precise than looking at the syringe tip. We defined as a convention that any snapshot for 2D XY variables is taken with respect to the laser probe position. Within a preset `syringeCentered` for the motion script, we set the motion script's `offset` variable to `laserToNozzle`, the offset between laser and syringe. The script applies the `offset` variable to any executed move command. When previewing, we simply set the `offset` variable to the null vector with another preset.

⁹Leather hard refers to a pre-firing stage of ceramic production where vessels are partially dried but still contain some moisture to support additional fabrication operations like carving, applying secondary parts, and slip addition.

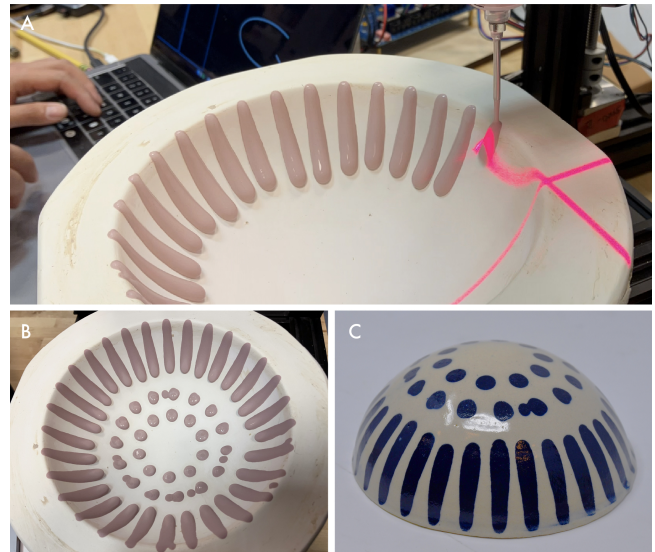


Figure 12: Dripping slip onto plaster mold for inlay decoration. A) Dripping in a circle formation on the vertical walls of the mold creates long droops. B) The colored slip on the mold, prior to adding the white slip. C) The object after casting, de-molding, and firing.

At execution time, we could jog the machine to position the laser at a point in the XY plane where we want the circle of slip dots to be positioned, using the laser cross as a preview. We would then select the `syringeCentered` preset and trigger execution, moving the syringe along the selected circle. Furthermore, before executing we also had to pick the extrusion amount depending on the desired dot size (using the reference sheet we made prior) and set the `zFloor` variable value to match the height of the vessel's surface at that particular point, to ensure the nozzle end would come close to contact with the surface—important for good drop deposition.

We also implemented teardrop-shaped droops by adding a variable controlling the teardrop length and additional extrusion in the tail of the teardrop. While decorating the first plate's flat section, we noticed that the plate's rim would be nice to decorate, and since it was slanted we modified the motion script by adding the option to define a secondary Z value for the end of the teardrop move (Fig. 11C and D, see plate rim).

This workflow demonstrates the benefits of programming machine motion interleaved with motion execution, as the resulting slip deposition pattern would be difficult to simulate numerically. Instead, MINTS lets us iteratively build the pattern through variable changes based on prior results (G3).

6.4.3 Dripping slip on plaster. Plaster molds are commonly used in ceramics for slip casting, where 3D vessels are produced by removing moisture from the slip with the plaster mold [Han et al. 2025; Lyons 2024]. While slip casting typically involves pouring slip until the mold is full, we were inspired by the technique of design studio Joachim-Morineau who create functional and sculptural pieces by dripping slip paths into a mold [2024]. We experimented with applying CNC to this method by executing a sequence of varying

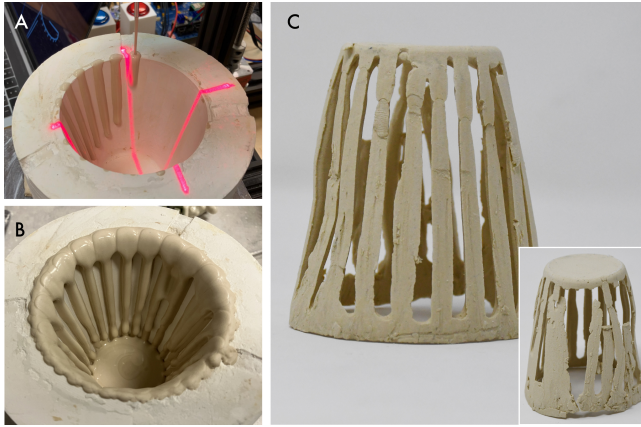


Figure 13: Experimental slip structure. A) We deposit a large quantity of slip at regular intervals along the mold’s inner rim. The slip slides along the mold’s wall, creating long thin channels. B) The object in the mold. C) Object after de-molding and firing (front and back).

dripping patterns on the mold. We used the same motion script as in the previous demonstration, but adjusted the slip recipe to be a more liquid casting slip and increased the extrusion amount to obtain long droops. Since larger amounts of slip were necessary, we used the refill script whenever the syringe was nearly empty. In this demonstration, the specific Z value of the syringe mattered less than in the slip trailing case, but we relied on the laser pointer to precisely position the dripping points exactly on the molds. We created an inlay pattern in a half-dome shape by dripping colored slip in a geometric pattern, then pouring white slip in the mold (Fig. 12), and a sculptural piece with thin structures created by the dripping channels formed by slip along the mold’s walls when depositing it at regular intervals along the rim (Fig. 13). The sculptural piece was very fragile and some of its arms broke during demolding due to being too thin to detach properly (see Fig. 13C inset).

Although the resulting artifacts are more experimental than our work applying slip to thrown vessels, they demonstrate how MINTS supports the quick progression from inspiration to application of novel techniques on an experimental system. The plaster dripping workflow took a very short time to develop with our system (it required merely changing variable values from the previous demonstration’s script). Performing this complete end-to-end fabrication workflow helped us diagnose pain points across toolpath programming, machine design (e.g., need for a larger syringe), and material (e.g., need to develop a better casting slip recipe to avoid cracking). This highlights the benefits of framing motion programming as a flexible practice that can be performed both before and during fabrication, for final artifact production as well as for machine and material research and development.

7 Discussion

We developed MINTS to explore how representing machine toolpath as dynamic expressions evaluated just before execution could

extend the benefits of programmatic toolpath control into the execution phase. In this section, we discuss future research directions to develop systems for end-user control at both planning and execution phases, as well as remaining challenges at the intersection of end-user programming and interactive fabrication.

7.1 Towards end-user control at *all* stages

MINTS allowed us to author parametric programs *before* any fabrication begins (planning phase), and to execute these programs with parameters decided at different moments in time, including *after* fabrication starts and right up until an individual command is dequeued and executed (execution phase). Our demonstrations highlight how decision-making happened at different times during each design workflow. We summarized all demonstrations in Table 1, highlighting that important design decisions were made at both planning and execution stages. We believe that letting users take decisions during *all* stages of the design process offers significant benefits: some design decisions such as the noise function chosen to perturb the painted lines in the ink painting example (Section 4) need to be carefully pre-planned and iterated on (either through preview or multiple trial fabrication runs), while other decisions such as triggering a paint brush ink recharge, or indicating the Z-height to deposit slip on a thrown vessel, are easier to take while observing real physical conditions.

Prior work has demonstrated the merits of rapid machine program development within a quick iteration loop of planning and fabrication [Fossdal et al. 2021, 2023; Subbaraman and Peek 2022; Twigg-Smith and Peek 2023], as well as the potential of making real-time decisions while the machine is fabricating [Moyer et al. 2024b, 2026; Willis et al. 2010]. Our work builds on these findings to demonstrate the usefulness of *integrating* both approaches, to let end-users take design decisions at multiple stages in a fabrication workflow. Unlike in prior systems for real-time modifications during execution, users aren’t limited to a pre-defined editable representation with a prescribed design space. Our approach lets users define custom parametric toolpath representations, allowing them to plan execution-time modifications at the level they deem appropriate for their particular use case.

MINTS shows one possible way of exposing control to end-users at planning and execution phase, specifically through *imperative programming* in the planning phase, and a *graphical user interface* in the execution phase. We demonstrate that these control mechanisms are effective, with imperative programming supporting the definition of precise and repeatable motions, while the GUI provides a way to quickly modify subsequent motions. In the future, other ways of specifying motion commands in both phases can be explored, such as using embodied actions to plan machine motion [Bourgault et al. 2025], or using live coding to generate motion commands on-the-fly [pixelpusher 2026]. A key challenge to resolve towards exposing flexible control over the toolpath during fabrication lies in alleviating risks and failures arising from tampering with machine operation live. Fabrication operation are irreversible, meaning that it is important to provide ways to preview and simulate the impact of a user action before applying it; machine motors have physical torque limits that must be respected, meaning that it is necessary to help users stay within acceleration bounds.

Table 1: Overview of design workflow for all demonstrations. At planning stage, we wrote motion scripts (`</>`) and set some script variables (`x`) to be kept fixed for the entire fabricated object. At execution stage, we interacted with the system both while the machine was actively moving (`▶`) and while it was idling after finishing current queued motions (`■`), to modify script variables (`x`) and instantly trigger motion scripts (`🕒`)

Workflow	Decision time	
	Planning stage	Execution stage
Ink painting (Section 4)	<code></></code> Brush rinsing and dipping path <code></></code> Perlin noise lines pattern <code>x</code> Ink well positions, paper size	<code>🕒 ▶</code> Timing and color of brush ink recharge <code>x ▶</code> Scale of the noise displacement
Improvisational loops (Section 6.1)	<code></></code> Vessel profile shape <code></></code> Loop shape <code>x</code> Printing parameters <code>x</code> Vessel center and profile shape	<code>🕒 ▶</code> Position and size of loops
Textural patterns (Section 6.2)	<code></></code> Vessel profile shape <code></></code> Wavy texture path <code>x</code> Preset variants of the wavy texture path <code>x</code> Printing parameters <code>x</code> Vessel center and profile shape	<code>x ▶</code> Wavy texture style along vessel
Pitcher as kit-of-parts (Section 6.3)	<code></></code> Shape of the different parts <code>x</code> Printing parameters <code>x</code> Vessel center	<code>🕒 ▶</code> Z-location of the different parts <code>x ▶</code> Parameters of the parts (eg: slant angle, handle length)
Slip machine utils (Section 6.4.1)	<code></></code> Drop deposition motion <code></></code> Go to target position	<code>x ■</code> Position of refill station <code>x ■</code> Droplet size
Vessel slip decoration (§6.4.2) Slip dripping on mold (§6.4.3)	<code></></code> Circular drop pattern <code></></code> Circular teardrop pattern <code>x</code> Offset between laser and syringe	<code>🕒 ■</code> Refill <code>x ■</code> Radius and number of repetitions of pattern

7.2 Towards end-user programming for interactive fabrication

While developing the demonstrative workflows, we identified limitations of the current implementation of MINTS. Both technical and conceptual improvements are needed to better support end-user programming in interactive fabrication contexts.

Improving programming syntax and programming environment. Due to the relative simplicity of the syntax we introduced with the `$goto` function, the end-user needs to write nested function calls in order to describe more complex expressions (e.g. the function `applyNoise` in Listing 2). Expressions often are easier to write by introducing temporary variables (e.g. `x` and `y` variables in `dipIn` function), which requires a longer, full-function syntax rather than a short inline expression, as we had originally envisioned for the `$goto` function. While nesting function calls is a practical solution, it can make the code more convoluted. The current syntax is functional and expressive but relatively contrived and hard to read in complex examples. In future work, a domain-specific programming syntax that caters to such use cases could improve the programming experience. Furthermore, just like in other creative and design use cases [Korosteleva and Sorkine-Hornung 2023; Mei et al. 2024; Pollock et al. 2024; Ye et al. 2020], we see the development of domain specific languages (DSL) as a promising avenue to support people

in tackling more scoped use cases (e.g., describing an interactive fabrication workflow by composing known operations). Our system architecture for motion command evaluation at execution-time will directly facilitate such work.

In addition, improving the overall programming environment UI of our prototype implementation could enhance the programming experience: in the future we would like to integrate variable declaration within the scripting environment (e.g., inspired by the way Unity components declare attributes exposed in the editor¹⁰, or Processing Tweak¹¹), add linting and auto-complete support for both user-defined variables and system-specific syntax, and improve the syntax to be less error-prone (e.g., modify the keyword `self`, which could cause collisions with a browser variable in Javascript).

Extending programming and debugging tools. Programming motion command expressions that refer to motion execution time variables can be challenging when the variables are involved in many procedural expressions (e.g., used repeatedly in a `for` loop). This is because the outcome of such expressions is not fully determined until motion execution (and variable value modification), making it hard to conceptualize the potential outcome without actual execution. In addition, developers must author logic to work

¹⁰<https://docs.unity3d.com/Manual/unity-attributes.html>

¹¹<https://processing.org/environment/>

in *relative* mode with respect to any variable that we wish to modify during execution. For instance, in Listing 3 (l.23–27, and l.31) the function `calculateRadiusAtPt` computes the resulting radius for a given toolpath point by gradually incrementing a radius variable in script scope, in order to take into account changes of the `deltaRadiusPerLayer` variable at any point during execution. Originally, we had considered computing `radiusAtPt` (l.33) as: `self.deltaRadiusPerLayer * 1`, which yields a very similar result if we assume that `deltaRadiusPerLayer` stays fixed during execution, but breaks down if we modify this variable, because the expression describes global effects of `deltaRadiusPerLayer` across all motion commands. The challenge of predicting program output in MINTS for complex programs reflects established challenges in software development in other contexts, with the unique facet that programs lead to fabricated objects with less reworkability than digital applications. While the previewing feature we implemented (both during script authoring and track execution) was helpful to us in identifying bugs or unintended effects when developing the demonstrative workflows, we see potential to develop more sophisticated code inspection and debugging tools [Li et al. 2020], to flatten the learning curve. In the future, we see opportunities to deploy MINTS among relevant user groups, such as digital fabrication practitioners, in order to evaluate the usability and learnability of the system, and to identify areas for environment support in understanding and predicting program behavior.

8 Conclusion

We developed MINTS, a software tool that lets end-users program machine behavior as dynamic expressions at the planning phase, and then modify motion commands right up until they are sent to the machine at the execution phase. To achieve that, we made three key technical contributions: a system architecture for communicating with a CNC machine and supporting interactive modifications of machine motion; a syntax that lets users describe planned motion as dynamic expressions evaluated during execution; and a controllable queue of motion commands that facilitates executing motion commands and UI commands in the asynchronous motion execution context. We demonstrated that this paradigm for motion programming can support the development of complex fabrication workflow, both within established (pen-plotting, clay 3D printing) and experimental techniques (slip deposition).

Acknowledgments

We express our gratitude to Ilan Moyer for contributing to develop and maintain Stepdance, and for insights into how to achieve real-time control from a desktop computer, to Devon Frost for helping frame this project, and to Alejandro Aponte for his precious support in developing the slip drip machine. This research was funded in part by the NSF CAREER Program (Award: 2441766), and the Computing Research Association CISE REU Student Funding Program.

References

- T. Alderighi, L. Malomo, T. Auzinger, B. Bickel, P. Cignoni, and N. Pietroni. 2022. State of the Art in Computational Mould Design. *Computer Graphics Forum* 41, 6 (2022), 435–452. doi:10.1111/cgf.14581 _eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14581>.
- Amy Goodchild. 2024. Chaos in the medium: watercolour plotting. <https://www.amygoodchild.com/blog/watercolour-plots>

- Fiona Bell, Camila Friedman-Gerlicz, and Leah Buechley. 2025. Biomaterial Recipes for 3D Printing: A Cookbook of Sustainable and Extrudable Bio-Pastes. In *Proceedings of the Nineteenth International Conference on Tangible, Embedded, and Embodied Interaction (TEI '25)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3689050.3704427
- Marcel Borowski, Luke Murray, Rolf Bagge, Janus Bager Kristensen, Arvind Satyanarayan, and Clemens Nylandstedt Klokmoose. 2022. Varv: Reprogrammable Interactive Software as a Declarative Data Structure. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. Association for Computing Machinery, New York, NY, USA, 1–20. doi:10.1145/3491102.3502064
- Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D³ Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec. 2011), 2301–2309. doi:10.1109/TVCG.2011.185
- Samuelle Bourgault, Alejandro Aponte, Megumi Ondo, Emilie Yu, and Jennifer Jacobs. 2025. WORM: Programming Collaborative Robots Through Manual Actions for Craft-Aligned Digital Fabrication. In *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology (UIST '25)*. Association for Computing Machinery, New York, NY, USA, 1–19. doi:10.1145/3746059.3747616
- Sam Bourgault and Jennifer Jacobs. 2024. Millipath: Bridging Materialist Theory and System Development for Surface Texture Fabrication. In *Proceedings of the 2024 ACM Designing Interactive Systems Conference (DIS '24)*. Association for Computing Machinery, New York, NY, USA, 50–68. doi:10.1145/3643834.3661599
- Samuelle Bourgault, Pilar Wiley, Avi Farber, and Jennifer Jacobs. 2023. CoilCAM: Enabling Parametric Design for Clay 3D Printing Through an Action-Oriented Toolpath Programming System. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3544548.3580745
- Leah Buechley, Jaime Gould, and Fiona Bell. 2024. CeraMetal: A New Approach to Low-Cost Metal 3D Printing with Bronze Clay. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3613904.3642155
- X. Chermain, G. Cocco, C. Zanni, E. Garner, P. A. Hugron, and S. Lefebvre. 2025. Atomizer: Beyond Non-Planar Slicing for Fused Filament Fabrication. *Computer Graphics Forum* 44, 5 (2025), e70189. doi:10.1111/cgf.70189 _eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.70189>.
- Xavier Chermain, Cédric Zanni, Jonàs Martínez, Pierre-Alexandre Hugron, and Sylvain Lefebvre. 2023. Orientable Dense Cyclic Infill for Anisotropic Appearance Fabrication. *ACM Trans. Graph.* 42, 4 (July 2023), 68:1–68:13. doi:10.1145/3592412
- Claywoven. 2026. Claywoven/Nozzleboss-Claywoven. <https://github.com/Claywoven/Nozzleboss-Claywoven> original-date: 2026-05-31T03:53:46Z.
- Matt DesLauriers. 2026. mattdes/canvas-sketch. <https://github.com/mattdes/canvas-sketch> original-date: 2018-06-14T06:01:46Z.
- Simon Duenser, Roi Poranne, Bernhard Thomaszewski, and Stelian Coros. 2020. RoboCut: hot-wire cutting with robot-controlled flexible rods. *ACM Transactions on Graphics* 39, 4 (Aug. 2020), doi:10.1145/3386569.3392465
- Duet3D. 2026. Duet Web Control Manual. https://docs.duet3d.com/User_manual/Reference/Duet_Web_Control_Manual
- Evil Mad Scientist. 2019. Introduction – AxiDraw CLI Reference. https://axidraw.com/doc/cli_api/#introduction
- Jack Forman, Mustafa Doga Dogan, Hamilton Forsythe, and Hiroshi Ishii. 2020. DefeXtiles: 3D Printing Quasi-Woven Fabric via Under-Extrusion. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. Association for Computing Machinery, New York, NY, USA, 1222–1233. doi:10.1145/3379337.3415876
- Frikk Fossdal, Rogardt Heldal, and Nadya Peek. 2021. Interactive Digital Fabrication Machine Control Directly Within a CAD Environment. In *Symposium on Computational Fabrication*. ACM, Virtual Event USA, 1–15. doi:10.1145/3485114.3485120
- Frikk H Fossdal, Vinh Nguyen, Rogardt Heldal, Corie L. Cobb, and Nadya Peek. 2023. Vespidae: A Programming Framework for Developing Digital Fabrication Workflows. In *Proceedings of the 2023 ACM Designing Interactive Systems Conference*. ACM, Pittsburgh PA USA, 2034–2049. doi:10.1145/3563657.3596106
- Camila Friedman-Gerlicz, Deanna Gelosi, Fiona Bell, and Leah Buechley. 2024. WeaveSlicer: Expanding the Range of Printable Geometries in Clay. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, Honolulu HI USA, 1–16. doi:10.1145/3613904.3642622
- Devon Frost, Raina Lee, Eun-Ha Paek, and Jennifer Jacobs. 2024. SketchPath: Using Digital Drawing to Integrate the Gestural Qualities of Craft in CAM-Based Clay 3D Printing. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3613904.3642684
- Aditya Ganeshan, Kurt Fleischer, Wenzel Jakob, Ariel Shamir, Daniel Ritchie, Takeo Igarashi, and Maria Larsson. 2025. MiGumi: Making Tightly Coupled Integral Joints Millable. *ACM Trans. Graph.* 44, 6 (Dec. 2025), 193:1–193:14. doi:10.1145/3763304
- Neil Gershenfeld, Quentin Bolsée, and Robert Hart. 2022. Urumbur: Minimal Machine Building. In *Proceedings of the 7th Annual ACM Symposium on Computational Fabrication*. ACM, Seattle WA USA, 1–2. doi:10.1145/3559400.3565599
- Andrew Gleadall. 2021. FullControl GCode Designer: Open-source software for unconstrained design in additive manufacturing. *Additive Manufacturing* 46 (Oct. 2021),

102109. doi:10.1016/j.addma.2021.102109
- Xinyue Gui, Ding Xia, Wang Gao, Mustafa Doga Dogan, Maria Larsson, and Takeo Igarashi. 2025. Draw2Cut: Direct On-Material Annotations for CNC Milling. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/3706598.3714281
- Bo Han, Jared Lim, Kianne Lim, Adam Choo, Ching Chiuan Yen, Genevieve Ang, and Clement Zheng. 2025. Slip Casting as a Machine for Making Textured Ceramic Interfaces. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, 1–18. doi:10.1145/3706598.3714396
- Shiqing He and Eytan Adar. 2020. Plotting with Thread: Fabricating Delicate Punch Needle Embroidery with X-Y Plotters. In *Proceedings of the 2020 ACM Designing Interactive Systems Conference (DIS '20)*. Association for Computing Machinery, New York, NY, USA, 1047–1057. doi:10.1145/3357236.3395540
- Heinz-Loepmeier. 2026. Heinz-Loepmeier/nozzleboss. <https://github.com/Heinz-Loepmeier/nozzleboss> original-date: 2021-10-09T16:35:31Z.
- Yuichi Hirose, Mark Gillespie, Angelica M. Bonilla Fominaya, and James McCann. 2024. Solid Knitting. *ACM Trans. Graph.* 43, 4 (July 2024), 88:1–88:15. doi:10.1145/3658123
- Mare Hirsch, Gabrielle Benabdallah, Jennifer Jacobs, and Nadya Peek. 2023. Nothing Like Compilation: How Professional Digital Fabrication Workflows Go Beyond Extruding, Milling, and Machines. *ACM Trans. Comput.-Hum. Interact.* 31, 1 (Nov. 2023), 13:1–13:45. doi:10.1145/3609328
- Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. 1986. Direct Manipulation Interfaces. In *User Centered System Design*. CRC Press. Num Pages: 38.
- Jennifer Jacobs, Joel Brandt, Radomir Mech, and Mitchel Resnick. 2018. Extending Manual Drawing Practices with Artist-Centric Programming Tools. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, Montreal QC Canada, 1–13. doi:10.1145/3173574.3174164
- Jasper. 2026. Claywoven – coded craft. <https://claywoven.com/>
- David Jourdan, Pierre-Alexandre Hugron, Camille Schreck, Jonàs Martínez, and Sylvain Lefebvre. 2023. Shrink & Morph: 3D-printed self-shaping shells actuated by a shape memory effect. *ACM Trans. Graph.* 42, 6 (Dec. 2023). doi:10.1145/3618386 Place: New York, NY, USA.
- Daphna Kaplan, Mirela Ben Chen, and Yoav Sterman. 2025. ConTextural: A Toolpath-Based Texture Editing Tool for Extrusion 3D Printers. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3706598.3713753
- Ken Perlin. 2002. Improved Noise reference implementation. <https://cs.nyu.edu/~perlin/noise/>
- Jeeun Kim, Clement Zheng, Haruki Takahashi, Mark D Gross, Daniel Ashbrook, and Tom Yeh. 2018. Compositional 3D printing: expanding & supporting workflows towards continuous fabrication. In *Proceedings of the 2nd ACM Symposium on Computational Fabrication*. ACM, Cambridge Massachusetts, 1–10. doi:10.1145/3213512.3213518
- Clemens N. Klokmoose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: Shareable Dynamic Media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. Association for Computing Machinery, New York, NY, USA, 280–290. doi:10.1145/2807442.2807446
- Maria Korosteleva and Olga Sorkine-Hornung. 2023. GarmentCode: Programming Parametric Sewing Patterns. *ACM Transactions on Graphics* 42, 6 (Dec. 2023), 1–15. doi:10.1145/3618351
- Gierad Laput, Xiang 'Anthony' Chen, and Chris Harrison. 2015. 3D Printed Hair: Fused Deposition Modeling of Soft Strands, Fibers, and Bristles. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. Association for Computing Machinery, New York, NY, USA, 593–597. doi:10.1145/2807442.2807484
- David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. 2018. Evaluation Strategies for HCI Toolkit Research. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, Montreal QC Canada, 1–17. doi:10.1145/3173574.3173610
- Jingyi Li, Joel Brandt, Radomir Mech, Maneesh Agrawala, and Jennifer Jacobs. 2020. Supporting Visual Artists in Programming through Direct Inspection and Control of Program Execution. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3313831.3376765
- Jingyi Li, Sonia Hashim, and Jennifer Jacobs. 2021. What We Can Learn From Visual Artists About Software Development. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3411764.3445682
- Jingyi Li, Eric Rawn, Jacob Ritchie, Jasper Tran O'Leary, and Sean Follmer. 2023. Beyond the Artifact: Power as a Lens for Creativity Support Tools. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3586183.3606831
- Licia He. 2024. 20+ Questions About My Plotter Painting Practice. https://www.eyesofpanda.com/project/plotter_painting_q_a/
- Geoffrey Litt, Josh Horowitz, Peter van Hardenberg, and Todd Matthews. 2025. Malleable software: Restoring user agency in a world of locked-down apps. <https://www.inkandswitch.com/essay/malleable-software/>
- Marco Livesu, Stefano Ellero, Jonàs Martínez, Sylvain Lefebvre, and Marco Attene. 2017. From 3D models to 3D prints: an overview of the processing pipeline. *Computer Graphics Forum* 36, 2 (2017), 537–564. doi:10.1111/cgf.13147 _eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13147>
- Daniel Cardoso Llach. 2015. *Builders of the Vision: Software and the Imagination of Design*. Routledge, New York. doi:10.4324/9781315798240
- Kent Lyons. 2024. Shape Cast: Automating 3D Design for Plaster Molds in Ceramic Slip Casting. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. ACM, Honolulu HI USA, 1–7. doi:10.1145/3613905.3651020
- Shiran Magrisso, Moran Mizrahi, and Amit Zoran. 2018. Digital Joinery For Hybrid Carpentry. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. Association for Computing Machinery, Montreal QC, Canada, 1–11. doi:10.1145/3173574.3173741
- Ali Mahdavi-Amiri, Fenggen Yu, Haisen Zhao, Adriana Schulz, and Hao Zhang. 2020. VDAC: volume decompose-and-carve for subtractive manufacturing. *ACM Transactions on Graphics (TOG)* 39, 6 (Nov. 2020), 203:1–203:15. doi:10.1145/3414685.3417772
- MarlinFirmware. 2011. Marlin. <https://marlinfw.org/>
- Yuxuan Mei, Benjamin Jones, Dan Cascaval, Jennifer Mankoff, Etienne Vouga, and Adriana Schulz. 2024. FabHacks: Transform Everyday Objects into Home Hacks Leveraging a Solver-aided DSL. In *Proceedings of the 9th ACM Symposium on Computational Fabrication*. ACM, Aarhus Denmark, 1–16. doi:10.1145/3639473.3665788
- Daniela Mitterberger, Selen Ercan Jenny, Lauren Vasey, Ena Lloret-Fritsch, Petrus Aejmelaelus-Lindström, Fabio Gramazio, and Matthias Kohler. 2022. Interactive Robotic Plastering: Augmented Interactive Design and Fabrication for On-site Robotic Plastering. In *CHI Conference on Human Factors in Computing Systems*. ACM, New Orleans LA USA, 1–18. doi:10.1145/3491102.3501842
- Ilan E Moyer, Samuelle Bourgault, Devon Frost, and Jennifer Jacobs. 2024a. Don't Mesh Around: Streamlining Manual-Digital Fabrication Workflows with Domain-Specific 3D Scanning. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3654777.3676385
- Ilan E Moyer, Samuelle Bourgault, Devon Frost, and Jennifer Jacobs. 2024b. Throwing Out Conventions: Reimagining Craft-Centered CNC Tool Design through the Digital Pottery Wheel. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–22. doi:10.1145/3613904.3642361
- Ilan E Moyer, Devon Frost, Emilie Yu, Maria Yang, and Jennifer Jacobs. 2026. StepDance: A Toolkit for Redesigning CNC Machines Using Physical Metaphors. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems (CHI '26)*. Association for Computing Machinery, New York, NY, USA, 1–34. doi:10.1145/3772318.3791107
- Stefanie Mueller, Anna Seufert, Huaishu Peng, Robert Kovacs, Kevin Reuss, François Guimbretière, and Patrick Baudisch. 2019. FormFab: Continuous Interactive Fabrication. In *Proceedings of the Thirteenth International Conference on Tangible, Embedded, and Embodied Interaction*. ACM, Tempe Arizona USA, 315–323. doi:10.1145/3294109.3295620
- David Noble. 2017. *Forces of Production: A Social History of Industrial Automation*. Routledge, New York. doi:10.4324/9780203791806
- Joyce E Passananti, Emilie Yu, Timea Tihanyi, Tobias Höllerer, and Jennifer Jacobs. 2026. Clay ARTools: Precise Machine Toolpath Editing for Clay 3D Printing With Craft-Inspired Direct Manipulation Tools in AR. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems (CHI '26)*. Association for Computing Machinery, New York, NY, USA, 1–19. doi:10.1145/3772318.3791878
- Patt Vira. 2023. Code tutorials - Perlin Noise Field. <https://www.pattvira.com>
- Nadya Peek, James Coleman, Ilan Moyer, and Neil Gershenfeld. 2017. Cardboard Machine Kit: Modules for the Rapid Prototyping of Rapid Prototyping Machines. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, Denver Colorado USA, 3657–3668. doi:10.1145/3025453.3025491
- Huaishu Peng, Jimmy Briggs, Cheng-Yao Wang, Kevin Guo, Joseph Kider, Stefanie Mueller, Patrick Baudisch, and François Guimbretière. 2018. RoMA: Interactive Fabrication with Augmented Reality and a Robotic 3D Printer. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3173574.3174153
- Franklin Pezutti-Dyer and Leah Buechley. 2022. Extruder-Turtle: A Library for 3D Printing Delicate, Textured, and Flexible Objects. In *Proceedings of the Sixteenth International Conference on Tangible, Embedded, and Embodied Interaction (TEI '22)*. Association for Computing Machinery, New York, NY, USA, 1–9. doi:10.1145/3490149.3501312
- pixelpusher. 2026. liveprinter. <https://github.com/pixelpusher/liveprinter> original-date: 2018-06-02T07:35:55Z.
- Josh Pollock, Catherine Mei, Grace Huang, Elliot Evans, Daniel Jackson, and Arvind Satyanarayan. 2024. Bluefish: Composing Diagrams with Declarative Relations. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24)*. Association for Computing Machinery, New York, NY, USA, 1–21. doi:10.1145/3654777.3676465

- Josh Pollock and Arvind Satyanarayan. 2026. GoFish: A Grammar of More Graphics! *IEEE Transactions on Visualization and Computer Graphics* 32, 1 (Jan. 2026), 549–559. doi:10.1109/TVCG.2025.3634250
- Processing Foundation. 2001. Welcome to Processing! <https://processing.org/>
- Jake Robert Read, Leo McElroy, Quentin Bolese, B Smith, and Neil Gershenfeld. 2023a. Modular-Things: Plug-and-Play with Virtualized Hardware. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems*. ACM, Hamburg Germany, 1–6. doi:10.1145/3544549.3585642
- Jake Robert Read, Nadya Peek, and Neil Gershenfeld. 2023b. MAXL: Distributed Trajectories for Modular Motion. In *Proceedings of the 8th ACM Symposium on Computational Fabrication (SCF '23)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/3623263.3623362
- Reinder Nijhoff. 2018. Turtletoy. <https://turtletoy.net/>
- Mitchel Resnick, Brad Myers, Kumiyo Nakakoji, Ben Shneiderman, Randy Pausch, Ted Selker, and Mike Eisenberg. 2005. Design Principles for Tools to Support Creative Thinking. (Jan. 2005). doi:10.1184/R1/6621917.v1
- Alec Rivers, Ilan E. Moyer, and Frédo Durand. 2012. Position-correcting tools for 2D digital fabrication. *ACM Transactions on Graphics* 31, 4 (Aug. 2012), 1–7. doi:10.1145/2185520.2185584
- Studio JOACHIM-MORINEAU. 2024. Moca Tableware - Ceramics by Studio Joachim-Morineau. <https://moca.studiojoachimmorineau.com/>
- Blair Subbaraman, Nathaneal Bursch, and Nadya Peek. 2025. It's Not the Shape, It's the Settings: Tools for Exploring, Documenting, and Sharing Physical Fabrication Parameters in 3D Printing. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–19. doi:10.1145/3706598.3713354
- Blair Subbaraman and Nadya Peek. 2022. p5.fab: Direct Control of Digital Fabrication Machines from a Creative Coding Environment. In *Proceedings of the 2022 ACM Designing Interactive Systems Conference (DIS '22)*. Association for Computing Machinery, New York, NY, USA, 1148–1161. doi:10.1145/3532106.3533496
- Blair Subbaraman and Nadya Peek. 2024. Playing the Print: MIDI-Based Fabrication Interfaces to Explore and Document Material Behavior. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems (CHI EA '24)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/3613905.3650966
- Blair Subbaraman and Nadya Peek. 2026. From Copy/Paste to Copying Pastes: Supporting Replication in an Online Digital Fabrication Community. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems (CHI '26)*. Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3772318.3791321
- Haruki Takahashi and Homei Miyashita. 2017. Expressive Fused Deposition Modeling by Controlling Extruder Height and Extrusion Amount. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, Denver Colorado USA, 5065–5074. doi:10.1145/3025453.3025933
- Rundong Tian and Eric Paulos. 2021. Adroid: Augmenting Hands-on Making with a Collaborative Robot. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. ACM, Virtual Event USA, 270–281. doi:10.1145/3472749.3474749
- Rundong Tian, Vedant Saran, Mareike Kritzler, Florian Michahelles, and Eric Paulos. 2019. Turn-by-Wire: Computationally Mediated Physical Fabrication. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. ACM, New Orleans LA USA, 713–725. doi:10.1145/3332165.3347918
- Jasper Tran O'Leary, Gabrielle Benabdallah, and Nadya Peek. 2023. Imprimer: Computational Notebooks for CNC Milling. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. ACM, Hamburg Germany, 1–15. doi:10.1145/3544548.3581334
- Jasper Tran O'Leary, Eunice Jun, and Nadya Peek. 2022. Improving Programming for Exploratory Digital Fabrication with Inline Machine Control and Styled Toolpath Visualizations. In *Proceedings of the 7th Annual ACM Symposium on Computational Fabrication*. ACM, Seattle WA USA, 1–12. doi:10.1145/3559400.3561998
- Jasper Tran O'Leary, Thrisha Ramesh, Octi Zhang, and Nadya Peek. 2024. Tandem: Reproducible Digital Fabrication Workflows as Multimodal Programs. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, Honolulu HI USA, 1–16. doi:10.1145/3613904.3642751
- Hannah Twigg-Smith, Jasper Tran O'Leary, and Nadya Peek. 2021. Tools, Tricks, and Hacks: Exploring Novel Digital Fabrication Workflows on #PlotterTwitter. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3411764.3445653
- Hannah Twigg-Smith and Nadya Peek. 2023. Dynamic Toolchains: Software Infrastructure for Digital Fabrication Workflows. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. ACM, San Francisco CA USA, 1–20. doi:10.1145/3586183.3606802
- Unity Technologies. 2022. Unity 2020.3.43f1. <https://unity.com/releases/editor/whats-new/2020.3.43f1>
- Joshua Vasquez, Hannah Twigg-Smith, Jasper Tran O'Leary, and Nadya Peek. 2020. Jubilee: An Extensible Machine for Multi-tool Fabrication. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. ACM, Honolulu HI USA, 1–13. doi:10.1145/3313831.3376425
- Karl D.D. Willis, Cheng Xu, Kuan-Ju Wu, Golan Levin, and Mark D. Gross. 2010. Interactive fabrication: new interfaces for digital fabrication. In *Proceedings of the*

fifth international conference on Tangible, embedded, and embodied interaction. ACM, Funchal Portugal, 69–72. doi:10.1145/1935701.1935716

Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. 2020. Penrose: from mathematical notation to beautiful diagrams. *ACM Trans. Graph.* 39, 4 (Aug. 2020), 144:144:1–144:144:16. doi:10.1145/3386569.3392375

Fanchao Zhong, Yonglai Xu, Haisen Zhao, and Lin Lu. 2023a. As-Continuous-As-Possible Extrusion-Based Fabrication of Surface Models. *ACM Transactions on Graphics* 42, 3 (June 2023), 1–16. doi:10.1145/3575859

Fanchao Zhong, Haisen Zhao, Haochen Li, Xin Yan, Jikai Liu, Baoquan Chen, and Lin Lu. 2023b. VASCO: Volume and Surface Co-Decomposition for Hybrid Manufacturing. *ACM Trans. Graph.* 42, 6 (Dec. 2023), 188:1–188:17. doi:10.1145/3618324

Amit Zoran and Joseph A. Paradiso. 2013. Freed: a freehand digital sculpting tool. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, Paris France, 2613–2616. doi:10.1145/2470654.2481361

A Machine control delay and precision

Delay between action and execution. Since the backend execution loop is 1ms, the maximum delay between triggering an action from the backend and the actual start of executing that action is 1ms. The latency for websocket communication between UI and backend running on the same computer was measured on a Macbook M5 Pro at around 4ms roundtrip on average, meaning that a one-way message may have a 2ms delay. In practice, for jogging, at the speeds we consider (e.g., 20mm/s for the Axidraw) a 3ms delay amounts to overshoots of less than 0.06mm when the machine is jogging and the user releases the keyboard key.

Machine motion velocity while executing commands. For the demonstrations, we use fairly low velocity targets and velocity limits, in order to stay safely within machine acceleration constraints. Velocity targets: 40mm/s for Axidraw, 20mm/s for Ender and Potterbot, 5mm/s for Potterbot with 3mm nozzle used only for the big pitcher (to facilitate layer adhesion between the thin coils). Velocity limits were set per-axis, and adjusted empirically depending on the expected torque necessary to actuate the axis: 50mm/s for all belt-driven axes, 20mm/s for Ender Z-axis (lead-screw), 2mm/s for Potterbot Z-axis (lead-screw with heavy load), and 0.05mm/s for Potterbot extruder (lead-screw with high torque).

Precision of control. Our system sends target displacement values per stepper motor to the Stepdance board through USB Serial. We avoid quantization artifacts by normalizing the displacement values by a user-specified maximum expected number of full steps of the stepper motor per 1ms period, which we set to 5 steps for all demonstrations (corresponding in machine space to a speed of 62.5mm/s for the Ender and Axidraw, and 125mm/s for the Potterbot—safely above the operating speeds we work at). The smallest increment we can send is thus $5 \cdot 1/128 = 0.04$ step, corresponding to a linear distance of $5 \cdot 10^{-4}$ mm for the XY axes of the Axidraw and Ender, and of 10^{-3} mm for the XY axes of the Potterbot. We use this quantized displacement value to update simulated machine state in the backend, to avoid rounding errors accumulation. Empirically, we do not notice visible stepping artifacts or imprecisions, although operating the machines outside of the restricted ranges we tested (e.g., very fast or with very small displacements) may lead to quantization errors becoming more apparent, or to rounding errors due to stepper motor resolution (the controller allows a resolution of 1/16 steps). In future work, introducing features such as homing and position sensing could help resolve imprecisions and error accumulation.