Deep Shot: A Framework for Migrating Tasks Across Devices Using Mobile Phone Cameras

Tsung-Hsiang Chang [*]
MIT CSAIL
vgod@mit.edu

Yang Li
Google Research
yangli@acm.org

ABSTRACT

A user task often spans multiple heterogeneous devices, e.g., working on a PC in the office and continuing the work on a laptop or a mobile phone while commuting on a shuttle. However, there is a lack of support for users to easily migrate their tasks across devices. To address this problem, we created Deep Shot, a framework for capturing the user's work state that is needed for a task (e.g., the specific part of a webpage being viewed) and resuming it on a different device. In particular, Deep Shot supports two novel and intuitive interaction techniques, *deep shooting* and *deep posting*, for pulling and pushing work states, respectively, using a mobile phone camera. In addition, Deep Shot provides a concise API for developers to leverage its services and make their application states migratable. We demonstrated that Deep Shot can be used to support a range of everyday tasks migrating across devices. An evaluation consisting of a series of experiments showed that our framework and techniques are feasible.

Author Keywords

Multi-device environment, mobile interaction, camera, computer vision, task migration.

ACM Classification Keywords

H5.2 [Information interfaces and presentation]: User Interfaces: Input Devices and Strategies, Interaction Styles.

General Terms

Design, Human Factors

INTRODUCTION

The landscape of personal computing has shifted from one computer per user to multiple heterogeneous devices per user [7]. To carry out an everyday task, such as finding a restaurant for dinner, a user often switches from one device to another according to the situation. For example, a user has looked up the directions to the restaurant on her PC at home but then redoes the search on her phone for navigation in her car. A recent study found that this and other common tasks, such as email and web browsing, were the source of the most frustration while switching between different devices [12].

Figure 1. A user takes a picture of the screen of her computer and then sees the application with the current state on her phone. Our system recognizes the application that the user is looking through the camera, automatically migrates it onto the mobile phone, and recovers its state.

The lack of tool support for migrating tasks across devices has also been pointed out by several previous studies. A survey [18] conducted in 1997 showed that 62.9% of people stated they transferred information for completing a task on other devices "by hand", i.e., reading a text string on a display and typing it on another computer. A non-trivial number of people transferred data through shared files, FTP, or emails. Surprisingly, a more recent study in 2008 [7] showed that people were still using these old-fashioned mechanisms plus emerging cloud services (e.g., Google Docs) to transfer information across devices. Although cloud services and ubiquitous access to the Internet seem to be an antidote, the study found people were still frustrated as they have to manually reconstruct their work state, e.g., opening and locating the part of a PDF article that was viewed on the previous computer to continue reading. Furthermore, moving between heterogeneous devices (e.g.,

CHI 2011, May 7–12, 2011, Vancouver, BC, Canada.

[*] This work was conducted during an internship at Google Research.

a PC and a mobile phone) amplifies the task resumption overhead due to various contextual and resource constraints [2,3,12].

Prior work made substantial progress in providing more integrated user experience for task migration across devices (e.g., [8,15,16,19]). However, existing solutions are insufficient in two ways. First, prior work primarily focused on infrastructure for transferring data across devices, not user interaction. Compared to moving data or application windows around on a single computer by drag-and-drop, there is no similarly easy method for cross-device migration. Secondly, existing tools are mostly document-centric with little support for recovering a work state [2,3]. Manually recovering a work state requires users to deal with many details that can distract them from the task that they want to resume.

To address these issues, we present Deep Shot, a framework that supports task migration by allowing users to transfer not only documents but also application states across devices using a mobile phone camera. Deep Shot provides two novel camera-based interaction techniques, *deep shooting* and *deep posting*. These two techniques allow seamless and intuitive migration of user tasks from one device to another by one uniform operation: taking pictures.

Deep shooting allows a user to capture and to persist the *deep information*, i.e., the information behind the raw pixels, such as application states, with a camera-like mobile phone application in a single click (see Figure 1). The captured work state can be resumed immediately on the mobile phone, opened later, or migrated to another device with deep posting, which pushes deep information to a device with a camera as well.

To support deep shooting and deep posting, we created a framework, Deep Shot, for application developers to easily incorporate these techniques into their applications. It includes two key ideas. First, Deep Shot uses robust computer vision algorithms to identify what portion of the screen the user is looking at through the camera. We conducted two experiments to show the feasibility of this technology. Second, Deep Shot requires the applications to encode the deep information as Uniform Resource Identifiers (URIs) so that a task can be resumed even using different applications, such as viewing a Microsoft Outlook contact's information with a native Android application.

We make the following contributions in this paper:

- Deep shooting and posting: two novel techniques for pulling and pushing a task (information or application states) using a mobile phone camera.
- Deep Shot: a framework for developers to easily make their application states migratable across devices.

In the rest of the paper, we first clarify our motivation using a running example in which a user searches for a restaurant and discuss how Deep Shot supports this task by allowing a user to easily migrate the task across devices. Next, we discuss the design of our framework, and implementation details. We also show how developers can leverage our framework to enable deep shooting and posting in their applications. We then discuss the range of scenarios that Deep Shot can address. Finally, we describe an evaluation of our techniques and framework, and conclude with related and future work.

MOTIVATION

Here we discuss why it is important to address task migration across devices. Let us assume a user, Bob, is searching for a restaurant for dinner on Yelp at home. Bob has read several reviews of a restaurant on his desktop computer. He decides to try one restaurant and clicks on the map on the review page to read the driving directions. Everything is going smoothly until he needs to leave home and move the directions to his mobile phone for navigation in his car. How can Bob open the same region of the map on his phone?

Bob could manually type the restaurant's address or name and search on the phone. Or he could click "Link" on Google Maps to get a bookmarkable URL of the current region, and email that URL to himself so that he can look for the email and open the URL in it on his phone later. These approaches generally require the user to perform two steps: 1) *inspecting* the internal state of the application, e.g., the URL, and 2) *copying* it by hand or via a temporary medium, e.g., a file or an email, from one device to another.

The inspecting step varies widely depending on the applications. In a web page or web application, the Uniform Resource Locator (URL) on the address bar often represents the application state that a user intends to transfer. However, in many web applications using Ajax, the URL no longer represents the current state of the application. A user is often required to perform extra steps to retrieve the real "bookmarkable" URL, such as what Bob would do in Google Maps. However, many Ajax and desktop applications do not have a URL that represents what the user is viewing and working on. Tools have been developed to overcome this problem by recording the commands needed to return a page [10]. However, a desktop application's state is generally inaccessible by end users.

The copying step requires a user to either manually re-enter the information on another device, e.g., when transferring a small piece of information such as a short URL or the name of a landmark, or understand and deal with low-level operations such as how to save information in files and use file transferring software.

Anecdotally, people often take a picture of a particular region of interest (ROI) on the monitor using a camera, which is generally available on modern mobile phones. This method utilizes the camera as a physical tool to directly inspect and copy the information at the same time. In Bob's scenario, he could capture the portion of the map he needs in one simple step, i.e., taking a picture. This

method is simple, independent of the application the user is using, and avoids many of the hassles of manual inspection and copying. However, it is limited in that information being transferred is encoded in raw pixels and will not allow a user to perform further interaction, e.g., panning or zooming a map.

DEEP SHOOTING & POSTING

Inspired by the picture-shooting metaphor above, we designed and implemented deep shooting and posting, two novel techniques that are as simple to perform as taking a picture, but copy *deep information* behind the raw pixels of the captured region, that is, the application state.

With deep shooting (see Figure 1), Bob can copy a specific region of the map displayed on his computer's screen to his mobile phone by simply taking a picture of it with the phone's camera. The same region is then shown on the phone automatically. More importantly, the captured map remains interactive on the mobile phone. In other words, Bob can pan the map to see the area that is not originally captured by the camera, or zoom in to see more details of the streets.

Based on the picture captured with deep shooting, our system automatically identifies the captured area on the screen and the front-most application containing that area. Our system then pulls information from the application and sends it to the mobile phone that took the picture. The information is encoded as a URI, which has been accepted as a standard way to launch applications on contemporary mobile operating systems such as Apple iOS and Android. Therefore, the user can view or manipulate the extracted information on the mobile phone with native applications.

As a complement to deep shooting, deep posting allows a user to push information from a mobile phone to another device. Let us assume Bob has opened the restaurant review page on his mobile phone to write a review, but soon decides he would rather continue this task on his desktop computer, where it is easier to type. To do so, Bob aims the mobile phone camera at the computer screen with the review page still shown on the phone. Once deep posting is activated (e.g., via a hot-button on the phone), the review page becomes semi-transparent so that the user can see through it and know which part of the screen he is targeting at. Based on the screen region as seen through the camera, once Bob confirms, deep posting identifies the intended computer screen and automatically opens the same review page on it.

Deep posting employs the same mechanism as deep shooting in identifying the target computer screen and the specific region on the screen that the user sees through the camera. However, unlike the deep shooting, deep posting does not need to identify which application the user is looking at, since the application for handling the information being posted may not be running.

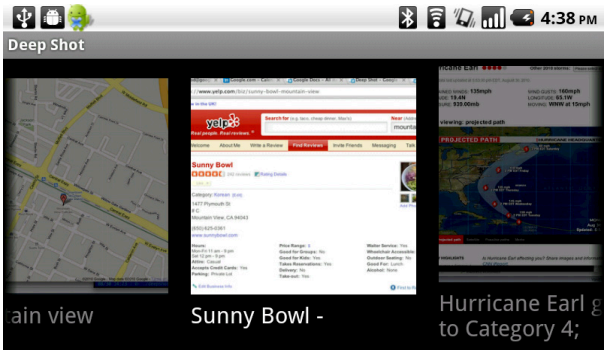

Figure 2. The Deep Shot gallery allows a user to quickly launch an application with a previously captured work state. A user can flip left or right on the touch screen to browse the gallery.

The deep shooting application running on the mobile phone maintains the history of deep shots that a user has taken. Similar to browsing photos in a photo gallery application, a user can browse all of his deep shots (see Figure 2). Each shot in this gallery shows the title and the thumbnail of the captured application. With the gallery, a user can directly launch a desired application with its captured state on the phone. The gallery provides a simple interface for users to manage their tasks and switch between them.

Currently, the Deep Shot framework is designed for migrating tasks across personal devices. Thus, before using deep shooting or posting, a user needs to log into a remote server with the user's credential on each personal device, and the credentials can be stored in the devices thereafter. Therefore, this authentication step only needs to be performed once for each device. We will discuss the possibilities of eliminating the authentication process in the Future Work section.

THE DEEP SHOT FRAMEWORK

To support deep shooting and posting, we designed the underlying Deep Shot framework with two goals in mind. First, from the user's perspective, deep shooting and posting should be as easy to use as taking a picture with an ordinary camera. Therefore, the user should not have to do any network configuration beforehand nor pair any devices to use Deep Shot. Second, from the developer's perspective, the Deep Shot framework should be easy to integrate with her applications. Developers should not need to worry about the communication between devices nor understand how to detect what portion of screen the user is looking at through the camera.

To achieve these goals, we have to carefully choose the technologies for the link layer and the network layer. Many options exist for the link-layer technologies, such as IrDA, USB, FireWire, Ethernet, Bluetooth and WiFi. We chose WiFi/Ethernet for their ubiquity on almost all devices and then we can utilize the standard TCP/IP stacks. For the network layer, device discovery and association are still challenging obstacles today. Since we want to focus on migration across personal devices, we decided to base our

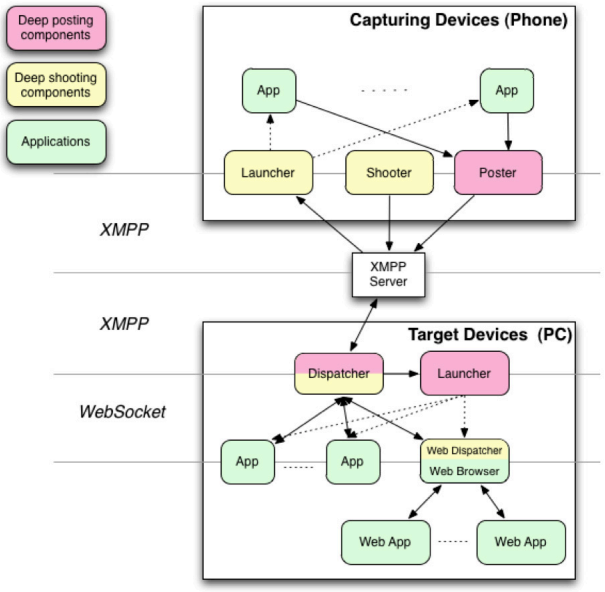

Figure 3. The system architecture of Deep Shot. Solid lines represent direct messages between components, whereas dotted lines represents the launching signal sent from the launcher.

framework on an instant messaging (IM) architecture, which was previously used in cross-device infrastructures such as PIE [16]. Thus, we can build on top of standard TCP/IP and avoid the problems of dynamic IP addresses, private IP addresses behind Network Address Translation (NAT) gateways, and firewalls that block connections from the outside. However, this architecture requires an authentication step before using our system. Fortunately, authentication only needs to be performed once for each device and does not add any cost for using Deep Shot thereafter.

We chose Extensible Messaging and Presence Protocol (XMPP), also known as Jabber, for our IM protocol. One reason is that XMPP supports logging in with the same user account from multiple different devices. A user account with a device has a unique identifier of the form "user@server/device." This allows users to set up all of their devices with the same user name. Since XMPP can list all of a user's presences across devices, users do not have to manually add their devices into their contact list. Also, the size of a XMPP message is not limited, which means we can send relatively large data, e.g., a JPEG photo, through a typical message packet without hacking the protocol.

System Components

Deep Shot's architecture is shown in Figure 3. The pink components are required for deep posting, whereas the yellow ones are required for deep shooting. There are five roles in our system: a *shooter,* a *poster,* a *dispatcher, launchers* and *applications (apps)*. The shooter and the poster only run on a *capturing device*, e.g., a mobile phone equipped with a camera. The dispatcher runs on a *target device*, which accepts a deep shooting or posting request from a capturing device. The launchers run on both sides of the system. On the capturing device, the launcher launches mobile applications to recover a work state captured by deep shooting, whereas on the target device it launches desktop applications to present a work state that is posted by deep posting.

Protocol Design

Here we describe the protocols between each pair of system components. To simplify the design of our protocols, the messages exchanged among all components are structured and encoded in the JavaScript Object Notation (JSON) key-value pairs. Besides, all binary data, e.g. images, are encoded in Base64 so we can include them in standard XMPP messages.

Deep Shooting: Shooter-Dispatcher Protocol

Once a user uses the shooter to take a picture of the region of interest on a computer monitor, an XMPP message with the picture and a subject *deepshot.req* indicating a deep shooting request is sent to each available device.

When the dispatchers running on the target devices receive the request message, they immediately take a screenshot of the entire screen of their devices. Each dispatcher then matches the picture it receives against the screenshot. The matching algorithms locate the region that the user was looking at through the camera. Then the dispatcher sends a new message with the x-y coordinates of the corners of the region and the central point of the region to the front-most application overlapping the center point, through a WebSocket connection.

After the application has handled the message and returned a response, the dispatcher inserts the application name, and the thumbnail of the matched region on the screenshot. Both are useful for browsing the deep shooting history on the capturing device. Finally, the dispatcher sends the response to the shooter via the XMPP server.

Deep Shooting: Application-Dispatcher Protocol

The dispatcher is designed as a daemon that always runs in the background on all personal devices. The dispatcher has the user's credentials, so it is always connected to the XMPP server. Therefore, any device can obtain the availability of any other device from the XMPP server.

The dispatcher communicates with each application using a dedicated WebSocket connection. WebSocket is a new protocol that supports full-duplex and bi-directional communication over a TCP socket. We choose WebSocket for two reasons. First, it is being standardized by the IETF and W3C, and modern browsers already support WebSocket. Thus, we can easily implement a browser extension as a second-level dispatcher for web applications. Second, since the traditional TCP socket is the most pervasive inter-process communication (IPC) mechanism, and WebSocket is a simple extension of TCP sockets, traditional desktop applications can support it easily.

Each time an application that supports Deep Shot is launched, it registers itself with the dispatcher through a

WebSocket connection on a TCP port. A registration process starts after the standard WebSocket handshaking. The application sends out a registration message with its name. If the dispatcher accepts the registration, it returns an OK message, or else it returns a decline message indicating the reason and closes the connection. To support deep posting, an application sends the command "accept *URI_SCHEME*", which indicates what types of URI schemes it accepts. For example, an email client can register the "mailto:" scheme, and a web browser can register the "http:" and "https:" schemes. Once the registration is completed, this WebSocket connection should be kept persistent until the application is closed so the dispatcher can proactively notify the application when a request is coming.

Once an application has dealt with the dispatcher's request, it replies with a message consisting of at least a URI that encodes the state of the application or the information to expose. If needed, the application can attach offline resources or files in the response message. Each attached file is stored in a JSON structure with the file name and the content of the file.

Deep Shooting: Dispatcher-Launcher Protocol

After the dispatcher receives a reply message from the application, it routes that message with a subject "*deepshot.resp"* back to the capturing device that sent out the request.

On the capturing device, a launcher waits for the "*deepshot.resp*" messages. Once a response message arrives, the launcher decodes the message and writes all attachments to the storage on the device. Finally, it opens an appropriate application that handles the URI replied from the target device to recover the work state and resume the task flow.

Deep Posting Protocol

The deep posting protocol is based on the same foundation we used in deep shooting, including JSON structures and XMPP communication. The key role in our system for deep posting is the *poster* that runs on a capturing device. The poster accepts requests from the applications that support our Deep Shot framework. The posting requests should consist of at least a URI representing the internal state of the application.

Once the poster receives a posting request from an application, it opens the camera and overlaps the screenshot of the application on the viewfinder so that a user can see the target device and the information to post at the same time. After the user has confirmed the target device through the viewfinder, the poster creates a "*deeppost.req*" message with the picture taken by the camera and the request from the application. Finally, this message is sent to all available devices, in the same way as deep shooting.

After the dispatchers running on the user's other devices receive the "*deeppost.req*" message, each of them runs the same vision algorithm used for deep shooting to match the picture taken by the user against its screenshot. If a dispatcher finds a match, it routes the request to the launcher.

As we mentioned before, applications register the types of URI schemes they support. The deep posting launcher requires this information to launch an appropriate application for the given URI in the request. Each application may register multiple URI schemes. If a URI scheme can be accepted by multiple applications, the launcher either opens a dialog so the user can choose an application or just launches a previously specified default application.

Screen Matching Algorithms

Once a device receives a Deep Shot request, it takes a screenshot of the entire monitor. It then extracts visual features from the screenshot and the picture taken by the camera, using a computer vision algorithm, Speeded-Up Robust Features (SURF) [4]. SURF is robust against scaling and rotation, and faster and more robust than Scale-Invariant Feature Transform (SIFT), another popular feature extraction algorithm.

We use SURF to detect the key points, which are represented by feature vectors, on the screenshot and on the picture respectively (see Figure 4). We then compute the cosine similarity between each pair of key points and find the nearest neighbor for each point. Finally, with the paired key points, a homography (the perspective transformation between two planes) can be calculated to find the projective plane on the screen image. Thus, the region of the screen that the user sees through the camera can be located.

Content and State Encoding

A migration process of an application consists of transferring not only its content but also its states. With our framework, developers can store arbitrary offline content, e.g., files, as an attachment in a Deep Shot request and encode the application states into a URI. A URI is the key element to resuming a work state in Deep Shot. A URI can be application independent. For instance, "content://contacts/15" opens a contact manager to show the person with the id 15; "geo:*latitude,longitude"* shows the given location in a map application; and "document://chi2011.pdf/3" represents the third page of the file "chi2011.pdf". Furthermore, developers can append the zoom level, the scrolling position, and all necessary information of this document to the URI as needed. We do not limit the length of a URI so arbitrary states of an application can be encoded.

Recently, some application frameworks such as Three20 (http://three20.info) have started to support URL-based navigation in traditional applications. Mobile operating systems such as Android and iOS also support launching applications with standard URIs (e.g., http:, tel: and geo:). Deep Shot allows applications to create their own URIs,

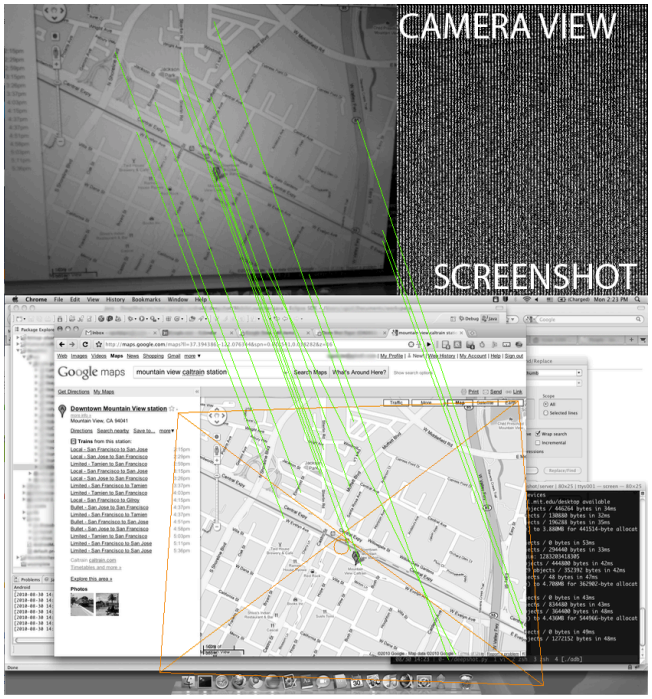

Figure 4. The screen matching algorithms match the picture (at the top) against the screenshot (at the bottom) and find a projective plane (the orange convex) on it.

although it is advisable to be compatible with public standards.

Bootstrapping with a Default Responder

To handle the work state of various applications, the Deep Shot framework, needs to be integrated into those applications by their developers. To deploy such a framework, we need to address how to bootstrap its usage when application developers have not yet adopted the framework. Therefore, we implemented a default responder in the dispatcher to handle the case in which the target application does not support Deep Shot.

If the application that the user is taking a photo of is not registered with the dispatcher, the default responder replies the screenshot of the entire screen to the capturing device as well as the coordinates of the matched region. Therefore, users can acquire a clear version of the screen, i.e., without any noise and distortion caused by the physical camera. They can also zoom in to see more detail and pan to other parts of the screen that were not in the original picture. In addition, as the dispatcher takes the screenshot, it also detects clickable URLs and information of interest such as phone numbers or addresses, using the operating system's accessibility API. These metadata are also transferred along with the screenshot, so the user can tap on a URL or a phone number on the screenshot to launch a browser or dial the number directly.

Supporting Web and Desktop Applications

Deep Shot is a general framework that supports traditional desktop applications as well as web applications. We discuss how Deep Shot supports these two kinds of applications in this section.

Most modern web applications are written in JavaScript and run inside a web browser, while their data are stored on remote servers. To further bootstrap the Deep Shot framework and support this kind of application, we created a *web dispatcher*, implemented as a Google Chrome browser extension, which has three important features.

First, the web dispatcher acts as a second-level dispatcher. It routes messages from the first-level dispatcher to the appropriate web page and sends reply messages back (see Figure 3).

Second, the web dispatcher is a default responder for all web applications that do not support Deep Shot. If the web dispatcher gets a request asking for data from a site that does not register itself with Deep Shot (discussed in a later section), it will only return the URL to that site as a default response. The URL on the address bar often maps to the state of the current web application. However, some Ajax applications do not have this property or hides their real URL on purpose. Fortunately, the last feature of our dispatcher addresses this problem.

Last, the web dispatcher can inject a script into a web application that allows Deep Shot to extract the application's state, without the application knowing about Deep Shot. This is possible because, as a browser extension, the web dispatcher is capable of injecting any content into any web page from the browser.

In addition to web applications, desktop applications could be more difficult to migrate since their developers need to make additional effort to encode the application states into a URI. Fortunately, most mobile versions of a desktop application are simplified and only provide the key features on the mobile devices. This means the developers do not need to encode the complete state of their applications, but can focus on a small set of key states. For example, the key states of a word processor may only consist of the cursor position, the zoom level, and the scrolling position of the document. The other states, such as the view mode and the toolbar's style, could be unimportant because the mobile word processor does not have these adjustable features.

IMPLEMENTATION

We implemented Deep Shot to support both deep shooting and posting. On the mobile side, we chose the Android platform and implemented the system in Java on a Google Nexus One phone. On the other side, we implemented the dispatcher and the launcher in Python on a laptop computer. We also set up a XMPP server using Jabber on a Linux machine. Besides disabling the message size limit in Jabber's default configuration, we did not modify Jabber.

DEVELOPING DEEP SHOT EXTENSIONS

To minimize the effort for developers to incorporate Deep Shot into their applications, we created a Java library that

implements the dispatcher-application protocol and hides the WebSocket connection inside the library.

The library has a DeepShot class that has one public method, `void addListener(Listener listener, String app_name, String[] accepted_uris)`, for applications to register themselves to listen to Deep Shot requests. The `Listener` interface has only one method, `DeepShot.Response onShot(DeepShot.Request req)`, where the `Response` contains a URI and optional file attachments, and the `Request` contains the four corner points and the center point of the ROI. Deep posting also has a similar API, `void post(DeepPost.Request req)` in a class DeepPost, where `DeepPost.Request` contains a URI and optional file attachments.

For web developers, we also provides a JavaScript function, `DeepShot.addListener(listener)`, from a browser extension, so web developers can simply hook their web applications into Deep Shot. For example, Google Maps does not show the URL of the current region of the map in the address bar. To extract the real URL of the current map, we inject the following script:

```
if(window.DeepShot){
  DeepShot.addListener(function(request){
   return {"uri": document.getElementById("link").href};
  });
}
```

With this script, even if Google Maps does not support Deep Shot, users can still use deep shooting to open any computer map on their phone.

SCENARIOS

In this section, we illustrate four typical scenarios that can be accomplished by deep shooting and posting.

Scenario 1: Taking information to go (PC to mobile phone)

This is the classic scenario that motivated us to develop deep shooting. People usually work on desktop computers or laptops at work or home. Before moving to another place, they may look up the information related to that place on their computers. However, as the information may be hard to remember, they often write down the information on a piece of paper or look up the same information again on their mobile phones. In this scenario, people can take the information with them using deep shooting. For example, people could carry a part of a map or the address of the next meeting place so that they do not need to look it up again. People could even capture a YouTube video being played and later resume watching the video from where they left off on a mobile phone.

Scenario 2: Viewing or saving mobile phone content on PCs (mobile phone to PC)

People generate various kinds of lightweight information on mobile phones, such as photos, contacts, or unfinished readings. For lightweight tasks, such as transferring a photo in a phone to a PC so that more people can see it, using existing software tools for syncing up mobile phones with PCs is cumbersome (e.g., a user might need to plug in the cable and find the right folder). With deep posting, users can simply aim their camera at the target computer monitor with the photo still shown on the mobile phone. The photo will be automatically transferred and opened on the target monitor. Deep posting also allows an application to post information at a specific position and size, as seen through the camera, on the target screen, e.g., a Post It application, which cannot be achieved by Bluetooth-based sync tools.

Scenario 3: Using mobile phones as a bridge between PCs (PC to PC via a mobile phone)

USB flash drives are widely used to share files among computers. People are used to saving the information they want to share as files onto a USB drive, and then taking the drive to another computer. In this kind of scenario, deep shooting can be used to extract information from an application (e.g., running on an office PC) and automatically transfer it to a mobile device. The user can then take this mobile device to a home PC and post the extracted information or work states onto it with deep posting.

Scenario 4: Sharing content between mobile phones (mobile phone to mobile phone)

Although it is still rare to share information between multiple personal mobile devices, it is common to share information between mobile phones owned by different people. Researchers have developed techniques to address this need. For example, bumping is a synchronous gesture to connect two mobile devices [9]. Although the current Deep Shot framework does not support communication between multiple users' devices, deep shooting and posting can be used to locate the devices as well as the region of information to share across multiple users. For example, a user could take a picture of a contact displayed on another person's mobile phone with deep shooting, and then the full contact information would be automatically transferred to our phone. This scenario potentially requires a different authentication mechanism though.

USABILITY ANALYSIS AND TECHNICAL EVALUATION

We analyze and evaluate this work from three perspectives. From a user's perspective, we analyze the interaction model and the usability of deep shooting and posting. From a developer's perspective, we analyze the usability and the utility of the API that we provide for developers. Lastly, from a technical perspective, we evaluate the performance of our framework and the feasibility of using a camera to locate a region on a monitor.

Interaction Model and Usability Analysis

Traditional GUI applications on PCs provides an action such as "send this to …" in their menus to let users send local files to remote devices. However, we argue this model is less intuitive than the deep shooting and posting. For the same task of migrating applications across devices, in the traditional model, a user would need to select the source application, the data of interest, and the target device from a list of names or identifiers in multiple steps with a GUI, which can distract the user from the task. In contrast, deep

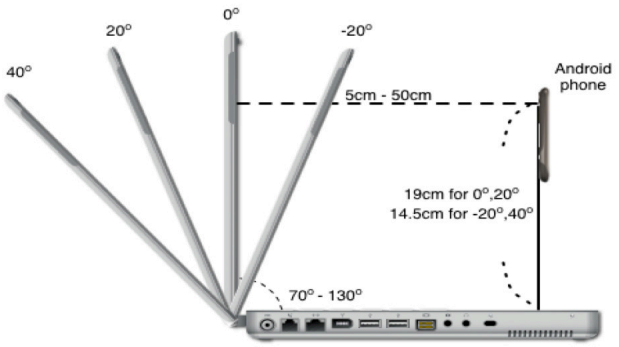

Figure 5. The setup of the reliability experiment.

shooting and deep posting adopts an old technique—taking pictures using a camera—to simultaneously identify the source device, the data to transfer, and the target device, all in one action of taking pictures of computer screens, which is just as easy as doing so of the real world. This is consistent with the informal feedback we collected from users.

Since there is no extra step beyond taking pictures, the learnability and the memorability of our techniques are as good as using ordinary cameras on mobile phones. The efficiency of our techniques is related to two factors. One is the steps performed by the user, and the other is the performance of our system in terms of speed and accuracy. To do a deep shooting or a deep posting, a user needs to perform three steps: launching our application on a phone, locating the target window on a device through the viewfinder, and pressing the shutter. These steps are exactly the same as taking a picture using a camera application on a phone. Therefore, our techniques are as fast as taking a picture from a user's perspective. The other factor, the speed and accuracy of our system, will be discussed later from the technical perspective. Finally, because the steps a user needs to perform are minimized, the type of error that may occur is taking a wrong region on the screen. However, the user can simply discard an incorrect capture and redo the procedure. In addition to user errors, our system may have errors while matching pictures against screenshots. We will examine this kind of error with a controlled experiment in the following sections.

Usability of Our Framework

As a framework, we provide a simple but powerful API that consists of two functions for developers. One is for an application to register itself with our system, and the other is to respond when a capture event occurs. We argue that these two functions are easy to use for developers who are experienced in event-driven programming.

A potential difficulty with incorporating Deep Shot into an application is to identify what components and information are located in the given region of interest. Fortunately, modern operating systems already include this functionality, known as hit testing, in their accessibility

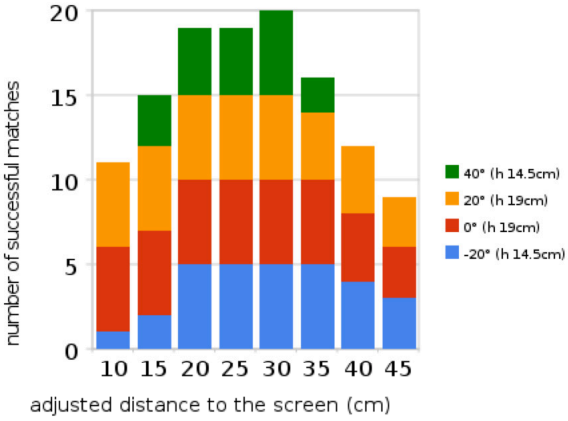

Figure 6. The results of the reliability experiment. The total number of trials for each setting is 20.

APIs. Developers may use these APIs to determine which component is hit given any point on the screen.

Technical Evaluation

Finally, we evaluate our system from a technical perspective. We set up two experiments to explore whether using a camera to locate a region on a monitor is feasible in terms of speed and accuracy. The first experiment was to test the speed of our system, and the second one was to test the accuracy of our image-matching algorithm.

Experiment 1: Speed Performance

We used a laptop, a 15-inch MacBook Pro with a high-resolution 1680×1050 monitor as the target device, and a Nexus One running Android 2.2 as the capturing device that takes 512x384 pictures. The capturing device was held by the experimenter at a distance of 20 to 40 cm and a pitch angle of ±20° so that about ⅓ of the screen could be seen through the viewfinder.

We tested four target applications: photos (from Google Street View), short textual information (from Yelp.com), long textual article with a few images (from CNN.com), and maps (from Google Maps). For each application three photos were taken using deep shooting under the setting described above.

The average time of the whole procedure across 12 trials (3 pictures for 4 applications) was 7.7 seconds (SD 0.3 seconds). By examining the average time of each step, we found the network transmission occupied about 50% of the total time, while the rest processing time was spent on the target device (34%) and the capturing device (16%). The transmission caused a significant portion of the latency because our current implementation attaches raw images captured by the camera in the messages and these messages were routed via an external server. As a result, we can significantly reduce the latency of our system by improving the transmission efficiency, e.g., using only visual features instead of the entire image and not using a third-party server. We will discuss the possibilities in the Future Work section.

Experiment 2: Reliability

In this experiment, we wanted to test the accuracy of our image-matching algorithm under typical conditions for taking a picture of a screen as well as extreme conditions that are not so common. Our experimental setup is shown in Figure 5. We used a 15-inch MacBook Pro laptop so that we could easily adjust and measure the pitch angle of the screen. We tested four pitch angles for the screen with respect to the phone: −20°, 0°, 20°, and 40°. The laptop shows a full-screen browser (Google Chrome) with a web page of the most popular local restaurant on Yelp, which is a typical webpage consisting of text and images. An Android phone, Google Nexus One, was tied to an L-square ruler that is perpendicular to the floor. The height of the camera, which was measured from the surface of the laptop keyboard to the center of the camera lens, was fixed at 19 cm while the pitch angle was 0° or 20°, and 14.5 cm while the pitch angle was −20° or 40°. These height settings allowed the camera to focus around the same target on the screen, namely the name of the restaurant. We set the phone in front of the screen with a distance of 5 to 50 cm, as measured from the screen shaft to the camera lens. Finally, for each pitch angle, we took five pictures for every 5 cm between 5 cm and 50 cm, which resulted in a total of 200 pictures. This experiment was conducted in an office with ceiling fluorescent lights.

We used the algorithm mentioned before to match each picture against the screen displayed on the laptop. If the center of the matched region overlapped the expected region on the screen, it was considered as a successful match.

The results of this experiment are summarized in Figure 6. The chart shows the number of successful matches for each adjusted distance to the screen, instead of the distance measured to the screen shaft. Because the screen was tilted, we adjust the distance to the shaft by adding $h \tan \theta$, where h is the height of the phone and θ is the pitch angle. With this adjustment, we only show the results between 15 cm and 40 cm where all the settings have valid measurements.

The experiment showed that the matching algorithm was highly robust with a 97% success rate, when the camera was parallel to the monitor and the distance between them ranged between 10 and 40 cm. This range is sufficient to cover everything from a small region, such as a restaurant's name, to the entire screen (see Figure 7). Even when the camera was tilted, taking pictures in the range of 20–30 cm was still robust (96.7% success). The accuracy significantly decreased when the camera is too close to (< 10 cm) or too far (> 40 cm) from the screen, but these conditions are uncommon as users can seek the appropriate size of the target through the camera. The results of this experiment showed that using a camera with our algorithm was robust enough to locate a region on a monitor.

RELATED WORK

Several research projects have addressed the issues of migrating information across devices. Remote Clip [15] is a simple way to share information via a synchronized clipboard across multiple personal computers. However, this technique is only feasible for copying textual or selectable objects. Pick-and-drop [18] is a direct-manipulation technique to pick up an object on a computer and drop it on another. Hyperdragging [17] is a technique like drag-and-drop that transfers information across devices. However, these two techniques require special, uncommon devices (pen devices and augmented tabletops) so they cannot be easily deployed to the real world. Some tools [6,21] allow users to control applications remotely. In contrast, Deep Shot allows users to interact with the same content via native applications running on a local device, which eliminates the need to have a constant network connection.

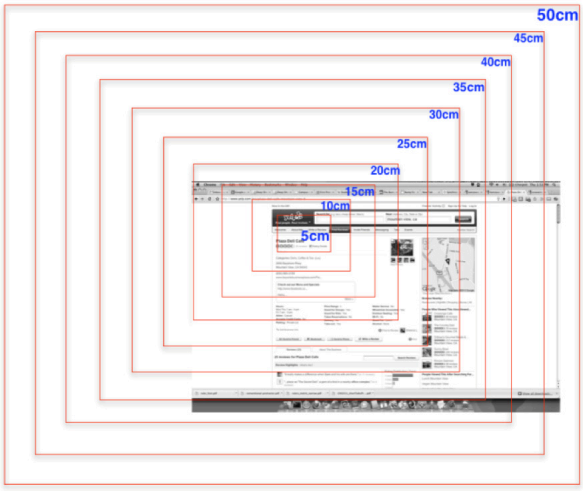

Figure 7. The regions the camera sees with a distance 5 to 50 cm away from the screen while the monitor is parallel to the phone.

Associating physical tags or bar codes to digital files is also a way to migrate information. Want et al. [22] describes using RFID tags to link physical objects to network services. Android phone users can install an application by scanning a QR code. The downside of these techniques is that they require special tags or codes that can only be read by machines. On the other hand, techniques such as [5,13, 14] based on only visual features have been proposed. Compared to these previous techniques, we used similar feature matching algorithms. However, these techniques only focused on file transfer or document manipulation for certain applications. In contrast, Deep Shot provides an extensible framework that enables an arbitrary application to migrate not only its content but also its runtime states across devices using a mobile phone camera.

PIE [16] is a general infrastructure for developers to create multi-personal-device services, which focuses on low-level mechanisms for sending information, commands, or events across devices. Device ensembles [19] also provided a viewpoint on how multiple devices communicate from the low-level link layer to the high-level application layer. In contrast with these projects, our framework provides a

high-level and unified way for developers to extend their applications and for users to invoke the migration using a mobile camera.

Recently, researchers have proposed Activity-Based Computing (ABC) [2,3], which considers a user activity as the basic computational unit. From the broad perspective of ABC [12], an activity can span across different users, devices, applications, and situations. Deep Shot deals with three major challenges in ABC: activity suspend and resume, activity roaming, and activity adaption, but we only focus on a subset of activities: single user, single application, and multiple devices.

CONCLUSION AND FUTURE WORK

We conclude by discussing the limitation of Deep Shot and possible extensions for future work.

Multiple users: Our current system finds possible target devices from a list of the user's online devices. It is easy to add other users' devices into the user's "friend list," so that they can be notified when a capture event happens. However, this would add extra effort of managing the device list. A possible solution is to replace the XMPP layer with a local service discovery protocol, such as Apple Bonjour, and broadcast the request to local devices.

Transmitting visual features instead of pictures: In the current implementation, we send pictures directly in a request, which raise privacy concerns since the devices that receive the request can "see" the pictures, especially for a multiple-user environment. Therefore, a possible solution is to extract the visual features directly on the capturing device and only send the feature vectors in a request. This could dramatically speed up the performance and also prevent malicious request sniffers. In addition, this could enable *real-time matching feedback* on the target screen, so users can be confident that the matching is successful and also know which region of the screen will be captured.

Limitation on feature matching: Feature matching may not work in some scenarios. For example, nothing can be extracted and matched if a user intends to capture a blank region. However, we can assume that no valuable information exist in this area and simply show the photo she took back to her. A more common problem is unfocused photos, although this could be solved with the real-time matching feedback we mentioned above.

This paper presented two novel interaction techniques, deep shooting and deep posting, to migrate a task across devices and a robust and extensible framework to support them called Deep Shot. We demonstrated that Deep Shot is reliable and feasible to support a range of everyday tasks migrating across devices using one simple gesture.

ACKNOWLEDGEMENT

We thank the anonymous reviewers, Rob Miller, and James Lin for their suggestions and feedback, and LaDawn Jentzsch for narrating our demo video.

REFERENCES

1. Bandelloni, R. and Paternŋ, F. Migratory User Interfaces Able To Adapt To Various Interaction Platforms. *International journal of human-computer studies,* 2004.
2. Bardram, JE. Activity-Based Computing: Support For Mobility And Collaboration In Ubiquitous Computing. *Personal and Ubiquitous Computing,* 2005.
3. Bardram, JE., Bunde-Pedersen, J., and Soegaard, M. Support For Activity-Based Computing In A Personal Computing Operating System. *Proc. CHI '06.*
4. Bay, H., Tuytelaars, T., and Van Gool, L. SURF: Speeded Up Robust Features. *Proc. CVIU,* 2008.
5. Boring, S., Altendorfer, M., Broll, G., and et al. Shoot & Copy: Phonecam-Based Information Transfer From Public Displays Onto Mobile Phones. *Proc. Mobility '07.*
6. Boring, S., Baur, D., Butz, A., and et al. Touch Projector: Mobile Interaction through Video. *Proc. CHI* '10.
7. Dearman, D and Pierce, JS. "It's On My Other Computer!": Computing With Multiple Devices. *Proc. CHI '08.*
8. Google Chrome to Phone. http://code.google.com/p/chrometophone.
9. Hinckley, K. Synchronous Gestures For Multiple Persons And Computers. *Proc. UIST '03*.
10. Hupp, D. and Miller, RC. Smart Bookmarks: Automatic Retroactive Macro Recording On The Web. *Proc. UIST '07*.
11. Karlson, A., Iqbal, S., Meyers, and Tang, J. Mobile Taskflow in Context: A Screenshot Study of Smartphone Usage. *Proc. CHI '10*.
12. Li, Y., and Landay, J. Activity-based prototyping of ubicomp applications for long-lived, everyday human activities, *Proc. CHI'08.*
13. Liao, C., Liu, Q., and et al. Pacer: Fine-grained Interactive Paper via Camera-touch Hybrid Gestures on a Cell Phone. *Proc. CHI '10*.
14. Liu, Q., McEvoy, P., and Lai, C.-J. Mobile camera supported document redirection. *Proc. MM '06*.
15. Miller, RC and Myers, BA. Synchronizing Clipboards Of Multiple Computers. *Proc. UIST '99.*
16. Pierce, J. and Nichols, J. An Infrastructure For Extending Applications' User Experiences Across Multiple Personal Devices. *Proc. UIST '08*.
17. Rekimoto, J. Pick-And-Drop: A Direct Manipulation Technique For Multiple Computer Environments. *Proc. UIST '97.*
18. Rekimoto, J and Saitoh, M. Augmented Surfaces: A Spatially Continuous Work Space For Hybrid Computing Environments. *Proc. CHI '99.*
19. Schilit, B and Sengupta, U. Device Ensembles. Computer, IEEE Computer Socity, 2004.
20. Tang, J., Lin, J., and et al. Recent Shortcuts: Using Recent Interactions To Support Shared Activities. *Proc. CHI '07*.
21. Tan, DS., Meyers, B., and Czerwinski, M. WinCuts: manipulating arbitrary window regions for more effective use of screen space. *Proc. CHI '04*.
22. Want, R, Fishkin, K, and Gujar, A. Bridging Physical And Virtual Worlds With Electronic Tags. *Proc. CHI '99.*