A Nested API Structure to Simplify Cross-Device Communication

Andy Wu, Sam Mendenhall, Jayraj Jog, Loring Scotty Hoag, Ali Mazalek
Synaesthetic Media Lab
GVU Center, Georgia Institute of Technology
TSRB, 85 5th St. NW, Atlanta, GA 30318, USA
{ andywu, smendenhall3, jayraj.jog, scotty.hoag, mazalek }@gatech.edu

ABSTRACT
In this paper we present Responsive Objects, Surfaces, and Spaces (ROSS) API, a tangible toolkit that allows designers and developers to easily build applications for heterogeneous network devices. We describe the unique nested structure of the ROSS framework that enables cross-platform and device development and demonstrate its capabilities using several prototype applications.

Author Keywords
API, Toolkits, Tangible User Interface, Network Protocol, Interactive Tabletop

ACM Classification Keywords
H.4.3 Communications Applications

General Terms
Design

INTRODUCTION

The Responsive Objects, Surfaces and Spaces (ROSS) Application Programming Interface (API) is a way for applications to run across a variety of platforms and devices: tabletop computers, touch-screen mobile devices, responsive walls, and interactive 3D spaces. The goal of the ROSS API is to simplify the complicated communication problems between new Tangible User Interfaces (TUIs) that researchers and developers have built and continue to build. These new TUIs often have new sensing, actuation, and display technologies that require significant programming efforts to communicate with other TUIs. The ROSS API uses a nested structure to integrate several devices and an abstract middle layer that makes cross-device communication less complicated.

The ROSS API allows applications to: exchange information about devices they are running on and receive real-time input data from other ROSS-enabled devices. In a ROSS world: your smartphone can be used as a multitouch controller to control your desktop computer, several iPads can be put together to form a larger multitouch display, or a Pan-Zoom-Tilt (PZT) outdoor camera can be manipulated by a remote user with a mobile device.

TEI 2012, Feb 19-22, 2012, Kingston, Ontario, Canada.

RELATED WORK

There has been a variety of work on developing toolkits for physical sensors, tools to support co-located groupware, and other API frameworks within the areas of ubiquitous computing and tangible interfaces. We present a brief overview of this research here, and discuss where the ROSS API fits into this space.

Physical Device Toolkits

Many custom tangible interfaces are constructed from a variety of small sensing and display components that are integrated into physical devices based on the kind of user interaction they need to provide.

Saul Greenberg [2] claims that toolkits should accomplish several goals in order to promote creativity. These goals can be summed up as low technology barrier, simplicity, and hardware/software design patterns. For example, the Phidgets [19] toolkit he created helps people leverage their existing knowledge and skills, allowing them to create interesting and novel interfaces in a very short amount of time. Phidgets have lowered the barrier to entry for physical user interfaces in the same way as widgets make Graphical User Interfaces (GUIs) easier to develop. Physical Widgets (Phidgets) serve as a set of 'plug-and-play' building blocks that support low cost sensing and control from a PC. They consist of hardware components such as sensors, motors, relays, and displays that connect to commodity desktop computers through a Universal Serial Bus (USB) interface. They are programmable through software, and support (as of August 2011), 20 different languages and development environments across the three major desktop operating systems (Windows, Mac OS and Linux). Phidgets.com notes: "All the USB complexity is taken care of by our robust API. Applications can be developed quickly by programmers using their favorite language." [20] While the Phidgets API allows application developers to easily combine a variety of physical widgets, it does not support the development of applications that combine these

physical sensors and devices with other platforms, such as multitouch and tangible interaction tabletops that are based on computer-vision fingertip and fiducial marker tracking.

Interactive Surface Toolkits

A related area of toolkit research has focused on single-display groupware, such as multitouch tables or interactive surfaces.

MT4j [21] stands for MultiTouch for Java. Its goal is to support different kinds of input devices but with special focus on multitouch support. It uses a hardware abstraction layer to support a number of different hardware input devices. Anyone can add functionality to support a new device by extending the abstract super class of all the input sources. MT4j contains a number of presentation layer affordances, such as scenes to separate various parts of applications, extensible components, a canvas and rendering layer. Additionally, these GUI components support event propagation, listening and processing, to handle the input events from hardware components.

reacTIvision [5] was originally created to support a tangible touch synthesizer application, reactTable [4]. reacTIvision is a high-quality open-source library for performing tangible and finger tracking on tabletop computers. The TUIO protocol developed for this library is used by a number of other tangible toolkits, for example, MT4j and is used to send information about object state and touch events. reacTIvision consists of a computer vision library that performs tangible and finger tracking, and sends Open Sound Control (OSC) messages using the TUIO protocol.

Augmented Reality/Virtual Reality Toolkits

VRToolkit [22] is a set of software tools for producing virtual reality applications. It enables programmers to write programs for virtual reality environments and lists among its many features: real-time physics and collision detection, fog support, support for human and other bipedal character formats, a number of 3D model formats including texturing and shading, and UI elements.

The Designer's Augmented Reality Toolkit (DART) [8] is a set of tools for rapid prototyping and development of augmented reality applications. It is built on top of Macromedia Director, to open up development of augmented reality experiences to a larger population. It contains drag-and-drop Director behaviors to communicate with cameras, the marker tracker, hardware trackers and sensors. ARToolKit [6] is another well-known software tool for rapid design and implementation of AR experiences.

Integrated Toolkit Approaches

While physical device toolkits provide support for a broad range of devices and enable a designer to assemble a custom configuration of hardware components for a specific target application, interactive surface toolkits allow designers to create multiple different applications for a given platform implementation. Prior work on tangible application development has shown that both these approaches are important in evolving the area of TUI design, and we thus feel they should not remain entirely separate from one another. For example, tangible objects tracked on an interactive tabletop might incorporate input from other sensors and actuators, such as buttons and LEDs, to provide an additional level of user interaction. Several prototype systems, such as Urp [16] and Sensetable [12], have explored these kinds of interactions within a limited scope. In order to further support these developments, the toolkits for shared tangible display surfaces need to support communication with other kinds of devices as well.

TwinSpace [14,18] is a conceptualized framework to support cross-reality interaction for collaborative work. Among other components, TwinSpace uses Open Wonderland as a MMO (Massively Multiplayer Online) collaborative virtual world platform and the EventHeap [3] service to communicate between devices and services in the physical and virtual spaces. One of several interfaces supported by TwinSpace is the interactive tabletop. There are two types of tabletop interfaces in TwinSpace. One uses a top-down infrared (IR) camera and IR pens to interact with the digital contents projected on the tabletop from a top-down projector. The other one is the Tangible Tracking Table [10], an interactive tabletop display that tracks multiple objects and finger touches on its tabletop.

Other past work on integrated application toolkits includes the iROS platform [13], which supports an interactive workspace that incorporates technologies on many different scales, from personal devices to large table and wall displays. The Papier-Mâché toolkit [7] provides an event-based model for application development using RFID-tagged, barcoded, or computer vision identified objects. Finally, HephaisTK [1] provides an approach for fusing messages that come from different human input modalities (e.g. speech, gestures) in multi-modal interfaces.

Universal Plug and Play (UPnP) is a network protocol for home appliances and consumer electronics to seamlessly discover each other's presence on the network and establish functional network services for data sharing. A UPnP compatible device from can dynamically join a network, convey its capabilities upon request, and learn about the presence and capabilities of other devices. The UPnP concept is very similar to ROSS. The greatest difference is the nested structure of ROSS, which is capable of interweaving a tangible user interface network, while UPnP defines client-server communication between devices.

Finally, the Synlab API [11] has integrated different kinds of tangible platforms into a shared architectural framework that could be easily expanded to incorporate a wide variety of platform types and technologies. The ROSS API builds on this research, extending it to provide a shared abstraction by which different responsive objects, surfaces, and spaces

within an interactive application environment can be assembled into nested relationships and programmed using a common approach. This abstraction provides a way for application developers to think about and design how different tangible devices can be configured to work together in an integrated tangible application system. The architecture and implementation of the ROSS API are described in the following section.

ROSS: RE-THINKING SOFTWARE IN PHYSICAL SPACE

An API is a software interface designed to enable communication between different software programs. While most software APIs target adjacent levels in the software architecture, ROSS is designed to communicate through several levels across multiple platforms that are very different but have common interactions. This abstraction is very similar to a Java Virtual Machine (JVM). The official core Java APIs have to run in the environment of the three editions of Java platforms, Java ME (Micro Edition), Java SE (Standard edition) and Java EE (Enterprise Edition). The underlying hardware platforms can be very different, but running on one of the Java platforms is a fundamental requirement. ROSS API is designed to establish communication between the hardware driver level to the software application level. To accomplish this, ROSS makes the interface between these levels abstract.

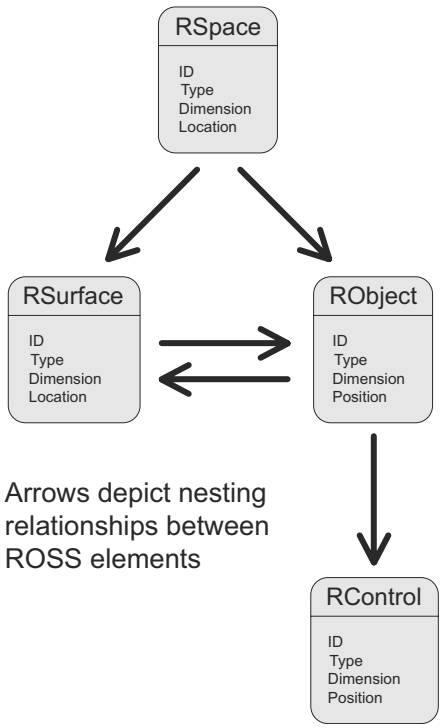

Figure 1. Core class structure of the ROSS API. RSpaces can have nested RSurfaces and RObjects. RSurfaces can only have nested RObjects and vice versa. RControls correspond to physical interface controls (e.g. buttons, dials) that can be attached to a RObject.

Most APIs are designed to support specific functions and do not interact with other levels in the software architecture. However, ROSS's nested structure makes interaction visible from one software level to other levels. The core of the ROSS API rests on the notion that the different objects, surfaces, and spaces in a responsive environment can exist in nested relationships with respect to one another. Figure 1 illustrates the nested structure of ROSS.

The nested structure of ROSS provides a common abstraction within which application developers can specify the particular configuration of their target platform. This is done using an Extensible Markup Language (XML) configuration file that describes a given platform's nested structure of spaces, surfaces, and objects, as well as its parameters such as ID, type, and dimensions. When an application receives such a configuration file from a candidate platform, it can further analyze it and determine whether or not the platform satisfies its needs.

What is ROSS?

Consider an office application that runs on an interactive tabletop that provides multitouch sensing, object tracking, and a surface display. Users interact with the application through tagged physical pucks and finger touches on the sensing surface. In this example, the table exists in a room embedded with location sensing technology, namely a camera mounted in the ceiling. The room is considered a responsive space. We can think of the interactive table itself as a responsive object that provides a nested sensing surface. Each tagged pawn or finger touch is another kind of interactive object that is in turn in a nested relationship to the sensing surface. Now imagine we replaced each puck with a multitouch enabled smartphone. These smartphones now each provide their own nested sensing surface, on which finger touches can again be seen as nested objects. The smartphones might also provide interaction through buttons, accelerometers or other physical control elements, which can be considered as nested controls that operate on the interactive object.

Design

With the emergence of popular mobile, tablet, tabletop, and tangible interfaces, interaction designers and digital artists now have a plethora of available interaction modes to experiment with. However, a particular direction that has been rarely explored is to utilize combinations of different platforms in a single unified design, even though such models have been suggested as early as 1994 in Mark Weiser's Tabs, Pads and Boards [17]. Several factors contribute to this stifled development. Construction of applications utilizing several platforms generally involves programming of low level networking protocols. In addition, a toolkit of this kind must also be very flexible and modular to allow for the addition of new platforms or sensing technologies.

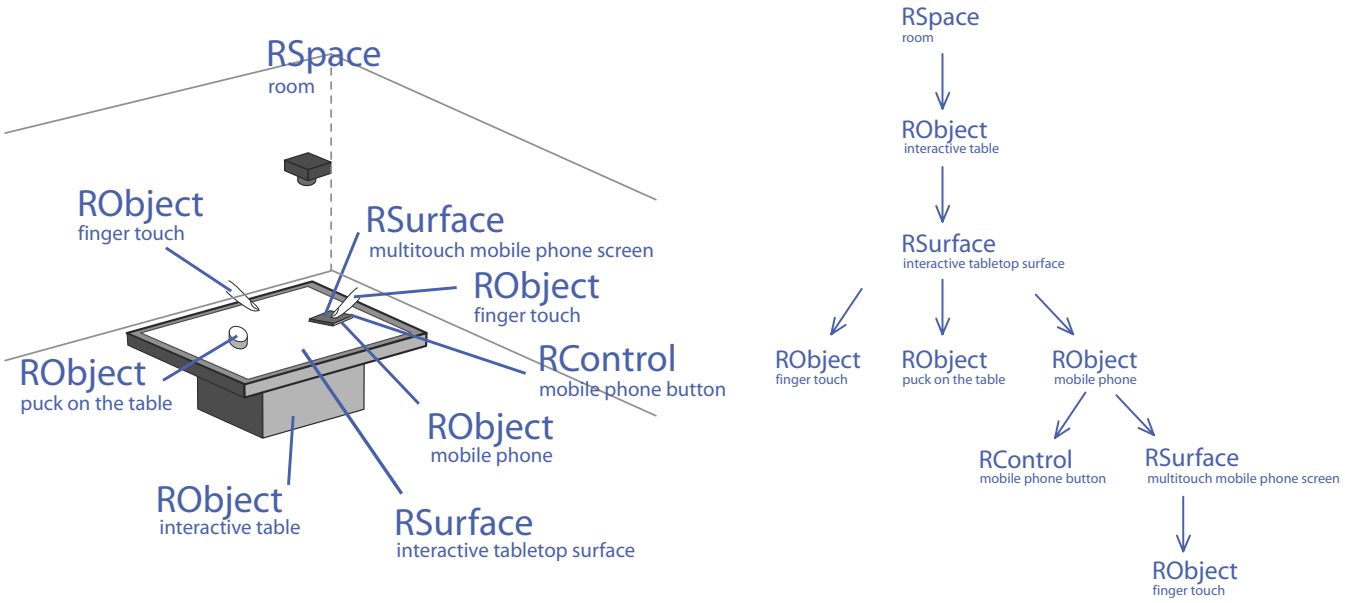

Figure 2. An example of ROSS's nested structure. In the left figure are the ROSS components in a room. There is an interactive tabletop with a puck and a mobile phone on its tabletop surface. Users interact with the tabletop and the objects on top of it. A sensor mounted in the ceiling keeps track of the location of the tabletop. The right figure shows the hierarchical nested structure.

Development Process

We inspected all the projects in our lab, as well as other canonical TUI examples (see [15]), when we created the specification for the ROSS API. The goal in the process was to find a method to simplify communication between all devices. We wrote down all possible inputs and outputs of a device. After that, we tried to establish connection from an output of one device to an input of another device. Ideas such as client-server architecture and peer-to-peer network for the inputs and outputs were discussed. But these fundamental network communication methods would complicate the framework when the number of devices increases. Moreover, separating inputs and outputs from a device destroys the unity of the device.

Changing the representation changes how people think. The nested structure changes how people think about the relationship between objects, surfaces, and spaces. In the left figure of Figure 2, there sit several separate objects in a room. To establish communication between the mobile phone and the tabletop, the most straightforward approach is to create a network connection between these two platforms. An experienced software developer knows how to create sockets and assigns one platform as the server and the other one as the client. When a touch happens on the tabletop, the mobile phone is not notified, as the event is not registered on the mobile phone. He has to register the mobile phone to listen for input events from the tabletop and vice versa to create a two-way communication. This structure and method of programming does not encourage developers (especially novices) to think about how their application solutions may incorporate multiple different platforms or devices at once. The development tools and approaches available thus tend to keep us developing within the confines of specific platforms rather than designing for complete physical spaces, with all the surfaces and devices they contain.

A different way of thinking is to relate all spaces, surfaces, and objects in a hierarchical nested structure. Since all these entities are connected like a tree-structure, there must be a route to navigate from one to another. If we move this relationship from the network level to a higher abstract level, programmers can bypass all the tedious low-level network programming. This type of thinking is logically plausible but with no support from any computer science theories at this moment. Nonetheless, we have implemented this concept by making the relationship abstract in a higher level.

Based on this concept of nested components, the core class structure of the ROSS API consists of a set of classes that represent responsive spaces (RSpace), responsive surfaces (RSurface) and responsive objects (RObject). RSurfaces are sensing surfaces, which actively report the detection and location change of RObjects, while RSpace can be deemed as a 3D version of RSurface. RObjects represent a vast variety of physical objects manipulable through tangible user interfaces. They could either be passive objects tracked by RSurface or RSpace, or active objects equipped with various sensors, or a mixture of both. RObjects could also have nested standard controls such as buttons, switches, sliders, and dials, represented by the RControl class. Each instance of an RSpace, RSurface, RObject, or RControl

knows its own type and unique identifier, and can also have a variety of other properties such as position, dimensions, orientation, state, etc.

COMMUNICATION ARCHITECTURE

ROSS is an API designed to integrate different tangible interaction platforms, such as multiuser tangible media tables, full-body interaction spaces, and RFID-tagged objects. The ROSS software developers had to think of creative methods to meet these constraints. Our lab designed the ROSS API to handle these challenges through the use of abstraction. We developed a common architecture (see Figure 3) for each node in a distributed ROSS application. We made the underlying network code as invisible as possible to the application designer. Nodes in a ROSS network automatically detect each other via multicast packets, and swap descriptions of their input device configurations. Developers can take advantage of the ease of XML format to describe complex nested relationships of objects, spaces, and surfaces. ROSS encapsulates all of a user's possible interactions with a device inside of a single representation, which can be fundamentally altered through the XML configuration file. Having the ability to quickly and easily swap out input methods adds an entirely different dimension of creative thinking to application development. Using the toolkit, designers gain a vocabulary for describing different sources of user input, and are free to tinker and stretch the boundary between the physical and digital worlds.

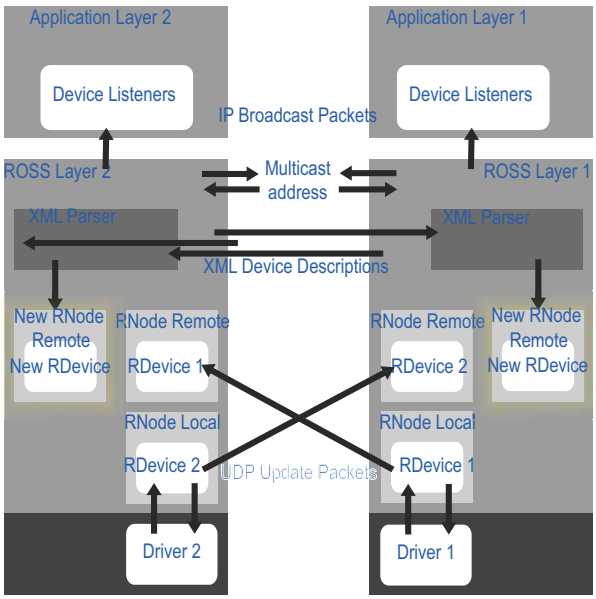

Figure 3. ROSS API communication architecture. The ROSS Layer serves as an interface between the device drivers and applications. Devices report events to the ROSS Layers in real-time via the Open Sound Control (OSC) protocol. Routing and device information exchange is done via XML between the XML Parsers.

The Nested Representation

Figure 4 shows a portion of the XML that is used to represent an acoustic-based sensing implementation of the TViews Table [9] platform for an application that uses button-enabled tagged TViews pucks. Currently, the XML files are created by hand using any standard XML editor. We will eventually provide a platform configuration tool that will generate an XML file based on how the platform components and parameters are configured in a drag-and-drop interface on screen.

```
<rossXML>
  <!-- Root object -->
  <robject>
    <!-- General tags, e.g. ID, name, etc. -->
    <name>TViews Table</name>
    <id type="int">1234567890</id>
    <description>...</description>
    <keywords>...</keywords>
    <location type="address">Atlanta, GA</location>
    ...
    <!-- Nested surface -->
    <rsurface>
      <!-- General tags -->
      ...
      <!-- Dimension in inches, e.g. rectangle, ellipse -->
      <dimension type="rectangle">
        <width>32</width>
        ...
      </dimension>
      ...
      <!-- Normalized vector, e.g. toward earth center (0, 0, -1) -->
      <orientation>...</orientation>
      ...
      <!-- Nested objects -->
      <robject>
        <!-- General tags -->
        ...
        <!-- Range of 64-bit TViews IDs configured with buttons -->
        <idfrom type="double">...</idfrom>
        <idto type="double">...</idto>
        <!-- Other general tags, e.g. name, description -->
        ...
        <!-- Nested control -->
        <rcontrol>
          <!-- General tags, sensor, protocol descriptions, etc. -->
          ...
        </rcontrol>
      </robject>
    </rsurface>
  </robject>
</rossXML>
```

Figure 4. The ROSS XML file of TViews Table. It shows the nested structure of ROSS. The TViews Table is an RObject that has a surface (RSurface). The surface has nested tagged TViews pucks (RObject) with buttons (RControl) on them.

APPLICATIONS

We describe several applications that have been created at various stages of the ROSS API development process.

ROSS Paint

ROSS Paint (see Figure 5) utilized an early version of the ROSS API. In this application, touch inputs from phones are shared between devices. A single component of the program listens for packets containing touch information from all other surfaces. Lines are drawn between received touches. Since the touch packets overlap in frequency, the lines are often drawn between user gestures, resulting in banded patterns of color.

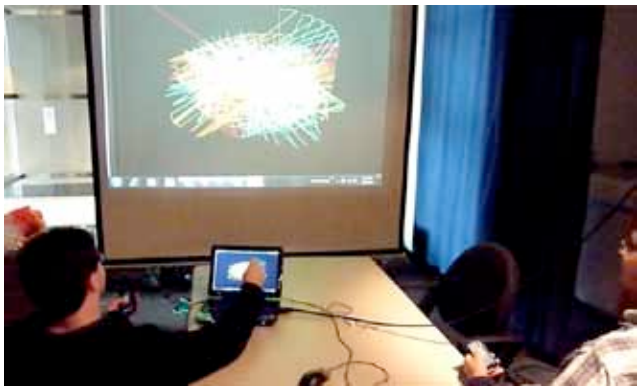

Figure 5. In this figure, three ROSS devices are working together to create an art piece using ROSS Paint. The left user sketches with his left thumb on the touch screen of a smartphone. He also holds a stylus in his right hand to sketch on a tablet computer. At the same time, the right user sketches with his left thumb on another smartphone. The projection screen is the mirrored image of the tablet screen.

ROSS Ripple

ROSS Ripple is an artistic demonstration of the capabilities of nested multitouch surfaces and tangible objects. User touches on either a phone or table surface cause colored "ripples" to emanate from the touch position and fade out over time. When a phone is held in the air, touches on the surface of the phone or table are virtually "synchronized" with the surface of the other device (see Figure 6). Ripples produced on the phone are thus produced on the table and vice versa, in positions relative to the size of each surface.

Figure 6. ROSS Ripple. The phone is away from the tabletop surface. The phone screen is mapped to the entire tabletop surface. Ripples on the table show up at corresponding positions on the phone's screen.

When a phone is placed on the table surface, it "merges" with the surface, and ripples created on the phone will spill out onto the table, and vice versa (see Figure 7). The simple ripples interaction would seem mundane on an ordinary desktop computer, where many other more visually dynamic effects have been implemented. However, visitors to our lab during demos and open houses have been very excited about the interaction between the surfaces. Seeing this interaction allowed them to imagine scenarios in which the image on the cellphone, now associated to the tabletop image in a synchronized manner, could reveal other layers of information such as the anatomy of the chameleon.

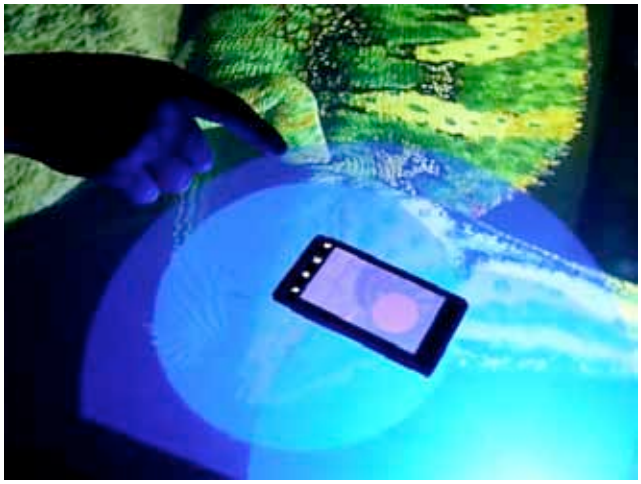

Figure 7. ROSS Ripple. The phone is placed on the tabletop. The phone's screen has become part of the tabletop surface. Ripples on the cellphone screen pass the physical boundary of the cellphone to the tabletop.

CHALLENGES

The greatest challenge we have encountered is the software development knowledge of the student programmers who use the ROSS API. Making ROSS easy to use even for novice programmers is important for its success.

Embedded Systems

Some TUIs use embedded hardware and software. These systems have limited speed, memory, and storage. Some of these devices do not even have required software libraries to run the basic networking services. How to build ROSS API on top of these tiny devices presents a significant challenge.

Developing for New Devices

How to allow an inexperienced ROSS developer apply the ROSS concept to a new device smoothly is a challenge. The recommended procedure is to create a ROSS XML with the nested structure for the device, create a ROSS XML parser that converts XML files to associated classes, establish OSC communication, and a ROSS manager that handles the incoming events. The core of the ROSS API is written in Java, an Object Oriented Programming (OOP) language. A ROSS API developer thinks in an OOP style. For a developer who favors a different programming paradigm, there is a lot of work to be done.

The ROSS XML file

Preparing the XML file remains tedious. The example XML in Figure 4 is only part of the entire file of an interactive table. It documents all necessary information of a device. There is no automatic way to generate this file as

yet. A ROSS user has to prepare this file to enjoy all the benefits of sharing resources among ROSS devices. A graphical interface that allows a user to add and delete elements in the XML file is a transient solution before we implement an automatic ROSS XML generator.

LESSONS LEARNED

The young software developers on this project approached the solution in an irregular way. The experience of using multiple computers to distribute a computing task through the network became a solution that passed to other members in the research lab later.

Abstraction

Closer to tangible interaction software, libraries like MT4j employ hardware abstraction layers to allow programmers to interact with hardware input devices without thinking about the working of those devices. They also enable programmers to add support for new devices by extending the capabilities of the abstract representations of hardware input devices. This allows use of the libraries in ways not anticipated by the designer.

We expect ROSS to play a similar role on programmer creativity. We provide abstractions for tangible input devices. In addition, the library allows programmers to treat objects, surfaces and spaces with the same spatial relationships as they have in real life. For example, a smartphone is a ROSS object, but it also has within it a ROSS surface. Similarly, a tabletop computer is a ROSS object, which contains a ROSS surface (the tabletop surface). When a smartphone is placed upon this surface, the surface then contains a ROSS object (the phone), which in turn contains another ROSS surface (the display). These real world spatial relationships are mirrored in software, while at the same time remaining silent about the type of tabletop computer and smartphone.

Tangible Promotes Dialogue between Developers

When presenting an idea for a cellphone application, one often has a phone or even an arbitrary object in hand to illustrate the concept. The physical object makes an idea easier to understand than presenting through slides or other indirect means. The ROSS API is designed for integrating tangible user interfaces. In other words, a lot of these interfaces are graspable. Usually when we discussed an idea in the lab, we picked up the object and demonstrated it to other colleagues. This characteristic of the "tangible" user interface helps make project conversations more fluid. On the other hand, programming interfaces with no physical body requires a lot of mental work. As a result, we see a lot of software diagrams that are composed of colorful blocks.

Educational

Some of the solutions provided by the student software developers may not be recognized as creative by professional or senior software developers. This situation happens often in a research lab whose members do not necessarily have a computer science background. For example, after several hours of effort, a novice software developer writes several Java classes to notify one class when a specific event happens. This mechanism is in fact a callback function that is widely used in Java programming.

Ease of Use and Creativity

ROSS provides a language for describing embedded user interactions across a well-defined and extensive range of devices. Such a vocabulary gives designers the opportunity to play with creating novel combinations of interactions. It also allows designers to bypass the technical details of network programming in favor of building and modifying ROSS XML configuration files. ROSS enables the design and construction of applications with much richer interactions.

Lowering barriers to entry in a field leads to innovation in that field. For example, Phidgets and Arduino put hardware sensors and other devices into the hands of hobbyists. Prior to this, hardware or embedded systems programming was very much a specialized field.

With ROSS API we aim to provide the same impetus to tangible interaction application development. ROSS API makes programming for varied platforms simple. It takes care of details such as networking, event handling and device management, allowing programmers to focus on making an engaging and creative application. Some evidence of this is already available. For ROSS Paint and ROSS Ripples, the applications themselves perform very little of the common tasks of establishing connections, managing networking, events and subscriptions, and other necessary tasks. Instead the application code consisted entirely of application logic.

SUMMARY AND FUTURE WORK

ROSS uses a fundamentally different design paradigm for specifying a networking API. Through its novel hierarchical nested structure, ROSS closely models the environments for which it is designed. These new concepts create a network of tangible user interfaces that promotes embodied interaction.

The unorthodox API architecture of ROSS opens another window for tangible user interface researchers to implement their applications in a different way. This novel method in API development has not yet been evaluated at a technical level. Therefore, we cannot make claims about its efficiency. However, once a TUI developer follows the specification defined in ROSS, she can create applications that communicate with heterogeneous devices and benefit from being part of the entire ROSS tangible network.

Our next steps in the development of the ROSS API include incorporating a greater number of platforms and devices beyond the interactive tables, phones, and custom spaces and devices that are currently supported. We also plan to

develop the platform configuration tool that will enable ROSS XML files to be automatically generated. Finally, we plan to eventually make the ROSS API openly available so that developers outside of our research lab and institution will be able to benefit from the ROSS approach. This will not only provide additional testing for our continued development of ROSS, but we hope will also invite others to contribute to the ROSS API codebase by incorporating their own favorite platforms and devices that may not be available to us in order to make the ROSS API as broadly useful and usable as possible.

ACKNOWLEDGMENTS

We gratefully acknowledge the support of Google and the research collaboration grant from Steelcase Inc. in conducting this research.

REFERENCES

1. Dumas, B., Lalanne, D., and Ingold, R. HephaisTK: a toolkit for rapid prototyping of multimodal interfaces. *Proceedings of the 2009 international conference on Multimodal interfaces*, ACM (2009), 231-232.
2. Greenberg, S. Toolkits and interface creativity. *Multimedia Tools and Applications 32*, 2 (2007), 139-159.
3. Johanson, B. and Fox, A. The Event Heap: a coordination infrastructure for interactive workspaces. *Mobile Computing Systems and Applications, 2002. Proceedings Fourth IEEE Workshop on*, (2002), 83-93.
4. Jordà, S., Geiger, G., Alonso, M., and Kaltenbrunner, M. The reacTable: exploring the synergy between live music performance and tabletop tangible interfaces. *Proceedings of the 1st international conference on Tangible and embedded interaction*, ACM (2007), 139-146.
5. Kaltenbrunner, M. and Bencina, R. reacTIVision: a computer-vision framework for table-based tangible interaction. *Proceedings of the 1st international conference on Tangible and embedded interaction*, ACM (2007), 69-74.
6. Kato, H., Billinghurst, M., Poupyrev, I., Imamoto, K., and Tachibana, K. Virtual object manipulation on a table-top AR environment. *Augmented Reality, 2000. (ISAR 2000). Proceedings. IEEE and ACM International Symposium on*, (2000), 111-119.
7. Klemmer, S.R., Li, J., Lin, J., and Landay, J.A. Papier-Mache: toolkit support for tangible input. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2004), 399-406.
8. MacIntyre, B., Gandy, M., Dow, S., and Bolter, J.D. DART: a toolkit for rapid design exploration of augmented reality experiences. *Proceedings of the 17th annual ACM symposium on User interface software and technology*, ACM (2004), 197-206.
9. Mazalek, A., Reynolds, M., and Davenport, G. The TViews table in the home. (2007).
10. Mazalek, A., Winegarden, C., Al-Haddad, T., Robinson, S.J., and Wu, C.-S. Architales: physical/digital co-design of an interactive story table. *Proceedings of the 3rd International Conference on Tangible and Embedded Interaction*, ACM (2009), 241-248.
11. Mazalek, A. Tangible Toolkits: Integrating Application Development across Diverse Multi-User and Tangible Interaction Platforms. *Let's Get Physical Workshop, DCC'06*, (2006).
12. Patten, J., Ishii, H., Hines, J., and Pangaro, G. Sensetable: a wireless object tracking platform for tangible user interfaces. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM (2001), 253-260.
13. Ponnekanti, S.R., Johanson, B., Kiciman, E., and Fox, A. Portability, Extensibility and Robustness in iROS. *Pervasive Computing and Communications, IEEE International Conference on 0*, (2003), 11.
14. Reilly, D.F., Rouzati, H., Wu, A., Hwang, J.Y., Brudvik, J., and Edwards, W.K. TwinSpace: an infrastructure for cross-reality team spaces. *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, ACM (2010), 119-128.
15. Ullmer, B. and Ishii, H. Emerging frameworks for tangible user interfaces. *IBM Syst. J. 39*, 3-4 (2000), 915-931.
16. Underkoffler, J. and Ishii, H. Urp: a luminous-tangible workbench for urban planning and design. *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, ACM (1999), 386-393.
17. Weiser, M. The computer for the 21st century. *Scientific American 265*, 3 (1991), 94-104.
18. Wu, A., Reilly, D., Tang, A., and Mazalek, A. Tangible navigation and object manipulation in virtual environments. *Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction*, ACM (2011), 37-44.
19. Phidgets - Physical widgets for prototyping physical user interfaces. http://grouplab.cpsc.ucalgary.ca/phidgets/.
20. Phidgets Inc. - Unique and Easy to Use USB Interfaces. http://www.phidgets.com/.
21. MT4j - Multitouch For Java. http://www.mt4j.org/mediawiki/index.php/Main_Page.
22. WorldViz : Vizard. http://www.worldviz.com/products/vizard/index_b.html.