Visual Assistive Device Design and Prototyping in Preparation for User Studies

6.UAP Project Report

Peter Iannucci, MIT Department of EECS

Friday, May 20, 2011

1. Introduction

1.1. Existing Problem

Vision is a tremendously rich sensory input. Human adults allocate 140 million neurons to perform the staggering amounts of computation necessary to reduce the visual stimulus down to the salient details called for by the current situation[1]. For individuals who do not have access to vision as a primary means of spatial exploration and feature extraction, navigating unfamiliar environments or avoiding environmental hazards are difficult tasks, and may require the individual to obtain assistance from a friend, companion animal, or passerby.

In order to promote increased independence, participation, and quality of life for a large class of users with perceptual or cognitive disabilities, a group of principal investigators (Teller, Glass, Miller, Roy, and Torralba) led by Professor Seth Teller from the Robotics, Vision, and Sensor Networks (RVSN) group at CSAIL have proposed an expansive five-year program of fundamental research in the adaptive perception, inference, learning, and interaction planning methods required to realize assistive devices that can

[1] Leuba, G., and Kraftsik, R. Changes in volume, surface estimate, three-dimensional shape and total number of neurons of the human primary visual cortex from midgestation until old age. *Anatomy and Embryology 190* (1994), 351–366. 10.1007/BF00187293.

substitute for a human assistant. One instantiation of the techniques they plan to develop will be a wearable device equipped with a rich sensor suite for environmental awareness and one or several channels of communication with a human user with perceptual or cognitive disabilities, for instance via screen, Braille display, keyboard, speech, and haptics. The parameters and uses of the system are being determined through ongoing and past dialogue with residents of The Boston Home in Dorchester, MA, a specialized care facility for people with progressive neurological diseases, as well as interaction with a blind member of the project staff and other domain experts in accessible technology.

Based on these discussions, one focus of the proposed algorithmic development is toward assisting a user in navigation tasks like avoiding hazards, traveling to a destination, getting one's bearings when dropped off near a familiar location, and navigating stores, hotels, airports, and other public spaces. The proposal also considers identifying, locating, and providing information about objects, recognizing people, providing awareness of the "lay of the land" in unfamiliar places, and providing running commentary on nearby objects. Basic problems include automatically recognizing and understanding the user's environment, interpreting the user's intentions, entering into deliberate and helpful dialog with the user, and selecting relevant information to convey to the user out of a flood of input.

1.2. Subproblem

For several reasons, and while the process of procuring funding is ongoing, a minimally functional system prototype is the right first step toward carrying out the proposal. Our reasoning follows, along with precisely what we mean by a minimally functional prototype.

First, it will be valuable to learn as much as possible about how the types of interaction so far considered between the user and the assistive device will contribute to the user's ability to navigate and participate in the world. We expect that if we had a working system online today, then seeing how users interacted with it and which functions were more helpful than others would reveal e.g. what sensors, algorithms, and human interface the device will ultimately need to possess; in what areas to focus our efforts; what level of cognition the user expects from the device in responding to queries; and how to structure the human-machine interaction for maximum utility.

Second, the construction of a prototype allows us to start making architectural decisions and evaluating their compatibility with the interaction modalities that turn out to be most useful. For example, we might find that a speech-centric interaction introduces so much latency as to preclude timely guidance, or we might find that users have difficulty reading a Braille display located on their torso, or reading one while on the move. Separately, we might confirm our expectation that transmitting sensor data over Wi-Fi and doing all the video processing on a powerful, non-mobile server (e.g. in a dedicated server facility) provides adequate reliability, performance, and flexibility for prototyping, or we might find that unreliable Wi-Fi hand-offs preclude useful testing under conditions of mobility. Any decisions we make at this point must permit us to remain flexible in order to avoid duplication of effort. A major goal of this work is therefore to develop a system that is modular and extensible, enabling (as far as possible) seamless integration with existing motion planning and environment visualization code within the RVSN group, and keeping system components decoupled to support experimentation.

Lastly, a prototype will allow us to start collecting and storing sensor data for later playback and analysis. We expect that many of the machine learning problems involved in scene understanding and object recognition will require extensive real-world training and evaluation data, and we are eager to start collecting this data to accelerate work on these parallel projects.

1.3. Limitations on Prototype Functionality

Because we wish to evaluate a design which cannot be realized in its entirety today, we are forced to make certain compromises. For instance, it is not currently practical to build all the computation into the worn device, which is why have limited ourselves to designs which transmit sensor data (perhaps eventually in some digested, rather than raw, form) from the worn device to an off-body server. More significantly, the assistive or interactive functions of the device, considered independently, cannot depend on any existing software algorithm for reliable object recognition across the thousands of classes and countless unique instances that make up the real world. For our prototype, therefore, each component that does not yet exist will be isolated from the rest of the system by a stable communication API. A software module should eventually expose the proper functionality through this interface. In the mean time, we can recruit a human

“Wizard of Oz” to do the dirty work of categorizing objects, for instance. By keeping this human hidden from the ultimate user of the system, and supplying the human with the same sensor data we would make available to an algorithm, we hope to permit a degree of realism in our trials. For the person using the system, “realism” means that the system is not a surrogate for a distant human assistant (though such a system might be helpful). The wizard should not accept arbitrarily complex spoken requests, and the scope of the wizard’s verbal feedback to the user should be well specified. For the wizard, realism means that the system cannot be made to perform beyond the bounds of its sensors. For this reason, the wizard must be spatially isolated from the user, so that the only useful input s/he receives is through the communication API. Once we are able to observe how the user interacts with the system, we will be better able to appraise what the human “wizard” is doing that is helpful, and to use this appraisal to inform our interaction planning and algorithmic investigations.

1.4. Setting Objectives

Setting objectives for our project was an iterative process. We initially settled on an outline of what sorts of on-line trials we wanted to enable, based on which sensors we felt we could deploy on the prototype. In particular, our choice of the Microsoft Kinect depth camera limits us for the most part to indoor operation, with portable scanning LIDAR a more expensive alternative better suited to the outdoors. An ideal depth sensor would provide both color and spatial information over a wide angle of view, with dynamic range large enough to detect both objects held in the user’s hand (anything the user touches is worthy of special attention) and cars down the street (or other potential hazards, with as much lead time as possible), and with enough precision to detect cracks on the sidewalk and other tripping hazards; it would be light and small enough to be comfortably worn on the person, and efficient enough to run off of a small battery for hours; and it would collect data quickly enough to avoid blurring of fast-moving objects. While most humans are born with a set of these ideal sensors built right into our faces, the electronics aisle at the hardware store is a bit more limiting, and commercial units can offer at best a few of these features. We chose the Kinect for its resolution, acceptable dynamic range, portable drivers (under the libfreenect project), light weight, and low cost. LIDAR sensors tend to have a larger "dead zone" around the sensor where depth information is not available, and even the Kinect in its default configuration has difficulty with objects nearer than about three feet, which limits it to providing color information for held objects. While we are led to believe that this minimum distance can be altered, investigation in that direction was outside the scope of our work.

Other sensors we hoped to deploy included an inertial measurement unit to track the motion of the depth camera as the user moves, enabling accurate reconstruction of the three-dimensional surroundings, and one or more microphones to capture the user's voice and the environment. Eventually, the system should provide spoken feedback only when it would be conversationally polite to do so (or at least when no one is talking), and it should recognize commands without requiring the user to first press a button or otherwise "key" the microphone. We considered mounting stand-alone speakers and microphones to a worn vest with the other components, but decided that an Android smartphone would provide a more integrated solution.

Based on the available sensors, we envisaged trials focused on indoor blind or blind-folded navigation and object identification. We set out to design and build a "wizard of Oz" prototype with as many human wizards as necessary behind the scenes to make it work. We would consider ourselves successful if we delivered a trial-ready system and we could train other workers to use it. Our other parameters were to maintain compatibility with existing communication, motion planning, and data visualization code developed by the RVSN group, to produce a system that was extensible and modular to support future experimentation, and to incorporate the capability to record and play back sensor inputs and system state.

1.5. Participants

This joint project between Yafim Landa and myself was carried out under the supervision of Professors Seth Teller and Rob Miller, with whom we met for regular status reports. We shared our hardware with graduate student Jon Brookshire, who developed it for a portable mapping system with overlapping goals.   Dr. Albert Huang supported us in our use of the Lightweight Communications and Marshalling library software, as well as the Kinect driver. Yafim's and my specific contributions are detailed in the section on implementation. We were both heavily involved in the minutiae of the design phase.

2. Design

2.1. System Decomposition

The first stage of our project was to design the prototype. In order to meet our objectives of modularity, extensibility, and compatibility with existing code, while still permitting us to invoke GUI and visualization toolkits as needed in whichever language was most convenient, we decided that it would be best to decouple the system into a web of independent modules (typically processes) separated by network links. These modules could run in the same process or even in the same thread if so desired, so long as the communication framework did not impose this as a requirement. Furthermore, they could run on one machine or several as needed, offering a smooth path toward future scalability in processing power, yet ultimately permitting the entire system to be run unmodified on a single worn device once that becomes feasible. Network-based decoupling also permits wireless tablets like iPads or Android devices to be incorporated into the system as wizard interfaces or diagnostic aids, and simplifies the process of swapping out human- and computer-based processing stages, since the human wizard is not limited to interacting with one particular machine. This last point became more significant as the design grew from the original proposal (as represented in Fig. 1) to include more than one wizard.

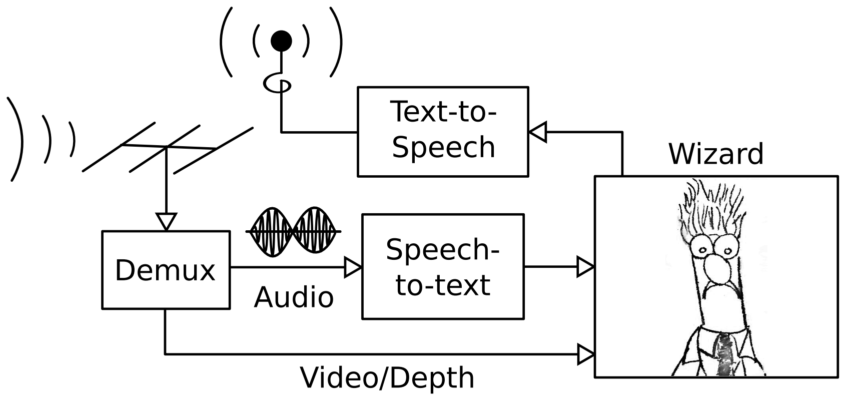


Fig. 1. Block diagram of one-wizard system, with the wizard handling user requests, processing the video feed, and providing guidance.

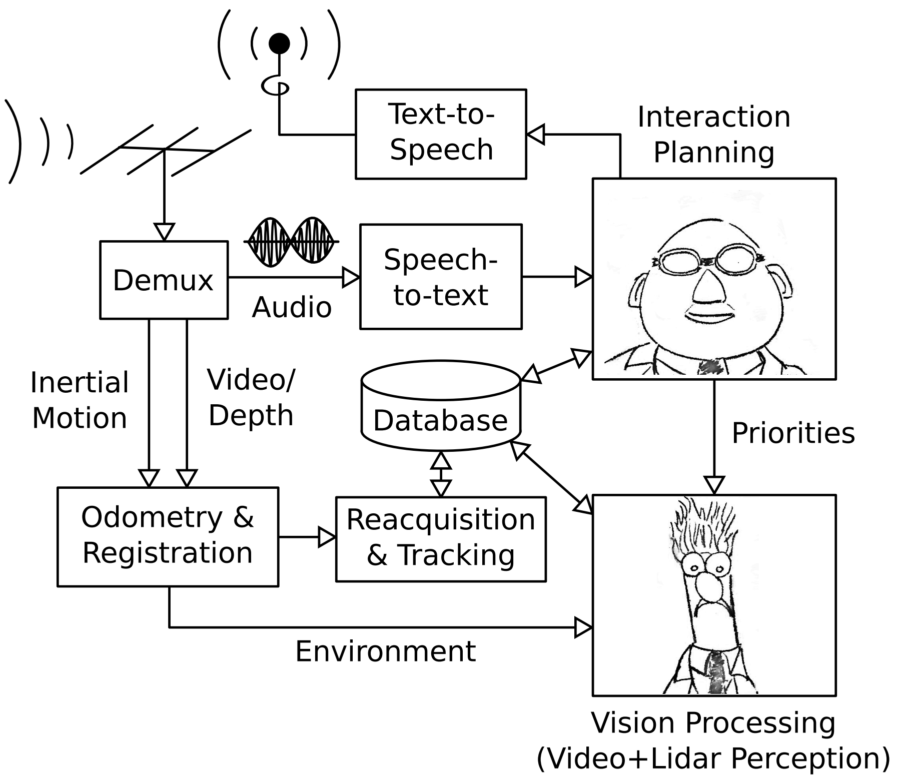


Fig. 2. Block diagram of two-wizard system.

After considering the various typing, speaking, gesturing, and planning responsibilities of the human wizard in the original grant proposal, we felt that the tasks assigned to this individual were potentially overwhelming. As an alternative, we proposed that the tasks of the wizard should be broken up between two people: a “planner” responsible for interacting with the user and planning how to accomplish tasks set by the user, and a “lookout” responsible for identifying nearby objects based on a priority list of what the planner currently considered interesting (Fig. 2). The planner sees only a schematic view of the space around the user, populated with objects tagged by the lookout, while the lookout sees the full camera feed, superimposed with objects that he had already tagged, plus the list of priorities set by the planner. When the user asks a question, the audio is transmitted to the planner after passing through automated voice transcription software. The planner decides what sort of objects need to be identified to answer the question, and provides this information to the lookout. The lookout quickly circles relevant objects in the camera's field of view and tags them with some information about what they are. All tagged objects are stored in a database, and motion data from the inertial measurement unit and the camera are merged to track objects and build up a persistent model of the world. When the planner has accomplished the task or gathered the right information, they can key in appropriate feedback to the user, which will then be passed through a text-to-speech system. Based on an optimistic assessment of system complexity, we expected to be able to produce skeleton implementations of most of these modules, leaving a more advanced treatment of each to future work.

We retained the more conservative one-wizard design to serve as a milestone (Fig. 1). In this design, besides automatic logging of all network traffic, there is no persistence. The wizard is provided with a machine transcription of requests from the user, and possibly also a copy of the raw audio (to make up for deficiencies in the machine transcription). Feedback is keyed in as before and fed through a text-to-speech system.

2.2. Communication

Inter-module communication, both on one machine and between machines, is achieved using the Lightweight Communications and Marshalling library (LCM). Developed for the MIT DARPA Urban Challenge Team, LCM is portable across languages and operating systems and conveniently based on a publish-subscribe model. LCM is responsible for data serialization, deserialization, packetization and validation. It permits messages much larger than the Ethernet MTU, though it makes no delivery guarantees when using UDP as the underlying transport (which it does by default). It supports rich, nested static data types, and can automatically generate C, Java, and Python code for interacting with these structures[2]. The LCM library is bundled with logging and replay tools suitable for our purposes.

While LCM ordinarily runs over UDP multicast, we had difficulty getting Android to interoperate without setting up a TCP tunnel. Making the switch brought Android on board and gave us better delivery guarantees, but it also interposes machinery with internal state and timeouts between us and the network, potentially affecting our ability to recover quickly from outages and Wi-Fi handoffs. This gave us cause for concern, and we planned to spend some time evaluating this decision once the system was fully implemented.

3. Implementation

3.1. Final Block Diagram

The system as actually prototyped is shown in Fig. 3. Note that functionality is spread across at least five different machines: a laptop worn by the user, an Android phone worn by the user, a server in the Google cloud that does audio transcription, a “discovery” server that publishes the IP address of the LCM TCP service at a fixed URL,

[2] HUANG, A., OLSON, E., AND MOORE, D. LCM: Lightweight communications and marshalling. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (October 2010).

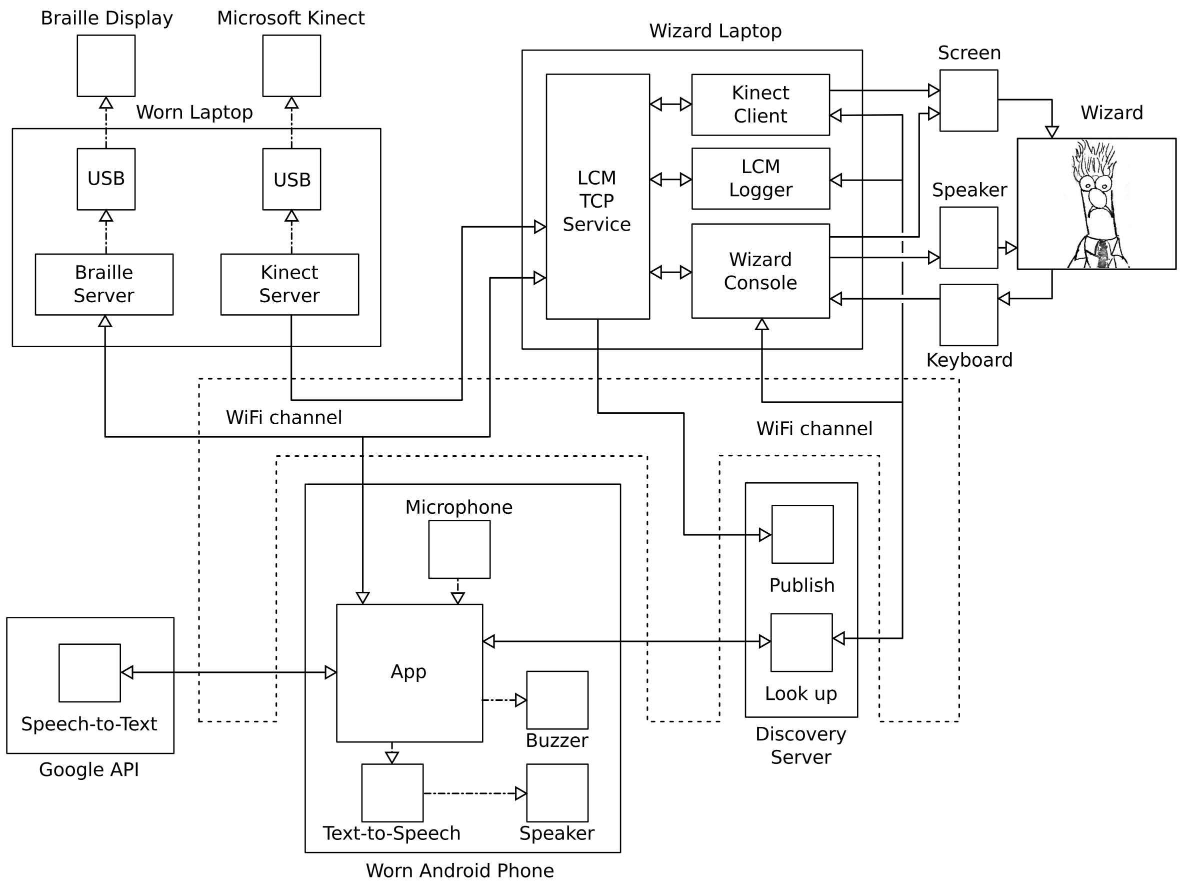


Fig. 3. Block diagram of one-wizard system as implemented.

and the wizard's laptop. More details will follow in the section on hardware. The significant increase in complexity over the design of figure 1 reflects realities of implementation. One addition is support for a portable Braille display, with the Android phone providing haptic feedback via its buzzer in order to draw the user's attention to non-spoken messages as they appear on the Braille display.

3.2. LCM Channels

Our prototype uses two LCM channels: one for Kinect data, and one for interaction with the user. The Kinect datagrams are identical to those defined by RVSN's kinect driver, and the server code is modified trivially to permit LCM to be redirected over TCP. Further changes to the pre-existing Kinect client are discussed below.

The second LCM channel, called `TRANSCRIPT`, carries messages of two types: `audio_msg_t` and `transcript_msg_t` (Listing 1). Each type begins with a header indicating the source and destination of the message (since transmissions are bidirectional), and contains a message body with either compressed (3gp) audio or plain text. The timestamp in the message header is set at the transmitter, and served as a useful metric of latency during our evaluation.

```
struct header_t
{
    int64_t  timestamp;

    int8_t   source;
    int8_t   destination;
    const int8_t SOURCE_WIZARD = 0;
    const int8_t SOURCE_USER_SPEECH = 1;
    const int8_t SOURCE_USER_TYPED = 2;

    const int8_t DESTINATION_WORN_SPEAKER = 0;
    const int8_t DESTINATION_BRAILLE_DISPLAY = 1;
    const int8_t DESTINATION_WIZARD = 2;
}

struct transcript_msg_t
{
    header_t header;
    int32_t  transcript_data_nbytes;
    byte     transcript_data[transcript_data_nbytes];
    double   srec_confidence;
}

struct audio_msg_t
{
    header_t header;
    int32_t  audio_data_nbytes;
    byte     audio_data[audio_data_nbytes];
}
```

Listing 1. LCM data type specification for the `TRANSCRIPT` *channel.*

Receivers subscribed to the `TRANSCRIPT` channel used the `source` and `destination` fields for selectivity. For instance, the Android phone played back messages destined to `WORN_SPEAKER`, and responded to `BRAILLE_DISPLAY` messages by printing them on its screen and briefly activating the phone's buzzer. Messages from the wizard do not make use of the `srec_confidence` metric, which is populated when an audio transcript is retrieved from Google. The worn laptop listens to the `TRANSCRIPT` channel to pick out messages destined to `BRAILLE_DISPLAY`.

3.3. Hardware

Jon's vest prototype is shown in Figure 4 with the addition of the Braille display and the carried laptop. We envision the Braille display being mounted to a belt clip with an adjustable swivel, and being oriented so that if the user places the palm of one hand flat on their hip, the display surface is comfortably perpendicular to the vertical swing of the elbow. The vest has an existing belt loop of nylon webbing material which may be sufficient for this purpose, but to avoid accidents with the expensive Braille display when donning or removing the vest, we believe that a separate belt may be necessary. The carried laptop might be mounted on the user's back or placed in a backpack, but we did not approach this problem or find a solution to the thermal issues accompanying the backpack proposal. Since battery life is an issue for all the worn equipment, we hope that the laptop's heat production can be kept manageable. Measurements of power consumption due to Kinect data streaming will be presented in the evaluation section.

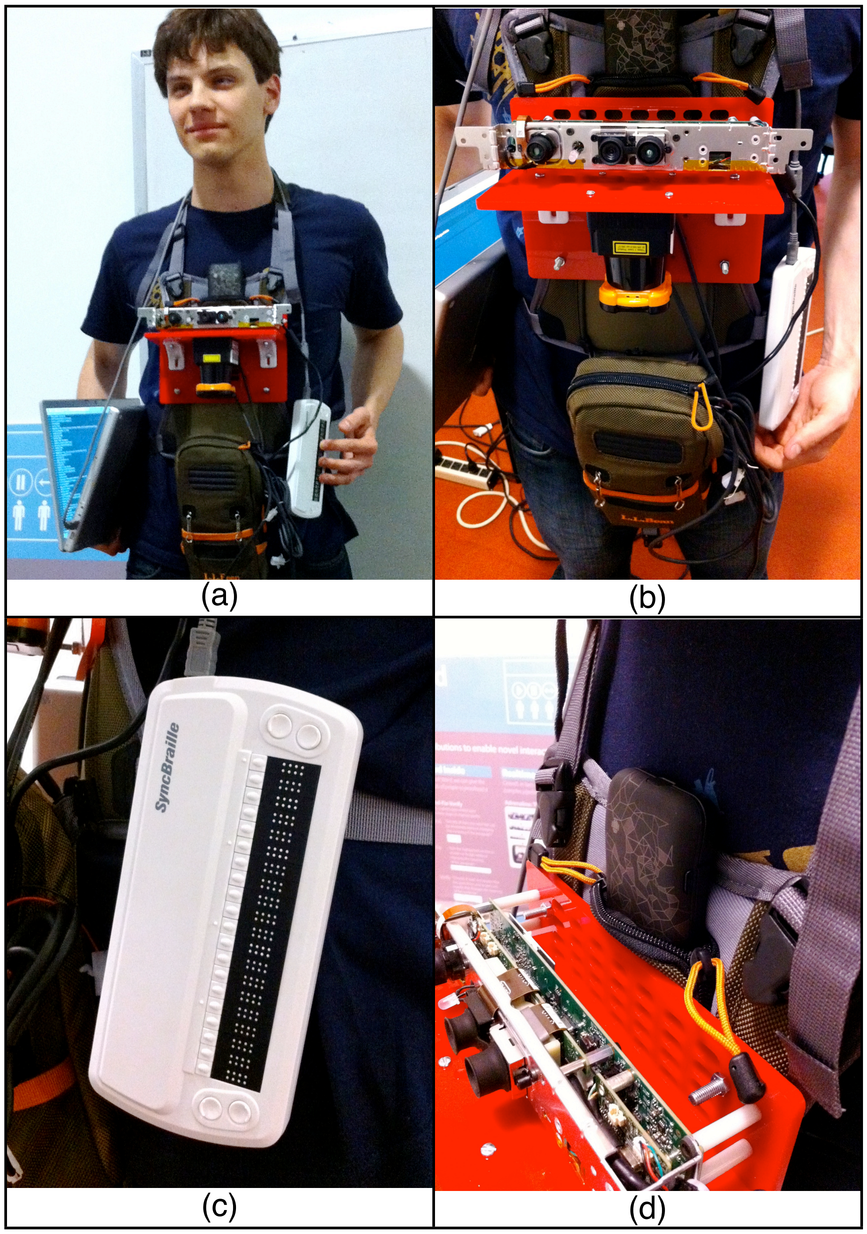


Fig. 4. Vest prototype detail showing (a) positions of chest sensor platform, battery pack, Braille display, and carried laptop; (b) close-up of sensor platform showing Kinect on top and LIDAR beneath; (c) SyncBraille display unit; and (d) central location of Android phone on chest.

Our experience with the prototype vest suggests that a front-heavy unit supported by the combination of a padded neck strap and a belt loop may result in user discomfort. The central position of the battery pack in front of the groin area may also result in discomfort, depending somewhat on the adjustment of the neck strap. The main weight seems to be the battery. A light, minimal power supply, even at some expense, would seem to be a good investment before embarking on extended user trials. We also found that depending on the wearer's posture and the adjustment of the neck straps, the Kinect sensor may end up positioned high enough on the chest that significant navigational hazards at e.g. chair level fall entirely below its field of view.

The $2000 SyncBraille unit displays 20 two-by-four cells at a time, and has affordances for scrolling through a larger body of text or "clicking on" one of the twenty characters. It measures 18.3 x 8.5 x 2 cm and 280 grams. Since it receives both power and control over USB, the SyncBraille unit does not require a separate battery. The BRLTTY project provides suitable Linux drivers and a user-mode API – complete with Python bindings – that provides access to the display and input functions of the unit. Determining the most comfortable placement of the Braille display should be straightforward with the assistance of a Braille-literate individual.

We borrowed an HTC Dream "G1" Android smartphone from Prof. Rob Miller's UID group. It measures 11.8 x 5.6 x 1.7 cm and 158 grams, features a 528 MHz ARM11 processor, and runs Android 1.5. We did not load a SIM card, but used the phone exclusively on Wi-Fi. The phone also has a digital compass, accelerometer, and GPS, making it (potentially) a surrogate IMU. We did not exploit the latter functionality. The phone's LCD screen displays a copy of each Braille message for the benefit of Braille-illiterate users. We positioned it on the vest with the display hidden and only the camera button, speaker, and microphone accessible. The camera button keys the microphone.

Prototype development took place primarily on Yafim's and my personal MacBook Pro laptops. The process of installing software dependencies was sufficiently arduous that we were forced to limit our attention to as few machines as possible; however, Jon has procured hardware suitable for use as either the carried (worn) laptop or the wizard laptop.

3.4. Android application

Our Android application was written by Yafim. Intended to provide access to text-to-speech, speech-to-text, compass, accelerometer, and GPS functionality, development focused on the audio, haptic, and text features. On startup, the phone queries our service discovery mechanism (more on that later) to determine the IP address of the LCM TCP hub. We were unable to get LCM over UDP multicast working with the Android phone, but given the issues we also had with our OS X machines, this may not be an issue with Android. After connecting to the LCM TCP service, the phone listens for messages on the `TRANSCRIPT` channel, prints them onscreen, and speaks them aloud if they have the appropriate header. If a new message is received before the last text-to-speech operation has completed, the phone ends the old operation and starts the new one immediately. This is the correct fall-back behavior in case the wizard needs to inform the user of a hazard, but the Android API also supports queueing a message for playback after the existing message has completed. We envision an easy-to-toggle mechanism at the wizard end which selects between these two behaviors, with emergency traffic (see below regarding canned responses and shortcut keys) automatically having priority. If a message goes out to the Braille display, the phone activates its buzzer to let the user know that fresh information is available.

The Android app is also responsible for recording the user's speech. Ideally, the phone would be continuously recording and running a speech detector on the audio. Code exists within the RVSN group for some of this functionality. For our prototype, the phone begins recording when the user presses the camera button on the side of the phone, and ends recording with a second press. This introduces several delays. First, the recording will clip the beginning of the user's utterance unless s/he pauses briefly between pressing the button and speaking. This would not be an issue if we were recording continuously, since we could keep past audio in a circular buffer and thereby "anticipate" a button push or a positive speech detection. The second source of delay comes from the fact that the compressed audio file provided by the recording API arrives only at the end of the utterance. At that point, the phone transmits the file over LCM to the wizard, who can play it back. Ideally, the audio would be streamed to the wizard as it was being recorded; as it stands, if the users speaks for ten seconds, the wizard cannot even begin to contemplate a response until they finish. Our estimate of the latency of this system follows in the section on evaluation.

Once the phone has transmitted an audio file over LCM, it optionally solicits speech-to-text transcription via a Google API. The transcript and its confidence level are transmitted over LCM to the wizard conditionally on the confidence exceeding some preset threshold. The phone can buzz to indicate to the user whether a transcription was successful. Our experience indicates that the phone's built-in speech-to-text functionality vastly outperforms the Google API. Unfortunately, Android 1.5 does not support simultaneous audio recording and speech transcription, and we decided that recording audio to the log was more important than receiving a reliable text transcript.

The Android app was written for API level 3 (Cupcake). Upgrading the operating system (and the hardware) would allow us to take advantage of simultaneous recording and transcription support built into Honeycomb. We expect it will be straightforward to add functionality that transmits IMU data (compass, accelerometer, GPS) over LCM.

3.5. Wizard application

The wizard application was primarily written by myself. It uses Python for flexibility, and runs at the command line. On launching the script, an interactive prompt is printed, along with a guide to the available commands. A help function makes the application self-describing. By using features from `libreadline` and the utility `rlwrap`, the interactive prompt is made to sit beneath a scrolling view of all the text and audio messages that have passed over the `TRANSCRIPT` channel during the interaction session. See Fig. 5 and 6 for screenshots of the Kinect video feed receiver and the wizard application, respectively.

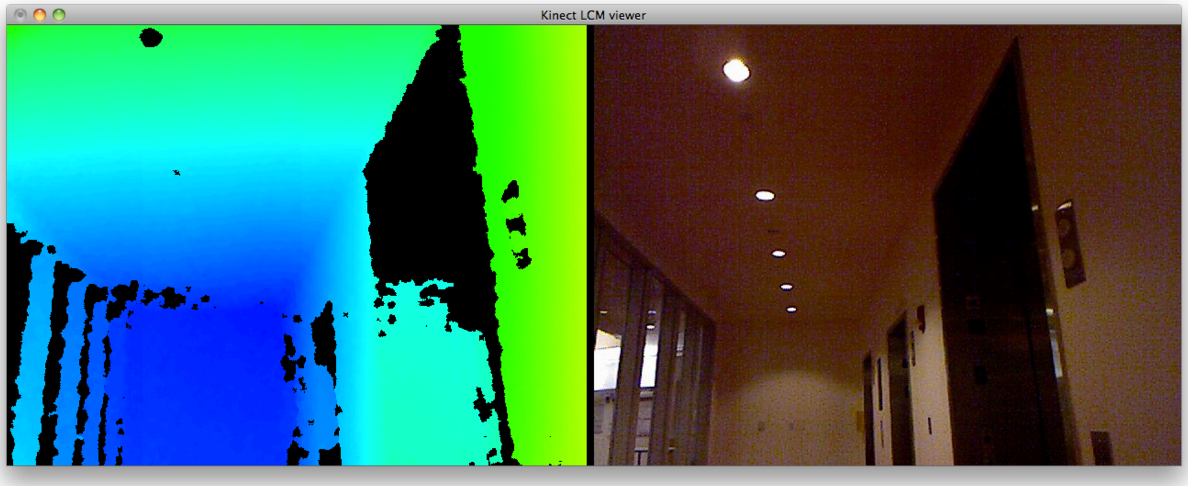
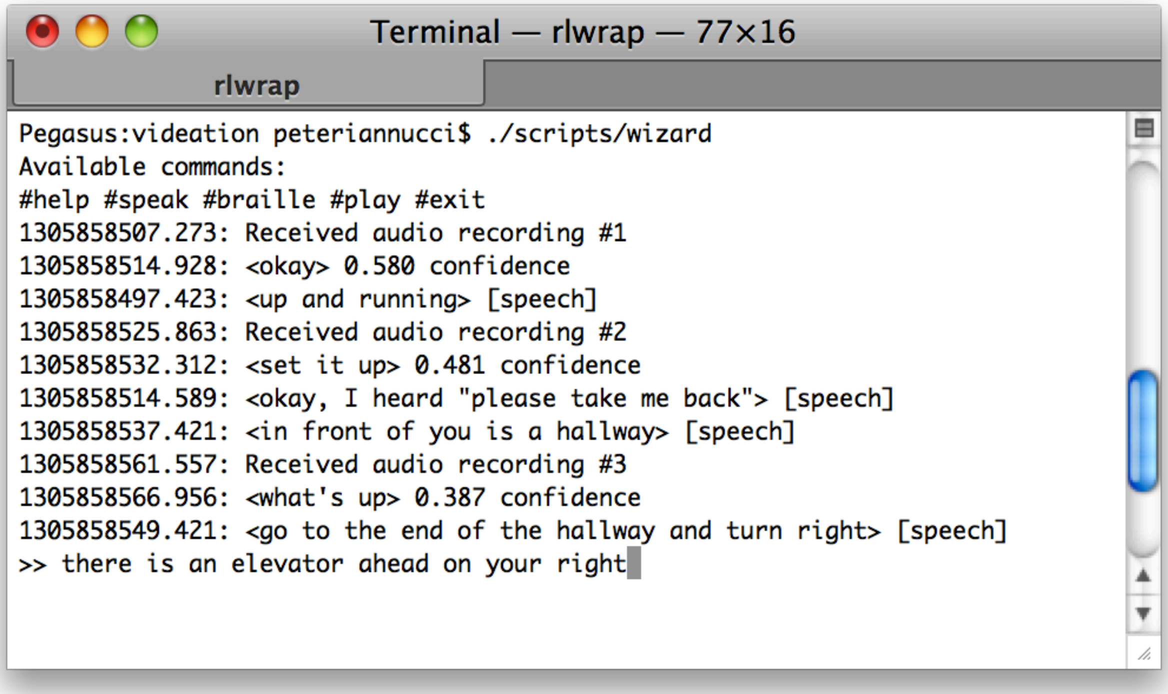


Fig. 5. Screenshot of the Kinect video feed. The upward angle of view will require mechanical adjustment.



```
Pegasus:videation peteriannucci$ ./scripts/wizard
Available commands:
#help #speak #braille #play #exit
1305858507.273: Received audio recording #1
1305858514.928: <okay> 0.580 confidence
1305858497.423: <up and running> [speech]
1305858525.863: Received audio recording #2
1305858532.312: <set it up> 0.481 confidence
1305858514.589: <okay, I heard "please take me back"> [speech]
1305858537.421: <in front of you is a hallway> [speech]
1305858561.557: Received audio recording #3
1305858566.956: <what's up> 0.387 confidence
1305858549.421: <go to the end of the hallway and turn right> [speech]
>> there is an elevator ahead on your right
```

Fig. 6. Screenshot of the wizard command-line interface. Second and third audio transcripts are incorrect (note low confidence scores).

When the human wizard types a message and presses enter, the wizard application encodes the message to be played out loud by the Android phone. By introducing a switch “#b” or simply “b” at the beginning of the line, the human wizard can specify that a message should be sent to the Braille display instead. For faster response to hazards, the wizard application supports shortcut keys with canned responses taken from a configuration text file. For instance, pressing 1 and enter causes the phone to immediately say “stop”.

The wizard application continually listens for messages on the `TRANSCRIPT` channel and prints them out above the interactive prompt. When the application receives a compressed .3gp audio file from the phone, it invokes the `ffmpeg` utility in the background to obtain decompressed audio samples, and plays them back for the user through the `pyaudio` module. Each recording is assigned a sequential number, and these numbers are printed along with the notification that an audio message has arrived. If the human wizard wants to listen to a recording again, s/he can type “#play” to hear the last recording or e.g. “#play 3” to hear a specific recording. In the LCM log file (details follow), the audio is stored in 3gp format. When the phone succeeds in obtaining a text transcript (from the Google API or otherwise), the wizard application prints it onscreen.

We realize that the command line interface is potentially cumbersome and unfamiliar, and in the interest of making the human wizard’s job as easy as possible, we expect that the tool will eventually need to be migrated to a GUI. One step I explored in this direction was to port the `kinect-glview` stream receiver application developed by RVSN from C to Python/wxPython/OpenGL, allowing the Kinect camera feed to be integrated into the UI of a Python application.

3.6. Infrastructure

In addition to the wizard application, I implemented a simple service discovery mechanism to communicate the LCM TCP service's IP address to each module, as well as a number of utility scripts to simplify the operation of the system. The LCM TCP service lives in the Java LCM module at lcm.lcm.TCPService, and can be invoked from the command line with a reference to the appropriate .jar file. When launched, it opens a socket on port 7700 and listens for clients attempting to connect. Before launching the JVM, my `lcm-server` wrapper script checks to see if a server is already running on some other machine, and looks up the IP address of whichever network interface can route to MIT (18.0.0.0). It then issues an HTTP `GET` request to the discovery service to upload IP address, and sets a trigger to upload a null address (0.0.0.0) whenever the JVM terminates.

The discovery service consists of a PHP script at a fixed URL <http://iannucci.scripts.mit.edu/up.php?ip=x.x.x.x>. The script opens a file called "down.html" in the same directory and writes the IP address passed in the URL into the file. While it would be possible for the discovery service to directly determine the IP address of the machine that connected to it, rather than accepting an argument through the URL, this behavior would differ in the presence of network address translation or a proxy. Assuming that either all of the system components have public IP addresses or they are all behind the same NAT, the correct behavior is for the script to accept the IP address of the LCM TCP server's routable interface. To retrieve the stored IP address, a client of the discovery service issues a `GET` request to <http://iannucci.scripts.mit.edu/down.html>.

Another utility script runs the LCM datatype preprocessor to generate language-dependent source files for Java and Python. The output of the preprocessor is not included in version control. Other utility scripts invoke `lcm-logger`, `lcm-logplayer`, `kinect-lcm`, `kinect-glview`, and `lcm-spy` with the appropriate command-line arguments to discover and connect to the LCM TCP service. The `lcm-logger` script takes advantage of the underlying command's ability to automatically date- and time-stamp file names. Each script detects the case where the discovery service reports "0.0.0.0" as the IP address of the server, and fails with an appropriate error message.

Yafim wrote a Python script that interacts with the BRLTTY API to listen for and display messages on the Braille display. It currently truncates to 20 characters, but the API provides support for detecting scroll up/down key presses on the display unit. This script runs on the carried (worn) laptop.

4. Evaluation

4.1. Technical Evaluation

We needed a quantitative basis for evaluating our design decisions surrounding TCP LCM transport over Wi-Fi – in particular, how the TCP transport would compare with alternatives which maintain less network state in the presence of wireless outages – and we wanted to ensure that the voice recording and playback procedure introduced a tolerable amount of latency into question-and-answer round trips. In order to minimize weight and heat, we also needed lower bounds on the required performance of the worn and wizard laptops. We therefore set out to measure the latency associated with providing feedback to the user under various network conditions, as well as the computational, network, and storage loads on the two laptops while the system was recording Kinect data.

Our first experiment was to determine the worst-case latency associated with recovery from Wi-Fi hand-offs. We planned to issue 20 pings and 20 LCM messages per second, first over TCP and then over UDP, from a transmitting laptop moving back and forth along the Infinite Corridor, infamous for its Wi-Fi hand-offs. At the receiver, we logged the times (to the microsecond) when each packet was received, and computed the rate using a moving window average. We expected that a typical hand-off event would look something like Fig. 7, with the stateless protocols recovering fastest.

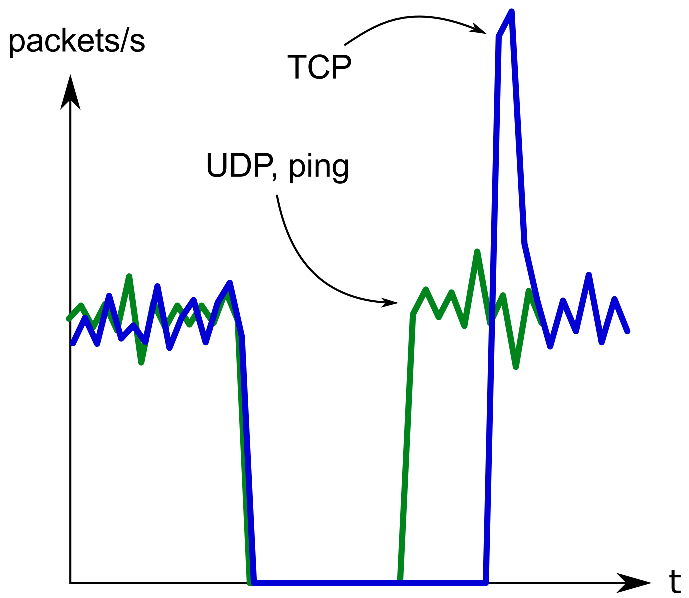


Fig. 7. Expected outcome for typical Wi-Fi hand-off

When we set up automated tools to collect this data, what we actually observed was much messier and more difficult to quantify. Our experience suggests that hand-offs vary from mild/unnoticeable (observed during some trials in Stata) to completely disruptive (certain parts of the Infinite Corridor). In the worst cases, the transmitting laptop disassociated from the network completely for an extended period of time, and manual intervention was required to restore connectivity. In the best cases, the hand-off resolved itself in about a second.

We are aware of three resolutions to the hand-off issue. The simplest is to maintain an ad-hoc wireless link between the worn laptop and the wizard laptop. This would require the wizard to work on the move, and we expect that it would degrade the realism of the user trials to have the wizard maintain the necessary proximity.

The second possibility is to equip the user with a 4G or WiMAX modem, and direct all the traffic over that link. Depending on the performance of those networks, we expect that this could shift the hand-off issue from the 100ft distance scale to the 1000ft scale, which would be a significant improvement and possibly practically sufficient.

A third possibility might be to take advantage of the Stanford OpenFlow Wireless project's Lossless Handover demo at ACM MOBICOM 2009[3] [4]. They designed a network stack that can connect to multiple access points simultaneously, solving the hand-off problem before it occurs.

Our second experiment was designed to measure round-trip latency introduced by the system. We found that for a certain repeated challenge response ("ping" and "pong"), keying the microphone, transmitting, and waiting for the wizard to receive the message and respond through the text-to-speech system took on average 6.2 seconds. By measuring the same round-trip time without the system being involved, we obtained a lower bound of 1.1 seconds for the ideal case of a system that introduces no delays at all. Accounting for the repetition of the users's queries (since the prototype requires them to record audio and then wait for it to be played back to the wizard) raises the practical latency lower bound for our design to 2.1 seconds.

Our third experiment measured resource requirements associated with the Kinect video feed. We were unable to get Kinect streaming over UDP multicast to work between the two laptops for reasons we still do not understand, but TCP Kinect streaming performed adequately in our Stata trial (described in the next section).

---

[3] Lossless Handover with n-casting between WiFi-WiMAX on OpenRoads, *Kok-Kiong Yap* and *Te-Yuan Huang* (Stanford University, USA); *Masayoshi Kobayashi* (NEC System Platforms Labs, USA); *Michael Chan* (Stanford University, USA); *Rob Sherwood* (Deutsche Telekom R&D Lab, USA); and *Guru Parulkar* and *Nick McKeown* (Stanford University, USA)

[4] Demonstration video available at <http://www.youtube.com/watch?v=ov1DZYINg3Y>

The uncompressed Kinect stream easily exceeds 40 MB/s. Our logged data rate for JPEG compressed RGB (quality 80%) and zlib compressed depth was 1.06 MB/s. We found CPU utilization in the neighborhood of 25-35% of a single core on a 2.66 GHz Intel Core 2 Duo machine to be typical for decoding the JPEG-compressed Kinect data stream, and 70-80% for encoding. These figures are potentially operating-system dependent, but they set a rough speed lower bound of 2.2 GHz for the worn machine. We also measured power dissipation of 13.5 W associated with transmitting the stream, on top of an idle power dissipation of 11.4 W. For a MacBook Pro with a roughly 50 Wh battery, this corresponds to a battery lifetime for user trials of about two hours, not including power for the Wi-Fi transmitter. The Kinect itself is powered by the external battery worn by the user.

In our 383 second trial, we logged 405 MB of data. Over the course of two hours, the system would generate 126 GB of data, which could easily fit on a laptop if the experimenters wish for the entire system to be mobile.

4.2. Usability Evaluation

The second component of our evaluation was to measure our success in enabling the Videation project to do user trials in the future. For this purpose, we ran an untethered navigation assistance trial in the Stata center with Yafim volunteering as the device wearer. Using guidance from the system, Yafim was able to successfully navigate from an unknown location to a known location in a different part of the building. We ran into issues with our LCM client on Android while running this trial, which could reliably send but could often not receive messages. The problem vanishes when the Android debugger is attached. We were able to work around the issue either by attaching the debugger or by having Yafim watch the text messages sent out by the wizard on his carried laptop screen. We logged the entire trial, and during log playback we are able to verify that all the components work properly. Provided the issues with LCM on Android can be overcome, and the vest can be modified to mount the carried laptop securely, we believe that the system is in good shape for user trials.

Our final objective was to transfer our knowledge to other members of the RVSN group to make sure nothing is lost as Yafim and I leave for the summer. With the help of sophomore Cristina Lozano and graduate Michael Fleder, we set out to get the software installed and working on a clean system. In the process, we compiled the setup guide for OS X linked from the Appendix. We plan to bring the software up on Michael's Linux desktop and the laptop procured by Jon in the near future, and at that time we will place setup instructions for Ubuntu in the same location.

5. Discussion

5.1. Future Work

Next steps for this project include training additional individuals in the use of the one-wizard system and using their feedback to improve the wizard application; making adjustments to the vest prototype (making the angle of view of the Kinect adjustable, moving or downsizing the battery, and mounting the Braille display and laptop) to prepare for user testing; and building more resiliency into the system. We experienced some unexplained reliability issues associated with the `libfreenect` driver at the heart of the Kinect software, and a work-around needs to be developed for them.

Beyond user trials, forward directions for the wizard software include gaining capabilities for odometry and spatial persistence/mapping, conversion to a graphical UI, and (depending on feedback from wizard trainees) splitting of wizard tasks between two people. One feature we anticipate being particularly useful is the ability to pause, rewind, and play back the Kinect feed at the wizard interface. This will reduce the need for the user to hold the camera stable while the wizard tags objects, and will help the wizard to “remember” the locations of misplaced objects or the faces of passed pedestrians.

Another avenue of attack will be reducing the CPU and network utilization associated the Kinect stream. This will leave more headroom for vision processing and rich spatial visualization.

Appendix

A.1. Setup Guide for OS X

See <http://groups.csail.mit.edu/rvsn/wiki/index.php?title=Videation_Project> for detailed installation instructions on OS X.