Malleable User Interface Toolkits for Cross-Surface Interaction

James R. Eagan
LTCI, Telecom ParisTech, Université Paris-Saclay
75013 Paris, France
james.eagan@telecom-paristech.fr

ABSTRACT
Existing user interface toolkits are based on a single user interacting with a single machine with a relatively fixed set of input devices. Today's interactive systems, however, can involve multiple users interacting with a heterogeneous set of input, computational, and output capabilities across a dynamic set of different devices. The abstractions that help programmers create interactive software for one kind of system do not necessarily scale to these new kinds of environments. New toolkits designed around these environments, however, need to be able to bridge existing software and libraries or recreate them from scratch. In this position paper, we examine these new constraints and needs. We look at three strategies for software toolkits that help to bridge existing toolkit models to these new interaction paradigms.

Author Keywords
user interface toolkits, cross-surface interaction, instrumental interaction, malleable software

INTRODUCTION

The design of user interface toolkits has changed relatively little since the first graphical interfaces began to appear: users interact with interface controls, or widgets, using a pointing device and a keyboard. While these toolkits have seen minor improvements over time, such as handling pointing with a finger or performing gestures, the overall design approach has changed little.

A programmer creates an interface using a collection of widgets. They typically come from a standard set of pre-defined widgets, but programmers may propose their own set of supplemental widgets. They may modify the behavior of an existing widget in some small way, such as a custom list view that might show font names rendered in the relevant font. Or they may provide wholly new behaviors, such as a double-ended range slider or an interactive graph view.

Presented at the *CHI 2017 Workshop on HCI.Tools: Strategies and Best Practices for Designing, Evaluating, and Sharing Technical HCI Toolkits*

Ultimately, the applications programmers create are heavily influenced by the set of widgets provided by the toolkit. These widgets, however, were designed around at a time when one user interacted with one machine, and applications consisted of a single process running on that machine.

These assumptions start to break down in the context of multi-surface or cross-device interactions, where a logical application—from the user's point of view—might involve processes running across multiple devices, such as a phone, tablet, tabletop, wall, or motion tracking system.

Moreover, adapting an existing application's interaction or functionality to handle unanticipated usage scenarios may be cumbersome. A user who wishes to add a new toolbar button for a frequently-used task, or a teacher who wishes to extend an email client's data detectors [5,6] to link course numbers to an intranet course management website, or a technical writer who wishes to reference BibTeX citations in email client or presentation tool would be hard-pressed to do so without modifying existing applications' source code.

Currently, developers of systems that permit such kinds of interaction in new environments or such malleable interfaces must create explicit support on an ad-hoc basis. We need to provide programmers the appropriate tools and abstractions on such concepts that make creating future applications feasible in the same way the current UI toolkits greatly reduced the barriers to making graphical user interfaces.

THREE CHALLENGES FOR TOOLKIT RESEARCH

We view three primary challenges for toolkit research in such future applications: handling the heterogeneity of future application contexts with multiple users, multiple machines, and multiple interaction modalities; making software more malleable to support users' own particular needs in their own situated contexts; and bridging between the current state-of-the-art and future development models.

Multiple users, machines, interaction modalities

Application developers can no longer make the assumption that there will be one user, interacting with a single machine using a single mouse and a single keyboard. Applications in multi-surface environments, for example, may run on one or multiple machines, and the number of machines present may evolve during a single interaction session.

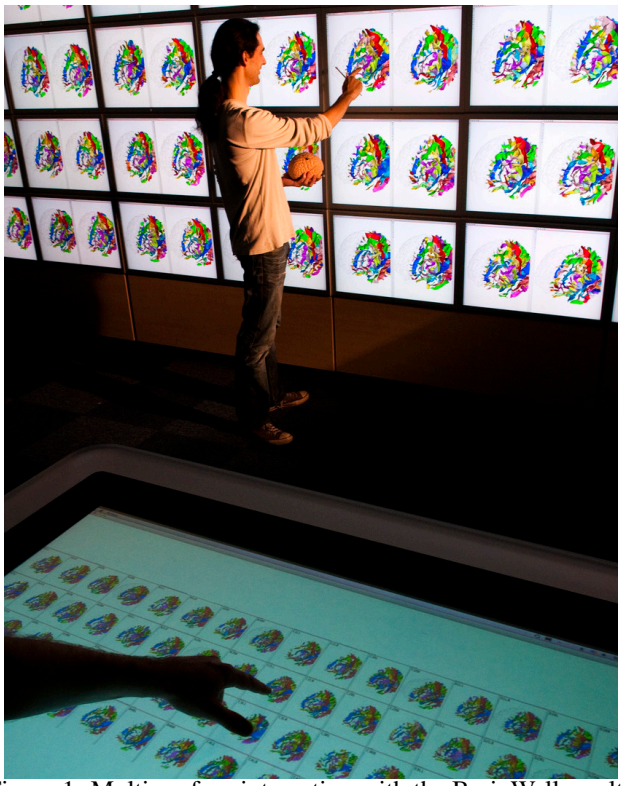

Figure 1: Multi-surface interaction with the BrainWall: multiple users interact with a table, wall-sized display, and physical interface props.

Figure 1 shows a real user scenario from our work with Substance Grise to create the BrainWall application [3]: A neuroscientist has the latest 3D brain scans of healthy and unhealthy brains on her smartphone. She enters the room and places her phone on an interactive tabletop to bring her data into the environment. Around the table, she and her colleagues arrange the brains to better facilitate comparison. To get a better view of the brains, she and her colleagues move in front of a wall-sized display that mirrors the display on the table, showing a high-resolution image of the brain scans. A colleague uses an instrumented wooden chopstick to point at a related structure on a plastic model of a brain, causing all of the brains displayed on the wall and table to re-orient their 3D views to that part of the brain. Another colleague takes out a tablet and selects one of the brains on the wall. On his tablet, he begins annotating the different structures and changing their colors, using the tablet as a personal workspace before sharing them with the group.

As this scenario shows, the BrainWall application actually consists of multiple applications running on multiple devices: a data provider on the neuroscientist's phone, an organizer on the tabletop, a viewer on the wall, a 3D tracker for the physical brain and pointer, and an annotator on the colleague's tablet.

Even without each of these individual components, the application would continue to function but without that component's capabilities. A scientist could continue to sort brains without the wall, or view the brains without the 3D tracker, etc. These devices may dynamically come and go, as when the colleague takes out a tablet to join the environment and annotate brains, or if the neuroscientist leaves the room to take a phone call.

Moreover, each of these different devices offers a different computational and interactional profile, with different memory, storage, processing, communication, input, or output characteristics. An application developer must manage the complexities of this heterogeneity and dynamicity manually.

To help with these challenges and to provide developers with a set of abstractions that simplify data sharing strategies and different functionalities between devices, discovery, and interaction, we created the Shared Substance prototype [3], described below. While this approach reduces the barrier to creating multi-surface applications, programmers must master a new "data-oriented programming" model and still must explicitly manage the specifics of data sharing strategies (such as via local replication or remote querying). Some of these details may intrinsically require specific consideration from the programmer. For the rest, we need to develop a collection of appropriate abstractions much as undo managers have freed programmers from needing to explicitly support such capabilities.

Making software malleable

Current applications are designed for a particular context of use, with developers making a certain set of assumptions about how the software will be used. It is not possible, however, for designers to foresee and anticipate the myriad ways that a user may make use of the software.

Current software toolkits provide relatively little support for end-users to extend the capabilities of their software. Some systems do provide support for users to create macros to automate certain actions, as in Microsoft's Office suite or with AppleScript interfaces, but these are limited to the customization hooks that developers explicitly embed in their software and maintain independently of their exposed functionalities. As such, developers must work explicitly to support these features and thus expose a larger surface area for potential bugs.

Some applications do provide support for plugins, but these interfaces are up to the individual application developer. Each application developer must create her own infrastructure for detecting, loading, unloading, and sandboxing such plugins on an ad-hoc basis. As a result, each application, if it provides a plugin interface at all, offers a differing degree of access to program concepts and objects, and each modification creator must learn the specific intricacies of that particular program.

What is missing is explicit support in the toolkit to create generalizable application objects that programmers can re-use. In the 1980's and 90's, it was common for applications to provide their own "macro" capabilities, where users could automate their software using macro scripts. Each application provided its own set of capabilities, using its own specific macro scripting language. Today, Mac applications built with the standard Cocoa toolkit are automatically scriptable using AppleScript and support a standard set of universal objects and

commands common to GUI applications: opening windows, selecting the frontmost document, clicking buttons, etc.

If application developers explicitly support it, they can add such higher-level concepts as messages in an email program or todo items in a task manager. Nonetheless, application developers must explicitly provide such support, and the user is limited to the specific hooks provided by the developer.

Supporting such kinds of customization should be an automatic consequence of using standard toolkit elements and design patterns for interactive software. If a user wishes to, for example, overlay subtitles downloaded from the internet on a movie file downloaded from the iTunes store, it should be feasible for the user to be able to connect a subtitles loader to the video playback controller, even if the application developer did not anticipate such a feature.

Recreating the universe

We have created various toolkits that attempt to explore these concepts [2–4]. One of the challenges in creating new ways of building interactive software is the bootstrapping problem. If the toolkit is completely built from scratch, all applications in the environment need to be created from scratch. This approach offers great flexibility, but requires significant development effort and tends to yield "toy" examples.

The Shared Substance [3] environment is an example of a toolkit built from scratch. Shared Substance is based on a data-oriented programming model similar to object-oriented programming in the sense that data associated with program concepts can be grouped together into objects and can have associated methods. In data-oriented programming, however, these methods are separable from the underlying data into *facets*, or collections of methods. Thus, an object running on a tabletop might offer a different set of functionalities than an object running on a smartphone, despite using the same underlying set of data. Data itself are organized into trees, providing a scene graph that can be shared across the different devices in a multi-surface environment.

Shared Substance programmers thus choose which subtrees to make available to other devices. The toolkit provides builtin discovery capabilities. When data are shared, or new devices become available, programs can either replicate the data by maintaining a cloned copy that must be kept in sync with its source, or they can mount the data, assuring that the original always maintains an authoritative copy of the data. Adding new functionality involves associating new Facets, or collections of methods, that can be attached to different parts of the data.

While this approach provides a set of transparent abstractions that frees the programmer from many of the challenges of multi-surface interactions, it does require programmers to think about and write software in a different way. We found that the mental gymnastics of contorting one's brain into a new way of thinking hindered the development of software in this environment and thus required a lengthy transition period for developers to adapt to this new model. Moreover, any new capabilities or applications, such as displaying a new kind of data on the wall, involved writing the code from scratch.

Bridging to legacy software: Scotty

Scotty [2] uses a different philosophy. Its goal is two-fold: to provide a test-bed for exploring instrumental interaction [1] and to provide a toolkit for the development of malleable applications. Rather than create a toolkit from scratch built around these concepts, we built Scotty as a meta-toolkit that grafted new capabilities into Cocoa. Thus, existing Cocoa applications can benefit from Scotty's new capabilities without modification of their source code.

Scotty is thus able to give arbitrary Cocoa applications the ability to load Scotty plugins that can be built using concepts of instrumental interaction. Scotty instrument plugins draw upon the Scotty toolkit to provide lenses into the underlying application's objects, views, and controllers. As such, creating the subtitles modification described above is "simply" a matter of identifying the playback window's playback controller and method to extract the current time. Scotty itself provides tools for helping a plugin developer to inspect and make sense of a host application's interface and core program objects. Adding this new functionality is thus a matter of attaching a transparent overlay window, loading a Python module that decodes subtitles, etc. In about 150 lines of Python code, a programmer can "teach" Quicktime Player to load subtitles from an external file, overlay them on the screen, and integrate them to the playback of the movie.

This approach has the advantage of quickly being able to take advantage of the full ecosystem of existing Mac applications. In theory, researchers are not bound by what they can develop from whole cloth. If an existing program offers the core functionality necessary, it should be straightforward to incorporate it into a research prototype.

The reality, of course, is different: design choices made by the original developers—and their rationale—are invisible. Shoe-horning them into a new environment, with different core assumptions and core values, can involve considerable effort. In the case of instrumental interaction, for example, Mac applications just aren't designed with this style of interaction in mind. A developer may end up spending as much effort adapting an existing application to a new interaction paradigm as she would have implementing a proof of concept. Only in hindsight can she evaluate whether that effort is worth the benefits of having a real application running in a real environment versus a functional proof-of-concept research prototype.

Webstrates

In between these two approaches, we built Webstrates as a sort of putty to build shareable dynamic media on top of existing web technologies. Developers can take advantage of their knowledge of HTML, JavaScript, and CSS to build webstrates in this new environment. The learning curve to be at least functional in this environment is relatively low. Moreover, existing web applications and libraries are readily available so long as they meet or can be made to meet certain core assumptions of Webstrates (notably that the DOM in a web browser is no longer ephemeral).

Webstrates combines several appealing properties for the exploration of new kinds of applications: it is compatible with

a large selection of applications that meet the requirements above, developers can leverage their existing knowledge and experience, web environments present a relatively low barrier to entry, and the webstrate canvas itself is a relatively open environment in which developers can experiment freely.

In contrast to a strict bridge such as Scotty, where application developers must fit within the constraints of the Cocoa development environment first, and then figure out how to express their new interactions within its concepts, webstrates allow developers to focus first on their concepts, and then on the constraints of the web framework. Both approaches use a bridging approach to bootstrap the development environment, but Webstrates finds a more lightweight balance than does Scotty.

CONCLUSIONS

Modern toolkits need to break the core assumptions implicit in historical interface toolkits. No longer do we live in a world of a single user at a single keyboard and mouse, interacting on his or her own. One or multiple users may interact with one or multiple devices in dynamic environments with different interactive and computational properties. The interactive metaphors that worked in historic environments do not necessarily hold up in the face of these new additions. We thus need to build new toolkits to help programmers build interactive software that can handle these changing constraints.

We have presented a collection of three toolkits that we have built, deployed, and published. Through this experience, we have explored different styles of implementing new interaction models and new ways of modeling interactive software; and new ways of building off of the existing set of tools that have been created over the course of the WIMP and early post-WIMP era.

ACKNOWLEDGEMENTS

This work has been funded in part by Digitéo, the French National Research Agency (ANR), and the Région Île-de-France.

REFERENCES

1. Beaudouin-Lafon, M.: Instrumental interaction: an interaction model for designing post-wimp user interfaces. In: CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems. pp. 446–453. ACM (2000)

2. Eagan, J.R., Beaudouin-Lafon, M., Mackay, W.E.: Cracking the cocoa nut: user interface programming at runtime. In: UIST '11: Proceedings of the 24th annual ACM symposium on User interface software and technology. pp. 225–234. ACM (2011)

3. Gjerlufsen, T., Klokmose, C.N., Eagan, J., Pillias, C., Beaudouin-Lafon, M.: Shared substance: developing flexible multi-surface applications. In: CHI '11: Proceedings of the 2011 annual conference on Human factors in computing systems. pp. 3383–3392. ACM (2011)

4. Klokmose, C., Eagan, J., Baader, S., Mackay, W., Beaudouin-Lafon, M.: Webstrates: Shareable Dynamic Media. In: UIST '15: ACM Symposium on User Interface Software and Technology. pp. 280–290. ACM (Nov 2015)

5. Nardi, B.A., Miller, J.R., Wright, D.J.: Collaborative, programmable intelligent agents. Communications of the ACM 41(3), 96–104 (1998)

6. Pandit, M.S., Kalbag, S.: The selection recognition agent: Instant access to relevant information and operations. In: IUI '97: Proceedings of the 2nd International Conference on Intelligent User Interfaces. pp. 47–52. ACM (1997)