Processing Electromyographic Signals for Use as Computer Input

Adam Van Aken

4/19/15

Abstract

With the advancement of technology, daily interaction with computers is being relied on more and more. However, conventional interaction methods remain difficult or impossible to use for people with certain physical disabilities. One potential solution is to capture functioning electromyographic (EMG) signals from the user's muscles to be translated into computer input. We seek to show that EMG signal recognition is a viable alternative input method for people with certain physical disabilities. Modern signal processing is accurate enough for us to develop a proof of concept input method for both keyboard and mouse interaction. With further testing, a complete replacement for keyboard and mouse that relies only on the user's EMG activity could be achieved.

1 Introduction

Signal Processing and Human-Computer Interaction often appear together, such as in speech recognition and gesture recognition, to create computer input. The case that this report will examine, however, is using electromyographic (EMG) signals as a means for gesture recognition. Electromyography is the measurement and evaluation of bio-electric signals as they travel through skeletal muscles. Signal processing is very broad and can include the processing of any kind of signal, but in general it is the processing of an n-dimensional stream of input – quite often 2D. Human-Computer Interaction studies many things, such as speech-to-text input, gaze-tracking, keyboard input, etc. The goal is to connect these two fields and analyze how they can benefit from one another. Most interactions with computer software can be achieved with just pointing and clicking [2]. However, keyboard input is also very important. Therefore, we want to discover if EMG is a viable input method for people with specific disabilities that might restrict them from using conventional computer input methods (e.g. mouse/trackpad and keyboard).

Concisely, the topic at hand is to capture EMG signals in the forearm and use them to recognize the gesture formed by that user. EMG has been tried with electroencephalography (EEG), as a means of computer control before, specifically 2D mouse control [1]. EEG is very similar to EMG, but instead of recording skeletal muscle impulses, the electric signals along the scalp (commonly though of as "brain waves") are measured. EMG has also been used to recognize gestures without the help of EEG. However, our research has not found any examples of pure EMG-based mouse and keyboard input. Our goal is to combine research on EMG-signal-to-gesture detection with research on mouse control and ultimately achieve successful keyboard input. Hopefully the result can be used by persons with certain disabilities that prevent them from using conventional input methods.

2 Related Work

There are some existing technologies that attempt to provide alternate methods of mouse control. These methods are often aimed at enabling people with certain disabilities to control a mouse cursor. Barreto [1] mentions a few of them, such as *Tonguepoint* and *Headmouse*. *Tonguepoint* is a mouthpiece with a joystick that the user controls with their tongue. There are also buttons that the user can press with their tongue to emulate a mouse click.

Headmouse tracks a white ball attached to the user's forehead in a manner very similar to eye-gaze tracking. To click, the user simply dwells the cursor upon the desired click site. Though *Tonguepoint* might be cumbersome, the idea sufficiently provides mouse control - provided the user has fine tongue control. *Headmouse*, similarly, technically could provide both mouse and keyboard control (with an on-screen keyboard), but it has its own problems. Barreto addresses some problems with gaze tracking, such as the "Midas Touch" problem, which is essentially "ghost input." This is also a problem with the *Headmouse*. If the user is simply dwelling over a spot on the screen (e.g. watching the clock, reading an article, or watching a video) the user may not want to click, but the *Headmouse* and many eye-gaze tracking solutions will simulate one. This phantom click is known as a "Midas Touch" by some researchers [1, 4].

Eye-gaze tracking has additional implementation difficulties. The software must be calibrated with respect to the distance between the user and the optical sensors. This calibration provides a "sweet spot" of effective use. If the user moves in or out of that plane, the software must be re-calibrated. This poses implementation problems, especially for people with disabilities that may make the calibration process more difficult.

Barreto's individual research contribution was using EEG and EMG for cursor control [1]. Their solution used EEG signals to turn the software on and off, and EMG signals on the head and neck to control the mouse cursor. This control includes X and Y axis movement as well as left-click and pressing 'ENTER.' The nodes to read the EMG signals were placed as seen in **Figure 1**.

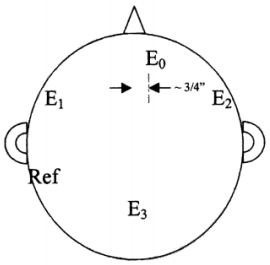

Figure 1: Electrode placement for EMG nodes [1]

As shown in **Figure 1**, E_0 is somewhat offset from the center of the head. The technique for exact placement can require at least one assistant with a measuring tape. This fact ultimately makes such technology potentially difficult to be setup solely by the user – it is not as simple as putting on a cap.

Other groups have used only EEG signals for their mouse control [1, 6, 11]. Some of the focus has been on detecting the "mu rhythm," which is defined as a sinusoidal brainwave in the 8-12 Hz range [11]. Sensors on the head and neck capture the EEG signals which are translated into mouse control. Event-Related Desynchronization, or ERD, is when the "contralateral mu rhythm is suppressed during the preparation of the movement." [6] After training with visual biofeedback, the user learns how to control these ERDs and, therefore, to control the mouse with their EEG signals. However, even after extensive training, users never gain complete control. A huge problem with using EEG alone is noise. EMG signals tend to be much more noisy than EEG signals in that they can have larger amplitudes [1]. Because of this excess noise, placing EEG nodes on the head runs the risk of EMG contamination every time the user blinks, moves their head or eyes, swallows, etc. Any number of repetitive actions involving the head or face can overwhelm the EEG signals, thus resulting in noisy data that cannot be accurately interpreted.

Saponas [7] researched how to process EMG signals and interpret them as gestures. They were able to use eight EMG sensors wrapped around a user's forearm to detect up to 18 gestures with high accuracy: tap, lift, extend, and apply pressure for each of the four main fingers (excluding the thumb) on one hand.

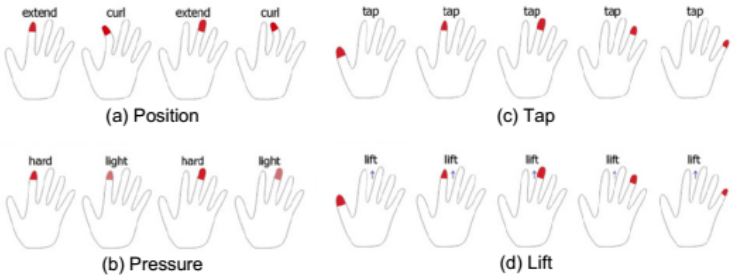

Figure 2: Above you can see the gestures detected [7]

The results were very good with highly accurate gesture detection ranging from 70% to 95% accuracy. They were also able to recognize six different grasps [8]. Again, their results had very high accuracy, depending on the amount of training data that they fed into their machine learning algorithm.

To be processed, signals are often divided into $250ms$ segments and each segment is treated as an independent data point. In his first paper, Saponas and team processed their EMG signals using this common technique [7]. They then ran their samples through a band-pass filter to remove signal ranges that they have shown are less useful. This left them with signals ranging from 2 Hz to 55 Hz and from 65 Hz to 102 Hz. The 55 Hz to 65 Hz band was removed since 60 Hz is a frequency present in many computing environments, from power lines and appliances, and is a risky range for noise pollution [7]. They calculated three features for each signal: Root Mean Square (RMS), Frequency Energy, and Phase Coherence.

First, they took the RMS of the amplitude of the EMG potential. RMS is a measurement of the magnitude for a quantity that is varied – especially a quantity that is sinusoidal. Since their experiment uses eight channels, they were able to create 28 features per data point by taking the ratio of each given channel with every other channel (in an "8 choose 2" scenario).

Next, the Fast Fourier Transform (FFT) is calculated for each sample, and its amplitude is squared. They calculate the frequency energy by using the FFT on each sample. "Frequency energy is indicative of the firing rate of muscle activity" [7]. The FFT calculated the energy at each frequency, and another 10 features for each sample are given by the sum of the energy for all of the eight channels into 10 Hz bins.

Finally, the phase coherence measurement is often used in EEG signal processing and relates to the fixed relationship among waves. Another 28 features were created by taking the ratios of each pair of waves' average phase for all channels. These 74 total features per data point were then run

through their detection algorithm to classify the gesture.

To classify their data points against all of the possible gestures, they would take the majority winner over the course of the entire stimulus. In their test runs, they used 2 seconds for each stimulus. **Figure 3** shows how each classified segment for their 2 second stimuli is tallied to determine the winner.

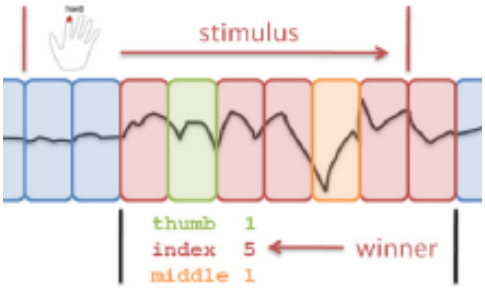

Figure 3: Gesture Classification[7]

One additional type of mouse control is Chin's research [2]. They experimented with cursor control using EMG and eye-gaze tracking (EGT). Their EMG sensors were placed on the face so that they could supplement their EGT results, as EGT accuracy is notoriously problematic. However, using EMG to aid the EGT results proved very successful, as their results had very high accuracy – down to one pixel.

These many different attempts at controlling a mouse, especially by signal processing, are proof of the concept. To a computer, mouse control is identical to keyboard control – it is simply a stream of input. By coupling the signal processing with the research by Saponas group for detecting gestures with EMG, controlling a mouse and keyboard by EMG signal processing seems possible.

3 Control Design

3.1 Mouse Control

Before implementing any electromyographic control, we must first define how our control will work. We will be using the *Myo* by Thalmic Labs [9] to implement our control. Fortunately, the *Myo* API comes with mouse control. All we need to do is enable or disable it as appropriate. Once enabled, the cursor moves as the user moves the device. Accelerometers in

the *Myo* detect 3-axial motion that the *Myo* translates into 2-axial cursor control.

The *Myo* recognizes five gestures: swipe left, swipe right, spread fingers, fist, and touch thumb to middle-finger. Using these five controls, we design a state diagram for mouse control, as depicted in **Figure 4**.

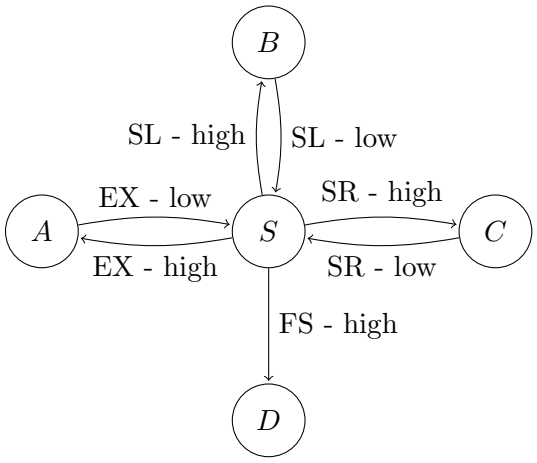

Figure 4: Our state diagram for mouse control. Control begins at node S

EX	extend	A	cursor movement
SL	swipe left	B	left-click
SR	swipe right	C	right-click
FS	fist	D	switch to keyboard mode
TM	thumb-to-middle-finger	S	starting state

Table 1: A key for the gestures in **Figure 4**

The above state diagram indicates how the mouse control starts and transitions. The "high" and "low" indicates whether the gesture is starting or stoping. In the cases of nodes A, B, and C, the gesture starts (high), the reaction happens, and the gesture stops (low) which returns us to the starting state. This behavior is helpful since, for example, state A handles cursor movement. While the person's fingers are extended, the cursor will follow the arm movement. This allows for more accurate stoping and starting

of cursor movement. Specifically, initial tests showed that using an entire gesture (i.e. high and low) to toggle between cursor lock/unlock states was not very accurate. While attempting the "lock" gesture in order to stop the cursor over a particular spot, the cursor shook and moved off of the desired location.

Gesture 5 is used to lock/unlock the *Myo*'s detection. This feature is to prevent ghost input. Node D has no return capabilities in this diagram, because state D represents the portion for keyboard control, which is defined below.

3.2 Keyboard Control

Now we define our keyboard control implementation. For rudimentary American English, we ultimately would like at least all 26 letters, 10 digits, *space*, *backspace*, and four punctuation marks: *comma*, *period*, *question mark*, and *exclamation mark*. However, we only have four recognizable gestures. It would be cumbersome to map these four gestures to 42 keystrokes. A possibility is to have a keyboard styled like old SMS keyboards: "*ABC*" typing such that repeated gestures cycle through a distributed set of keystrokes. However, those methods usually relied on a dedicated *space* and *backspace* buttons. Following that method, we would be left with two gestures and each to cycle through 20 keystrokes. This is not ideal.

Another possibility is to increase our "vocabulary" of recognized gestures by measuring the length of time between the high and low signals of a given gesture. This, though possible, would probably require an annoyingly high amount of precision by the user, and it also limits the potential words per minute (WPM) of the typist.

To solve our problem we will use an Ordered-Gesture Tree (OGT). With four gestures, we can compress 16 recognized gestures into binary commands. That is, two consecutive gestures will map to a given control. 4^2 will give us 16 and 4^3 will give us 64 usable options. This fully satisfies our desired alphabet of keystrokes. However, we could do better if we introduced a second *Myo*, giving us a 24 gesture alphabet (four gestures per device by itself, and 16 2-gesture concurrent combos by using both devices at once). 24^2 gives us 576 which can certainly map an entire keyboard of buttons (many input buses use 8-bits, for 256 inputs).

However, for the purposes of our investigation, we will only use one *Myo* device with two-part commands, such that we can handle 16 keystrokes. We will include *space*, *backspace*, *period*, and the first half of the alphabet: *a-m*

(excluding *j*, for its limited frequency, and replacing it with functionality to return to the mouse control state).

Figure 5 below shows our OGT for the control sequence of our keyboard control. Each pair of gestures, for example (EX) - (EX), will result in a keystroke. In that example, "A" would be typed. If any initial gesture is followed by (TM), thumb-to-middle-finger, the control sequence will return to the start state. This is to cancel out of an accidentally initiated gesture sequence. (FS) - (FS) will return the control to the mouse state diagram shown in **Figure 4**.

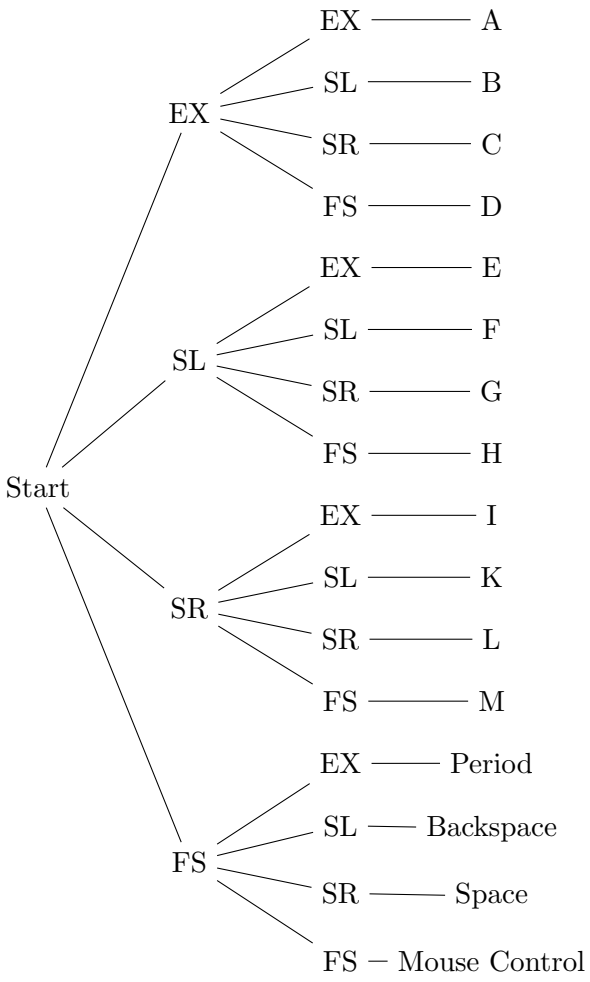

Figure 5: Our OGT for keyboard control. The gestures are abbreviated as defined in **Table 1** and **Table 2** below. Each leaf corresponds to the associated keystroke or behavior.

EX: extend	SL: swipe left	SR: swipe right
FS: fist	TM: thumb-to-middle-finger	

Table 2: A reference table for the gesture abbreviations in **Figure 5**

4 Work and Methods

After designing the state diagrams and OGT, the implementation was very straightforward. The *Myo* developer kit is very intuitive, and the code is based on Lua. A *Myo* script is designed to be activated upon a specific computer application coming into the foreground of the computer. We enable our script regardless of which application enters the foreground. This makes the script always active so that the user can control mouse/keyboard for any window.

The *Myo* API comes with mouse control, so implementing our state diagram for the cursor was very easy. Implementing the OGT functionality was also simple with the build in API calls to trigger input. Expanding our specific designs to a different platform and/or set of gestures would, therefore, require additional work to implement the OS input controls. However, the software is the easy part. Our work here is to show whether the effort is worthwhile. In other words, to show whether EMG recognition is powerful enough to handle mouse and keyboard input.

5 Results

5.1 Mouse

Mouse results were recorded as two data points: click speed and accuracy. To test we will run a Mouse Accuracy and Pointer Click Training software found online[1]. The software spawns targets at random positions on the screen where they will grow to a maximum size and then shrink away. The user can set the target spawn rate as well as how large the targets grow. For our tests we will use a slow spawn rate and a medium target size. We will compare the results with the *Myo* to a standard mouse and trackpad.

5.1.1 Click Speed

The trials run in 15 seconds. We will define our measurement of Click Speed as the number of targets clicked in that 15 seconds. For the *Myo*, our mean and minimum were 5, but our maximum was 7. For both a conventional mouse and a trackpad, 8 was the significant average. Though the *Myo* averaged slower than the other devices, it was not terribly slow.

[1] http://mouseaccuracy.com/

5.1.2 Accuracy

We will score accuracy as $\frac{misclicks}{total\ clicks}$. For the *Myo*, our average accuracy was 80.2%. Trackpad and mouse scored at 96.3% and 87.2% respectively. Interestingly, the trackpad beat the mouse for accuracy. However, most importantly the *Myo* was 23% closer to the mouse results than the mouse-trackpad comparison. Since the mouse is commonly used over the trackpad, despite its lower accuracy measurements, this strongly suggests that the *Myo* can hold its own as a device for mouse control.

5.2 Keyboard

Much like mouse control, we recorded two data points for keyboard control. We will split the keyboard results into two sections: words per minute and typing accuracy. This should fully capture both desired properties of a typing input method. While a typist wants speed, they also want to be accurate. Speed often inversely affects accuracy, but any effective input method should have a good balance (or allowance) of both.

5.2.1 Words Per Minute

Words per minute, or WPM, is often used as a representation of typing speed. If we want to be able to use the *Myo* to replace conventional keyboards users will have to be able to reach competitive WPM speeds. However, our first goal is to make a suitable input method for people with certain disabilities such that they cannot use a conventional mouse or keyboard. The average speed for transcription from one study in 1998 showed 33 WPM [5]. The definition of WPM defines a "word" to be 5 keystrokes including spaces [10].

In order to accurately test the WPM, we had our subjects transcribe a given script. Listed beneath the desired output were the exact gestures required to achieve that output. This reduced, to a degree, negative effects on typing speed due to the learning curve of learning a new alphabet. To compare a typist using our alphabet with an average typist on a QWERTY keyboard would under-represent the potential speed, as one might have been using a QWERTY keyboard for decades but only just begun with our *Myo* alphabet.

In our initial tests, we had speeds of around 5 WPM including correcting all mistakes to achieve 100% accuracy. Without correcting for mistakes, our speed did increase to 6 WPM with 95% accuracy.

5.2.2 Typing Accuracy

The *Myo* comes with a default calibration profile that is agnostic of device placement as well as skin properties. It should work relatively the same for all orientations and all people. However, it also allows for individual calibration which seems to work best if done every time the device is put on (calibration requires only about a minute, which is not terrible especially since EEG node placement can take much longer than that). For our accuracy tests, we used both the default calibration and custom calibrations.

As mentioned in the section above, without correcting for mistakes the subjects had around 95% accuracy while attempting to type quickly with a custom calibration profile. However, in the specific test for accuracy, the subjects went through all sequences of commands. This resulted in 93.3% accuracy. Visual feedback was output to indicate if the "first-step" gesture was misread so that the subjects could terminate the gesture sequence and begin again. Time was also not pressured for this test, giving the subjects ample time to make their gestures. With the default calibration we saw accuracy of 88.6%. This, too, was with visual feedback and no time pressure.

For both calibration profiles we also recorded the most common mistake. The *middle-finger-to-thumb* gesture was consistently the least accurately interpreted, most often being mistaken for the *spread* gesture. This inaccuracy is not very surprising for the custom calibration profile due to how the calibration is designed. It records all of the *other* four gestures, but does not record this fifth gesture. It also records the arm's resting state. From those recordings it generates a profile, interpolating for the *middle-finger-to-thumb* gesture.

6 Discussion and Future Directions

One of the most challenging pieces of implementing a successful gesture recognizer, such as the *Myo*, is to make it environmentally agnostic. That is, more gestures may eventually be recognized, but it is difficult to do so accurately with different skin types, amounts of arm hair, levels of skin oils, etc. The *Myo* should work roughly the same on any person's arm and in any orientation. Being this flexible comes at the cost of additional gestures. However, as we showed, this limited amount of gestures can be enough to comfortably control an entire set of keyboard input.

We considered certain things that we did not look into in detail, such as handedness. Could a right-handed mouse user be more accurate using their left hand with the *Myo* than with a mouse? It could be a more desirable

backup in the event of a limited right-hand.

Also, certain gestures are often more reliable than others. After having discovered that in testing, perhaps there could be better designs that focus less heavily on inaccurately-matched gestures.

We also have not tested our system on people with disabilities that might limit their conventional input abilities. Some future tests should be run to compare our *Myo* system to existing input methods for people with disabilities. However, we expect that it will work well for certain disabilities.

Additionally, a user cannot use a conventional keyboard very well while also using the *Myo*. The muscle activity for typing ultimately unlocks the device and then induces gesture recognition. This could be solved by disabling the script between uses (which can be done with *Myo* mouse control, but a regular mouse is needed to re-enable it).

6.1 Mouse

Mouse control was the easiest to handle with the *Myo* since it is built into the *Myo* API. The accuracy was not too much worse than conventional mouse accuracy. However, the speed was a bit less. With respect to **Figure 4**, we noticed that the cursor would tend to jump while the user was trying to return from state A to S, or while the low end of the *extend* gesture was detected. This is due to the users being unable to keep the device still while ending the *extend* gesture. One possible solution could be to increase the accelerometer threshold necessary to generate movement.

If we added a second device, we could also increase the speed. The second device could handle clicks while the user keeps the first device steady. This would help to solve the problem mentioned above. Also, in turn that would help increase the accuracy by replacing misclicks with intended clicks.

6.2 Keyboard

There are difficulties in accurately measuring potential WPM with our system. There is certainly a learning curve associated with using a new technology and alphabet as our test subjects are new to the muscle movements as well as our alphabet. Perhaps a better comparison would be the average speeds of a typist using a new keyboard layout (e.g. Dvorak) to someone with our *Myo* implementation. This could mitigate the learning curve.

Our choice of OGT layout was not based on any planning or trials. In fact, we think future research of our project could explore different mappings.

Perhaps a more intuitive map could be made that makes use of the subject's muscle memory for a standard keyboard layout. However, if the goal is to provide a method of input for users who cannot use conventional keyboards, perhaps that muscle memory does not exist anymore.

Huffman Coding [3] could be used here to generate an OGT with more potential. Huffman Coding generates trees based on frequency or weight of the data. A Huffman tree could be generated from frequency of input characters in a given language. The more frequent (or higher weighted) a character is, the shorter the path on the OGT would be. However, with our gestures we could also consider some other factors: muscle strain, mental strain, ease of gesture, accuracy of recognition, etc. Combining factors such as these with the Huffman tree could lead to some very interesting algorithms for OGT generation. As the keyboard mapping is handled in software, it would be very simple to trial some different layouts.

However, the *Myo* was designed to be able to work in parallel with a second device on the user's other arm. As briefly discussed before, this gives us many more gestures to work with. If everything is a 2-gesture combo, we have 576 options. However, with 24 unique single gestures, half of those could be reserved for 1-gesture commands. We could map, for example, the more common letters and punctuation to these 1-gesture commands. This leaves another 12, from which we can have 144 2-gesture combos to satisfy the other necessary input.

Having a certain number of 1-gesture commands reserved for common input would almost certainly increase the potential WPM. Also, a more intuitive gesture-to-input map could increase speeds and accuracy. Once a user gets used to this input, it will be interesting to reevaluate their speed and accuracy.

7 Conclusion

The *Myo* seems like a feasible replacement for mouse and keyboard input. With a little more research into the specifics of the implementation, such as keyboard mapping, the use of two devices, OGT generation, etc., the input could hopefully be made much more comparable to conventional input methods. It certainly seems to be the case that EMG signal recognition has come a long way such that it can be used in HCI like this. The work that Saponas [7] has done to pave the way for EMG gesture recognition, as well as the great work at Thalmic labs for developing the *Myo*, combine to give us a new technology that could have a huge effect on the future of HCI.

8 Acknowledgements

I would like to thank John Dougherty for all of his help and advisement, without which this thesis would not have been possible. Also, a big thank you to Thalmic Labs and their work with the *Myo*, without which, again, this thesis would not have happened.

References

[1] Barreto, A. B., Scargle, S. D., Adjouadi, M., et al. A practical emg-based human-computer interface for users with motor disabilities. *Journal of Rehabilitation Research and Development 37*, 1 (2000), 53–64.

[2] Chin, C. A., Barreto, A., Cremades, J. G., and Adjouadi, M. Integrated electromyogram and eye-gaze tracking cursor control system for computer users with motor disabilities. *Journal of Rehabilitation Research and Development 45*, 1 (2008), 161.

[3] Huffman, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E. September* (1952), 1098–1101.

[4] Jacob, R. J. K. The use of eye movements in human-computer interaction techniques: What you look at is what you get. *ACM Trans. Inf. Syst. 9*, 2 (Apr. 1991), 152–169.

[5] Karat, C.-M., Halverson, C., Horn, D., and Karat, J. Patterns of entry and correction in large vocabulary continuous speech recognition systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 1999), CHI '99, ACM, pp. 568–575.

[6] Pfurtscheller, G., Flotzinger, D., and Kalcher, J. Brain-computer interface: A new communication device for handicapped persons. *J. Microcomput. Appl. 16*, 3 (July 1993), 293–299.

[7] Saponas, T. S., Tan, D. S., Morris, D., and Balakrishnan, R. Demonstrating the feasibility of using forearm electromyography for muscle-computer interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2008), ACM, pp. 515–524.

[8] Saponas, T. S., Tan, D. S., Morris, D., Balakrishnan, R., Turner, J., and Landay, J. A. Enabling always-available input with muscle-computer interfaces. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology* (2009), ACM, pp. 167–176.

[9] Thalmic Labs Inc. Myo, 2015. [Online; accessed 14-Feb-2015].

[10] Wikipedia. Words per minute — wikipedia, the free encyclopedia, 2015. [Online; accessed 2-April-2015].

[11] Wolpaw JR., M. D. J. Multichannel eeg-based brain-computer communication. *Electroencephalogr Clin Neurophysiol. 90*, 6 (1994), 444–449.

9 Appendix I. Raw Trial Data

	Trial 1		**Trial 2**	
	Click Speed	Accuracy	Click Speed	Accuracy
Myo	5.5	81.25%	6.0	79.15%
Trackpad	8.0	94.45%	8.0	100.0%
Mouse	8.0	88.9%	8.0	86.35%

Table 3: Raw data from the two trials for mouse results

	Char./min.	**Mistakes**
Custom Calibration	45.0	3.0
Default Calibration	35.0	4.0

Table 4: Total raw data from two trials for keyboard results

10 Appendix II. Myo Script Code

The following code was adapted from Thalmic Labs PowerPoint Connector script[2]:

```
-- flags and private variables
supportShuttle = false
keyboardControl = false
-- We will use -1 to indicate no first input received
keyboardFirstInput = -1
firstInputTime = 0
appName = "none"

function leftClick()
    myo.mouse("left", "click")
end

function rightClick()
    myo.mouse("right", "click")
end

function conditionallySwapWave(pose)
    if myo.getArm() == "left" then
        if pose == "waveIn" then
            pose = "waveOut"
        elseif pose == "waveOut" then
            pose = "waveIn"
        end
    end
    return pose
end

function typeKey(pose)
    if keyboardFirstInput == 1 then
        if pose == "spread" then
            myo.keyboard("a", "press")
        elseif pose == "left" then
            myo.keyboard("b", "press")
        elseif pose == "right" then
```

[2]https://market.myo.com/app/5474c658e4b0361138df2a9e

```
            myo.keyboard("c", "press")
        elseif pose == "fist" then
            myo.keyboard("d", "press")
        end
    elseif keyboardFirstInput == 2 then
        if pose == "spread" then
            myo.keyboard("e", "press")
        elseif pose == "left" then
            myo.keyboard("f", "press")
        elseif pose == "right" then
            myo.keyboard("g", "press")
        elseif pose == "fist" then
            myo.keyboard("h", "press")
        end
    elseif keyboardFirstInput == 3 then
        if pose == "spread" then
            myo.keyboard("i", "press")
        elseif pose == "left" then
            myo.keyboard("k", "press")
        elseif pose == "right" then
            myo.keyboard("l", "press")
        elseif pose == "fist" then
            myo.keyboard("m", "press")
        end
    elseif keyboardFirstInput == 4 then
        if pose == "spread" then
            myo.keyboard("period", "press")
        elseif pose == "left" then
            myo.keyboard("backspace", "press")
        elseif pose == "right" then
            myo.keyboard("space", "press")
        elseif pose == "fist" then
            keyboardControl = false
        end
    end
    -- We registered key, now clear input
    keyboardFirstInput = -1
end

function shuttleBurst(edge)
```

```
    if shuttleDirection == "forward" then
     if edge == "on" then
        onWaveRight(edge)
    end
    elseif shuttleDirection == "backward" then
     if edge == "on" then
        onWaveLeft(edge)
        end
    end
end

function onSpread(edge)
    if keyboardControl then
        if keyboardFirstInput == -1 and edge == "on" then
            keyboardFirstInput = 1
            firstInputTime = myo.getTimeMilliseconds()
            myo.setLockingPolicy("none")
        elseif edge == "on" then
            typeKey("spread")
        end
    else
        if edge == "on" then
            myo.controlMouse(true)
            myo.unlock("hold")
            myo.notifyUserAction()
        else
            myo.controlMouse(false)
            myo.notifyUserAction()
        end
    end
end

function onWaveLeft(edge)
    if keyboardControl then
        if keyboardFirstInput == -1 then
            keyboardFirstInput = 2
            firstInputTime = myo.getTimeMilliseconds()
            myo.setLockingPolicy("none")
        else
            typeKey("left")
```

```
        end
    else
        leftClick()
    end

end

function onWaveRight(edge)
    if keyboardControl then
        if keyboardFirstInput == -1 then
            keyboardFirstInput = 3
            firstInputTime = myo.getTimeMilliseconds()
            myo.setLockingPolicy("none")
        else
            typeKey("right")
        end
    else
        rightClick()
    end
end

function onFist(edge)
    if keyboardControl then
        if keyboardFirstInput == -1 then
            keyboardFirstInput = 4
            firstInputTime = myo.getTimeMilliseconds()
            myo.setLockingPolicy("none")
        else
            typeKey("fist")
        end
    else
        keyboardControl = true
    end
end

function onDoubleTap(edge)
    if keyboardControl then
        if keyboardFirstInput == -1 then
            myo.setLockingPolicy("standard")
            myo.lock()
```

```
        end
        keyboardFirstInput = -1
end

function onPoseEdge(pose, edge)
    if pose == "waveIn" or pose == "waveOut" then
        local now = myo.getTimeMilliseconds()

        if edge == "on" then
            -- Deal with direction and arm
            pose = conditionallySwapWave(pose)

            if pose == "waveIn" then
                shuttleDirection = "backward"
            else
                shuttleDirection = "forward"
            end

            -- Extend unlock and notify user
            myo.unlock("hold")
            myo.notifyUserAction()

            -- Initial burst
            shuttleBurst(edge)
            shuttleSince = now
            shuttleTimeout = SHUTTLE_CONTINUOUS_TIMEOUT
        end
        if edge == "off" then
            myo.unlock("timed")
            shuttleTimeout = nil
        end
    elseif pose == "fingersSpread"then
        onSpread(edge)
    elseif pose == "fist" and edge == "on" then
        onFist(edge)
    elseif pose == "doubleTap" and edge == "on" then
        onDoubleTap(edge)
    end
end
```

```
-- All timeouts in milliseconds
SHUTTLE_CONTINUOUS_TIMEOUT = 600
SHUTTLE_CONTINUOUS_PERIOD = 300

KEYBOARD_COMBINATION_TIMEOUT = 5000

function onPeriodic()
    local now = myo.getTimeMilliseconds()
    if (now - firstInputTime) > KEYBOARD_COMBINATION_TIMEOUT then
        -- Reset input
        if (keyboardFirstInput ~= -1) then
            myo.setLockingPolicy("standard")
            myo.lock()
        end
        keyboardFirstInput = -1
    end

    if supportShuttle and shuttleTimeout then
        if (now - shuttleSince) > shuttleTimeout then
            shuttleBurst()
            shuttleTimeout = SHUTTLE_CONTINUOUS_PERIOD
            shuttleSince = now
        end
    end
end

function onForegroundWindowChange(app, title)
    appName = app
    return true
end

function activeAppName()
    return appName
end
```