TangiMap: Context-Aware Generative Function Mapping for Tangible Input Devices

Yunyi Zhu
yunyizhu@mit.edu
MIT CSAIL
Cambridge, Massachusetts, United States

Chang Xiao
xchang@bu.edu
Computer Science
Boston University
Boston, Massachusetts, United States

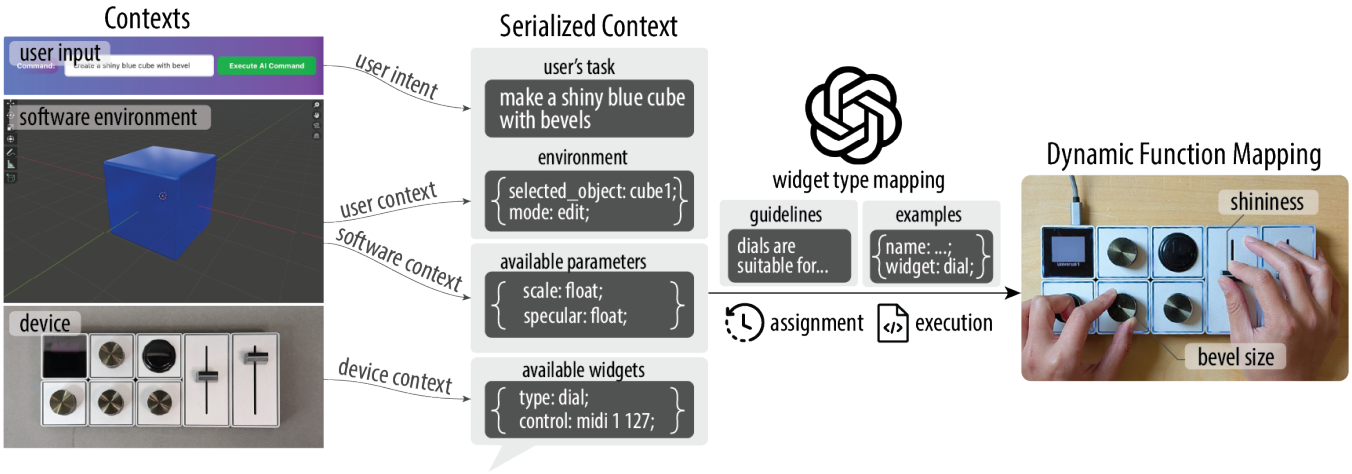

Figure 1: *TangiMap* bridges tangible input devices with dynamic software contexts by generating adaptive function mappings for hardware widgets. The system includes context from multiple sources: user intent, user environment, software parameters, and device configuration. These contexts are serialized into a structured representation and processed by an LLM together with mapping guidelines and examples to generate dynamic mappings. Functions are then assigned and executed on available tangible widgets, enabling real-time, context-aware interaction.

Abstract

While tangible input devices with physical controls such as knobs and sliders enable efficient interaction, they often lack the flexibility to adapt to dynamic contexts, limiting their use to predefined applications with fixed functionalities. In this paper, we introduce *TangiMap*, a proof-of-concept system that bridges tangible input devices with dynamic software contexts by generating adaptive function mappings for hardware controls. To achieve this, *TangiMap* analyzes the user's declared hardware specifications, the available controls, and the target software context, and uses large language models (LLMs) to assign suitable functions to each control element. To support diverse contexts, *TangiMap* presents multiple strategies such as API queries and screen parsing. We demonstrate the system through four scenarios: interactive adjustment of natural language parameters in 3D modeling, live music performance, task-aware mapping of on-screen controls in image editing, and dynamic puppeteering in animation.

UIST '26, Detroit, MI, USA

ACM ISBN 979-8-4007-2856-3/2026/11
https://doi.org/10.1145/3830398.3830602

CCS Concepts

• **Human-centered computing** → **Interactive systems and tools**; *Human computer interaction (HCI).*

Keywords

Tangible Input Device; Ephemeral UI; Large Language Models; Function Mapping

ACM Reference Format:
Yunyi Zhu and Chang Xiao. 2026. TangiMap: Context-Aware Generative Function Mapping for Tangible Input Devices. In *The 39th Annual ACM Symposium on User Interface Software and Technology (UIST '26), November 02–05, 2026, Detroit, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3830398.3830602

1 Introduction

Touch is the first sense to develop in humans and remains central throughout life for perceiving both our bodies and the physical world [8]. From early hand tools to today's complex mechanical panels, people have built devices that respond to the hand with tangible feedback, enabling efficient and precise work. In the digital era, this tangible interaction remains indispensable: a keyboard enables eyes-free, rapid text entry; an editing console supports intuitive,

simultaneous parameter changes for digital design; and a game controller affords quick, embodied reactions in fast-paced play. These examples show how physical controls have stood the test of time, adapting to different contexts while remaining a go-to solution. Extensive research on tangible interfaces has demonstrated that physical controls (such as dials, knobs, sliders, and buttons) enable efficient, precise, and often simultaneous manipulation of parameters, leading to measurable performance benefits and higher user preference compared to purely graphical user interfaces (GUI) [36].

Despite these advantages, tangible interfaces face a fundamental limitation: their functionality is fixed to the hardware form and must be explicitly mapped to software functions. A slider may adjust brightness in a photo editor or volume in a music mixer, but such mappings are typically predefined or manually assigned by the user. This rigidity is acceptable in stable contexts such as gaming, where mappings rarely change, but in complex software environments with multiple modes and evolving workflows, reusing the same physical controls across contexts demands constant remapping. This manual overhead is tedious, and it is also impractical to rely on universal presets across all scenarios, since users often own different types and configurations of tangible devices.

Moreover, recent advances in large language models (LLMs) have further renewed interest in natural language interfaces (NLIs), which allow users to specify tasks in free-form language. Such interfaces support fluid expression of intent but struggle to provide precise, fine-grained, and interactive parameter control. This gap has motivated the emergence of ephemeral UIs, which are lightweight, temporary controls that appear on demand [32]. Yet such fast-changing, dynamic scenarios are particularly challenging for tangible input, whose physical rigidity resists the flexibility required by these new interaction paradigms.

These observations raise a central research question: **can we make tangible interfaces as dynamic and adaptive as software widgets, automatically assigning appropriate functions to physical controls based on the user's current context?** To address this, we present *TangiMap*, a proof-of-concept system that bridges tangible input devices with dynamic software contexts by generating adaptive function mappings for hardware widgets, individual physical input controls on a tangible device, such as knobs, sliders, and buttons (Figure 1). To achieve this, *TangiMap* analyzes the user's context, the type of available widgets, and the specifications of the target software, and uses large language models (LLMs) to assign suitable functions to the tangible widgets.

Our work makes three primary contributions: (1) an end-to-end pipeline for dynamically mapping software functionalities onto tangible input devices; (2) a method for serializing diverse software and device environments into natural language contexts; and (3) four application scenarios that demonstrate the potential of *TangiMap* across different uses, software support and device types.

2 Related Work

2.1 Adaptive Tangible Interface

Tangible interaction has long been a central topic in HCI, offering people precise and intuitive ways to engage with digital systems [24, 29, 34, 39, 40]. However, tangible interfaces are typically fixed in form and function, and thus often regarded as less flexible than software-based alternatives. From the early days of the field, researchers have envisioned tangible interfaces that could adapt their functionality to different contexts and tasks. *Bricks* [6] introduced graspable interfaces where physical objects serve as handles to digital information, while *Tangible Bits* [17] envisioned seamless integration between atoms and bits.

Research has explored different approaches to creating adaptive tangible interfaces. One direction is *shape-changing systems* that physically transform to match their function. *inFORM* [7] uses 900 actuated pins to dynamically render physical constraints and affordances, while *Materiable* [25] simulates different material properties through the same physical surface. More recently, *ReactFold* [38] introduces a method for tracking paper as an interface, enabling flexible functionality and dynamic shape transformations. Shape-changing has also been applied at the scale of individual input controls. For example, *KnobSlider* [20] combines the affordances of a knob and a slider into a single shape-changing device, supporting eyes-free professional use without the bulk of traditional mixing consoles, and *DynaKnob* [35] explores how a physical knob can dynamically reconfigure both its shape and haptic force feedback. *ShapeTape* [13] is a flexible curve input that senses bending and twisting along a flexible strip, and digital tape drawing [2] adapts the two-handed curve-layout technique of automotive designers to a large display.

Another direction is to build *modular systems* that enable functional reconfiguration through rearrangement. *Topobo* [28] introduces kinetic memory where identical components learn different behaviors, *Siftables* [23] changes function based on spatial arrangement, and Glauser et al. [10] demonstrates a hardware implementation of modular and reconfigurable interfaces for animation rigging. Greenberg and Boyle's customizable physical interfaces [11] let users bind physical controls to the functions of conventional applications, *VoodooSketch* [3] extended interactive surfaces with physical palettes whose sketched or plugged-in controls are assigned to functions on the fly, and *Dynamic Tangible User Interface Palettes* [30] repurposed spatially tracked, paper-like palettes as reconfigurable control surfaces.

Most relevant to our work is research on *reprogrammable interfaces*, where fixed physical controls can be reprogrammed with different digital functions based on context. Fiebrink et al. [5] demonstrated how the same tangible widgets could be repurposed for different groupware tasks. *SLAP Widgets* [37] maintained physical form while dynamically relabeling controls via rear projection. In the computer music domain, *SensOrg* [31] matched control modules to musical functions through ergonomic, modality-driven design, and the *reacTable* [19] assigned synthesis roles to tangible objects through spatial proximity grammars on a tabletop. Recent work such as *AdapTUI* [14] automatically optimizes the spatial layout of tangible interfaces when users move between environments for AR applications.

However, all previous work requires manually specifying the functionality of tangible widgets, domain-specific rule frameworks, or operates only within predefined environmental contexts. Our work advances this line by inferring mappings from the user's task context through LLM semantic reasoning, without per-application setup, enabling the automatic and intelligent generation of tangible functions across diverse hardware and software settings.

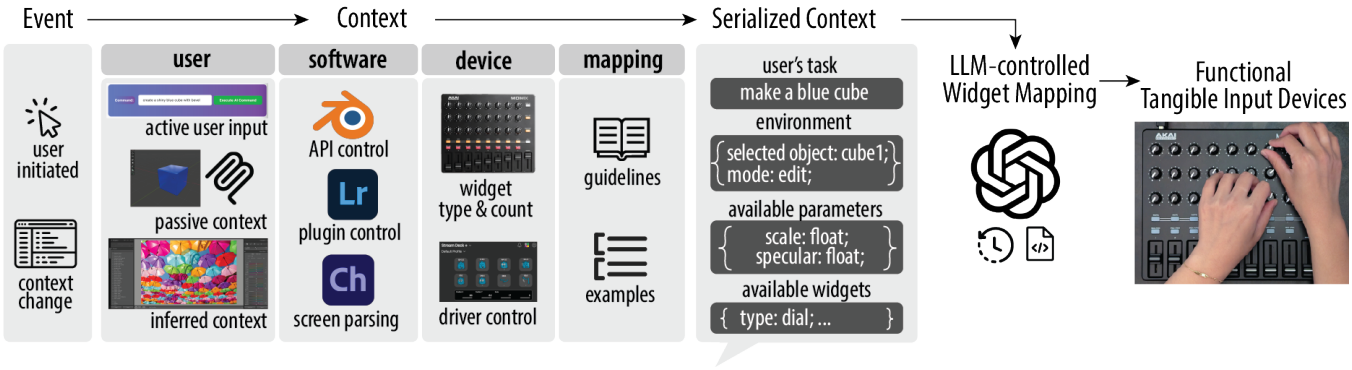

Figure 2: The *TangiMap* pipeline. The system begins with an event and gathers context from the user, software, and device, using methods suited to the specific environment. This context is serialized into a representation of the task, environment, parameters, and available widgets. An LLM-controlled mapping agent then selects relevant parameters, assigns them to widget types, and binds them to specific controls on the device.

2.2 Ephemeral and Generative User Interfaces

For decades, HCI has explored on-demand interfaces that appear when needed and adapt to user context. Early work concentrated on automatically generating *graphical* controls tailored to applications, devices, and user abilities. *Design Galleries* [22] constructs a tailored interface for navigating parameter settings in computer graphics and animation. *Huddle* [26] automatically produced task-based interfaces for coordinating multiple connected appliances by modeling content flow across devices. *Supple* [9] formalized interface generation as an optimization problem, enabling run-time personalization to a user's devices, tasks, preferences, and motor/vision capabilities, and showing speed and accuracy gains for users with motor impairments. *Bespoke* [33] transformed command-line functionality into graphical controls, and *ORCLayout* [18] used constraint-based methods to synthesize adaptive GUI layouts. While effective, these approaches are typically generated from a predefined function set and require substantial upfront modeling.

Recent advances in LLMs have enabled *generative* and *ephemeral* UIs that leverage richer context using natural language intent, operation history, and user preferences to create controls on demand. *DynaVis* [32] generates transient, task-specific controls for data visualization directly from natural language. *Biscuit* [4] scaffolds LLM-generated code with short-lived, contextual interfaces that help users steer and validate generation. *CrowdGenUI* [21] incorporates user preference data to produce interface variants that better match user goals and preferences.

Beyond the screen, generative approaches have begun to extend into tangible interaction. *SHAPE-IT* [27] translates natural-language requests into physical transformations on a shape display, effectively generating tangible affordances on demand. *Generative Muscle Stimulation* [16] connects an LLM with vision to localize objects in situ and actuate users' muscles to accomplish tasks, blending perception, planning, and embodied actuation.

Our work builds on this trajectory by bridging fixed tangible controls and dynamic task environments. We demonstrate that generative interaction is not limited to pixels on a screen, but can also reconfigure physical affordances on-the-fly.

3 System Structure

We present *TangiMap*, a system that dynamically maps software functionalities to tangible input widgets in a context-aware manner. As illustrated in Figure 2, the pipeline begins with an event, either initiated by the user or triggered by a change in context. To generate task-appropriate mappings, *TangiMap* integrates context from three sources: (i) the user, including explicit input, supporting context, or inferred intent; (ii) the software, including functions and tunable parameters obtained through APIs, plugins, or screen parsing; and (iii) the device, including widget types, counts, and control.

The collected context is serialized into a structured description of the current task, relevant parameters, and available widgets. An LLM-controlled mapping agent consumes this description, together with mapping guidelines and examples, to generate a widget–function mapping that respects user context (D1), software and device context (D2), widget compatibility (D3), and mapping consistency over time (D4). The resulting bindings are deployed to the physical controller, enabling real-time interaction with immediate feedback.

User Context (D1). Software typically encompasses more functionality than hardware can support. *TangiMap* identifies which functions are relevant to the user's current task and intent.

Software and Device Context (D2). *TangiMap* needs to know what parameters the software exposes, how to control them, and how to detect widget interactions on the connected device.

Widget Compatibility (D3). Different widget types suit different parameter types. *TangiMap* matches each function to the most appropriate control (button, knob, slider, etc.).

Mapping Consistency (D4). Generic input devices require users to relearn mappings each time they change. *TangiMap* preserves consistency over time to reduce this cognitive overhead.

Because *TangiMap* operates across different applications and hardware, we separate application-agnostic components (mapping guidelines, context processing, and mapping policy) from application-specific components (user context collection, and software and

device context collection). In the following subsections, we elaborate on each design consideration and discuss strategies to ensure portability across software environments and input devices.

3.1 User Context (D1)

Extracting user context is important for deciding which software functions are relevant and worth mapping to the tangible input device. User context extraction is inherently application-specific, as it depends on how much information the application exposes and the level of intention the user reveals. Some applications provide APIs that enable detailed monitoring, while others restrict access. To accommodate this variability, *TangiMap* employs three ways for extracting and serializing user context: active user intent input, passive user context, and inferred user context. The three types of context can be used together to provide a more complete picture of the user context.

Active user intent input. *TangiMap* serializes active intent when it is explicitly expressed by the user. In natural language interfaces, for example, the user directly specifies their task, allowing the system to identify and prioritize functions most relevant to the current task. When active intent is available, it is serialized by directly incorporating the user's stated task into the system.

Passive user context. In most graphical user interfaces, users do not directly state their goals but instead interact with the software through its interface. Some applications provide APIs or tool calls that allow *TangiMap* to extract contextual information, such as the currently selected object, the active window, and a history of recent events. This contextual data is then incorporated into the widget mapping process by being serialized into descriptive inputs that help align available functionalities with the user's current interaction state.

Inferred user context. When applications do not expose APIs or tool calls, *TangiMap* infers context through screen parsing. Specifically, we use a vision–language model (VLM) to parse the user's current application state, including the active window and the cursor-focused element. The VLM also identifies graphical interface widgets such as sliders and buttons together with their labels. These elements are then serialized and inserted into the system as a list of parameters currently showing on the interface.

3.2 Software and Device Context (D2)

Extracting software and device context requires identifying which parameters the software exposes, determining how those parameters can be controlled, and understanding how hardware widgets can be used to control them. Software and device context enable *TangiMap* to generate functional mappings between applications and tangible devices. Same as user context, software and device context extraction is application-specific: different programs provide varying levels of API or plugin access, and devices vary in how their drivers expose control pathways. To accommodate this, *TangiMap* supports multiple strategies for extracting and representing software and device context, allowing the system to function across various types of environments.

Software Context. The software context consists of two main elements: the available parameters and their corresponding control functions. The format of the software context representation is inspired by the Model Context Protocol (MCP) [1] tool call. Specifically, the software context is expressed as a JSON file that lists parameters together with their descriptions, types, ranges, and control functions. This structured representation is then passed to the system to support context-aware mapping.

Different applications expose varying levels of API access and control, requiring different strategies for extracting software context. When an application provides a formal API, *TangiMap* can directly query internal functions and parameters, generating a structured context from the specification. If only plugins or extensions are available, *TangiMap* uses them to capture the subset of parameters they expose. In the absence of both APIs and plugins, the system falls back on screen parsing to identify control opportunities such as hotkeys.

Device Context. The device context consists of two main elements: the type and count of available widgets, and the control pathways that connect these widgets to software functions. Widget types and counts can be specified manually or extracted automatically by capturing an image of the device and processing it with a vision–language model (VLM). Control pathways depend on the device's drivers: some devices use closed drivers that are difficult to extend, while others provide open protocols. To ensure consistency across devices, *TangiMap* represents all controls using the MIDI protocol. For native MIDI devices, the MIDI input is used directly, and for non-MIDI devices, we use or write a plugin that generates MIDI output on a virtual channel so that the system can listen for widget events and trigger the corresponding software functionality.

3.3 Widget Compatibility (D3)

To guide the system in generating appropriate mappings, *TangiMap* incorporates application-agnostic context in the form of general guidelines and concrete examples. The guidelines capture heuristics about which widget types are best suited for different parameter categories, while the examples provide reference mappings drawn from existing devices and workflows.

Mapping Guidelines. We used an LLM to generate heuristic guidelines indicating which types of parameters are most suitable for different tangible input widgets. For example, continuous parameters without defined bounds are better suited to dials, parameters with limited ranges, such as brightness, are better suited to sliders, while discrete actions such as toggles or mode switches are more appropriate for buttons. These guidelines were then reviewed and refined through author inspection to ensure their applicability and consistency with established interaction design practices.

3.4 Mapping and Execution (D3, D4)

The LLM-controlled widget mapping system integrates encoded contexts from the user, software, device, and general guidelines to generate mappings between software functions and tangible widgets. It then executes these mappings by binding functions to specific device controls, transforming the device into a functional controller for the target application. Figure 3 illustrates this process,

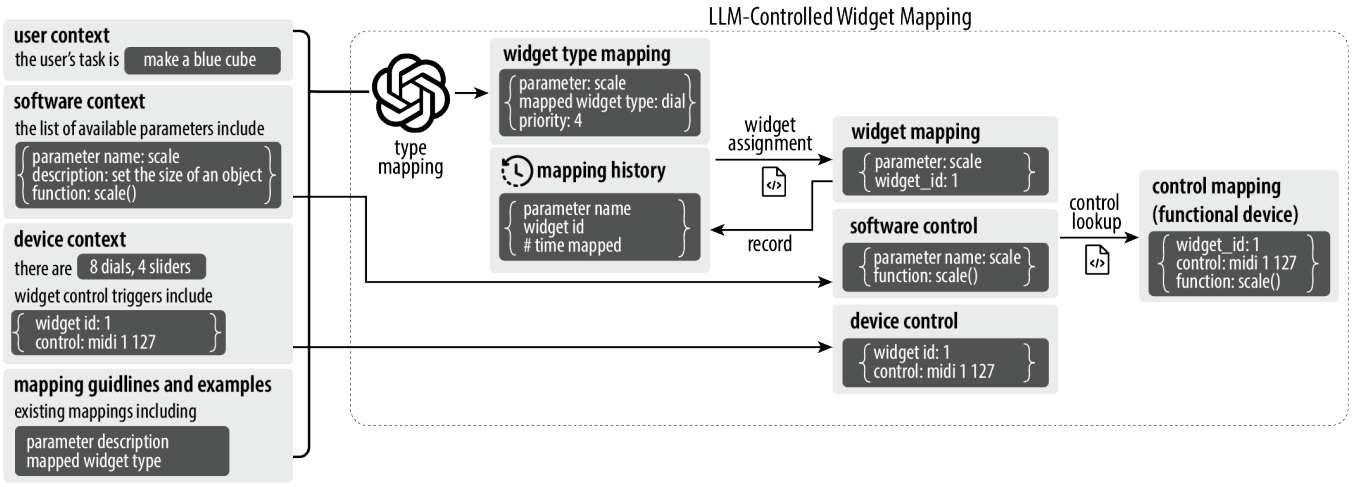

Figure 3: The LLM-controlled widget mapping process. User, software, and device contexts are combined with mapping guidelines and examples to generate a list of relevant parameters and their corresponding widget types. The system then includes mapping history to assign parameters to specific widgets. Finally, it performs a control lookup to bind software functions to device controls, producing the final executable control mapping.

where the contexts are serialized in a single LLM call, followed by conflict resolution, widget assignment, and control lookup. Appendix B provides pseudocode for the full pipeline.

Widget Type Mapping. *TangiMap* uses a single structured prompt consisting of a role directive followed by four context sections: user context, software context, device context, and design guidelines. Only the software-context section is application-specific: for Blender, we link the Python API documentation and enable the model's web search; for the other applications, parameters are provided as a JSON dictionary (Section 3.2). An example prompt is provided in Appendix A. The system performs single-pass inference with an LLM (Gemini 2.5 Flash) over the encoded contexts and selects parameters from the software context that are relevant to the user's current task. These parameters are mapped to appropriate widget types (e.g., buttons, sliders, dials) based on the provided guidelines and examples (D3), and returned as structured JSON, where each entry specifies a software function, its assigned widget type, and a priority score (1-10) reflecting its relevance to the user's context. The JSON output is parsed before further processing, and the inference call is retried if parsing fails. Capacity constraints are enforced in-prompt through per-type widget counts, so the LLM is aware of the number of available widgets on the device and does not exceed its capacity.

Conflict Resolution. Before any widget is assigned, the system enforces device capacity: in rare cases where the widget type mapping stage proposes more functions of a type than the device has widgets of that type, the candidates are sorted by priority score and only the top-ranked ones within each widget type are retained, dropping the lowest-ranked. Together with the in-prompt capacity constraints, this guarantees that widget assignment never has to consider more candidates than the device can host.

Widget Assignment. After capacity has been enforced, the system assigns each surviving function to a specific widget on the device. This step consults the user's mapping history first to keep the mappings consistent with past setups (D4). When a function has been mapped before, the system reuses the same widget to preserve familiarity. In cases of overlap, the function that has been mapped more frequently is given priority, and the losing function falls back to the next unassigned widget of its type.

Control Lookup. The final stage of the pipeline is control lookup, which establishes the connection between tangible widgets and software functionality. The system queries the software context to identify the function associated with each parameter and the device context to determine the corresponding MIDI channel and control change (CC) number for the widget, resolved via a per-device profile recorded once at setup. It then listens on the specified MIDI channel and executes the mapped software function whenever the widget is interacted with. This step closes the loop between tangible input and software control, completing the mapping pipeline and enabling real-time interaction.

4 Technical Evaluation

We evaluate whether TangiMap's widget-type assignments (e.g., knob, slider, button, pad) align with human-authored reference configurations. This evaluation is a sanity check of the widget-type mapping stage in isolation, and does not assess spatial assignment and overall user preference (see Section 6).

4.1 Dataset

We collected 16 human-authored reference configurations drawn from real-world device profiles for fixed-functionality devices and publicly available preset libraries for configurable devices. To construct each configuration, we collected device documentation and

reference images of physical devices from publicly available manufacturer websites and community-shared preset libraries. Each image was manually inspected and encoded as a structured JSON capturing device metadata (device name, manufacturer, model, target software, and use case), a full control inventory organized by type, and human-authored widget-to-function assignments.

The resulting set spans 8 fixed-layout and 8 configurable-layout devices and 305 controls in total, with per-device inventories ranging from 7 to 30 controls. It covers knobs, dials, faders, buttons, pads, encoders, switches, jog wheels, and trackballs, across categories including color grading panels, MIDI and DJ controllers, modular editing consoles, and game controllers, targeting software from Ableton Live and Rekordbox to Lightroom, Photoshop, and Spotify. Table 1 in Appendix C lists every configuration individually.

4.2 Setup and Metric

To evaluate our system against the human-authored reference data, we convert the raw data into prompt/answer evaluation pairs: the prompt simulates the TangiMap context input, including user context, software context, and the hardware control inventory, while the human-authored reference answer specifies the human-authored widget-to-function assignments.

For each configuration, we provided TangiMap with the device's control inventory (widget types and counts) and the software context (active functionality list with descriptions), and then prompted the system to assign a widget type to each functionality. The system outputs a list of widget type mappings, which are compared with the human-authored reference assignments.

To measure whether the system correctly reasons about widget modality (e.g., assigning a continuous parameter to a knob or fader rather than a button), we measure the **reference alignment rate**: the proportion of functionality–widget pairs in which the system's predicted widget type matches the widget type in the human-authored reference configuration.

4.3 Results and Discussion

Across all 16 configurations, TangiMap achieved an aggregate reference alignment rate of 99.1%. 15 of the 16 configurations reached 100% alignment with the human-authored references. The only misaligned mapping comes from the TourBox device with the Clip Studio Paint drawing software, where the TangiMap system maps "switch between previous and current colors" to a button and "clear selection" to a knob, while the human-authored reference does the opposite. This result suggests that the widget-type mapping stage produces assignments consistent with how human authors assign widget types in these configurations.

5 Applications

In this section, we demonstrate how *TangiMap* supports creative workflows through four scenarios: (1) **Interactive adjustment of NLI-generated parameters**, where widgets are mapped to the adjustable controls exposed by natural-language commands; (2) **Live performance in computer music**, where a generic mixer replaces dedicated hardware for a custom-written application; (3) **Color grading on the same mixer**, where the controls that suit audio mixing are repurposed for image adjustment; and (4) **Puppeteering for animation**, where widgets drive live parameter changes for expressive motion. Each scenario exercises a distinct interaction mode and varies in both the software and device context.

5.1 Interactive Adjustment of NLI-Generated Parameters

As natural language interfaces become more prevalent, researchers have explored ways to help users explain their goals with greater precision [32]. One approach is to present ephemeral, custom-generated UIs that let users adjust parameters of functions related to a user's command. Because these interfaces are created on the fly and exist only temporarily, no existing system can dynamically map them to tangible input devices. Consequently, users miss out on the precision and immediacy that tangible widgets provide.

TangiMap addresses this gap by mapping the generated parameters onto the tangible widgets available to the user, extending the benefits of tangible interaction to systems that rely on ephemeral UIs. We demonstrate this through the example of adjusting parameters in natural language 3D modeling with Blender (Figure 4).

User Context. While interacting with the Blender interface, the user begins by entering a natural language command, such as "create a shiny blue cube with bevel." (Figure 4a) At this stage, the user context includes the prompt to the NLP interface, the Blender commands generated by the interface, and the current scene state, including selected objects, workspace, active windows, and all manipulable objects. (Figure 4b)

Software and Device Context. *TangiMap* then identifies relevant parameters from the software context, which consists of the tools exposed through Blender's MCP plugin and the Blender Python API (bpy), which is provided through including its source link. (Figure 4b) These parameters are assigned to the available widgets of the Monogram Creative Console[1], used here as the tangible input device. Dynamic functionality is achieved by enabling MIDI mode in the device's driver, which allows flexible assignment of controls to widgets.

Mapping and Execution. The resulting mappings are communicated to the user through a visualization that replicates the device layout, with each widget annotated by a generated label explaining its functionality (Figure 4c). For example, the system may assign a knob for adjusting the "bevel size" and a slider for controlling the "shininess." Once mapped, the user can adjust parameters directly by interacting with the device (Figure 4d).

5.2 Live Performance in Computer Music

In computer music performance, tangible control remains essential: performers depend on physical controls for eyes-free, real-time manipulation. The domain is therefore served by dedicated hardware such as professional DJ systems, which are expensive and fixed device layout and software application. A performer who works with different software, or with a custom application of their own, usually has no preset to fall back on. We therefore demonstrate this scenario on a computer music application written with an AI coding

[1] https://monogramcc.com/

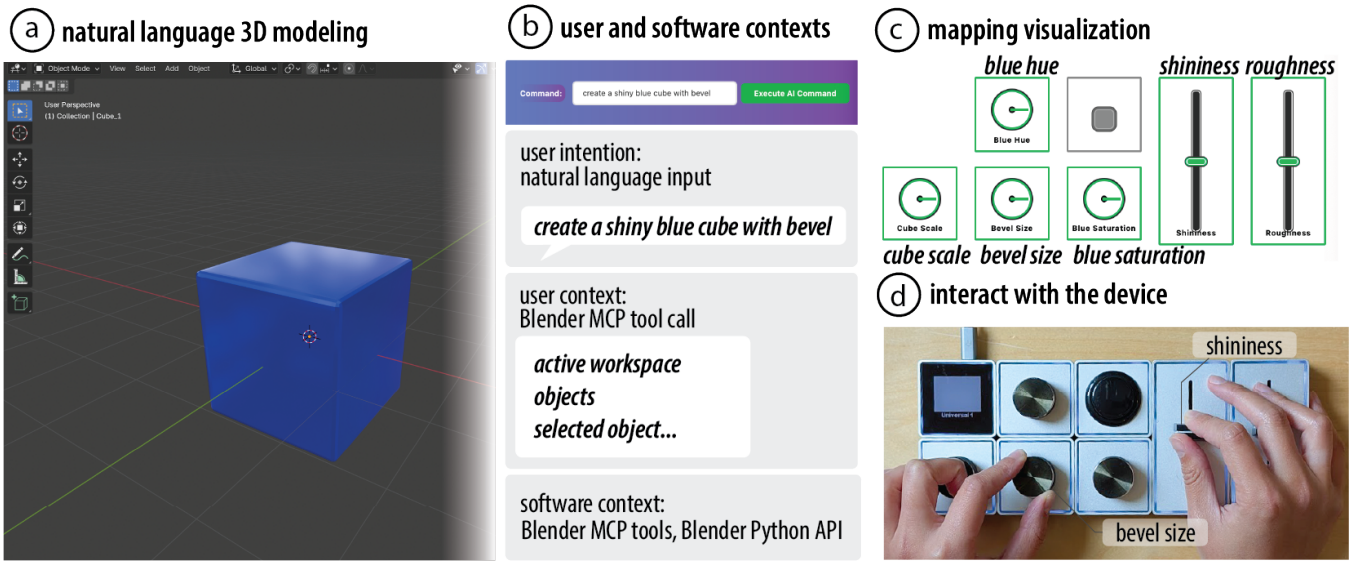

Figure 4: *TangiMap* enables interactive adjustment of parameters generated from natural language input. (a) The user performs 3D modeling in Blender using natural language input, "create a shiny blue cube with bevel". (b) *TangiMap* collects user and software contexts through the user's input and the Blender API. (c) Relevant parameters are selected and mapped onto the tangible device. (d) The user adjusts these parameters, such as "bevel size" and "shininess", by turning knobs and moving sliders.

agent (Claude Fable 5), driven by an affordable, general-purpose MIDI mixer.

The application comprises two playback decks, a two-channel mixer with three-band EQs and filters, beat-synchronized effect units, a sampler, and a track library, exposing over 100 controllable functions through a control registry (Figure 5a). By reading that registry directly, *TangiMap* produces the application's first hardware mapping. The tangible input device is an Akai MIDIMix, a compact DAW mixer built for studio production.

User Context. Because the application is custom-written, it exposes its full state, and user context is collected directly as the log of recent user control actions. From this state, *TangiMap* infers the performer's current workflow phase, such as browsing the library, preparing the set, or executing a mix transition (Figure 5b).

Software and Device Context. The software context is read live from the control registry, which lists every function with its description, value range, and current value. Because the registry exposes more functions than the device has widgets, the inferred workflow acts as a filter: for each phase, the registry reports the subset of functions most relevant to it. Both decks' tempo faders are retained in every phase, so beatmatching remains available at any moment. The MIDIMix's widgets are accessed directly through MIDI.

Mapping and Execution. Remapping requires a manual button press on the *TangiMap* visualizer, so the mapping never changes unprompted mid-performance (Figure 5c). This keeps the performer, rather than the system, in control of when the hardware reconfigures. In our demonstration, the performer creates a mapping prior to their performance, which they then practice and perform with (Figure 5d). The current mapping is communicated through the device-layout visualization with generated labels.

5.3 Repurposing the Same Mixer for Color Grading

Color grading places the same demands on tangible control as audio mixing: continuous parameters, adjusted simultaneously, with the eyes on the image rather than on the controls. Dedicated consoles such as the Loupedeck+[2] or the DaVinci Resolve Color Panel[3] serve exactly this need, but they are typically single-purpose, with fixed widget mappings that cannot be adapted to other software or to additional functions within the same software, and their limited physical space restricts the number of parameters that can be controlled at once. Because the widget types a mixer offers are the ones color grading needs, *TangiMap* can instead repurpose the same MIDIMix from the previous scenario, for which no Lightroom preset exists.

User Context. The user navigates to the color channel adjustments in Lightroom (Figure 5e). *TangiMap* parses the interface with a vision–language model (Gemini 2.5 Flash) to identify the active window, panel, and visible controls, thereby capturing the user's current editing focus (Figure 5f). In this case, the parsed context includes hue, saturation, and luminance adjustments for eight color channels: red, orange, yellow, green, aqua, blue, purple, and magenta. The captured context updates automatically as the user scrolls to different parts of the application.

Software and Device Context. Lightroom functions are accessed via the MIDI2LR plugin, which exposes them as fixed MIDI

[2]https://loupedeck.com/us/products/loupedeck-plus/
[3]https://www.blackmagicdesign.com/products/davinciresolve/panels

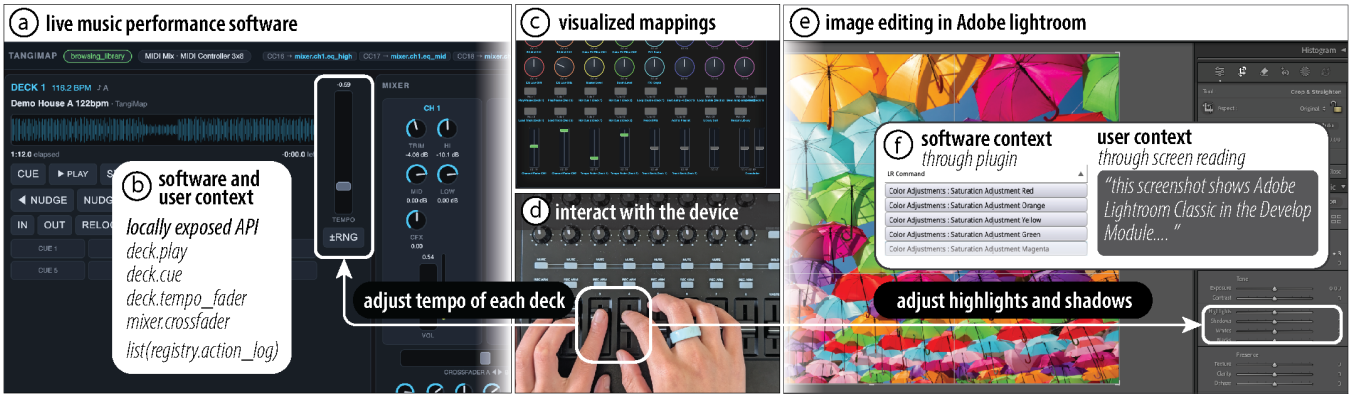

Figure 5: *TangiMap* maps the same generic mixer (Akai MIDIMix) onto two different applications. (a) The performer works in a custom-written computer music performance application. (b) *TangiMap* reads software and user context directly from the application's locally exposed API and control-action log. (c) The generated mapping is visualized on the device layout. (d) The performer adjusts the tempo of each deck on the mixer. (e) The same mixer then serves color grading in Adobe Lightroom, where the user adjusts highlights and shadows. (f) The Lightroom software context is captured through the MIDI2LR plugin and user context through screen parsing.

commands. We create a profile in MIDI2LR that maps all available functions to unused channels, and use a custom MIDI remapper to dynamically redirect the mixer's inputs to the channels corresponding to the active parameters (Figure 5f).

Mapping and Execution. *TangiMap* assigns the mixer's knobs and faders to the controls currently visible in Lightroom and communicates these mappings back to the user through a visualization of the device layout annotated with generated labels (Figure 5e). The user can then adjust multiple parameters simultaneously on the MIDIMix device, combining the embodied efficiency of tangible controls with the flexibility of the software interface (Figure 5e). As the context updates, the mappings adapt accordingly. For example, when the crop window is active, the knobs are remapped to parameters such as rotation angle.

5.4 Puppeteering for Animation

In video editing and animation, puppeteering refers to the live control of characters or objects, supporting both real-time performance and the rapid creation of animated sequences, and is often more efficient than traditional timeline editing [15]. Effective puppeteering typically requires external input devices with one-to-one mappings between controls and actions, as well as simultaneous manipulation of multiple parameters, which makes conventional GUIs insufficient. While existing software can support such workflows, setting up the necessary framework often requires manual configuration.

TangiMap streamlines the setup of puppeteering by automatically mapping available animation triggers onto a tangible device, allowing users to establish a working configuration in a single step (Figure 6). We demonstrate this through an application built on Adobe Character Animator, a tool for designing and animating characters, using the Elgato Stream Deck+[4] as the tangible input device.

[4]https://www.elgato.com/us/en/p/stream-deck-plus

User Context. Because Character Animator does not provide an SDK, the system captures the user context through screen parsing with a vision-language model (Gemini 2.5 Flash). The interface is analyzed to extract text labels and action icons from cropped regions of the recorded screen, identifying the set of triggers available for the active character (Figure 6b).

Software and Device Context. Similar to the user context, the software context consists of the parsed controls identified through screen parsing. On the device side, we implemented a custom Stream Deck plugin that supports updating button icons from local images and remapping their output MIDI channels to align with the extracted controls (Figure 6c).

Mapping and Execution. *TangiMap* assigns the parsed animation triggers to individual buttons on the device and updates each button to display the corresponding trigger image. The user can then act out the character actions and expressions in real time (Figure 6d).

6 Discussion and Future Work

In this paper, we designed and implemented a first working prototype of a pipeline that automatically selects and maps software functionalities onto tangible input devices. *TangiMap* is an early exploration of generative function mapping rather than a fully validated interaction technique, and this discussion accordingly focuses on its current boundaries.

Dynamic vs. Fixed Mappings. Tangible controls support eyes-free, rapid, and embodied interaction, and it is possible for on-the-fly remapping to undercut these benefits by requiring users to relearn the interface. We see dynamic mapping as suited to software with large or uniquely generated parameter spaces, where no fixed preset can anticipate which functions will be relevant, as in our 3D modeling. Fixed mappings remain preferable for workflows that depend on muscle memory, such as instrument-style playing in live

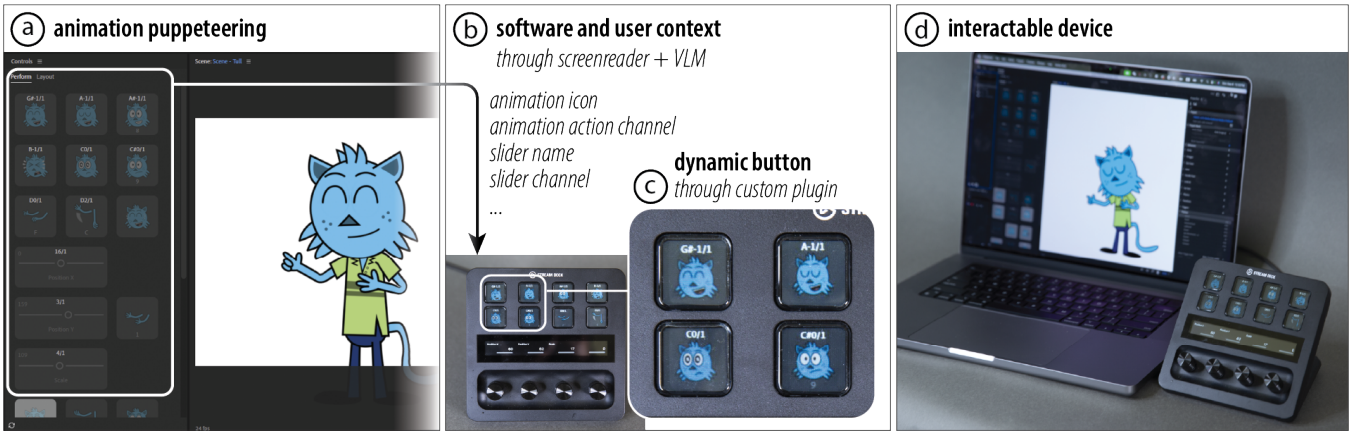

Figure 6: *TangiMap* enables automatically setting up puppeteering system for animation in Adobe Character Animator. (a) The Adobe Character Animator interface with triggers and parameter sliders. (b) User and software contexts are captured through screen parsing with a VLM, extracting text and icons from the interface. (c) The triggers are mapped onto the Elgato Stream Deck+, with a custom TangiMap plugin updating button icons to display the corresponding trigger images. (d) The user can interact with the device to act out the animation in real time.

music. The two are not mutually exclusive. In future work, we plan to include the functionality where users can pin individual controls while others remap, or freeze a *TangiMap*-generated mapping as a fixed preset.

6.1 Limitations

6.1.1 System Design Limitations.
The following limitations concern how *TangiMap* reasons about and presents its mappings internally.

Spatial Layout Consideration. *TangiMap* focuses primarily on widget type mapping and overall system development. Spatial layout is addressed only through considerations of mapping consistency. In future work, we plan to incorporate broader design considerations around spatial layout, such as grouping widgets into logical clusters and distributing them ergonomically across the device.

User Preference and Manual Override. Users may have specific preferences about how their widgets are mapped. Our current implementation uses automated control, prioritizing mappings based on usage frequency so that more frequently used controls are assigned first. In future work, we plan to explore ways to allow users to provide more explicit preference input, such as loading a preference configuration, manually adjusting mappings for the system to learn from, and providing a rollback path to an earlier configuration.

Widget Visualization. The current implementation of *TangiMap* runs on a PC and sometimes relies on an on-screen overlay to communicate widget mappings to the user, requiring users to shift their attention between the screen visualization and the physical device, which introduces an added layer of mental load. In one of the applications, we show how modern configurable input devices, such as the Elgato Stream Deck+, can display mapping labels directly on the widgets themselves.

6.1.2 Infrastructure Limitations.
The following limitations concern the external hardware and software environments that *TangiMap* depends on.

Hardware Constraints. *TangiMap* can only repurpose the widgets available on a given device, and cannot overcome the physical constraints of limited or fixed hardware layouts. When a device provides only a small number of controls, the range of parameters that can be mapped remains restricted. The widget vocabulary is also conventional: knobs, sliders, and buttons cover only part of the input space, and tasks such as manipulating deformable surfaces or character kinematics may be better served by devices such as *ShapeTape* [13], potentially with force feedback. In the future, we plan to support modular or self-assembling hardware, such as reconfigurable input widgets [12], and to extend the device guidelines so that such affordances can be expressed in the prompt.

Application Access Limitation. *TangiMap* spans a range of access levels: a full local API in the computer music application, an MCP plugin in Blender, a fixed-channel plugin in Lightroom, and no programmatic access at all in Character Animator, where we fall back on screen parsing. Screen parsing works but must be tuned per application, and as tool interfaces such as MCP become standard that fallback should keep narrowing.

6.2 Toward Screen-Free Interaction

While the current implementation of *TangiMap* functions as an add-on to existing GUI interfaces, it points to the future possibility of tangible input devices that extend beyond screen-based interactions. Across our applications the LLM is engaged in two ways. In 3D modeling it is explicit: the user states an intent and the mapping follows from it. In color grading, puppeteering, and live performance it is implicit, inferring intent from the interface or performance state without being addressed directly. Both directions lead away from the screen. Explicit use could move to voice, asking aloud for

the color controls rather than reaching for a panel, while implicit use could draw on ambient sensing, such as a camera watching the physical workspace, to follow context without interrupting the task. The output side can leave the screen as well: in sound production the audio is itself the output, monitored directly without visual panels, which is part of why mixers remain the primary control surface, and future outputs could be physical, through shape-changing interfaces or other dynamic objects. Together these describe a loop in which context arrives without a screen, control happens in the hand, and the result is heard or felt rather than displayed.

7 Conclusion

In this paper, we presented *TangiMap*, a proof-of-concept system that leverages large language models to dynamically map software functionalities onto tangible widgets by analyzing user, software, and device contexts. Through four application scenarios in 3D modeling with natural language interfaces, live music performance, image editing with simultaneous parameter adjustments, and animation puppeteering, we demonstrated how *TangiMap* enables flexible, context-aware use of existing tangible devices and can operate across different types of software and hardware with varying capabilities. This first exploration highlights the potential of combining tangible interfaces with adaptive mapping to move beyond single-purpose workflows and toward more versatile interactions with tangible input devices.

Acknowledgments

This work was supported in part by a Gemini Academic Program Award. The initial work was conducted while Yunyi Zhu was an intern and Chang Xiao was a Research Scientist at Adobe Research. We thank Dacia Saenz, Dave Werner, Luis Figueroa, Mike Berry, Michael Vangen, Eric Sanders, Eunyee Koh, and Alex Nikiforov for their early feedback on the project, and Alex in particular for lending us the Monogram Creative Console.

References

[1] Anthropic. 2024. Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol.

[2] Ravin Balakrishnan, George Fitzmaurice, Gordon Kurtenbach, and William Buxton. 1999. Digital tape drawing. In *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology (UIST '99)*. 161–169. https://doi.org/10.1145/320719.322598

[3] Florian Block, Michael Haller, Hans Gellersen, Carl Gutwin, and Mark Billinghurst. 2008. VoodooSketch: extending interactive surfaces with adaptable interface palettes. In *Proceedings of the 2nd International Conference on Tangible and Embedded Interaction (TEI '08)*. 55–58. https://doi.org/10.1145/1347390.1347404

[4] Ruijia Cheng, Titus Barik, Alan Leung, Fred Hohman, and Jeffrey Nichols. 2024. BISCUIT: Scaffolding LLM-Generated Code with Ephemeral UIs in Computational Notebooks. In *Proceedings of the 2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. https://doi.org/10.1109/VL/HCC60511.2024.00012

[5] Rebecca Fiebrink, Dan Morris, and Meredith Ringel Morris. 2009. Dynamic mapping of physical controls for tabletop groupware. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. 471–480. https://doi.org/10.1145/1518701.1518778

[6] George W Fitzmaurice, Hiroshi Ishii, and William AS Buxton. 1995. Bricks: laying the foundations for graspable user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '95)*. 442–449. https://doi.org/10.1145/223904.223964

[7] Sean Follmer, Daniel Leithinger, Alex Olwal, Akimitsu Hogge, and Hiroshi Ishii. 2013. inFORM: dynamic physical affordances and constraints through shape and object actuation. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. 417–426. https://doi.org/10.1145/2501988.2502032

[8] Matthew Fulkerson. 2013. *The first sense: A philosophical study of human touch*. MIT Press.

[9] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. 2007. Automatically Generating User Interfaces Adapted to Users' Motor and Vision Capabilities. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology (UIST '07)*. 231–240. https://doi.org/10.1145/1294211.1294253

[10] Oliver Glauser, Wan-Chun Ma, Daniele Panozzo, Alec Jacobson, Otmar Hilliges, and Olga Sorkine-Hornung. 2016. Rig animation with a tangible and modular input device. *ACM Transactions on Graphics* 35, 4, Article 144 (2016), 11 pages. https://doi.org/10.1145/2897824.2925909

[11] Saul Greenberg and Michael Boyle. 2002. Customizable physical interfaces for interacting with conventional applications. In *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology (UIST '02)*. 31–40. https://doi.org/10.1145/571985.571991

[12] Saul Greenberg and Chester Fitchett. 2001. Phidgets: easy development of physical interfaces through physical widgets. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology (UIST '01)*. https://doi.org/10.1145/502348.502388

[13] Tovi Grossman, Ravin Balakrishnan, and Karan Singh. 2003. An interface for creating and manipulating curves using a high degree-of-freedom curve input device. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '03)*. 185–192. https://doi.org/10.1145/642611.642645

[14] Fengming He, Xiyun Hu, Xun Qian, Zhengzhe Zhu, and Karthik Ramani. 2024. AdapTUI: Adaptation of Geometric-Feature-Based Tangible User Interfaces in Augmented Reality. *Proceedings of the ACM on Human-Computer Interaction* 8, ISS (2024), 44–69. https://doi.org/10.1145/3698127

[15] Robert Held, Ankit Gupta, Brian Curless, and Maneesh Agrawala. 2012. 3D puppetry: a kinect-based interface for 3D animation. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* (Cambridge, Massachusetts, USA) *(UIST '12)*. 423–434. https://doi.org/10.1145/2380116.2380170

[16] Yun Ho, Romain Nith, Peili Jiang, Steven He, Bruno Felalaga, Shan-Yuan Teng, Rhea Seeralan, and Pedro Lopes. 2026. Generative Muscle Stimulation: Providing Users with Physical Assistance by Constraining Multimodal-AI with Embodied Knowledge.

[17] Hiroshi Ishii and Brygg Ullmer. 1997. Tangible bits: towards seamless interfaces between people, bits and atoms. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '97)*. 234–241. https://doi.org/10.1145/258549.258715

[18] Yue Jiang, Ruofei Du, Christof Lutteroth, and Wolfgang Stuerzlinger. 2019. ORC Layout: Adaptive GUI Layout with OR-Constraints. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. 1–12. https://doi.org/10.1145/3290605.3300643

[19] Sergi Jordà, Günter Geiger, Marcos Alonso, and Martin Kaltenbrunner. 2007. The reacTable: exploring the synergy between live music performance and tabletop tangible interfaces. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction (TEI '07)*. 139–146. https://doi.org/10.1145/1226969.1226998

[20] Hyunyoung Kim, Céline Coutrix, and Anne Roudaut. 2018. KnobSlider: Design of a Shape-Changing UI for Parameter Control. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. 1–13. https://doi.org/10.1145/3173574.3173913

[21] Yimeng Liu, Misha Sra, and Chang Xiao. 2024. CrowdGenUI: Enhancing LLM-Based UI Widget Generation with a Crowdsourced Preference Library. *arXiv preprint arXiv:2411.03477* (2024). https://doi.org/10.48550/arxiv.2411.03477

[22] J. Marks, B. Andalman, P. A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. 1997. Design galleries: a general approach to setting parameters for computer graphics and animation. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*. 389–400. https://doi.org/10.1145/258734.258887

[23] David Merrill, Jeevan Kalanithi, and Pattie Maes. 2007. Siftables: towards sensor network user interfaces. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction (TEI '07)*. 75–78. https://doi.org/10.1145/1226969.1226984

[24] Roderick Murray-Smith, John Williamson, Stephen Hughes, and Torben Quaade. 2008. Stane: synthesized surfaces for tactile input. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. 1299–1302. https://doi.org/10.1145/1357054.1357257

[25] Ken Nakagaki, Luke Vink, Jared Counts, Daniel Windham, Daniel Leithinger, Sean Follmer, and Hiroshi Ishii. 2016. Materiable: Rendering dynamic material properties in response to direct physical touch with shape changing interfaces. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. 2764–2772. https://doi.org/10.1145/2858036.2858104

[26] Jeffrey Nichols, Brandon Rothrock, Duen Horng Chau, and Brad A. Myers. 2006. Huddle: Automatically Generating Interfaces for Systems of Multiple Connected Appliances. In *Proceedings of the 19th Annual ACM Symposium on User Interface*

Software and Technology (UIST '06). 279–288. https://doi.org/10.1145/1166253.1166298

[27] Wanli Qian, Chenfeng Gao, Anup Sathya, Ryo Suzuki, and Ken Nakagaki. 2024. SHAPE-IT: Exploring Text-to-Shape-Display for Generative Shape-Changing Behaviors with LLMs. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24)*. https://doi.org/10.1145/3654777.3676348

[28] Hayes Solos Raffle, Amanda J Parkes, and Hiroshi Ishii. 2004. Topobo: a constructive assembly system with kinetic memory. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. 647–654. https://doi.org/10.1145/985692.985774

[29] Valkyrie Savage, Andrew Head, Björn Hartmann, Dan B. Goldman, Gautham J. Mysore, and Wilmot Li. 2015. Lamello: Passive Acoustic Sensing for Tangible Input Components. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. https://doi.org/10.1145/2702123.2702207

[30] Martin Spindler, Victor Cheung, and Raimund Dachselt. 2013. Dynamic Tangible User Interface Palettes. In *Human-Computer Interaction – INTERACT 2013*, Paula Kotzé, Gary Marsden, Gitte Lindgaard, Janet Wesson, and Marco Winckler (Eds.). 159–176. https://doi.org/10.1007/978-3-642-40498-6_12

[31] Tamas Ungvary and Roel Vertegaal. 1998. The SensOrg: a musical cyberinstrument with a cognitive ergonomic touch. In *Proceedings of the 1998 IEEE International Conference on Systems, Man, and Cybernetics (SMC '98)*, Vol. 2. 1066–1071. https://doi.org/10.1109/ICSMC.1998.727830 ISSN: 1062-922X.

[32] Priyan Vaithilingam, Elena L. Glassman, Jeevana Priya Inala, and Chenglong Wang. 2024. DynaVis: Dynamically Synthesized UI Widgets for Visualization Editing. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. https://doi.org/10.1145/3613904.3642639

[33] Priyan Vaithilingam and Philip J. Guo. 2019. Bespoke: Interactively Synthesizing Custom GUIs from Command-Line Applications By Demonstration. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. https://doi.org/10.1145/3332165.3347944

[34] Elise Van Den Hoven, Joep Frens, Dima Aliakseyeu, Jean-Bernard Martens, Kees Overbeeke, and Peter Peters. 2007. Design research & tangible interaction. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction (TEI '07)*. 109–115. https://doi.org/10.1145/1226969.1226993

[35] Anke van Oosterhout, Eve Hoggan, Majken Kirkegaard Rasmussen, and Miguel Bruns. 2019. DynaKnob: Combining Haptic Force Feedback and Shape Change. In *Proceedings of the 2019 ACM Designing Interactive Systems Conference* (San Diego, CA, USA) *(DIS '19)*. 963–974. https://doi.org/10.1145/3322276.3322321

[36] Simon Voelker, Kjell Ivar Øvergård, Chat Wacharamanotham, and Jan Borchers. 2015. Knobology Revisited: A Comparison of User Performance between Tangible and Virtual Rotary Knobs. In *Proceedings of the 2015 International Conference on Interactive Tabletops and Surfaces (ITS '15)*. 35–38. https://doi.org/10.1145/2817721.2817725

[37] Malte Weiss, Julie Wagner, Yvonne Jansen, Roger Jennings, Ramsin Khoshabeh, James D Hollan, and Jan Borchers. 2009. SLAP widgets: bridging the gap between virtual and physical controls on tabletops. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. 481–490. https://doi.org/10.1145/1518701.1518779

[38] Chang Xiao. 2025. ReactFold: Towards Camera-based Tangible Interaction on Passive Paper Artifacts. In *Proceedings of the Nineteenth International Conference on Tangible, Embedded, and Embodied Interaction (TEI '25)*. 1–12. https://doi.org/10.1145/3689050.3704926

[39] Chang Xiao, Karl S. Bayer, Changxi Zheng, and Shree K. Nayar. 2019. Vidgets: modular mechanical widgets for mobile devices. *ACM Transactions on Graphics* (2019). https://doi.org/10.1145/3306346.3322943

[40] Daniel Campos Zamora, Mustafa Doga Dogan, Alexa F. Siu, Eunyee Koh, and Chang Xiao. 2024. MoiréWidgets: High-Precision, Passive Tangible Interfaces via Moiré Effect. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. https://doi.org/10.1145/3613904.3642734

A Example Mapping Prompt

```
You are an expert in mapping software functions to physical hardware
controllers for creative tools. Given the user's current context, the
software's controllable functions, and the connected device's widget
capacity, select the functions most worth placing on hardware and the widget
type each should get.

## USER CONTEXT
- Current request: "make the key light warmer and fine-tune the sphere's
  size and roughness"
- Workspace: Shading | Mode: OBJECT
- Active object: MetallicSphere_001; selected: [MetallicSphere_001,
  KeyLight]
- Scene stats: {objects: 5, meshes: 2, lights: 2, cameras: 1, materials: 3}
- Recent interactions: ["create a metallic sphere", "add a key light"]

## SOFTWARE CONTEXT
Software: Blender (3D creation suite, controlled via the Python bpy API over
MCP).
Python API reference: https://docs.blender.org/api/current/
You may use web search against that reference to check what parameters an
operation exposes and their value ranges.
Available MCP tools: execute_code, get_scene_info, get_object_info,
analyze_scene_composition, ...

## DEVICE CONTEXT
Device: Monogram Creative Control (modular console; profile recorded at
setup)
Widget capacity: 6 knobs (dials), 3 sliders, 4
buttons (keys)

## DESIGN GUIDELINES
- Propose the software functions this user most plausibly wants on physical
  controls right now, given their request and scene.
- Prefer continuous parameters on knobs/sliders; discrete actions on
  buttons/switches.
- Name each function as a concrete, adjustable parameter or action (e.g.
  "light.energy"), not a vague theme.
- Assign a widget TYPE only; a later assignment step selects the specific
  widget instance.
- priority is an integer 1-10 (10 = most important for this task);
  priorities decide what is dropped if capacity runs short.
- Never propose more functions of a widget type than the device capacity for
  that type.
- Output ONLY valid JSON exactly in the format below.

## OUTPUT FORMAT
{"mappings": [{"function": "<name>", "widget_type": "knob|slider|button",
  "priority": <1-10>}, ...]}
```

B Mapping Pipeline Pseudocode

```
procedure GenerateMapping(userCtx, softwareCtx, deviceCtx, guidelines, history):
    prompt <- Serialize(userCtx, softwareCtx, deviceCtx, guidelines)
    // capacity (per-type widget counts) is stated directly in the prompt

    // Stage 1: widget-type mapping (LLM)
```

```
for parseAttempt in 1..3:
    for apiAttempt in 1..3:
        response <- LLM(prompt)
        if request succeeded: break
        if transient API error (503/429/500) and apiAttempt < 3:
            wait, retry                          // path A: API retry
    if response is valid JSON: break
    // path B: malformed JSON retries the inference call itself
proposals <- ParseJSON(response)    // [{function, widgetType, priority}]

// Stage 2: conflict resolution
for each widgetType:
    sort proposals of widgetType by priority, descending
    keep the top deviceCapacity[widgetType]; drop the remainder

// Stage 3: widget assignment
for each (function, widgetType, priority) in kept, by priority desc:
    if history has a same-type, unclaimed preferred widget for function:
        widget[function] <- that widget          // reuse previous widget
    else if that widget is contested by another proposal:
        the more-frequently-mapped function keeps it; the other retries
        against the fallback pool
    else:
        widget[function] <- next unassigned widget of widgetType

// Stage 4: control lookup
for each function with an assigned widget:
    (channel, cc) <- deviceProfile[widget[function]]  // recorded at setup
    bind MIDI event (channel, cc) -> softwareCtx.control(function)

update history with the final assignments
return the executable mapping
```

C Evaluation Dataset

Table 1: The 16 human-authored reference configurations used for evaluation, grouped by device layout type.

Device	Software	Controls	Control Types	Category
Fixed-layout devices				
ADJ Hexcon	DMX Lighting	14	sliders, buttons	Lighting controller
Akai APC Key 25 MK2	Ableton Live	19	pads, encoders, buttons, keys	MIDI controller
DaVinci Resolve Mini Panel	DaVinci Resolve	30	knobs, trackballs, buttons	Color grading panel
Loupedeck+	Lightroom Classic	29	knobs, dials, buttons, pads	Color grading panel
Oberheim SEM	Analog Synthesizer	28	knobs, switches	Analog synthesizer
Teenage Engineering OP-Z	Sound synthesis	12	knobs, buttons	Sequencer
Pioneer DDJ-FLX4	Rekordbox	30	jog wheels, knobs, sliders, pads, buttons, encoders	DJ controller
Sony Camera Control Unit	Broadcast Camera	24	knobs, switches, buttons	Camera controller
Configurable-layout devices				
Xbox One Controller	Reaper	15	buttons, thumbsticks, triggers, d-pads	Game controller
Monogram CC	After Effects	12	knobs, dials, buttons	Modular editing console
Monogram CC	Premiere Pro	12	knobs, dials, buttons	Modular editing console
Monogram CC	Spotify	7	knobs, dials, buttons	Modular editing console
Elgato Stream Deck	Spotify	21	buttons, knobs	Multi-use
Loupedeck CT	Photoshop	18	knobs, buttons	Modular editing console
TourBox	Clip Studio Paint	14	knobs, dials, buttons	Digital art controller
TourBox	Spotify	20	knobs, dials, buttons	Everyday software