FrameKit: A Tool for Authoring Adaptive User Interfaces Using Keyframes

JASON WU*, Meta Reality Labs Research, Canada
KASHYAP TODI, Meta Reality Labs Research, USA
JOANNES CHAN, Meta Reality Labs Research, Canada
BRAD MYERS, HCI Institute, Carnegie Mellon University, USA
BEN LAFRENIERE, Meta Reality Labs Research, Canada

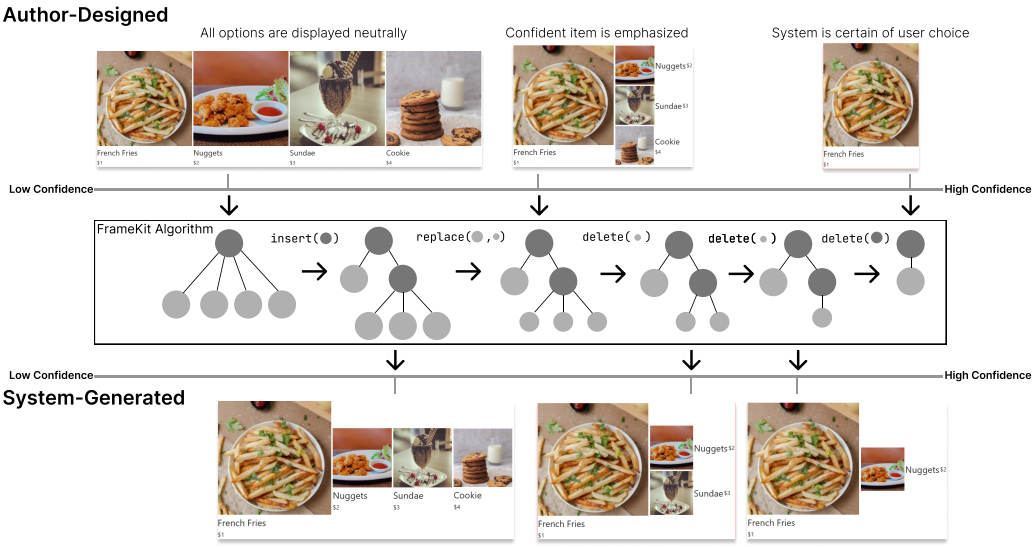

Fig. 1. FrameKit is a tool for authoring adaptive user interfaces based on a 'keyframe' metaphor. In the example above, an author designs an interface for a recommendation system that adapts the size of UI elements based on the system's *confidence* in a recommendation. The author provides three 'keyframe' designs for how the UI should look at different *confidence* values (top). The system applies a computational approach (middle) to generate additional variations by interpolating between the keyframes (bottom).

Adaptive user interfaces (AUIs) can improve user experience by automatically adapting how information and functionality are presented in a user interface. However, the dynamic nature and potentially numerous variations of AUIs make them challenging to author. In this paper, we present a generalized framework for defining adaptation as interpolations between UIs and introduce a computational approach for intelligently generating new variations of a UI from a small set of designs. Based on this approach, we develop *FrameKit*, an authoring tool with a programming-by-example interface that retains flexibility and control afforded by manual authoring while reducing effort through automatic generation. We demonstrate that FrameKit can support adaptations that typically require domain-specific toolkits, such as those found in context-aware applications, responsive UIs, and ability-based adaptation. We

*This author is also affiliated with Carnegie Mellon University. This work was done while the author was a Research Intern at Meta.

Manuscript submitted to ACM

evaluated FrameKit with ten front-end developers, who successfully authored AUIs after a short tutorial session and suggested that FrameKit provides an effective mental model for AUI authoring.

CCS Concepts: • **Human-centered computing → User interface toolkits**.

Additional Key Words and Phrases: Adaptive User Interfaces, Authoring Tools, Computational Design

ACM Reference Format:
Jason Wu, Kashyap Todi, Joannes Chan, Brad Myers, and Ben Lafreniere. 2024. FrameKit: A Tool for Authoring Adaptive User Interfaces Using Keyframes. In *29th International Conference on Intelligent User Interfaces (IUI '24), March 18–21, 2024, Greenville, SC, USA.* ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/3640543.3645176

1 INTRODUCTION

Adaptive user interfaces (AUIs) automatically reconfigure their appearance and behavior based on context to improve user experience. Despite their utility, the dynamic nature and potentially numerous variations of their appearance make AUIs challenging to author. In some cases, it is possible to manually author individual versions of an interface for each use-case (e.g., simplified versions of app launchers [5, 7]), but this strategy quickly becomes costly and time-consuming as the number of combinations of contextual factors grows. Instead, various approaches have been proposed for defining adaptive behavior at a higher level e.g., using objective functions [18, 22], policies [28], or abstract specifications [26, 27]. Compared to manual approaches, these techniques reduce effort but distance designers and developers from the final UI, since the authoring process becomes more abstract. This makes applying established workflows such as gradually refining mock-ups more difficult [13]. Existing tools for authoring AUIs are also domain-specific (e.g., an AR toolkit [22] is not designed for ability-based adaptation and vice versa), and they are often limited by platform-specific assumptions (e.g., CSS media queries for screen size). Collectively, this suggests a need for a more flexible and unified method of authoring AUIs.

To address the above challenges, we present *FrameKit* — a generalized authoring tool for 2-D AUIs that supports a novel automated workflow based on programming-by-example (PBE) and keyframing. FrameKit enables designers to guide an automated AUI generation algorithm by directly creating and manipulating point designs, analogous to "keyframes" used in video editing and animation. In our workflow, designers create a small set of design examples using a WYSIWYG interface, specify an arbitrary set of contextual adaptation parameters (e.g., the user's available attention, distance from user to interface, display dimensions, etc.), and associate their designs with different points in the adaptation space (i.e., keyframing). The system automatically generates new variations of the UI at unseen points in the adaptation space (called *frames*). The user can inspect and directly adjust the generated frames and save them as new keyframes, which are fed back into the system to guide future interface generation. FrameKit's workflow is flexible enough to be applied to a range of applications that were previously authored using domain-specific toolkits ranging from simple responsive layouts to complex context-aware XR UIs.

FrameKit is built on a novel computational approach developed in this work that defines adaption behavior as discrete interpolations between the tree structures of UI keyframes, based on a customized tree-edit distance metric [56]. First, hierarchical definitions are extracted from UI designs that describe their content, widgets, and structure. To generate a new version of the interface based on a set of context parameters, our algorithm finds relevant keyframes that have been previously associated with similar contexts. Finally, our algorithm "blends" together two of these designs using a sequence of discrete edit operations (based on tree-edit distance [56] and heuristics) that produce intermediate versions of the UI.

To demonstrate FrameKit's flexibility, we created a diverse set of example applications that include a responsive UI, ability-based adaptation, and a UI responsive to spatial context. To demonstrate FrameKit's usability, we conducted a user study with 10 front-end developers where we provided visual prompts (screenshots) and asked them to recreate examples of AUIs from past HCI literature. Although the original implementations of the prompts depended on domain-specific toolkits, all participants could successfully author the AUIs using FrameKit after only a short tutorial. Participants' comments suggest that FrameKit could integrate into existing front-end development workflows, and that the tool enabled an effective mental model of how to develop AUIs.

In summary, this paper makes the following contributions:

(1) A *computational approach* for UI adaptation that generates new variations of UIs from a small set of keyframes through interpolation.
(2) FrameKit, a *mixed-initiative tool* for authoring AUIs built on the above computational approach.
(3) A set of *example applications* that demonstrate how a wide range of existing adaptive UI behaviors can be achieved using a key-framed approach, including those used in existing contextually-aware applications, responsive UIs, and systems supporting ability-based adaptation.

2 RELATED WORK

Our work contributes to AUI research by providing a computational tool for authoring AUIs through a PBE workflow. To situate our work, we review related literature from *i)* adaptive user interfaces, *ii)* computational UI design tools, and *iii)* PBE authoring tools.

2.1 Adaptive User Interfaces

AUIs have been developed with the purpose of adapting to display parameters [52, 54, 57], user ability [53, 61, 67, 71], context [20, 43], user preferences [23, 26, 27, 64, 65], and tasks [21, 26].

A range of methods has been employed to define adaptive behavior. One approach is to develop tools and frameworks that contain pre-programmed logic for common adaptive patterns. For example, many commercial tools (e.g., Dreamweaver [9], WebFlow [8]) and frameworks (e.g., Bootstrap [1]) contain ready-made templates for adapting a website to mobile and tablet form-factors. Previous work has developed similar software frameworks to support context-awareness [20, 43], user behavior [28], and mixed-reality [34] UIs by providing developers with pre-built, composable modules for sensing, recognition, and adaptation. Model-based approaches have also been developed, which provide higher-level abstractions of widgets and behavioral patterns common in ubiquitous [59] and multi-device computing applications [47].

Other development approaches have used objective-based optimization to automatically adapt UIs by searching for layouts that optimize predefined metrics [23, 26, 27, 53, 57]. The ARNAULD [27] and SUPPLE [26] systems, enable developers to specify a high-level, formal definition of their UI that is used for dynamic generation. During runtime, these systems use an optimization algorithm that makes rendering decisions based on an objective function that captures user preferences or abilities. Similarly, recent approaches for the 3D placement of UI widgets in mixed reality applications applied optimized layouts based on the semantic properties of UIs [16, 44] and developer-tuned objective functions [22] such as reachability, visibility, and consistency.

FrameKit is designed to address some of the shortcomings of these existing AUI approaches. Compared to software toolkits and frameworks designed around a limited set of transformations, FrameKit provides a unified, domain-agnostic

framework for UI adaptation. Compared to approaches based on objective optimization, FrameKit's workflow allows designers to retain more control over the automated generation process by directly creating and editing UIs instead of manipulating more abstract parameters (e.g., objective weights).

2.2 Computational UI Design Tools

Numerous computational techniques, such as layout optimization and generation, have been integrated into tools that aid in the design process of complex layouts.

The space of possible UI layouts is very large, so some work has proposed surfacing relevant design examples to inspire new designs [29]. Webzeitgeist, for example, collected a large dataset of webpage layouts and mined the common design patterns used with different web pages [37]. RICO is a similar dataset collected from Android apps, and introduced an efficient machine learning model for finding designs with similar layouts [19]. Improvements have been made to improve the efficiency of exemplar search [12], and several tools have been developed that integrate these search capabilities using example galleries [40] and sketches [30].

An alternative to finding existing UI designs is to generate them. Numerous systems have applied optimization methods to generate a set of candidates or surface interesting design alternatives. Early work on layout design generation and suggestion focused on recommending graphic designs (e.g., posters) based on detected content types and a set of pre-defined optimization objectives [55]. Sketchplore was a UI prototyping/sketching tool that integrated a layout optimizer to provide design suggestions and alternative designs, which enabled users to quickly and efficiently explore a range of design alternatives [66]. Similarly, Scout was a system that enabled users to narrow a search space by specifying UI constraints for automated design generation [62]. GRIDS applied similar techniques earlier in the design process and supported creativity by producing a diverse set of starting points [18].

FrameKit builds on work in computational UI design that integrates automated capabilities into authoring tools (e.g., [62, 66]) to enhance designer workflows, but focuses specifically on creating a tool for designing AUIs, which often extend beyond the static UI layouts produced by previous tools. In addition, AUI authoring necessitates more controllable forms of generation (e.g., for a target context) rather than diverse generation (e.g., design inspiration), which led to the design of our interpolation-based approach.

2.3 Authoring UIs via Programming by Example

Authoring AUIs requires one to define the dynamic behaviors that extend beyond static layout design. Programming by Example (PBE) [51] and Programming by Demonstration (PBD) [48] make programming UI behaviors easier by learning the desired behavior from a small set of examples rather than a manually-specified definition [46].

Macros, or scripts that replay previous demonstrations, can be effective for simple interface tasks [49] but lack the capacity to generalize to new scenarios. Pavlov was an early system that applied PBD to author UI animations using a more generalizable *stimulus-response* framework, where temporal or action events (i.e., stimuli) were associated with graphical transformations and states (i.e., responses) via demonstration [68, 69]. The association process often relied on manually defined heuristics (i.e., rule-based inferencing) to extract logic from demonstrations or multiple examples [24, 50, 70]. Other more sophisticated reverse-engineering algorithms have been developed to infer the constraints [10, 35, 38, 39, 45], behaviors [25, 42], or definitions [33, 73] of existing UIs. In some cases, these approaches have been integrated into authoring environments for responsive adaptation [35] and adaptive AR scenes [60]. Recently, Large Language Models (LLMs) have shown promising capabilities in generating complex UI code from UI screenshots or simple sketches of UIs [11], further lowering the barrier to using PBE systems. While inference and generation

techniques have improved, many of the challenges of PBE systems remain. Since it is unlikely that any automated system will achieve perfect performance, PBE-based authoring systems are likely to benefit from user control and feedback [36, 50].

FrameKit contributes to this body of research by applying PBE to AUI authoring. FrameKit infers adaptive behaviors from a small number of manually authored UI examples called *keyframes*, which are associated with user-defined adaptation parameters. Most related to the present research is the Expresso environment, which used keyframes to infer CSS constraints while users were authoring responsive websites [35]. FrameKit, however, offers a more generalized approach that extends to complex adaptive behaviors through support porting modularization of adaptive behaviors, interpolating structurally distinct UIs, and support for multi-dimensional context. Importantly, FrameKit's automatic inference and generation abilities are tightly enclosed within a user-feedback loop, allowing for direct user control during the authoring process.

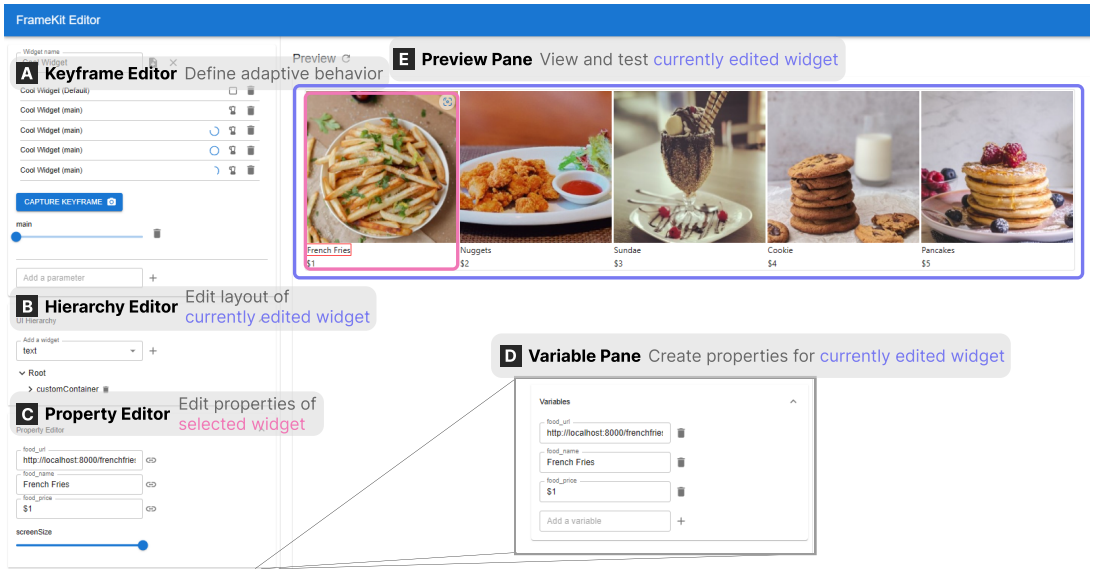

Fig. 2. FrameKit's authoring interface is divided into several regions. The Hierarchy and Property Editors (B & C) are used to create variations of the UI, which are associated with different parameters in the Keyframe Editor (A). The Preview Pane (E) is used to test the AUI. The Variable Pane (D) exposes editable values for use when nesting the current widget within other widgets.

3 FRAMEKIT: AN AUI AUTHORING TOOL

In this section, we introduce the design of FrameKit by first giving an overview of its authoring interface, and then providing a walkthrough scenario that illustrates how a designer could use it to author an AUI.

3.1 Interface Components

FrameKit's authoring interface was designed to mimic no-code or low-code GUI builder tools. The interface supports the creation of widgets that can represent portions or the entirety of a UI. Widgets can be combined to compose complex layouts and adaptive behaviors. The interface contains a Preview Pane (Figure 2E), which enables users to interact

with, and test, the current widget. Like many other GUI builder tools, UIs are represented as hierarchical structures composed of widgets and properties, similar to the DOM of a webpage.

The Hierarchy Editor pane (Figure 2B) allows users to edit the current widget by adding new subwidgets or training the layout structure through drag-and-drop interactions. Some subwidgets have editable properties (e.g., an image's source parameter), which can be viewed and modified in the Property Editor (2C). The Variable Pane (Figure 2D) allows the user to define new editable properties for the current widget that alter its layout or behavior. Finally, the Keyframe Editor (Figure 2A) is used to define the current widget's adaptive behavior. The top of the Keyframe Editor shows a list of keyframes (i.e., UI states associated with contextual states) associated with the current UI. Users can load keyframes, delete them, or toggle features such as UI interpolation. The Keyframe Editor allows users to define adaptation parameters (e.g., screen size), each represented by a slider. Users can create a new keyframe by i) dragging the context sliders to the desired values, ii) editing the current widget state using the WYSIWYG interface, and then iii) clicking "Capture Keyframe." Importantly, FrameKit makes no assumptions about the types of adaptation parameters that can be defined (they are not limited to common use cases such as screen size) making it more versatile for use cases not supported by existing tools that rely on pre-programmed behaviors or objective functions.

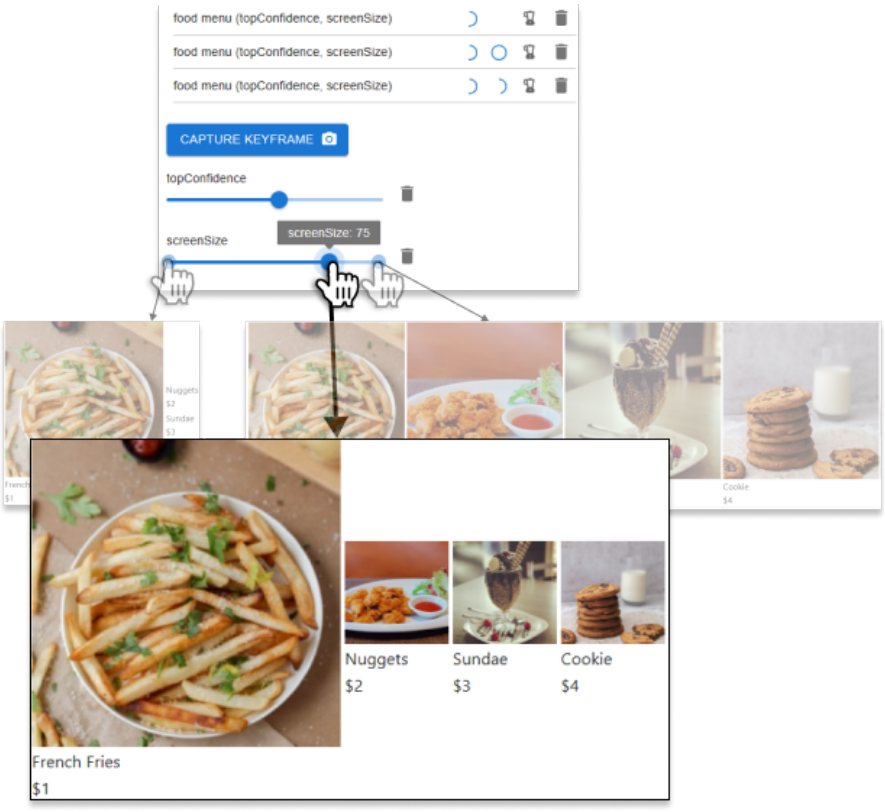

Fig. 3. FrameKit automatically generates variations of the UI based on keyframes associated with adaptation parameters. This figure shows the variations of a widget from the food menu UI that was generated from manually provided keyframes (the endpoints) which depend on two parameters.

3.2 Walkthrough

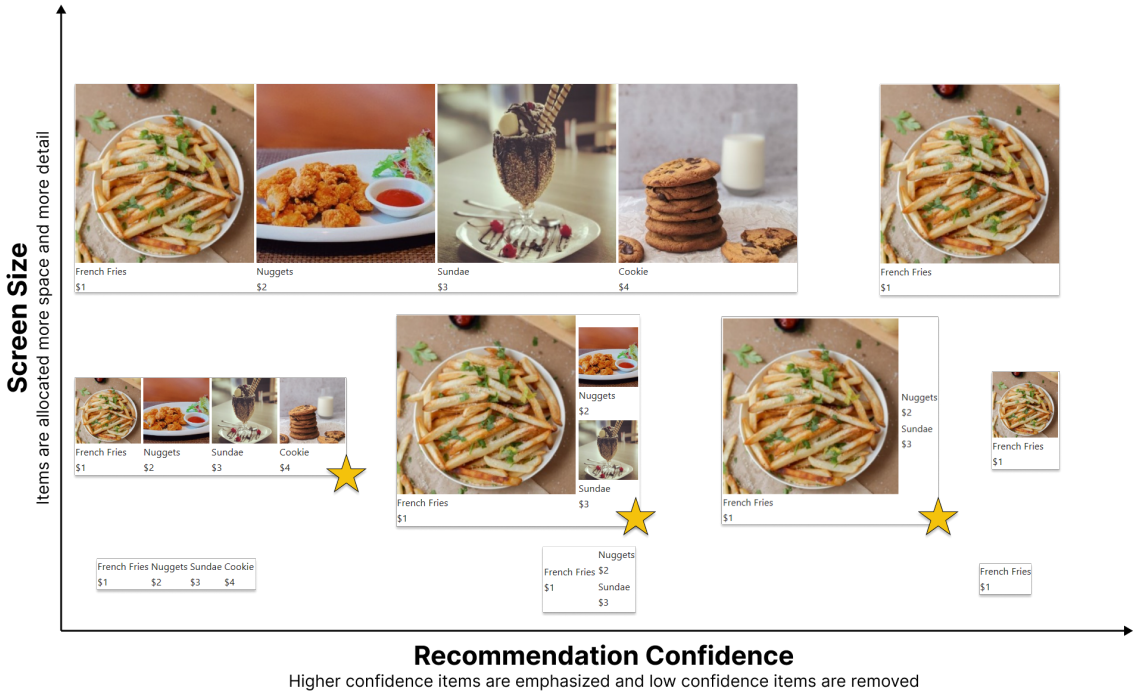

Fig. 4. The adaptivity of the Food Menu app represented in a 2-D space. There are two adaptation parameters, *i)* screen size and *ii)* recommendation confidence, and we show 9 adaptations of the app in this space, although there are more generated variations that are not shown due to space constraints. As screen size is increased, items are allocated more space and detail. As recommendation confidence is increased, higher confidence items are emphasized through relative size while low confidence items are removed. In this example, the UI variations with a star icon were automatically generated as frames, but the user made small edits, turning them into keyframes. A video is provided in the supplemental material to show the originally generated frames and user edits.

In this walkthrough, we illustrate how a designer could use FrameKit's workflow (Figure 5) to author an AUI. Alex is creating a context-adaptive food app (Figure 4) for a client's restaurant that adapts its layout to screen size and the output of an intelligent recommendation system.

3.2.1 Initial Design. Alex first uses FrameKit's WYSIWYG editor to create variations of the UI that correspond to different conditions. They think that the menu, for example, should present different layouts based on the screen size of the device and the confidence of the food recommendation system. To adapt to different screen sizes, the menu UI should display fewer details when screen real-estate is scarce (e.g., omitting an image preview or description text). The menu UI should also adapt to the recommendation system output by allocating more screen space to food items with higher confidence that the user might want them. Alex creates two widgets based on their designs: a food item widget that is used to render individual food items on the menu, and a menu widget that arranges multiple food items that have been imported as subwidgets.

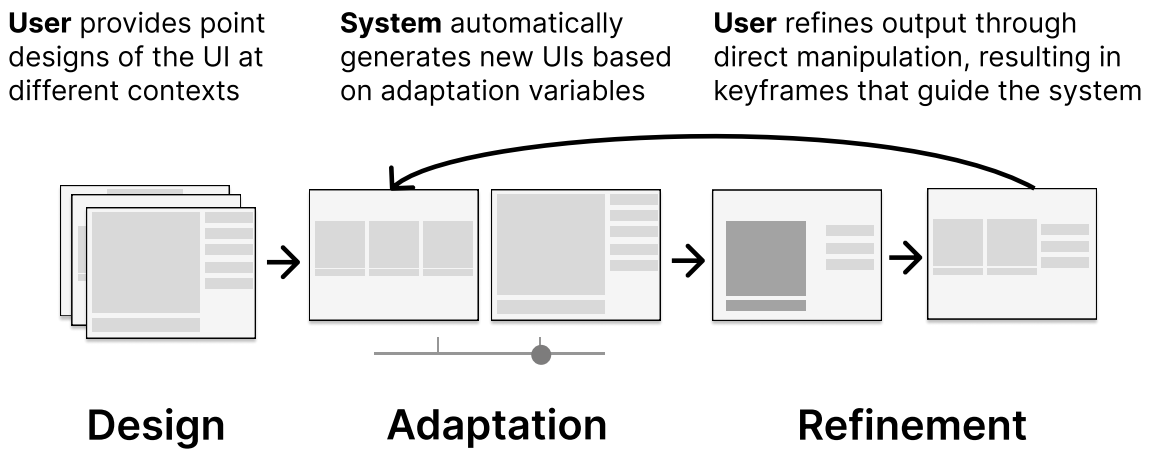

Fig. 5. FrameKit's overall workflow. Design phase: authors specify point designs of the UI associated with adaptation parameters. Adapt phase: tool generates additional variations of a UI for unseen contexts. Refine phase: user refines the generated output or provide additional keyframes to guide algorithm behavior.

3.2.2 Defining Adaptation. In order to add the adaptive behavior, Alex opens the food item widget and registers "screen size" as an adaptation parameter, which creates a slider to represent its value (Figure 3).[1] The default appearance of the food item widget (where all of the information is displayed) is appropriate when the screen size is large. For smaller screens, Alex drags the slider to the minimum value and then simplifies the food item widget by removing the image preview. Alex applies a similar process to the menu widget, starting by registering a new adaptation parameter called "item confidence." The default horizontal layout should be used when the recommendation engine assigns low or equal confidence values to all food items in the menu. Alex also creates an alternate menu layout that emphasizes the item with the highest confidence by displaying it in a separate container and saves a keyframe by dragging the "item confidence" slider to the maximum value.

3.2.3 Iterative Refinement. Alex can preview the UI's adaptive behavior by moving and dragging the sliders corresponding to adaptation parameters and viewing the generated UI in the preview pane (Figure 3). Alex can then refine the adaptive behavior of the app either by directly manipulating the generated output or by authoring an entirely new UI. Both actions result in a new keyframe that is incorporated into the system and helps guide it toward better generations.

4 COMPUTATIONAL APPROACH FOR UI ADAPTATION

FrameKit introduces a computational approach for adaptation that defines adaptive behaviors as interpolations between UIs. The input to our algorithm is a set of UIs (i.e., keyframes) that were previously associated with adaptation parameters, and a target context for which the algorithm generates a new variation of the UI. Our algorithm uses three steps (Figure 6) to generate new UIs from keyframes: *i)* input processing, *ii)* endpoint selection, and *iii)* UI blending (i.e., interpolation).

[1]The parameter name ("screen size") is chosen by the author and can be any arbitrary string. The system's adaptations are not hard-coded based on the selected name, but rather on the keyframes that the author has associated.

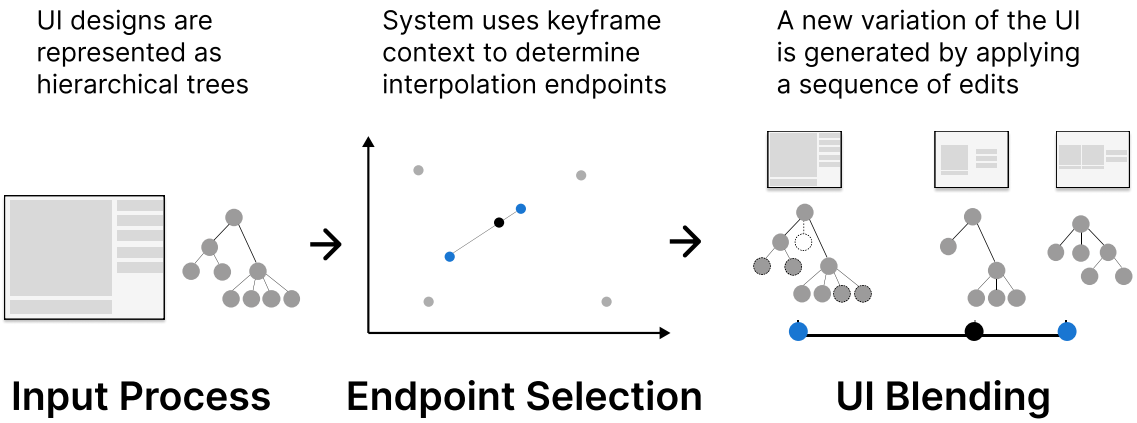

Fig. 6. An overview of FrameKit's computational approach to UI adaptation. Keyframes contain UI designs that are represented as trees, associated with context variables. The system finds relevant keyframes (endpoints) based on a target query. A edit sequence is computed from the endpoints which is used to generate additional variations.

4.1 Algorithm Input

The input to the adaptation algorithm is a set of keyframes, each consisting of a UI "point design" and corresponding values for adaptation parameters. The UI is internally represented as an ordered tree, where nodes correspond to widgets that are present in the layout. Nodes can contain attributes corresponding to parameters (e.g., an image's size and source URL). Some attributes, such as size, can be labeled by the user as "blendable," which allows them to be adjusted during the interpolation process. Context is represented by a multi-dimensional vector, where each dimension corresponds to a different user-defined parameter that affects the UI.

4.2 Endpoint Selection

To generate the UI corresponding to a context, the algorithm finds a set of relevant examples, called endpoints, that can be "blended" together to produce the final output. When there is only one adaptation parameter (one slider), two keyframes with the closest value greater than and less than the target value are chosen. When there are multiple dimensions, finding the smallest enclosing hull for the target context is computationally expensive, so an approximate approach is used that sequentially repeats the interpolation process for each adaptation parameter.

4.3 UI Blending

UI Blending is a feature that allows FrameKit to generate intermediate versions of the UI by interpolating the structure and attributes of the selected endpoints. If this feature is disabled, the system uses a "jump transition" to display an existing design instead of generating a new one. When blending is enabled, the algorithm first identifies widget attributes (e.g., widget size) that can be interpolated for a smooth transition. For our prototype implementation, we use simple linear interpolation, but more sophisticated types of interpolation (e.g., polynomial, spline, tapered) can be used to create softer transitions.

$$y = y_0 + \alpha(y_1 - y_0) \tag{1}$$

Equation 1 is used for the standard linear interpolation, where y_0 and y_1 are the endpoint values for a parameter and y is the computed output. The main idea is first to compute the difference between the two endpoints and then add a

portion of the difference (i.e., the alpha value) onto one of the endpoints to arrive at a "midpoint." To generalize this concept to discrete and structural changes, we use an implementation of the tree-edit distance (TED) algorithm [56] to calculate the "difference" between two trees by computing a sequence of "edit" operations that are needed to transform one tree to another. A portion of the edit sequence can be reapplied to one of the endpoints to generate an intermediate tree.

AUIs authored with FrameKit can be nested within one another, e.g., the user authors widget A, then includes it as a sub-component of widget B. When multiple adaptive widgets are nested, the adaptation parameters of the imported widget can be modified using the Property Editor and linked to the adaptation parameters of the parent widget. The layout of the child widget is first computed using the blending algorithm and inserted into the appropriate place in the parent's hierarchy. Our supplemental video shows an example of this process.

Reordering Heuristics. The sequence generated by the TED algorithm [56] is optimized to achieve the minimum possible number of operations. Since the default edit sequence may contain consecutive "delete" operations, simple truncation can result in missing or misplaced elements. Several heuristics were created to re-order and augment the default sequence to produce better intermediate outputs. Descriptions of these heuristics can be found in the appendix (Appendix B). At a high level, the heuristics *i)* re-order the sequence of insertions and deletions to group together edits that operate on container items, and *ii)* optimize the ordering of replacement edits with respect to other changes.

5 SYSTEM IMPLEMENTATION

FrameKit's authoring interface was implemented as a React.js web application that can be accessed through a browser. UI layouts are represented as JSON structures that are manipulated through interactions with the WYSIWYG editor and saved to local storage. JSONs are formatted as nested structures of widgets and their parameters, similar to the HTML DOM or other declarative UI definitions. The current prototype supports a small number of basic types of widgets (e.g., text, buttons, images), but this set could easily extended by the end user or at a toolkit level by updating FrameKit's JSON to code compiler. The front-end application uses HTTP requests to communicate with a Python server that executes the algorithms to adapt and render UIs (i.e., generating HTML and JavaScript code for the preview widget). Our current implementation uses a Python server for tree-edit distance computation with specialized libraries [56], but future versions could be implemented entirely using client-side JavaScript.

6 EXAMPLE APPLICATIONS

A key goal of FrameKit was to support the creation of AUIs for a wide range of different current and potential future applications. To demonstrate FrameKits's flexibility, this section describes three additional example applications (in addition to the recommendation-based UI described in Section 3.2) based on real-world applications and research prototypes [26, 52]. We describe: *i)* a responsive UI, *ii)* a UI that adapts to the motor abilities of a user, and *iii)* a waypoint UI for an AR application.

6.1 Responsive UI

The Responsive UI example is based on the Mozilla Firefox website, which was used in previous work [35] and includes several features that help it adapt to different viewport dimensions. First, the website includes responsive images whose display sizes are adjusted based on the width of the screen. The margins between some elements also shrink to preserve space. The layout generally transitions smoothly with changes in screen size, however, there are several distinct layouts that are snapped to at specific "breakpoint" widths. At each breakpoint, the header menu switches between several

Fig. 7. The Responsive (Top), Ability-based (Center), and AR Waypoint (Bottom) examples are represented along their adaptation parameters. The Responsive UI adapts to larger screen sizes by expanding the header menu and re-positioning the page content. The Ability-based UI is based on an example use-case of SUPPLE [26], where controls are made easier to tap while maintaining the same window size through container changes. The AR Waypoint is an example of a UI that shows more information at closer distances while also using directional indicators that adapt to the item's location in the user's field of view. The second frame of the Responsive UI, the second and third frames of the Ability-based UI, and the second frame of the AR Waypoint are automatically generated from the user-edited keyframes.

configurations: *i)* a horizontal menu, *ii)* a two row menu, and *iii)* a vertical menu that is revealed by tapping on a collapsed menu button.

To implement the Mozilla website in FrameKit, we observed how the layout changed as it approached the pre-set breakpoints. The main body of the website had two breakpoints, while the header menu had three, so these were implemented as separate subwidgets. The website body used four keyframes, where the first and last two were blended smoothly, and there was a jump transition (i.e., an instant change) between the second and third keyframes. The header bar was implemented using three keyframes with structural blending between them. Both the body and header widgets were imported into a main widget (i.e., nesting) to better coordinate their transition behavior.

6.2 Ability-Based UI

Adaptive UIs have been employed to accommodate users with motor or cognitive disabilities. This ability-based UI example is based on screenshots of SUPPLE [26] that adapted a lighting control panel for a user with motor impairments. The original interface contained elements such as sliders that required precise manipulation, so SUPPLE generated an alternative version where large, tappable buttons presented the same functionality. Because the buttons took up more space, the UI controls were re-organized in a tabbed container to maintain the same screen footprint.

The control panel contains several repeated elements (e.g., sliders) to control different lights. We identified parts of the UI that could be modularized into re-usable widgets, and we implemented widgets for brightness controls, lighting groups, and the main control panel. The brightness control widget had three states that traded input resolution for target size: *i)* a full slider, *ii)* 10 horizontally aligned buttons that supported the selection of brightness at increments of 10%, and *iii)* 5 horizontally aligned buttons that supported the selection of brightness at increments of 20%. The lighting group widget, used to organize the brightness controls, had two states: *i)* a single page with multiple panes for each lighting group, and *ii)* a tabbed container with a tab for each lighting group. The subwidgets were imported into the main widget that coordinated their transition behavior so that the lighting groups were tabbed containers when the brightness controls contained horizontal buttons.

Our re-implementation differs from the reference SUPPLE system in functionality. While SUPPLE automatically generates UI variations from the same high-level abstract specification, FrameKit needs prior knowledge of specific user groups and manual creation of keyframes for their preferences, like larger buttons for motor-impaired users. While we acknowledge that FrameKit sacrifices generative ability for manual control, we could imagine the approach being effective as a way to refine the output of other tools like SUPPLE at development time or as a way of exploring designs during prototyping.

6.3 AR Waypoint Marker

Finally, we present an AR waypoint application based on location markers in AR and 3D virtual environments. These markers are anchored to objects of interest in the environment, and provide more detailed information as the user gets closer to the object. Additional information that requires space to render is often occluded until the user is in close proximity, to avoid information overload (if there are multiple markers) and to mimic the effect of real-world visibility constraints.

We used FrameKit to author a waypoint UI for an AR shopping list app that displays the location of "Bell Peppers" in a grocery store. The waypoint UI responds to *i)* the distance between the user, and the item and *ii)* location of the item in the user's field of view. Initially, an icon is displayed to represent the item, and as it becomes closer, the UI increases in size and displays information about its name and distance. As the angle between the user's gaze and the item increases, the size of the directional indicator also increases. We created two keyframes to represent the waypoint: *i)* when the item is far and in center view, and *ii)* when it is near but in the periphery. FrameKit applied numerical interpolation to gradually adjust the size of the image and text elements and structural interpolation to simplify the UI with distance. Note that the example UI states shown in Figure 7 are rendered using simulated sensor values for viewing angle and distance, i.e., it does not include a real-time tracking implementation.

7 EVALUATION

We conducted a user study with 10 participants to *i)* determine whether participants could successfully use FrameKit to author AUIs, *ii)* measure perceived usability and usefulness, and *iii)* collect subjective feedback. Our goal was to evaluate FrameKit as a unified approach for AUI authoring, not to show that it could exceed the individual performance of existing domain-specific approaches for their target domains.

7.1 Participants

Ten participants with experience in front-end development were invited to participate in the study (6 male, 3 female, 1 non-binary; aged 21–41 years, mean age 28). Participants were recruited via electronic postings and word-of-mouth at

a technology company and a university. Participants had varying front-end experience: from basic HTML to 4 years as a lead UI designer and different toolkit preferences (e.g., HTML, React, Unity3D). Seven participants stated that they had some experience implementing responsive user interfaces (e.g., personal web pages). Three participants had implemented features that adapted a UI to something other than screen size (e.g., based on UI content, screen brightness, the detection of a user, or for accessibility purposes).

7.2 Procedure

A study design similar to those used by previous works [22, 35] was adopted to evaluate FrameKit. The study lasted 2 hours and consisted of three phases: *i)* a tutorial, *ii)* a usage session, and *iii)* a questionnaire. Participants were first provided with an overview of the study and completed a consent form (15 minutes).

During the tutorial phase (30 minutes), participants watched a series of pre-recorded videos that introduced them to FrameKit's authoring interface and functionality and walked them through the process of authoring the Food Menu app. Participants were asked to replicate the actions they saw in the tutorial videos and ask the researcher for assistance if needed.

During the usage session (60 minutes), participants were asked to recreate the Responsive UI and Ability-based UIs (Sections 6.1 and 6.2) based on screenshots of the original interfaces. In the final phase of the study (15 minutes), participants completed a questionnaire that collected information about their experience with FrameKit.

The questionnaire consisted of questions about *i)* demographics, *ii)* UI and AUI authoring experience, and *iii)* questions from the Technology Acceptance Model (TAM) [17], with additional free-form fields to gather explanations.

Due to limited time available during the study, simplified versions of the example applications (section 6) were used as prompts. Unlike prior work [35], prompts were constructed from screenshots instead of video because the target applications lacked "ground truth" transitions between UI states.

7.3 Results

In this section, we present the results of our study, based on *i)* usage results, *ii)* TAM questionnaire ratings, and *iii)* subjective feedback.

7.3.1 Usage Results. We used application logs from the study to determine re-creation success and measure usage of features. We considered a recreation to be successful if it was possible to reach all the states shown in the original prompt screenshots via some configuration of slider positions (i.e., adaptation parameters). Based on this criterion, every participant successfully implemented all the prompts, and many finished early. One participant (P2) experienced a bug that affected the UI generation capabilities of the algorithm. While they completed the prompt recreation task using "jump cuts" (i.e., disabled blending), it may have impacted their ratings. The bug was fixed immediately afterward and was not experienced by any other participants.

We measured number of keyframes participants created during the usage session (mean = 8.9 keyframes, SD = 3.0). Implementations of the Ability-Based UI had a larger range of keyframes than the Responsive UI (Ability: mean = 4.9 keyframes, SD = 2.2; Responsive: mean = 4 keyframes, SD = 1.25), possibly due to the structural differences (i.e., changes that required using different widgets rather than scaling) between the prompt screenshots.

Nine participants constructed complex widgets (that contained at least one user-authored subwidget), and four used the blending feature during their prompt re-creation task (others used jump transitions between manually created frames).

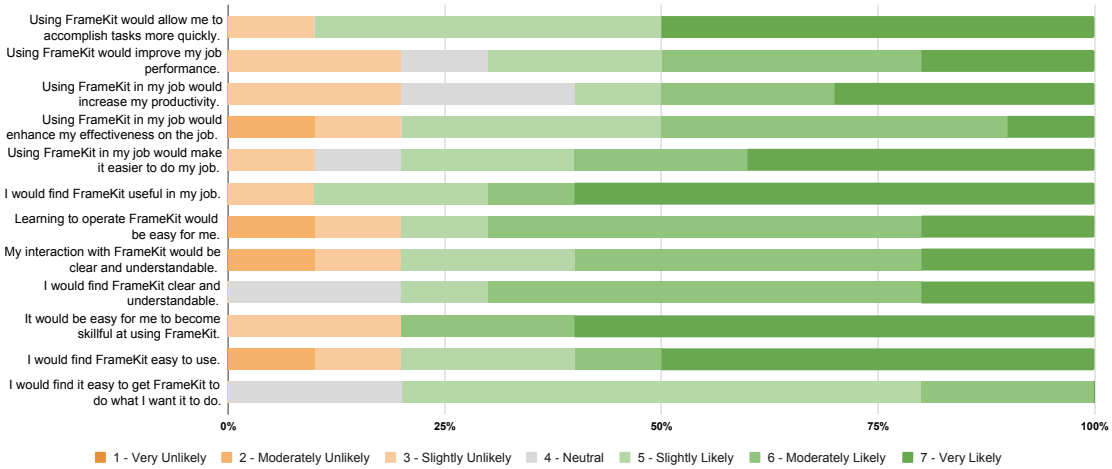

Fig. 8. Responses for each TAM question. Participants found FrameKit useful for creating adaptive UIs, both in terms of perceived usefulness and ease of use, with most participants agreeing that FrameKit would improve their effectiveness at authoring AUIs. Across all questions, participant responses suggested that FrameKit could be adopted into their workflows (mean = 5.51, SD = 1.44, median = 6).

7.3.2 TAM Questionnaire. The responses to the TAM questionnaire showed that participants would likely adopt FrameKit (mean = 5.51 out of 7, SD = 1.44, median = 6) across all of the questions. We further analyzed the TAM questions corresponding to *i)* perceived usefulness, and *ii)* perceived ease of use score.

Perceived Usefulness. Most participants perceived FrameKit as useful for authoring AUIs (mean = 5.51, SD = 1.47, median = 6). Participants attributed this to the WYSIWYG editing interface for layout creation and the keyframing technique for defining adaptive behavior. Participants who had less experience with front-end implementation appreciated the ability to easily create UIs without code, with some commenting that *"it would be impossible otherwise"* (P3). Others (P4, P7, P10) also found the keyframing technique to be useful and noted that the blend function (i.e., interpolation) *"would reduce a lot of work"* (P7).

Other participants (P2, P6) felt that a library or code-based framework would have been more useful for the responsive UI task than FrameKit. However, these domain-specific resources are often not available for more customized applications (e.g., ability-based adaptation).

Perceived Ease of Use. Participants also generally felt that FrameKit was easy to use (mean = 5.50, SD = 1.41, median = 6). Participants (P1, P3, P5) felt that the main WYSIWYG editing interface was intuitive and similar to existing tools, which made it easier to pick up. For the adaptive functionality, many participants indicated that there was an initial learning curve (P10) to understand how to use adapt UIs but made sense of them by relating concepts to those found in video editing software (e.g., Adobe Premiere) (P4) and animation software (P10).

Because FrameKit is a research prototype, it doesn't currently contain features such as auto-save, undo, or multi-selection, and some participants (P2, P5, P8) felt that this made the tool harder to use. However, these features would be straightforward to implement if FrameKit were to be integrated into a more polished tool.

7.3.3 Subjective Feedback. To gain additional insight into the user experience of FrameKit and further context on participants' ratings, we analyzed subjective feedback provided during the study and in the free-response fields of the post-study questionnaire.

Understanding AUI Authoring. Perhaps because complex AUIs (i.e., those that extend beyond responsive designs) are not a common feature in current applications, we found that many (7/10) participants did not have prior experience using or authoring them. We were thus interested in the learnability of FrameKit and asked participants to give us feedback on their process of understanding our tool's workflow and AUI authoring more generally. After the tutorial and some usage, many participants (8/10) agreed that learning to operate FrameKit would be easy for them, e.g., *"There's a bit of a learning curve, but given that adaptive modular interfaces are complex to conceive and manipulate, it is normal. I would say most of the complexity comes from the task, not from operating or working with the interface."* (P10). This was aided by the fact that FrameKit employs relatable concepts used in other types of creation software. P4, who was familiar with video editing tools, drew parallels to video editing tools *"[FrameKit's] close to Premiere and would be easy for me to use"*.

Other participants (P6, P9) noted that there was a "change in [my] process" (P6) that was needed to use FrameKit, *"I had to start thinking in [an adaptive] way, which I don't usually do"* (P6), and P9 noted that when using existing templates for responsive UIs, they usually worry about content before supporting screen responsiveness. Although P10 suggested the shift was manageable: "I think after a few complex examples, it would be pretty straightforward to build the correct mental model on how widgets should best be constructed to enable adaptive behaviors."

Feature Adoption. FrameKit contains features for *i)* modularizing adaptations (i.e., nesting adaptive widgets) and *ii)* blending UIs, and participants' feedback provided insights into their choice to use or not use these two capabilities.

Most participants (9/10) modularized their adaptive widgets during the usage session and expressed that it was a reliable way of reducing effort by *"using [the] same [widgets] more than one time"* (P8). P7 commented that compared FrameKit to a website builder (Framer [2]) with which they had experience, the widget abstraction feature implemented by FrameKit was useful for organizing pieces of adaptive behavior, *"Framer Motions has a function similar to this application's Blend function. Though I love the part that the parent container can influence their children's parameters."*

Fewer participants (4/10) used the blending feature. While blending can reduce effort by generating additional UI variations, some participants (P2, P6) expressed that they preferred manually creating the desired keyframe rather than refining an automatically generated one. Some participants expressed that they needed a deeper understanding of how the algorithm worked before they would feel comfortable using the blending feature: *"I feel like I need to understand how the blending works and, therefore, the algorithm behind blending, which is completely abstracted from me"* (P6). In addition, P2 expressed support for the concept of blending but some concern about being unable to undo if they made a mistake: *"The concept is very intuitive, but during usage, it's risky to do operations since there's no easy undo. If I forget to save the keyframe I need to restart again"* (P2). Overall, the above feedback suggests that more can be done to communicate how blending works, and to enable users to experiment with the feature without fear of losing progress. We believe that adding common editor features such as undo and auto-save would help enable such experimentation and address these concerns.

UI Authoring Workflow. A goal of the FrameKit system is to align more closely with the established workflows of UI designers and front-end developers, which start from mock-ups and point designs. We asked our study participants, who had a mix of experience in both design and front-end development, about their UI authoring workflows and how FrameKit might fit into them.

Several participants suggested that FrameKit could be a useful addition during the design phase, where UI designs are rapidly prototyped. Some (P1, P4, P5) found the WYSIWYG component of FrameKit to be useful for quickly building UIs that could be later augmented with adaptive functionality. P7, who worked as a lead UI designer for 4 years, commented, *"I can envision starting with Figma and using FrameKit to fill in the blank between the mobile and desktop design. This will save me a lot of work."* P10 suggested that FrameKit's role could also extend beyond design/prototyping and be a more effective implementation solution than code *"the keyframing approach ... previews states using a 'timeline' metaphor which is very useful for people who are visual (like I am). I think I would quickly realize great results with FrameKit, which would probably take me forever to realize with code"* (P10).

Despite the convenience of FrameKit's graphical interface, some participants (P1, P2 P6, P9) noted that code might still be needed or preferred in some scenarios. P1 and P2 envisioned scenarios in which writing code would give them more flexibility, e.g., *"If what you want to accomplish is supported with FrameKit, absolutely. But I think there are [other tasks like animation] FrameKit might not fully support"* (P1). One potential solution would be to implement FrameKit as a hybrid tool that could be used graphically but also generate code for additional customization. P9, an experienced full-stack web developer, made a similar suggestion, *"Code is much more flexible but could take longer to implement ... a hybrid code and FrameKit tool would be helpful."* Overall, we can imagine extending FrameKit to support additional features to complement existing tools and workflows.

8 DISCUSSION

Based on the design, implementation, and evaluation of FrameKit, we discuss ways that FrameKit could contribute to AUI authoring (i.e., supporting more complex AUIs and a useful mental model) and areas for further research.

8.1 Flexibility to Support Complex AUI Applications

Most existing AUIs, such as responsive web pages, apply minor transformations (primarily layout scaling or "jump transitions" at breakpoints) between variations, and existing authoring tools focus on supporting these basic scenarios while providing limited room for experimentation outside of this scope. Trends in AR and ubiquitous computing suggest that deeper, context-driven interaction will become more common. Based on this expectation, FrameKit supports more complex AUIs by providing flexibility in *i)* defining UI states, *ii)* representing contextual factors, and *iii)* authoring dynamic adaptation behaviors that depend on both.

First, FrameKit adopts a flexible hierarchical definition of UI layout similar to widely used approaches for web-based (DOM) and declarative interfaces. FrameKit's UI representation and additional features for importing and modularizing complex layouts made it amenable to WYSIWYG editing (unlike more abstract UI definitions [26, 58, 59]), which participants in our user study found helpful for building and refining keyframes. In addition, FrameKit also allows flexibility in defining adaptation context. Many existing tools [3, 8, 9] focus on a few pre-defined factors (e.g., screen width) and behaviors (e.g., motion libraries [2]), while leaving more customized use cases to manual scripting. In contrast, FrameKit allows authors to use arbitrary names or an arbitrary number of parameters to describe any contextual factors. Our Adaptive Food Menu example, an AUI that relies on multiple sources of context to determine optimal presentation, shows how complex behavior can be easily conceptualized in an author-defined adaptation space (Figure 1). Finally, FrameKit's adaptation process is based on interpolations of UIs, which has benefits in controllability and generation ability. Any variation of the UI is dependent on a small number of "endpoints" (i.e., a subset of all provided keyframes), unlike the global effects of adjusting weights on an objective function [44]. FrameKit also supports interpolating between dramatically different inputs (unlike previous approaches that only interpolated numeric attributes [35]), which can

reduce the number of manually authored keyframes that a user would have to provide. Our example applications (Figure 7) show that outputs generated by the system can be directly used or refined as new variations of the UI.

8.2 A Mental Model for Authoring AUIs

As with any task, the authoring of AUIs can benefit from an effective mental model that captures their design and implementation. Compared to "traditional" UI authoring, where authors map how data or interactions can be displayed in a graphical layout, AUIs introduce new challenges associated with dynamic adaptation. FrameKit's approach constitutes a *context-based* mental model of AUIs, where variations of the interface are mapped to specific instances of context. However, this is only one of many mental models that could also be applied to AUI authoring. For example, many UIs are currently implemented using a *rule-based* model, where *changes* in the UI are mapped to events e.g., *when* the window becomes smaller than a certain breakpoint, *then* switch to a new layout. This is unlike FrameKit's model, where the threshold for change is implicitly computed instead of explicitly set by the user. A *rule-based* model is advantageous in that it allows fine-grained control of exactly when changes occur; however, manually specifying event-based transitions may become infeasible for larger numbers of variations, as there n^2 transitions between n states. As previously mentioned, other established alternative models exist for AUI authoring as well, such as *task-based* [26, 58, 59] or *objective-based* [18, 22, 66] models that abstract the UI adaption and generation process. However, these models may be more difficult to integrate into existing UI authoring workflows.

Overall, our user evaluation provides initial support for a context-based model in that participants could successfully use the tool to re-create prior examples in the literature and connect its concepts to other familiar software (P4, P10). However like other mental models of adaptation, FrameKit makes trade-offs between authoring effort and control over the end result. Ultimately, by contributing to the body of approaches for AUI authoring, we hope to lay the groundwork for a future where designers and front-end developers have a greater set of tools at their disposal for developing AUIs, and are thus able to select the tool with the right trade-off for a given application.

8.3 Limitations and Future Work

Overall, FrameKit presents a promising alternative to existing approaches for creating AUIs, such as software frameworks [20, 28, 34, 43] and objective-based optimization [23, 26, 27, 53, 57]. The user evaluation showed that front-end developers were successful while using FrameKit to recreate simplified AUIs after a short tutorial. With expert use, it is possible to author more advanced AUIs that have complex behaviors (e.g., our three example applications). In this section, we discuss the limitations of our work and opportunities for future research.

UI Representation. FrameKit currently requires users to author keyframes by manipulating a tree-based representation of their UI, similar to other graphical GUI builders [4, 6]. While this is amenable to our layout blending algorithm (i.e., tree-edit distance), it offers less flexibility than free-form design tools (e.g., Figma [3]). This representation also introduces some ambiguities while authoring because the same visual result could be achieved using different hierarchical structures. Future PBE tools for authoring AUIs thus need more sophisticated reverse-engineering methods [10, 14, 73, 74] to infer structured representations from free-form designs.

Generation Algorithm. FrameKit currently uses a tree-edit algorithm to generate new UIs by "blending" keyframes. While this feature generalizes keyframe interpolation [25, 35] to work with structurally distinct inputs, it has several limitations.

First, the algorithm's performance depends on a UI's tree representation rather than its visual appearance. For example, inserting vertical containers into a horizontal stack and inserting horizontal containers into a vertical stack

would both achieve similar visual layouts (i.e., an evenly-spaced grid) but could lead to different intermediate states due to the different sequences of edits needed to reach other keyframe endpoints. Moreover, edit sequences have inherent limitations to the types of transitions they can generate (e.g., two edit actions cannot be applied in one step), necessitating additional manual refinement. Edit sequences also cannot easily be used for *extrapolation*, which could be useful for creative exploration during the authoring process. Finally, our algorithm's support of multi-dimensional contexts can be made more robust, as it currently relies on an approximate algorithm affected by the number and order of adaptation parameters. We successfully used FrameKit to author AUIs with up to three adaptation parameters but observed instability with further complexity. Recently, machine learning techniques have shown promise in training data-driven models to map UI layouts to a continuous embedding space [15, 31, 32, 41] which can be "decoded" back into plausible UIs.

Similarly, LLMs have also shown promising capabilities in understanding and generating UIs and layouts as code representations [63], which can be prompted to generate additional UI variations. Importantly, we view FrameKit's generalizable keyframing workflow of AUI authoring as agnostic to the underlying generation algorithm used and provides an effective framework for feedback and detailed design with AI-based tools. We anticipate that FrameKit could be extended to take advantage of newer approaches. For example, our current tree-edit algorithm could be augmented or replaced by machine-learning approaches, leading to improved UI quality and additional feature support (e.g., extrapolation).

User Evaluation. Our user evaluation was a preliminary evaluation of FrameKit in a laboratory setting. There is thus room for a deeper evaluation of FrameKit. For example, the layout blending algorithm may not have generated the ideal intermediate layout on its first iteration, so it would be valuable to understand how many manual edits are needed for the refinement of a great number of interfaces. The current evaluation also focused on eliciting usability feedback from participants, but we did not compare FrameKit to other baselines for AUI authoring, which are mainly research prototypes designed for specific application domains (e.g., AR [22, 26, 35]). Given that complex AUIs that extend beyond responsive designs have yet to be widely adopted, a fair comparison would require an introduction and training about a variety of tools and software libraries to yield insights about the trade-offs of each approach. Our study also focused on the recreation of simplified examples that were based on examples found in the research literature, and a longer study would allow participants to explore more features of our tool. Notably, due to time constraints, our current study focused AUIs with single-dimensional context, although our context-aware menu and AR waypoint example applications demonstrate that multi-dimensional context is possible. Furthermore, a longer-term study would enable participants to use FrameKit for creative and open-ended tasks, where participants could explore usage over time or apply our tool to their own projects. We are particularly interested in how tools like FrameKit could become integrated into the day-to-day workflows of designers and developers. An important part of this is finding an effective authoring interface that best surfaces the generative functionalities of our algorithm. Our current design (Figure 2) may be further tweaked through our currently collected feedback (e.g., to mimic the design of animation software) or subsequent interface evaluations.

9 CONCLUSION

This research introduced FrameKit, a mixed-initiative PBE tool built on a computational framework for authoring AUIs using keyframes. FrameKit introduces a novel workflow for authoring AUIs that *i)* retains a higher degree of user control over the adaptation process, and *ii)* supports many types of adaptation without any domain-specific tools or assumptions. To demonstrate FrameKit's flexibility in supporting a wide range of existing adaptive UI behaviors, we

used it to implement a responsive UI, an ability-based adaptation, and multiple context-adaptive UIs using keyframes. Finally, an evaluation study with 10 participants found that participants were able to develop AUIs after minimal training and suggested that FrameKit's authoring process serves as a useful conceptual model of UI adaptation.

ACKNOWLEDGMENTS

We thank Michelle Annett for proofreading and Krista Taylor for assisting with the user study. We thank our study's participants for their voluntary participation and time. Finally, we thank numerous members of the Meta Reality Labs Research team for insightful discussions and constructive feedback.

REFERENCES

[1] 2023. Bootstrap. https://getbootstrap.com/ Accessed 2023-08-19.
[2] 2023. Documentation | Framer for Developers. https://www.framer.com/motion/ Accessed 2023-09-01.
[3] 2023. Figma. https://www.figma.com/ Accessed 2023-08-19.
[4] 2023. Glade - A User Interface Designer. https://glade.gnome.org/ Accessed 2023-08-19.
[5] 2023. Samsung Easy Mode. https://www.samsung.com/au/support/mobile-devices/using-easy-mode/ Accessed 2023-08-19.
[6] 2023. Stetic GUI Designer. https://www.monodevelop.com/documentation/stetic-gui-designer/ Accessed 2023-08-19.
[7] 2023. Using the Simple Finder to help students stay on task. https://etc.usf.edu/techease/4all/learning/how-do-i-use-the-simple-finder-to-help-students-stay-on-task/ Accessed 2023-08-19.
[8] 2023. WebFlow. https://webflow.com/ Accessed 2023-08-19.
[9] Adobe. 2023. Adobe DreamWeaver. https://www.adobe.com/products/dreamweaver.html Accessed 2023-08-19.
[10] Pavol Bielik, Marc Fischer, and Martin Vechev. 2018. Robust relational layout synthesis from examples for Android. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.
[11] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712* (2023).
[12] Sara Bunian, Kai Li, Chaima Jemmali, Casper Harteveld, Yun Fu, and Magy Seif Seif El-Nasr. 2021. Vins: Visual search for mobile user interface design. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–14.
[13] Bill Buxton. 2010. *Sketching user experiences: getting the design right and the right design.* Morgan kaufmann.
[14] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*. 665–676.
[15] Chin-Yi Cheng, Forrest Huang, Gang Li, and Yang Li. 2023. PLay: Parametrically Conditioned Layout Generation using Latent Diffusion. *arXiv preprint arXiv:2301.11529* (2023).
[16] Yifei Cheng, Yukang Yan, Xin Yi, Yuanchun Shi, and David Lindlbauer. 2021. Semanticadapt: Optimization-based adaptation of mixed reality layouts leveraging virtual-physical semantic connections. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 282–297.
[17] Fred D Davis. 1989. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly* (1989), 319–340.
[18] Niraj Ramesh Dayama, Kashyap Todi, Taru Saarelainen, and Antti Oulasvirta. 2020. Grids: Interactive layout design with integer programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.
[19] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th annual ACM symposium on user interface software and technology*. 845–854.
[20] Anind K Dey, Gregory D Abowd, and Daniel Salber. 2001. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human–Computer Interaction* 16, 2-4 (2001), 97–166.
[21] Peitong Duan, Casimir Wierzynski, and Lama Nachman. 2020. Optimizing user interface layouts via gradient descent. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–12.
[22] João Marcelo Evangelista Belo, Mathias N Lystbæk, Anna Maria Feit, Ken Pfeuffer, Peter Kán, Antti Oulasvirta, and Kaj Grønbæk. 2022. AUIT–the Adaptive User Interfaces Toolkit for Designing XR Applications. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. 1–16.
[23] Anna Maria Feit, Lukas Vordemann, Seonwook Park, Caterina Berube, and Otmar Hilliges. 2020. Detecting relevance during decision-making from eye movements for ui adaptation. In *ACM Symposium on Eye Tracking Research and Applications*. 1–11.
[24] Gene L Fisher, Dale E Busse, and David A Wolber. 1992. Adding rule-based reasoning to a demonstrational interface builder. In *Proceedings of the 5th annual ACM symposium on User interface software and technology*. 89–97.
[25] Martin R Frank, Piyawadee" Noi" Sukaviriya, and James D Foley. 1995. Inference bear: designing interactive interfaces through before and after snapshots. In *Proceedings of the 1st conference on Designing interactive systems: processes, practices, methods, & techniques*. 167–175.

[26] Krzysztof Gajos and Daniel S Weld. 2004. SUPPLE: automatically generating user interfaces. In *Proceedings of the 9th international conference on Intelligent user interfaces*. 93–100.

[27] Krzysztof Gajos and Daniel S Weld. 2005. Preference elicitation for interface optimization. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*. 173–182.

[28] Camille Gobert, Kashyap Todi, Gilles Bailly, and Antti Oulasvirta. 2019. SAM: a modular framework for self-adapting web menus. In *Proceedings of the 24th International Conference on Intelligent User Interfaces*. 481–484.

[29] Scarlett R Herring, Chia-Chen Chang, Jesse Krantzler, and Brian P Bailey. 2009. Getting inspired! Understanding how and why examples are used in creative design practice. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 87–96.

[30] Forrest Huang, John F Canny, and Jeffrey Nichols. 2019. Swire: Sketch-based user interface retrieval. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–10.

[31] Forrest Huang, Gang Li, Xin Zhou, John F Canny, and Yang Li. 2021. Creating User Interface Mock-ups from High-Level Text Descriptions with Deep-Learning Models. *arXiv preprint arXiv:2110.07775* (2021).

[32] Naoto Inoue, Kotaro Kikuchi, Edgar Simo-Serra, Mayu Otani, and Kota Yamaguchi. 2023. LayoutDM: Discrete Diffusion Model for Controllable Layout Generation. *arXiv preprint arXiv:2303.08137* (2023).

[33] Yue Jiang, Wolfgang Stuerzlinger, and Christof Lutteroth. 2021. ReverseORC: Reverse Engineering of Resizable User Interface Layouts with OR-Constraints. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–18.

[34] Sarah Krings, Enes Yigitbas, Ivan Jovanovikj, Stefan Sauer, and Gregor Engels. 2020. Development framework for context-aware augmented reality applications. In *Companion Proceedings of the 12th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. 1–6.

[35] Rebecca Krosnick, Sang Won Lee, Walter S Laseck, and Steve Onev. 2018. Expresso: Building responsive interfaces with keyframes. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 39–47.

[36] Todd Kulesza, Weng-Keen Wong, Simone Stumpf, Stephen Perona, Rachel White, Margaret M Burnett, Ian Oberst, and Amy J Ko. 2009. Fixing the program my computer learned: Barriers for end users, challenges for the machine. In *Proceedings of the 14th international conference on Intelligent user interfaces*. 187–196.

[37] Ranjitha Kumar, Arvind Satyanarayan, Cesar Torres, Maxine Lim, Salman Ahmad, Scott R Klemmer, and Jerry O Talton. 2013. Webzeitgeist: design mining the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 3083–3092.

[38] David Kurlander and Steven Feiner. 1993. Inferring constraints from multiple snapshots. *ACM Transactions on Graphics (TOG)* 12, 4 (1993), 277–304.

[39] Tessa A Lau and Daniel S Weld. 1998. Programming by demonstration: An inductive learning formulation. In *Proceedings of the 4th international conference on Intelligent user interfaces*. 145–152.

[40] Brian Lee, Savil Srivastava, Ranjitha Kumar, Ronen Brafman, and Scott R Klemmer. 2010. Designing with interactive example galleries. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 2257–2266.

[41] Jianan Li, Jimei Yang, Aaron Hertzmann, Jianming Zhang, and Tingfa Xu. 2019. Layoutgan: Generating graphic layouts with wireframe discriminators. *arXiv preprint arXiv:1901.06767* (2019).

[42] Yang Li and James A Landay. 2005. Informal prototyping of continuous graphical interactions by demonstration. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*. 221–230.

[43] Brian Y Lim and Anind K Dey. 2010. Toolkit to support intelligibility in context-aware applications. In *Proceedings of the 12th ACM international conference on Ubiquitous computing*. 13–22.

[44] David Lindlbauer, Anna Maria Feit, and Otmar Hilliges. 2019. Context-aware online adaptation of mixed reality interfaces. In *Proceedings of the 32nd annual ACM symposium on user interface software and technology*. 147–160.

[45] Dylan Lukes, John Sarracino, Cora Coleman, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. Synthesis of web layouts from examples. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 651–663.

[46] Richard G McDaniel and Brad A Myers. 1998. Building applications using only demonstration. In *Proceedings of the 3rd international conference on Intelligent user interfaces*. 109–116.

[47] Giulio Mori, Fabio Paterno, and Carmen Santoro. 2004. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Transactions on Software Engineering* 30, 8 (2004), 507–520.

[48] Brad A Myers. 1988. *Creating user interfaces by demonstration.* Academic Press Professional, Inc.

[49] Brad A Myers. 1998. Scripting graphical applications by demonstration. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 534–541.

[50] Brad A Myers and Richard McDaniel. 2001. Demonstrational interfaces: sometimes you need a little intelligence, sometimes you need a lot. In *Your Wish is My Command*. Elsevier, 45–III.

[51] Brad A Myers, Richard McDaniel, and David Wolber. 2000. Programming by example: intelligence in demonstrational interfaces. *Commun. ACM* 43, 3 (2000), 82–89.

[52] Michael Nebeling and Moira C Norrie. 2013. Responsive design and development: methods, technologies and current issues. In *Web Engineering: 13th International Conference, ICWE 2013, Aalborg, Denmark, July 8-12, 2013. Proceedings 13*. Springer, 510–513.

[53] Michael Nebeling, Maximilian Speicher, and Moira Norrie. 2013. W3touch: metrics-based web page adaptation for touch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2311–2320.

[54] Jeffrey Nichols, Brad A Myers, and Kevin Litwack. 2004. Improving automatic interface generation with smart templates. In *Proceedings of the 9th international conference on Intelligent user interfaces*. 286–288.
[55] Peter O'Donovan, Aseem Agarwala, and Aaron Hertzmann. 2015. Designscape: Design with interactive layout suggestions. In *Proceedings of the 33rd annual ACM conference on human factors in computing systems*. 1221–1224.
[56] Benjamin Paaßen. 2018. Revisiting the tree edit distance and its backtracing: A tutorial. *arXiv preprint arXiv:1805.06869* (2018).
[57] Seonwook Park, Christoph Gebhardt, Roman Rädle, Anna Maria Feit, Hana Vrzakova, Niraj Ramesh Dayama, Hui-Shyong Yeo, Clemens N Klokmose, Aaron Quigley, Antti Oulasvirta, et al. 2018. Adam: Adapting multi-user interfaces for collaborative environments in real-time. In *Proceedings of the 2018 CHI conference on human factors in computing systems*. 1–14.
[58] Fabio Paternò. 2004. ConcurTaskTrees: an engineered notation for task models. *The handbook of task analysis for human-computer interaction* (2004), 483–503.
[59] Fabio Paterno', Carmen Santoro, and Lucio Davide Spano. 2009. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transactions on Computer-Human Interaction (TOCHI)* 16, 4 (2009), 1–30.
[60] Xun Qian, Fengming He, Xiyun Hu, Tianyi Wang, Ananya Ipsita, and Karthik Ramani. 2022. Scalar: Authoring semantically adaptive augmented reality experiences in virtual reality. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–18.
[61] Sayan Sarcar, Jussi PP Jokinen, Antti Oulasvirta, Zhenxin Wang, Chaklam Silpasuwanchai, and Xiangshi Ren. 2018. Ability-based optimization of touchscreen interactions. *IEEE Pervasive Computing* 17, 1 (2018), 15–26.
[62] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J Ko. 2020. Scout: Rapid exploration of interface layout alternatives through high-level design constraints. In *Proceedings of the 2020 CHI conference on human factors in computing systems*. 1–13.
[63] Zecheng Tang, Chenfei Wu, Juntao Li, and Nan Duan. 2023. LayoutNUWA: Revealing the Hidden Layout Expertise of Large Language Models. *arXiv preprint arXiv:2309.09506* (2023).
[64] Kashyap Todi, Gilles Bailly, Luis Leiva, and Antti Oulasvirta. 2021. Adapting user interfaces with model-based reinforcement learning. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–13.
[65] Kashyap Todi, Jussi Jokinen, Kris Luyten, and Antti Oulasvirta. 2018. Familiarisation: Restructuring layouts with visual learning models. In *23rd International Conference on Intelligent User Interfaces*. 547–558.
[66] Kashyap Todi, Daryl Weir, and Antti Oulasvirta. 2016. Sketchplore: Sketch and explore with a layout optimiser. In *Proceedings of the 2016 ACM conference on designing interactive systems*. 543–555.
[67] Jacob O Wobbrock, Shaun K Kane, Krzysztof Z Gajos, Susumu Harada, and Jon Froehlich. 2011. Ability-based design: Concept, principles and examples. *ACM Transactions on Accessible Computing (TACCESS)* 3, 3 (2011), 1–27.
[68] David Wolber. 1996. Pavlov: Programming by stimulus-response demonstration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 252–259.
[69] David Wolber. 1997. Pavlov: an interface builder for designing animated interfaces. *ACM Transactions on Computer-Human Interaction (TOCHI)* 4, 4 (1997), 347–386.
[70] David W Wolber and Brad A Myers. 2001. Stimulus-response PBD: Demonstrating "when" as well as "what". In *Your Wish is My Command*. Elsevier, 321–XVI.
[71] Jason Wu, Titus Barik, Xiaoyi Zhang, Colin Lea, Jeffrey Nichols, and Jeffrey P Bigham. 2022. Reflow: Automatically Improving Touch Interactions in Mobile Applications through Pixel-based Refinements. *arXiv preprint arXiv:2207.07712* (2022).
[72] Jason Wu, Siyan Wang, Siman Shen, Yi-Hao Peng, Jeffrey Nichols, and Jeffrey P Bigham. 2023. WebUI: A Dataset for Enhancing Visual UI Understanding with Web Semantics. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–14.
[73] Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P Bigham. 2021. Screen parsing: Towards reverse engineering of UI models from screenshots. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 470–483.
[74] Mulong Xie, Zhenchang Xing, Sidong Feng, Xiwei Xu, Liming Zhu, and Chunyang Chen. 2022. Psychologically-inspired, unsupervised inference of perceptual groups of GUI widgets from GUI images. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 332–343.

A PARTICIPANT-AUTHORED EXAMPLES

Figure 9 shows examples of the AUIs resulting from the usage portion of the study by participants who chose to use the blending feature, and FrameKit could generate variations (Figure 9B,E) from other manually authored keyframes.

B HEURISTICS FOR OPTIMIZING EDIT SEQUENCES

FrameKit generates new variations of a UI by interpolating between existing examples. The generation process is mainly based on a backtracking algorithm used to generate edit sequences (composed of *insert*, *replace*, and *delete* actions on nodes) between two trees [56]. The original algorithm produces the *shortest* edit sequence (i.e., minimizes cost)

between the two trees, but this doesn't result in the best intermediate output. For example, all sequences generated by the algorithms are ordered so that all *replace* actions come first, followed by *delete* actions, then *insert* actions, which could result in many intermediate states with missing elements. We briefly describe the heuristics that we used to automatically improve the edit sequence for discrete interpolation. Any remaining "bugs" or changes in generated frames can be manually fixed by the author during the refinement step. Note that while our current implementation uses manually-defined heuristics, this does not preclude the use of more intelligent algorithms that re-order the edit sequence using data-driven models. For example, a model of UI progression could be learned from a dataset of UIs rendered under different conditions [72] or generated by an LLM prompted with a small number of examples.

Node Matching. At a high level, the edit-distance backtracking algorithm proposed by Paassen [56] works by first creating a mapping between nodes in the source and target trees. Nodes without matches indicate insertions and deletions; otherwise, they result in replacement edits or are used to infer other types of edits. Thus, the matching function used to map nodes in the source and target trees can have a significant effect on the edit sequence.

The default algorithm requires that all attributes of two nodes must exactly match; however, other parts of our algorithm (e.g., parameter interpolation) may violate this assumption. We defined a list of parameters to be ignored by the node matching function: image dimensions, font size, container-specific parameters (e.g., number of columns in a grid), and user-added adaptation parameters.

Container Semantics. If the transition between two keyframes involves moving elements from one container (e.g., list of items) to another, we observed that the original edit sequence would first delete the original container, insert children nodes, then insert the final container (parent) node to minimize overall cost. This results in several intermediate frames where the children nodes are incorrectly laid out because they aren't grouped together. Therefore, we wrote a heuristic to identify inserted container nodes and corresponding children and reordered them.

For each edit e in the original sequence, if e is an *insert* edit, find list the children of the node corresponding to e. For each child e_c, find the *delete* action where it is removed from the original container. Group together corresponding pairs of *delete* and *insert* commands corresponding to each child so that they get executed simultaneously (no intermediate frame will be missing the element).

Replacement Re-ordering. Transitions between two keyframes can involve changing the type of widget used to represent a piece of content. For example, one keyframe may use toggle switches to represent a set of Boolean options while the other may use checkboxes. These transitions may result in *replace* edits. Based observations of test-cases development examples, we applied alternate orderings to these edits.

For each edit e in the original sequence, if e is a *replace* edit and corresponds to a node that is inserted into a container, we moved the e to after the container's insertion. This was done since the new widget type may only make sense in the new container. We applied this rule after the container semantics heuristic.

Fig. 9. Examples of keyframes from our user study. A) is a manually created SUPPLE keyframe (P10). B) is an automatically generated SUPPLE keyframe (P10). C) and D) are manually created Mozilla keyframes (P9). E) is an automatically generated Mozilla keyframe (P9).