Test Bed for Performance of Single Degree Freedom Movement

Justin Simon Tran

A thesis

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

2012

Committee:

Howard Jay Chizeck

Blake Hannaford

Program Authorized to Offer Degree:

Electrical Engineering

University of Washington

Abstract

Test Bed for Performance of
Single Degree Freedom Movement

Justin Simon Tran

Chair of the Supervisory Committee:
Professor Howard Jay Chizeck
Electrical Engineering

Research in brain computer interfaces (BCI) focus mainly on neural-prosthesis. Recently commercial electroencephalograms (EEG), a non-invasive BCI, have been introduced into the market which decreases the cost and increases the availability for everyday use. A standard evaluation of BCI performance is lacking, therefore in thesis work, a test bed is developed to measure performance of a BCI for single degree freedom movement. The classic video game Pong is used to measure and compare the performance of different users, input devices, and control algorithms. The first experiment recreated the Fitts' Law reciprocal tapping task to validate the of measure performance for different paddle sizes and distance between targets. The second experiment extends Fitts' Law to a Pong game. Our results for the second experiment show there is a main effect where paddle size affects throughput ($F(5,812) = 19.77$, $p < 0.0001$). Since there was a main effect, a pairwise t-test was used to compare performance between different paddle sizes.

Future research will compare the performance for different input devices and signal processing algorithms.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor Professor Howard J. Chizeck for his support and guidance throughout my graduate career. I would also like to thank Charlie Matlack for his insight and collaboration throughout my research; and Jessica Tran for helping me prepare for my defense and editing my thesis.

DEDICATION

I dedicate this thesis to my parents Simon and Julie Tran,

my sister Jessica Tran, and Martha Chan.

I. Introduction

Brain computer interfaces (BCI) create a direct communication channel between a users' brain and a device. Examples of medical BCIs include prosthesis, communication, and diagnostics. Commercial applications include video games and brain training. Recently commercial electroencephalograms (EEG) have become available on the market, which decreases the cost to obtain a BCI and increases accessibility to a larger audience. This makes it possible to test a large group of subjects; however, a standard method to evaluate BCI performance is lacking. Commercial manufacturers do provide hardware specifications, but these specifications may not translate to the performance experienced by the user. To solve this, I created a test bed to measure the performance of BCIs for single degree of freedom (DOF) movement using the classic video game Pong. The game Pong was selected because it has varying difficulty, allows for one and two player modes, and has an inherent measure of performance, score. Figure 1 below is a high level diagram of the system to develop the pong software.

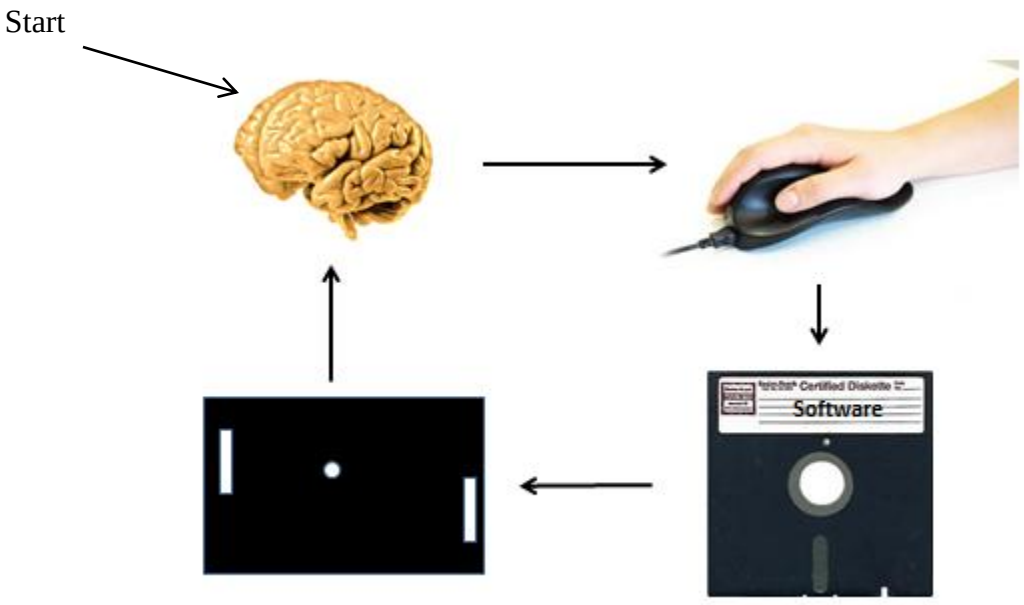

Figure 1: Block Diagram to create test bed

Starting at the users' brain, a signal is sent through the neuromuscular system to the hand which moves the mouse. The movement of the mouse creates an input to the Pong software. In the software, the start and stop position and movement time of the paddle is recorded during game play. The user sees their performance of the Pong task on a computer monitor and interprets the data. A modified mouse, set to move along a single line, is a familiar and reliable input device that allows us to rigorously test the Pong software.

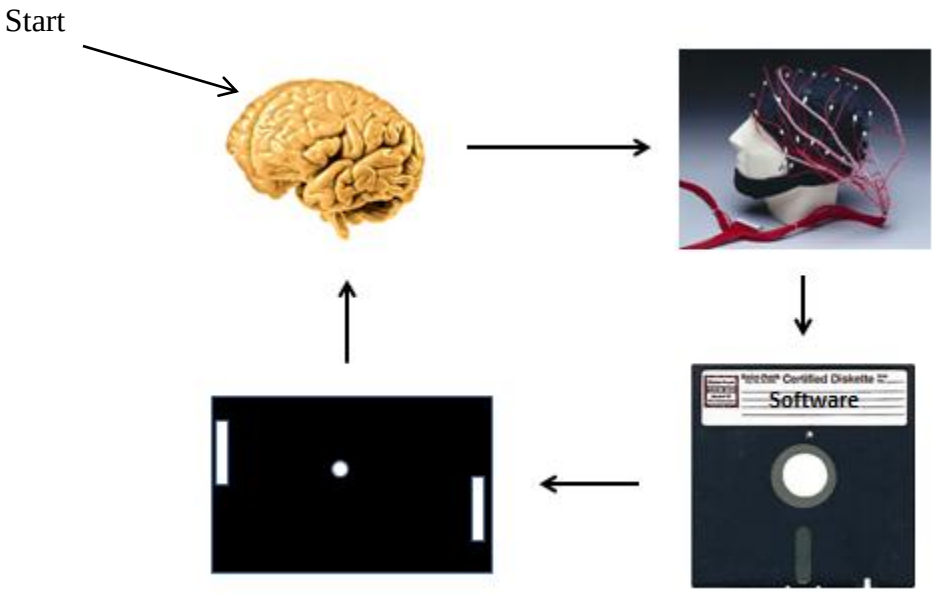

Figure 2: Block Diagram for Final Goal

After completing the Pong software, different input devices can be tested. Figure 2 shows the final goal of our research which is to replace the mouse input by a BCI. Using a BCI bypasses the neuromuscular system therefore directly linking the human brain to the software which measures performance of different users, input devices, and control algorithms.

The contributions of this research include:

1) Creating a test bed to capture user input data to complete a 1 DOF task
2) Analyzing input data to infer performance
3) Comparing the performance of different users, input devices, and control algorithms

These contributions are the first steps towards measuring the performance of BCIs for 1 DOF tasks.

II. Background and Literature Survey

A measure of performance, compatibility with hardware, and current applications of BCIs were considered in the test bed design. To measure performance a variation of Fitts' Law is applied to the Pong game using a continuous input signal to control the paddle.

2.1 Fitts' Law

Shannon's Information Theorem introduces the concept of channel capacity. A channel is a medium used to transmit a signal from a transmitter to a receiver. The medium of a channel may be a pair of wires, coaxial cable, band of radio frequencies, beam of light, etc. Shannon's theorem describes how much information can be transmitted over a channel given the bandwidth, strength of signal, and noise of signal (Shannon, 2001).

$$C = B \log_2 \left(1 + \frac{S}{N}\right) \quad (1)$$

where:

C is the channel capacity (bits/second)
B is the bandwidth of the channel (hertz)
S is the total received signal power over the bandwidth (watt)
N is the total noise or interference power over the bandwidth (watt)

Fitts' Law is an extension of Shannon's Information Theorem to the human motor system which predicts the time required to rapidly move to a target area as a function of the distance to the target and size of the target. Fitts used a reciprocal tapping task with a weighted stylus. The user was asked to move the stylus between two targets with a specified distance between the targets and specified width of target (Fitts, 1954). This study, referred to as Fitts' Law, quantified the difficulty of a single DOF task, predicted movement time to complete a task, and provided a measure of performance. For this test,

the medium of "the channel" extends to the human neuromuscular system. Channel capacity from Shannon's theorem is analogous to movement time, how fast a person can complete a task for a specified distance and accepted error.

$$MT = a + b\ \log_2\left(\frac{A}{W} + 1\right) \quad (2)$$

$$ID = \log_2\left(\frac{A}{W} + 1\right) \quad (3)$$

$$IP = \frac{ID}{MT} \quad (4)$$

where:

MT is the average time taken to complete the movement (seconds)
ID is the index of difficulty (bits)
IP is the index of performance (bits/second)
A is the distance from target
W is the width of the target
a represents the start/stop time of the device (intercept) and
b stands for the inherent speed of the device (slope).

The *a* and *b* constants can be determined experimentally by fitting a straight line to measured data.

The index of performance (IP) is analogous to channel capacity (C) from Shannon Information Theorem (bits/s), index of difficulty matches the log term (bits), and MT matches 1/B (in seconds) (MacKenzie, 1992). These equations give a value to how well a user can perform a task and how difficult a task is.

2.2 Brain Computer Interface

2.2.1 Types of BCI

There are three types of BCIs used to extract brain signals: invasive, partially-invasive and non-invasive.

Invasive

Invasive BCIs are implanted in the user's cortex specifically, electrodes inserted in the grey matter of the brain. This provides the highest signal quality, but may degrade overtime with scar-tissue formation. Scar tissue may damage the brain and negate the benefit of a BCI (Daly & Wolpaw, 2008).

Partially Invasive

Partially Invasive BCIs are implanted on the user's cortex which allows for high quality signal without the risk of scar-tissue formation. Electrocorticography (ECoG) measures brain activity using electrodes on the surface of the cortex while providing a higher signal quality than non-invasive BCIs. The sensors are not susceptible to as much noise from readings through the user's skull.

Non-Invasive

Non-invasive BCIs read brain activity without direct contact of the brain. The most studied is EEG, which uses electrodes placed on the scalp to read signals through the skull. EEGs are low cost, portable, and offer good temporal resolution but have low spatial resolution. Magnetoencephalography (MEG) records magnetic fields caused by electrical current from brain signals. The functional magnetic resonance imaging (fMRI) tracks brain activity by tracking changes in blood flow. MEG and fMRI have better temporal and spatial resolution than EEG, but the devices are large, thus limiting the location of use. Near infrared spectroscopy (NIRS) is similar to the fMRI since it measures the changes in blood flow, but is limited to cortical tissue, where fMRI measures activation throughout the brain (Zhang, Wang, & Fuhlbrigge, 2010).

2.2.2 BCI Tasks and Fitts' Law

Types of Movement Tasks

Path control and terminal aiming are two categories of movement control tasks. In path control tasks, subjects are asked to follow a given path as closely as possible. The performance criterion emphasizes minimizing the deviations from the path over its whole length. In terminal aiming tasks, subjects must reach an end target in a single movement. Performance is measured by the extent and direction of the difference between the final point reached by the subject and the target (Drury, Montzer, & Karwan, 1987).

Fitts' Law Tasks

Trajectory and pointer based tasks have been explored using Fitts' Law. Trajectory based tasks test movement in two degrees of freedom. An example of a trajectory based task is drawing a circle. Pointer based tasks test movement in a single degree of freedom. An example of a pointer task is moving a cursor "center out." This task measures the amount of time needed to move a cursor from the center of a computer screen to a target that is horizontal or vertical from the center.

The time given to complete a task must also be considered for trajectory and pointer based tasks. The different types of timing tasks are self-paced, speeded, neither self-paced nor speeded and externally paced. Self-paced tasks are tested at the rate the user wishes to complete the task. An example of a self-paced task is the original Fitts' Law reciprocal tapping task shown in Figure 3. The user has as much time needed to move the stylus back and forth. A speeded task has a time restriction to complete a task. For

example using the center out task, every 3 second interval a new target will appear for the user to hit.

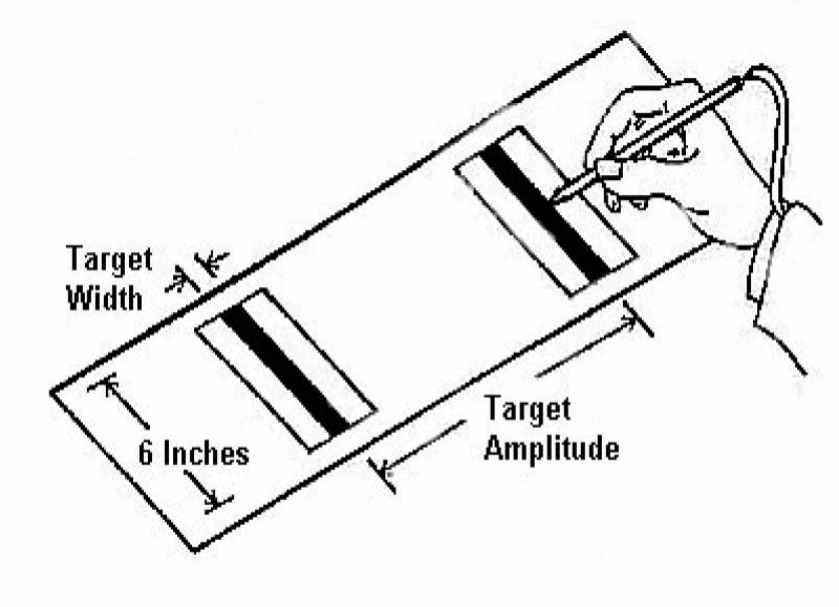

Figure 3: Continuous Fitts' Reciprocal Tapping Task

Beggs and Howarth (1972) examined a variation of a reciprocal tapping task. In this study, subjects moved a pencil in a motion like throwing a dart, moving alternately between a back contact plate and a target, paced by a metronome. The subjects were paced by a metronome, hence the movements were not self-paced nor speeded as in other studies. According to Fitts' Law the accuracy requirements should adversely affect movement velocity and in the faster metronome settings accuracy should deteriorate (Beggs & Howarth, 1972). A neither self-pace nor speeded task gives a recommended time to complete a task, but does not require the user to adhere to it. The data for this test was poorly described by Fitts' law since the task doesn't require the user to move as fast as possible.

2.2.3 BCI Applications

There are many applications of BCIs for improving the livelihood of users both medically and recreationally. Medical monitoring is one application of EEGs which are used to detect epileptic seizures, brain activity, and depth of anesthesia. Other medical applications include communication and control, which allow disabled patients to communicate through a virtual keyboard and control a mouse cursor.

There are many BCI applications for 'abled' users such as brain training exercises and video game control. For example, Neurotopia is a company that offers brain training for sports performance using EEGs. Neuro-feedback during athletic activities helps the user recreate conditions for optimal performance. 'Imagined' sporting actions and sports with minimal head movement have been analyzed to establish a criterion for optimal performance. The sports analyzed are cycling, marksmanship, and golf. Predictors for optimal performance were defined in two ways 1) the differences between expert and non-expert performance, and 2) in pre-shot EEG differences between successful and unsuccessful shots (Thompson, Steffert, Ros, Leach, & Gruzelier, 2008). The results from this EEG analysis are to be incorporated into athletic training in real time or post analysis.

The EPOC Emotiv and NeuroSky Mindset are commercial EEGs that are on the market. These two products offer connectivity with computer games. The EPOC Emotiv has been used to translate mental states into corresponding keyboard and mouse control sequences for applications. For example, in the game World of Warcraft, "the player's facial

expressions are used to trigger character animations, which reflect the user's emotional state" (Scherer, Pröll, Allison, & Müller-Putz, 2012). BCIs provide an alternative way to control the game and enhance the user's experience,

2.3 Pong and BCI

The classic video game Pong is a one dimensional task that can be used as a platform for BCI testing. The Berlin Brain Computer Interface uses an EEG to play Pong which they use two different control strategies. The first strategy studied is "stepwise displacement," where the paddle is moved by a fixed displacement step or remains at rest depending on the command signal. The second control strategy uses a "graded displacement" code that moves the paddle according to the strength of the command signal (Krepki, Curio, Blankertz, & Müller, 2007).

III. Experiment Design

In this thesis, we experimentally studied the performance of single degree freedom movement by recreating Fitts' work and extending it using Pong as a platform. The test bed is validated with a modified mouse input, this allows future tests to use different BCI input devices.

Playing the game Pong is the task selected to test for performance of single degree freedom movement. In order to apply Fitts' Law to Pong, the amplitude (A) and width of target (W) from equation (2) must be modified.

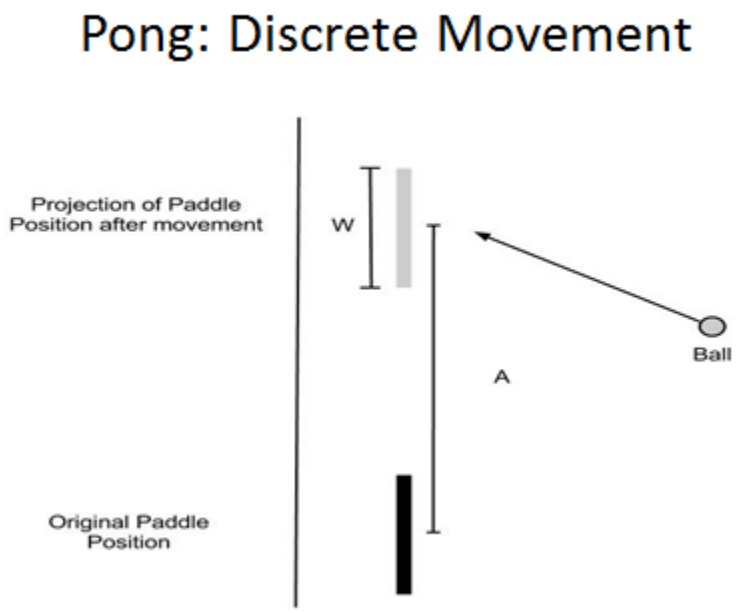

Figure 4: Fitts' Law applied to Pong

In Figure 4, the user is attempting to hit the ball with the paddle. The error is defined by the distance of the center of the ball from the center of the paddle. The amplitude is the magnitude of movement made by the paddle. A cycle in the Pong game is defined as time

when the ball strikes the left paddle, hits the right paddle, and returns to the left paddle. The movement time (MT) and magnitude of distance (A) are summed per cycle. This task uses a modified definition of Fitts' Law. Instead of a cursor moving an amplitude (A) to meet a target (W), the paddle (W) is moving an amplitude (A) to meet a moving Pong ball which is the target.

3.1 Measurements

The users' input controls the vertical position of the paddle. During a trial, the time and position of the paddle is measured when the paddle starts and stops moving. These measurements are used to calculated the amplitude (A) and movement time (MT) used in Fitts' Law. Also error is measured by measuring the position of the center of the paddle and the position of the ball along the plane of the left players paddle. The error is defined as the distance of the ball from the center of the paddle at the left plane of impact.

In addition to the Fitts' related measurements, amplitude (A) and movement time (MT), Pong game measurements are also made which are length of a rally per point, winner of a rally, and final score. The length of a rally is defined as the number of contacts made to the ball by a paddle per point.

3.2 Testing Procedure

3.2.1 Experiment 1

The first experiment is a recreation of the Fitts' reciprocal tapping task applied to Pong. Two Pong balls are placed a set an amplitude (A) apart and a paddle with a set size is used to transverse between them. The ball must be contained in the paddle and the paddle

must rest for 10 milliseconds in order for a hit to be registered. A counter next to each target allows the user to know that a hit is registered.

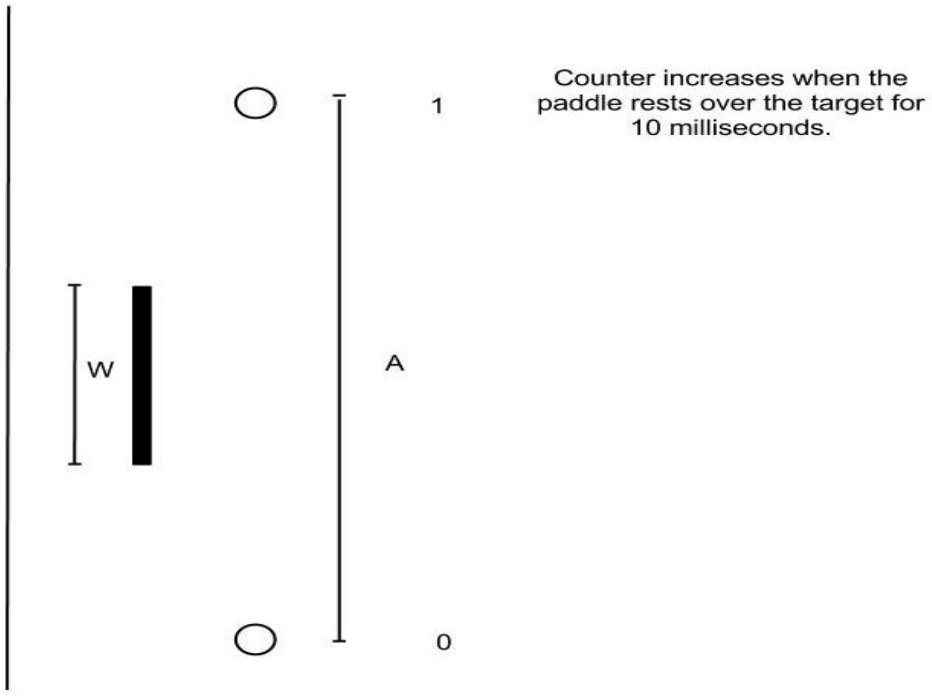

Figure 5: Illustration of Experiment 1

Method

Figure 5 above is an illustration of the test used in experiment 1. The 6 paddle sizes studied are (24, 48, 72, 96, 120, and 240 pixels). Four center-to-center distances between the target balls were studied (200, 400, 600, and 800 pixels) giving a total of 24 combinations of amplitude and target size.

The user is instructed to "Move the paddle alternately and make as many hits as possible within 15 seconds. Emphasize accuracy by matching the center of the paddle to the center of the ball. The counter next to the ball will increase by one when a hit is registered. The trial begins when the paddle is rested over the top ball for 10 milliseconds."

3.2.2 Experiment 2

The second experiment involves playing Pong against a computer opponent until the computer scores 11 points. The difficulty of the computer opponent is held constant for all trials, while the paddle size is varied (24, 48, 72, 96, 120, and 240 pixels). The speed of the ball is initialized in the start-up of a trial. The ball speed increases 10% with each hit during a rally. The ball speed resets to the original speed when a point is scored.

Method:

The following game settings were used during experiment 2.

Amplitude:

The vertical and horizontal components of ball speed vary with the number of regions on the paddle. The larger number of regions, the smaller the amplitude between each movement task. Eight hitting regions are used throughout the experiment.

Ball Speed:

The horizontal ball speed is initially set at 3 pixels per millisecond and the horizontal speed increases by 10% with each paddle hit. After a point is scored the speed resets to the original ball speed.

Computer opponent:

The computer opponent tracks the ball with an initial offset and randomly makes an additional offset once per cycle. This allows the computer opponent to make a mistake after a minimum number of cycles. In experiment 2 the AI setting is 6. To calculate the minimum number of cycles required for the computer to make a mistake use:

$$minimum\ number\ of\ hits = AI/2 - 1$$

IV. Hardware

A computer with two mice, an Intel Core i7-950 at 3.06 GHz and a Radeon 3450 ATI, 512 Mb were used for testing. These specifications exceed the requirements needed to run the Pong software for the experiment.

In Pong, the paddle is controlled by a mouse input which provides a continuous input signal. If future uses of the software, a BCI can be used in place of a mouse since a BCI provides a continuous input signal.

The Pong code uses the position of the cursor on the screen to produce paddle movement. This creates a problem for two player Pong with mouse input because the vertical movement needs to be controlled by two different mice. To solve this hardware problem, a hardware and software trick is used. Since the paddle only needs to control movement in a single degree of freedom, the mouse input can be divided into vertical and horizontal movement. In the code, the horizontal movement of the mouse is used to control the left players paddle. The vertical movement of the mouse is used to control the right players paddle.

The left player's mouse is manipulated to convert vertical input into horizontal movement of the cursor. The right player's mouse converts vertical input into vertical movement of the cursor.

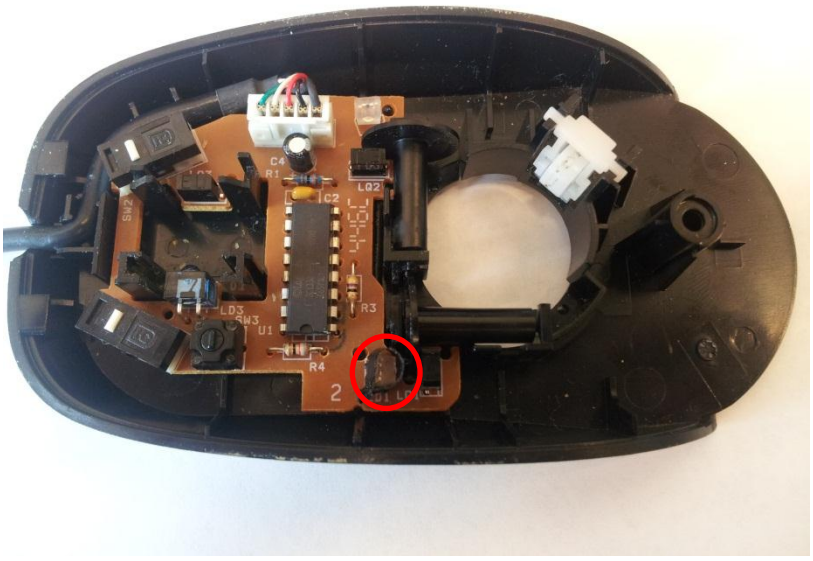

Figure 6: Modified Mouse for left player paddle

Figure 6 is a picture of the circuit of the left player's roller ball mouse. The LED for the horizontal input is blocked by the black electrical tape. This prevents light from travelling through the optical wheel to the photo diode. A roller ball mouse uses optical encoding disks to translate vertical and horizontal movement into light pulses which are read by a photo diode. If no light pulses are read by the photo diode, no movement is interpreted by the computer.

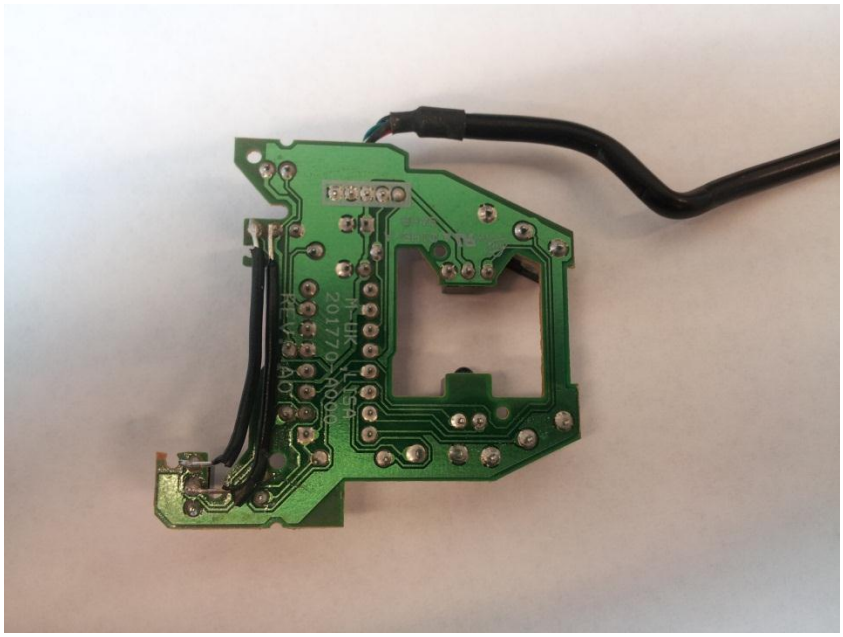

Figure 7: Modified Mouse for left player paddle

Figure 7 is a photo of the bottom view of the mouse. The vertical movement from the roller ball creates a signal from the LED/optical wheel/photo diode. This signal is routed to the horizontal receiver and results in vertical movement on the computer screen.

For the right players paddle, vertical movement of the cursor corresponds to vertical movement of the mouse. To modify the right players paddle, electrical tape is used to cover horizontal LED. This prevents the right players' mouse from interfering with the left players input signal.

V. Software

5.1 Pong Code

An open source version of Pong (Appendix A) written in C and compatible in Linux was needed to create a single degree freedom movement test for performance. These requirements allow the tester to change different settings in the game.

To measure performance using Fitts' Law, the code was altered to measure start and stop time; start and stop position; and position of the ball relative to the center of the paddle. The code was also altered to track game play statistics such as length of rally, score, and winner of each rally.

The Pong code used to design the tool is based on the Allegro game library. Appendix A discusses how to obtain and run the open source code on a Linux computer. Appendix B discusses the debugging fixes to the original code. Appendix C and D are the "engine" files used for Trial 1 and Trial 2.

5.2 Pong Game Performance

Pong measurements relevant to performance are score, length of rally, and winner of each rally. These variables provide additional information on user performance. The length of rally uses a counter to keep track of the number of hits within a rally. The counter increases by one each time the ball hits a paddle. The counter resets to zero when a player scores i.e. the paddle missed hitting the ball. At the end of a rally the length of the rally and the winner is recorded. The final score is recorded when the computer reaches 11 points. This information is saved in a text document for post-test analysis.

5.3 Fitts' Law and Pong

To monitor player performance using Fitts' Law, the code was modified to record the time and position when the paddle starts and stops moving. Also, the error is recorded by measuring the position of the center of the paddle and the center of the ball when it crosses the plane of the front edge of the left paddle.

Timer

In order to track start and stop times for Fitts' Law analysis, a timer must be implemented in the code. The Allegro library offers a timer with millisecond resolution (code displayed below).

```
//Timer
//1 tick per millisecond
install_timer();
install_int(tick, 1);
```

Time is recorded when a start or stop is identified, these conditions are addressed in the movement tracking section. A dwell time of 10 milliseconds is used to identify a stop which is compensated for in the code

Movement Tracking

Identifying when a stop occurs is needed to record the start and stop positions of a paddle. To implement this, the velocity of the paddle calculated using the current position of a paddle, the previous position, and the change in position as described in the code below.

```
int current_p1y = mouse_x;
int current_p1_slope = current_p1y-previous_p1y;

if(current_p1_slope == 0)
{
        counter++;
```

```
            if(counter == 10)
            {
                        start_flag = 0; //reset the start flag
                        paddle_movement++;
                        //a stop has occurred
            }
}
else
{
            //reset counter
            counter = 0;
}

if(start_flag == 0 && current_p1_slope != 0)
 {
            //a start has occurred
}
```

When the paddle is not moving the velocity of the paddle is zero. If it remains in the same place for ten milliseconds, the code identifies that a stop occurred. This dwell time is removed in the time measurement. When the paddle is moving and the previous velocity is zero, a start is identified. These two events are used mark the time and position of the paddle. The time and position are summed per cycle and recorded to a text document for analysis.

Accuracy in Fitts' Law is determined by the position of the ball which is relative to the position of the center of the paddle. When the ball is on the same plane as the surface of the left paddle, the position of the ball is recorded along with the position of the paddle.

These code modifications allow for the Pong game to be played against a computer opponent while recording information needed for Fitts' Law analysis.

VI. Discussion/Results

For this work, a single subject (the author) was used for evaluation. Evaluation on additional subjects will require IRB approval (pending).

6.1: Trial 1

Table 1 below shows the results for the 24 conditions tested using the Pong reciprocal tapping task. The left side of the table shows the test conditions of paddle size, amplitude, and index of difficulty of task. The right side of the table displays the results for a mouse input. These are movement time, percent error, throughput, and the rank of performance. Percent error is defined as the distance of the center of the paddle away from the center of the ball divided by the size of the paddle.

Table 1: Task Conditions and Performance Data for 24 Variations of a reciprocol tapping task

Tolerance and Amplitude Conditions			Mouse Input			
Paddle Size (pixels)	Amplitude (pixels)	ID (bits)	MT (milliseconds)	% Error	TP (bits/s)	Rank
24	200	3.22	572.89	25.5	5.62	10
24	400	4.14	594.56	18.3	6.97	2
24	600	4.70	715.69	24.5	6.57	3
24	800	5.10	818.64	23.8	6.23	5
48	200	2.37	476.26	20.3	4.97	14
48	400	3.22	542.32	24.0	5.94	7
48	600	3.75	531.40	19.5	7.07	1
48	800	4.14	779.67	20.7	5.31	11
72	200	1.92	421.00	23.6	4.55	18
72	400	2.71	470.00	19.3	5.77	8
72	600	3.22	503.77	19.4	6.40	4
72	800	3.60	587.00	23.4	6.13	6
96	200	1.62	351.39	20.1	4.62	17
96	400	2.37	420.30	23.1	5.64	9
96	600	2.86	542.90	24.2	5.26	12
96	800	3.22	643.53	19.1	5.01	13
120	200	1.42	392.74	15.9	3.60	20
120	400	2.12	438.12	18.3	4.83	15

120	600	2.58	593.29	17.5	4.36	19
120	800	2.94	611.65	16.3	4.80	16
240	200	0.87	370.37	9.4	2.36	24
240	400	1.42	407.12	14.9	3.48	23
240	600	1.81	518.86	14.6	3.48	22
240	800	2.12	597.30	21.8	3.54	21

The most difficult task to complete is the smallest paddle size with the largest amplitude. The easiest task to complete is the largest paddle size with the smallest amplitude. The best performance (Rank 1) occurred at paddle size 48, amplitude 600. The throughput used complete this trial is 7.07 bits/s due to a high index of difficulty and fast movement time. These results can be visualized in Figure 8 below.

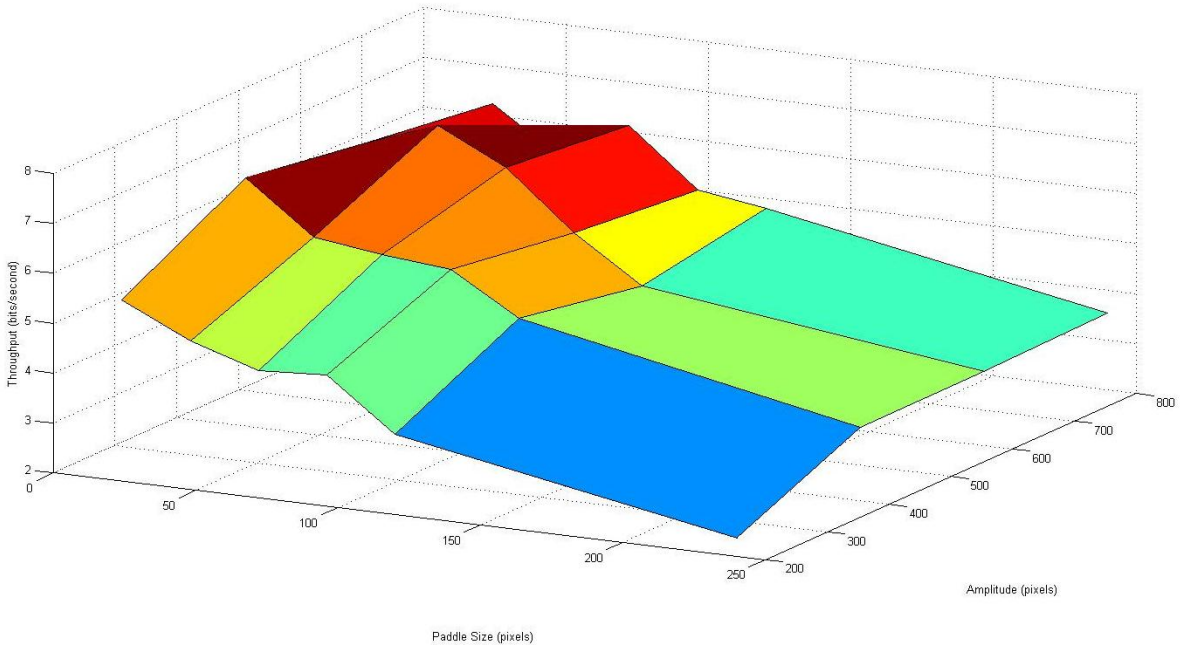

Figure 8: Three-dimensional representation of performance rate (TP) in bits per second for the reciprocal tapping task for Pong as a function of amplitude and tolerance requirements

Figure 8 above is a three-dimensional representation of Table 1. The x-axis has paddle size increasing, the y-axis shows the amplitude, and the z-axis is the throughput. There is a general trend that shows that easier tasks, larger paddle sizes and small amplitudes,

requires less throughput to complete than more difficult tasks, small paddles and large amplitudes.

In Figure 9 below, linear least squares regression is applied to the data collected from the trial with paddle size 240 with amplitude 400. Each data point is represents a single movement and the corresponding time to complete the movement. The *a* and *b* coefficients are determined empirically.

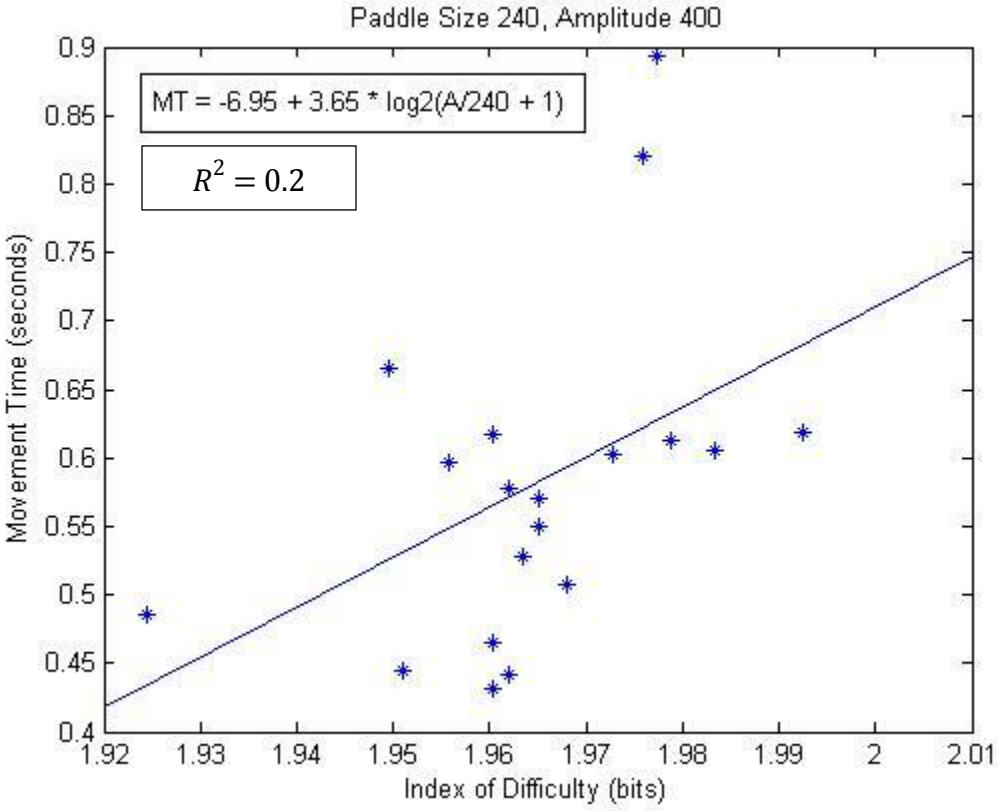

Figure 9: Linear Least Squares Regression for A = 400, Paddle Size = 240

The mean throughput to complete this task is 3.48 bits/second. Linear least squares regression produces the slope, *b,* and intercept *a,* used to predict the movement time for a given difficulty of task. In the original Fitts' Law test, the reciprocal of the slope was used as the index of performance. This technique to measure performance is invalid when

comparing the performance of different test conditions as discussed by Soukoreff & MacKenzie (2004).

6.2: Trial 2

For trial 2, the Pong game is played against a computer opponent until the computer scores 11 points. The paddle size is varied between each test. Figure 10 below plots the score versus the paddle size, the paddle size decreases from left to right.

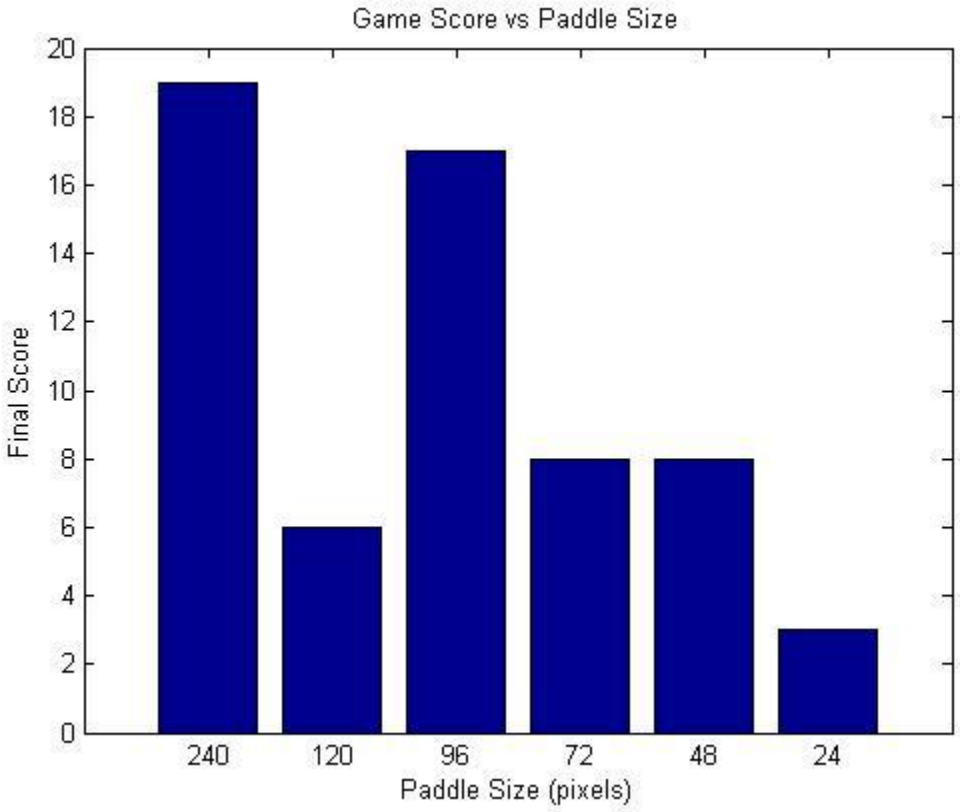

Figure 10: Score versus Paddle Size

With a larger paddle size, the ID of a game and TP required to complete a movement task are low. These two factors create an easier Pong game as seen in Figure 8. The final scores for the paddle sizes are (19, 6, 17, 8, 8, and 3). The score for paddle size 120 is

less than the score for paddle size 240 and 96. Additional performance measures will be explored to see why the score is significantly lower.

Figures 11-13 are histograms that show in greater depth the types of movements required to complete a Pong game trial for a single paddle size. The following results are for paddle size 240.

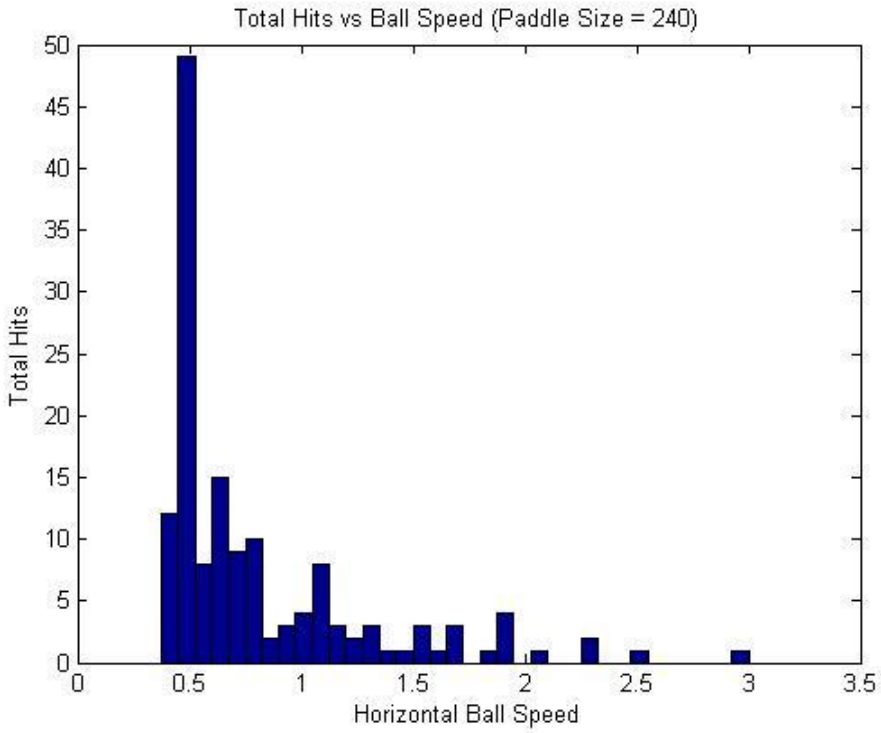

Figure 11: Total Hits vs Horizontal Ball Speed

Figure 11 shows the total number of hits versus the horizontal ball speed. The size of a bin is 0.075 pixels per second. Horizontal ball speed is the time for the ball to move from the center of the screen towards the left paddle plane of collision. From the histogram, we can see that the majority of hits occurred at low speed. The ball speed increases by 10% each hit during a rally. When a point is scored the ball resets to the initial ball speed. Most of the hits fall into the first 5 bins. This is because of the ball speed resetting at the

end of a rally, and the 10% increase in speed. When the ball speed increases by 10% at a small speed, the change in speed is small. As the ball speed increases, the change will also increase.

Figure 12 compares the Total Hits versus the Magnitude of Distance Moved. The size of a bin is 20 pixels. Most of the hits occurred in the first bin. The most common distance to move is under 20 pixels. These movements are when the ball is hit directly to the paddle from the computer opponent, or when small corrections are used to meet the ball. In general we expect to seem small movements for a large paddle size since the paddle covers the majority of the screen.

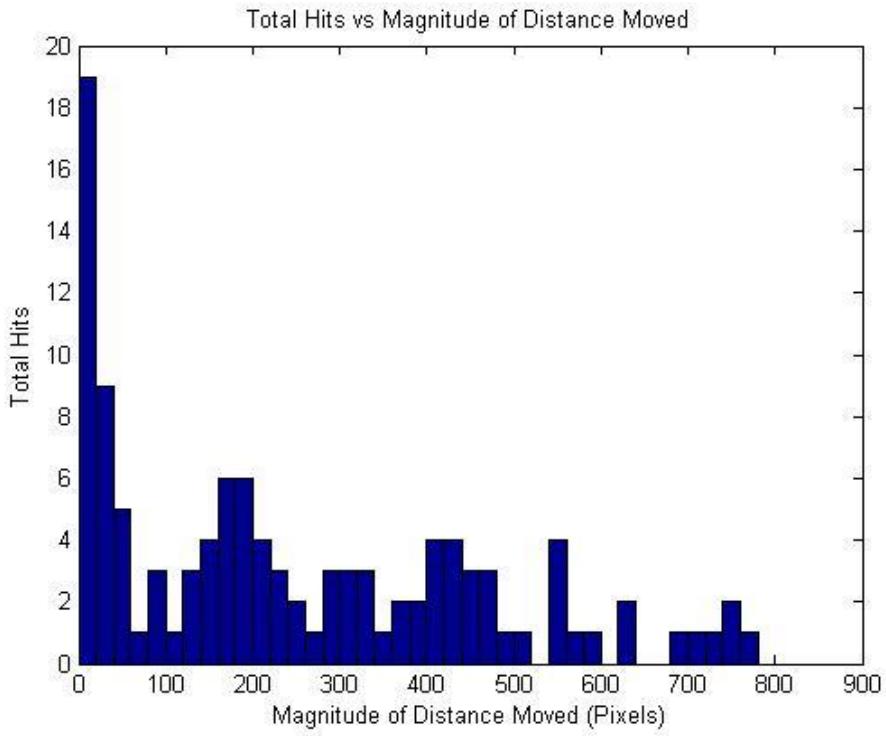

Figure 12: Total Hits vs Amplitude

Figure 13 compares the Total Hits versus the Movement Time of the paddle. The size of a bin is 100 milliseconds. The majority of movement time is less than 1.5 seconds. As the

ball speed increases the user is given less time to complete the task. The user is also instructed to emphasis accuracy while moving the paddle to the position of the ball as quickly as possible.

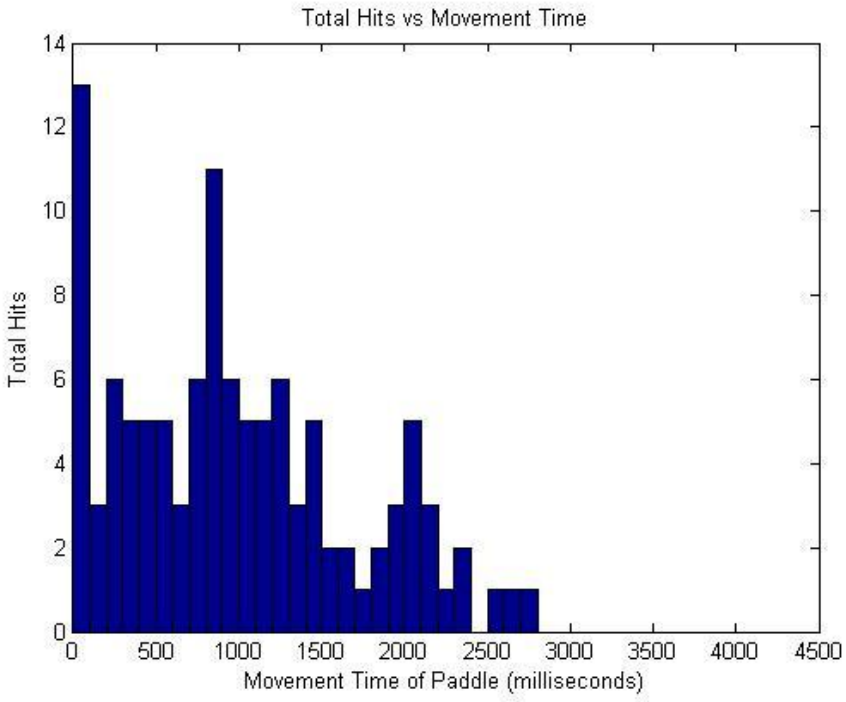

Figure 13: Total Hits vs Movement Time

Figure 14 displays the mean and variance of throughput used to complete a Pong game for different paddle sizes. It seems that the throughput required to complete a task increases as paddle size decreases. To test our hypothesis of decreasing paddle size increases throughput, a one way ANOVA was used. There is a main effect where paddle size affects throughput ($F(5,812) = 19.77$, $p < 0.0001$).

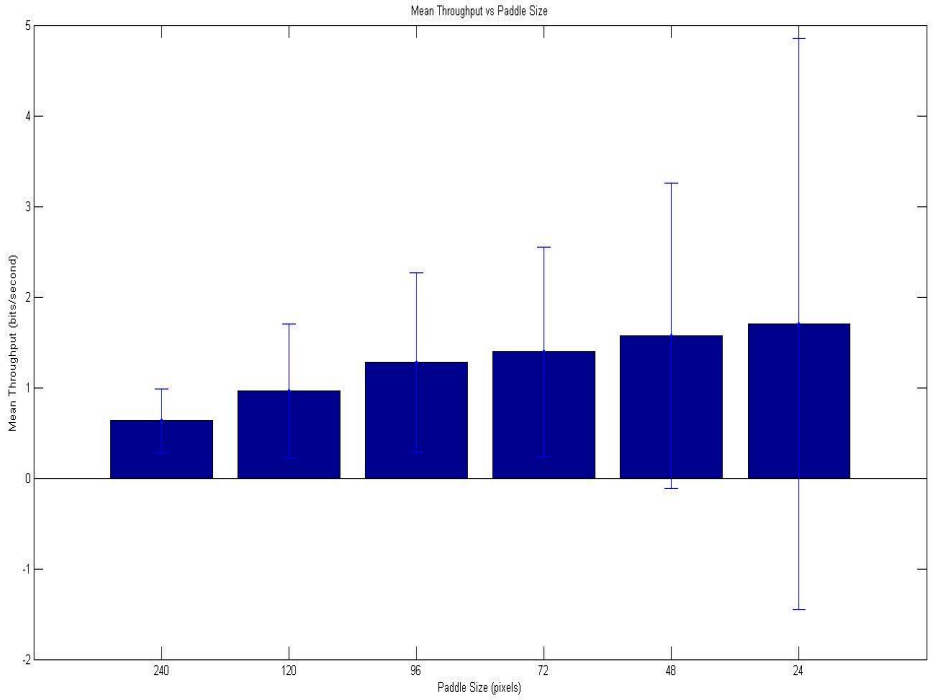

Figure 14: Mean and Variance of Throughput versus Paddle Size

Since a main effect was found, a post-hoc pairwise t-test with Bonferroni correction ($\alpha = \frac{0.05}{6} = 0.0083$), is used to test if the throughput to complete each task is statistically different for each paddle size. A Bonferroni correction is used to test each pair at the same statistical significance level, reducing false positives.

Table 2: Paired t-test to compare throughput for different paddle sizes

Paddle Size 1	Paddle Size 2	t-ratio	p-value	p < 0.05/6 = 0.0083
48	24	-0.02537	0.9798	ns
72	24	-1.99066	0.0469	ns
72	48	-2.425	0.0155	ns
96	24	-2.83919	0.0046	Yes
96	48	-3.53088	0.0004	Yes
96	72	-1.01253	0.3116	ns
120	24	-4.67957	0.0001	Yes
120	48	-5.72408	0.0001	Yes
120	72	-3.40132	0.0007	Yes

120	96	-2.5729	0.0103	ns
240	24	-6.85687	0.0001	Yes
240	48	-8.43065	0.0001	Yes
240	72	-6.14958	0.0001	Yes
240	96	-5.45589	0.0001	Yes
240	120	-2.68386	0.0074	Yes

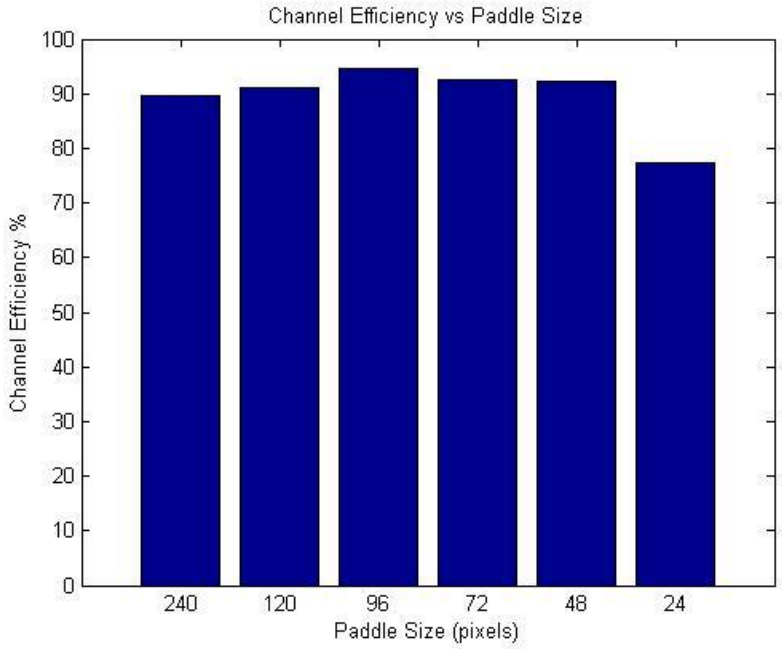

Figure 15: Channel Efficiency versus Paddle Size

Figure 15 shows the channel efficiency versus paddle size. Channel efficiency is calculated by taking the throughput for successful hits divided by throughput of hits and misses. The channel efficiency is an indicator of how well the users' effort goes towards performing the task. For the case of paddle size 24, the channel efficiency is about 75%. This means that 75% of the movements used to move the paddle results in successful hits.

When comparing the throughput for paddle sizes 240 and 120, it was found statistically significant that throughput was higher for paddle 120 over 240 ($t(812) = -2.86$, $p < 0.01$). The throughput for paddle sizes 120 and 96 were not found significantly different

(t(812)) = -2.57, ns). For paddle sizes (240, 120, 96), they have corresponding game scores (19, 6, 17, respectively). When analyzing the throughput and channel efficiency, the played the game proficiently for three paddle sizes. The low score at paddle size 120 is attributed to the lack of errors made by the computer opponent. With a larger test group, these observations can be reaffirmed.

VII. Conclusion and Future Work

This research has made progress towards measuring the performance of BCIs for single degree freedom movement. I created a test bed using Pong as a platform that accepts different inputs and measures movement time, amplitude, length of rally, and score. This data is analyzed to compare performance. By recreating the Fitts' test, we established the performance for different paddle sizes and distances, allowing us to extend Fitts' Law to Pong. Using a one way ANOVA, a main effect was found that paddle size affects throughput ($F(5,812) = 19.77$, $p < 0.0001$). Using a pairwise t-test with Bonferroni correction it was found that the performance for paddle size 120 is significantly greater than the performance for paddle size 240 ($t(812) = -2.86$, $p < 0.01$). Information about throughput, combined with channel efficiency, and score gives a comprehensive view on performance. Throughput is a measure of speed and accuracy. Channel efficiency indicates if the player is performing the task properly and the score indicates how well the player is completing the Pong task.

The test bed is not limited to comparing performance of different paddle sizes. Future research will examine different input devices and signal processing algorithms to control the paddle. A full medical EEG cap will be used to validate the test bed. With the medical EEG, different signal processing algorithms can be examined for performance to control the Pong paddle, such as signal reference montages, wave band analysis, and emotional state. There are a variety of signal reference montages such as averaged, bi-polar, Laplacian, and referential. An average montage uses the average of all electrodes as a reference signal, a Laplacian montage uses the electrodes around the electrode of

interest as a reference signal, a referential montage uses a specified electrode as its reference signal, and a bi-polar montage uses the difference between two electrodes as a signal. Wave bands are associated with different mental states which are: delta (0-4 Hz)/sleep, theta (4-8 Hz)/drowsiness, alpha (8-13 Hz)/relaxation, beta (13-30 Hz)/active thinking, and gamma (30 – 100 Hz)/multiple sensory. Emotional states are based on additional processing of wave bands. NeuroSky uses additional processing in their hardware to produce "Attention" and "Meditation" scores. These are proprietary formulas which combine different wave bands. Beta waves have greater weighting in the "Attention" score, while alpha waves have greater weighting in the "Mediation" score. After establishing a benchmark of performance, similar test will be performed on commercial EEGs.

Works Cited

Beggs, W., & Howarth, C. (1972). The movement of the hand towards a target. *Experimental Psychology*, 448-453.

Daly, J. J., & Wolpaw, J. R. (2008). Brain-computer interfaces in neurological rehabilitation. *Lancet neurology, 7*(11), 1032-1043.

Dobkin, B. H. (2007). Brain-computer interface technology as a tool to augment plasticity and outcomes for neurological rehabilitation. *The Journal of physiology, 579*(part 3), 637-642.

Drury, C., Montzer, M. A., & Karwan, M. (1987). Self-pace Path Control as an Optimiztion Task. *Systems, Man, and Cybernetics, 17*(3), 455-464.

Fitts, P. M. (1954). The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement. *Experimental Psychology, 47*(6).

Krepki, R., Curio, G., Blankertz, B., & Müller, K.-R. (2007). Berlin Brain - Computer Interface = The HCI communication channel for discover. *Internation Journal of Human-COmputer Studies, 65*, 460-477.

MacKenzie, I. (1992). Fitts' Law as a Research and Design Tool in Human-Computer Interaction. *Human-Computer Interaction, 7*(1), 91-139.

Montfor, N., & Bogost, I. (2009). *Racing the Beam: The Atari Video Computer System.* Mit Press.

Scherer, R., Pröll, M., Allison, B., & Müller-Putz, G. (2012). New Input Modalities for Modern Game Design and Virtual Embodiment. *IEEE Virtual Reality*, (pp. 163-164). Orange County.

Shannon, C. E. (2001). A Mathematical Theory of Communication. *ACM SIGMOBILE Mobile Computing and Communications Review, 5*(1), 3-55.

Soukoreff, R. W., & MacKenzie, I. S. (2004). Towards a standard for pointing device evaluation, perspectives on 27 years of Fitts' law research in HCI. *International Journal of Human-Computer Studies, 61*(6), 751-789.

Thompson, T., Steffert, T., Ros, T., Leach, J., & Gruzelier, J. (2008). EEG applications for sport and performance. *Methods, 45*, pp. 279-288. San Diego.

Zhang, B., Wang, J., & Fuhlbrigge, T. (2010). A Review of the Commercial Brain-Computer Interface Technology from Perspective of Industrial Robotics. *IEEE International Conference on Automation and Logistics*, (pp. 379-384). Hong Kong and Macau.

Appendix A – Pong Code Instructions

The instructions below describe how to download and run the original code on a Unix based computer. Special thanks to the author "pro-mole."

Instruction to run the code:

1.) On a linux computer, click on the link: http://www.allegro.cc/depot/Pong4Linux/
2.) Scroll the page and download pong-1.0.tar.gz
    http://www.allegro.cc/files/depot/2076/pong-1.0.tar.gz
3.) Extract the download file to your desired location
4.) Using the terminal, open the folder "pong-1.0" using the Unix commands:
    ls – list directory contents
    cd – change directory
5.) After navigating to the pong-1.0 folder, use the Unix command:
    make – GNU make utility to maintain groups of programs
    This will compile all of the files in the pong-1.0 folder
6.) Run the pong game with the command ./pong-1.0

This will run the pong game which can be played against a computer using the up and down arrow keyboard keys.

The code used in experiments 1 and 2 are located in appendix C and D. To run this code, open the pong-1.0 folder. This folder will contain two files engine.c and main.c. Copy and paste the code for the experiment you want to test into the engine.c file. Follow the steps above to run the code.

Appendix B – Debugging the Original Pong Game

When you play the original *Pong* game you will notice that the game doesn't operate properly. This section will go into greater depth on the modifications to the code.

Game Start Up

When the game is initialized in the original code, a pseudo random number is used to determine an angle the ball will initially travel. This code has the ball travel between -45 to 45 degrees, to the right players paddle to begin the game. Since the first ball always travels towards the right players paddle it may skew the test results. The code is fixed to randomly select the direction of the first ball.

Original Code

```
float dir = ((float)(rand()%9100-4600)/100) * (PI/180);
ball_hspeed = ball_speed * cos(dir);
ball_vspeed = ball_speed * sin(dir);
```

New Code

```
int possession = 2*(rand()%2)-1;
ball_hspeed = possession*ball_speed;
ball_vspeed = 0.0;
```

The new code random selects the direction of the first ball (left or right) during the startup of the game. The balls vertical component is set to zero to allow the players time to adjust to the controls when the game begins.

Pong Physics

The game pong is based on in-elastic collisions. The velocity of the ball in the game consists of a vertical and horizontal speed component. When the ball makes contact with a surface (paddle or wall), the balls speed will change proportionally. For example if the ball is moving towards the left player paddle and makes contact. The horizontal ball speed will change from negative (left direction) to positive (right direction). When the

ball makes contact with the top wall, the vertical component will change from positive (up) to negative (down).

Paddle Collision

The first major change is the algorithm for paddle collisions with the ball. The original code calculated the angle the ball hit the paddles using sines and cosines. This algorithm is incorrect because the physics of the desired game are violated (ie the ball doesn't bounce properly). To fix this, a constant called "ball_speed_modifier" is defined. This user defined value defines the number of regions on the paddle and affect how sharp of an angle the ball comes off the paddle. The less number of regions the sharper the angle.

Original Code for a paddle collision

```
ball_hspeed = -ball_hspeed;
ball_vspeed = ball_speed * sin(atan(ball_vspeed/ball_hspeed + (ball_y-P1_y)/(paddle_size/2)));
ball_hspeed = (ball_hspeed/abs(ball_hspeed)) * ball_speed * cos(asin(ball_vspeed/ball_speed));
ball_x += ball_hspeed;
```

New Code for a paddle collision

```
int ball_speed_modifier = 8;

ball_vspeed = (ball_y-P1_y)/ball_speed_modifier;
ball_hspeed = -ball_hspeed;
```

In the original code when the ball collides with the paddle, the balls horizontal component is reversed. The vertical component and horizontal component are changed based on the angle of the ball relative to the center of the paddle. The horizontal ball speed is then increased. This code is inefficient because of the number of calculations needed to determine.

The new code simplifies the calculation to basic arithmetic. The vertical ball speed is calculated based on the position of the ball relative to the center of the paddle divided by the ball_speed_modifier. The horizontal ball speed is reversed.

Graphics

The original *Pong* code uses the paddle width for the logic of a paddle collision and for drawing the paddle on screen. This creates a bug in the game where the ball can get trapped between the front and back edges of the paddle. To correct this, the width of the paddle should be set to 1, and the graphics should be scaled to the desired width.

Original Code

```
//Draw Paddles
int paddle_width = 4;

rectfill(buffer, 31, P1_y - paddle_size/2, 31+(paddle_width-1), P1_y + paddle_size/2, C_WHITE);
rectfill(buffer, SCREEN_W - 31, P2_y - paddle_size/2, SCREEN_W - 31 - (paddle_width-1),
P2_y + paddle_size/2, C_WHITE);
```

New Code

```
//Draw Paddles
int paddle_width = 1;

rectfill(buffer, 31, P1_y - paddle_size/2, 31+(paddle_width*4-1), P1_y + paddle_size/2,
C_WHITE);
rectfill(buffer, SCREEN_W - 31, P2_y - paddle_size/2, SCREEN_W - 31 - (paddle_width*4-1),
P2_y + paddle_size/2, C_WHITE);
```

The new code prevents the error from occurring and the graphics look identical to the original code.

Mouse Input

For the *Pong* game to be compatible with a BCI, a continuous signal input is needed to control the paddle. The original game uses discrete keyboard input. To solve this, the vertical and horizontal movement of the mouse is used to control the vertical movement of the paddle.

Original Code

```
//P1 Controls
	if (key[KEY_UP] && P1_y >= paddle_size/2)
	{
		P1_y -= paddle_speed;
	}
	if (key[KEY_DOWN] && P1_y <= (SCREEN_H-paddle_size/2))
	{
		P1_y += paddle_speed;
	}
```

New Code

```
	if(mouse_x > paddle_size/2 && mouse_x < SCREEN_H - paddle_size/2)
	{
		P1_y = mouse_x;
	}
```

Section IV describes the mice modification to translate vertical mouse movement to vertical paddle movement. The movement range of the mice is limited to prevent the cursor from going off screen.

Computer Opponent

The *Pong* game offers a single player mode against a computer opponent. The original opponent tracks the position of the ball up and down when the ball passes three quarters of the screen. If the balls vertical speed component is less than the speed of the paddle, then the computer paddle will make contact with the ball. Otherwise it will miss and the user will score a point.

Original Code

```
	if (ball_hspeed > 0 && ball_x >= (SCREEN_W - SCREEN_W/4 * (ball_hspeed/ball_speed)))
	{
		if (ball_y > (P2_y + paddle_size/2) && P2_y < (SCREEN_H - paddle_size/2))
		{
			P2_y += paddle_speed;
		}
		else if (ball_y < (P2_y - paddle_size/2) && P2_y > paddle_size/2)
		{
			P2_y -= paddle_speed;
		}
	}
```

This implementation of a computer opponent can create problems. If the center of the paddle matches the center of the ball, and the ball has no vertical component, then the game reaches an equilibrium state. Neither player can lose if they don't move their paddle.

A description of the original Atari *Pong* games computer opponent is described below. A variation of this strategy is implemented to prevent equilibrium from occurring.

> The AI moves the paddle to match the vertical position of the ball at any given time, appearing to follow it across the playfield…If the computer simply did this and matched the AI player's paddle position to that of the ball at all times, the result would be even worse than the perfect tic-tac-toe machine. The game wouldn't just be a draw-it would be one in which the human player could never score a single point. To avoid this blunder, Robot Pong's AI slows itself down, never quite following the ball exactly while still appearing to do so.
>
> When the ball is first served, the computer positions the AI paddle so that its top edge is vertically aligned with the ball… To help simulate the human error inherent in precise paddle positioning, the AI paddle skips its vertical adjustment every eight frames. The resulting behavior is visibly unnoticeable, but it allows the computer player's aim to drift enough that it occasionally misses the ball.
>
> (Montfor & Bogost, 2009)

In the new *Pong* code implementation, the computer opponent continuously tracks the ball. The paddle has an initial vertical displacement to prevent equilibrium, and a second displacement occurs when the ball hits the left players paddle. This creates an opponent that constantly tracks the ball, but also emulates human error.

New Code

```
int AI = 4;
int random_direction += (rand() % 2)*2*paddle_size/AI - paddle_size/AI;

if (ball_y > paddle_size/2-1 && ball_y < SCREEN_H - paddle_size/2)
{
```

```
            P2_y = ball_y + random_direction+ paddle_size/AI;
    }
```

The magnitude of the initial displacement and subsequent displacements are based on the AI difficulty level. This AI value defines the minimum number of paddle strikes by the left player required for the computer opponent to make an error.

Appendix C – Code: Trial 1 Fitts' Law Test

```
#include <engine.h>

#define BALL_SIZE (4) //4
#define VERSION "1.0"

int end_game, timer;
BITMAP* buffer;
SAMPLE *bump_bleep, *match_bleep;
FILE *filefitt, *filepong;

char allegro_error[ALLEGRO_ERROR_SIZE];

//Game Vars
static int P1_y, //Center position of Player 1
   P2_y, //Center position of Player 2
   ball_x, ball_y, ball_x2, ball_y2, //Center position of ball
         random_direction, AI,
    score_P1, score_P2,
   time_start, time_stop, time_delta, center_time, //right_player_collision_time,
   MT, A,
   DISTANCEBETWEENBALLS,
   P1_start, P1_stop, P1_delta,
   rally_number, rally_length, paddle_movement, current_p1y, previous_p1y, current_p1_slope,
previous_p1_slope, previous_previous_p1_slope,          start_flag, paddle_flag, counter, // In game
statistics
   ball_speed_modifier;

static float ball_hspeed,
   ball_vspeed;

//Game Options
static int paddle_size,
   paddle_width,
   paddle_speed,
   ball_speed,
   two_player;
static float speed;


//Initialization Functions
void init(int argc, char** argv)
{
   filefitt = fopen("FittS4T00.txt", "w");
   filepong = fopen("PongS4T00.txt", "w");

   end_game = 1; //On any error, get out!
   if (allegro_init() != 0)
   {
         fprintf(stderr,"ALLEGRO ABORTED!\n");
         return;
   }

   char x;
```

```
   int width = 1280,
           height = 960;

   speed = 2.0;
   paddle_width = 1;
   paddle_speed = 3;
   two_player = 0; //Set to 1 if you want 2 player, set to 0 for 1 player

   rally_number = 1;
   rally_length = 0;

   //Variables to Alter
   ball_speed = 3; //3
   AI = 6;    //Change this value to 4, 6, 8 for increasing difficulty of AI
   ball_speed_modifier = 8; //number of regions on paddle ie how much angle the ball gets of the paddle.
The lower the number the more angle (4,6,8) hardest --> easiest

   paddle_size = 96;//Paddle Size
   DISTANCEBETWEENBALLS = 60; // 2 x DISTANCEBETWEENBALLS = A

   while((x = getopt(argc,argv,"w:h:s:p:2H")) != -1)
   {
           switch(x)
           {
                   case 'w': //grid width
                           sscanf(optarg,"%d",&width);
                           break;
                   case 'h': //grid height
                           sscanf(optarg,"%d",&height);
                           break;
                   case 's': //game speed
                           sscanf(optarg,"%f",&speed);
                           break;
                   case 'p': //paddle size
                           sscanf(optarg,"%d", &paddle_size);
                           break;
                   case '2': //2P mode activated
                           two_player = 1;
                           break;
                   case 'H':
                   default:
                           printf("Usage: ./pong-%s [-w <width>] [-h <height>] [-s <speed>] [-p
<paddle_size>] [-2] [-H]\n", VERSION);
                           printf("\t-w W: defines the window width to W (default: 640)\n");
                           printf("\t-h H: defines the window width to H (default: 480)\n");
                           printf("\t-s S: defines the game speed to factor S (default: 1.0)\n");
                           printf("\t-p P: defines the size of the paddle to O (default: 64)\n");
                           printf("\t-2: enables 2-player mode\n");
                           printf("\t-H: prints this help.\n");
                           return;
           }
   }
   //initialize game components
   config(width, height);

   start();
```

```
}

//Game Configuration
//Based upon comandline expressions
void config(int w, int h)
{
   //Keyboard
   //No key repeating: only key presses!
   if (install_keyboard() < 0)
   {
         fprintf(stderr, "Trying to configure keyboard, couldn't: %s", allegro_error);
         return;
   }
   fprintf(stderr, "Keyboard set up succesfully!\n");

   //Sound
   //Yay, Sound! \o/
   if (install_sound(DIGI_AUTODETECT, MIDI_NONE, NULL) < 0)
   {
         fprintf(stderr, "Trying to configure sound, couldn't: %s", allegro_error);
   }
   else
   {
         fprintf(stderr, "Sound card set up succesfully!\n");
   }
   bump_bleep = load_wav("snd/beep.wav");
   match_bleep = load_wav("snd/deedoop.wav");

   init_gfx(w, h);
}

//Initialize graphics
void init_gfx(int w, int h)
{
   set_color_depth(24);
   if (set_gfx_mode(GFX_AUTODETECT_WINDOWED, w, h, 640, 480) != 0)
         printf("Error setting graphic mode\n");
   buffer = create_bitmap(SCREEN_W, SCREEN_H);
}

//Start the game
void start()
{
   //In Game Statistics
   previous_p1y = SCREEN_H/2;
   paddle_movement = 0;
   counter = 0;
   previous_p1_slope = 0;
   previous_previous_p1_slope = 0;
   start_flag = 0;
   paddle_flag = 0;

   //Initiallize

   P1_y = ball_y = SCREEN_H/2- DISTANCEBETWEENBALLS;
   P2_y = SCREEN_H/2 + paddle_size/AI;
```

```
   score_P1 = score_P2 = 0;
   ball_x = 38;
   ball_x2 = 38;
   ball_y2 = SCREEN_H/2 + DISTANCEBETWEENBALLS;
   time_start = 0;
   time_stop = 0;
   time_delta = time_stop-time_start;
   //right_player_collision_time = 0;
   center_time = 0;
   MT = 0; A = 0;
   P1_start = 0;
   P1_stop = 0;
   P1_delta = abs(P1_start - P1_stop);

   //Mouse Control
   install_mouse();
   scare_mouse();//Hide the Cursor on the screen

   srand(time(NULL));

   int possession = 2*(rand()%2)-1;

   fprintf(filepong, "The first ball is %d, (-1 for left player, 1 for right player) \n", possession);
   fprintf(filepong, "Width of Paddle %d, Number of Regions on Paddle %d, ball speed %d \n",
paddle_size, ball_speed_modifier, ball_speed);

   ball_hspeed = 0.0;
   ball_vspeed = 0.0;

   //Timer
   //60 ticks per second
   install_timer();
   install_int(tick, 1);  //1 millisecond per tick

   fprintf(stderr, "Timer routines set up succesfully!\n");

   end_game = 0;
}

//Game Loop Functions

void tick()
{
   ticks++;
}

//User input
void input()
{
   if (keypressed())
   {
            readkey();

            //Quit on Q
            if (key[KEY_Q])
            {
```

```
                end_game = 1;
                return;
        }
    }

    //P1 Controls
        if(mouse_x > paddle_size/2 && mouse_x < SCREEN_H - paddle_size/2){
                P1_y = mouse_x;    //Use the vertical mouse input to control height
        }

        //Movement Tracking
        current_p1y = mouse_x;
        current_p1_slope = current_p1y-previous_p1y;

        if(start_flag == 0 && current_p1_slope != 0) { // if the start_flag is off (ie. 0) and the paddle is
moving
                start_flag = 1; //The paddle is moving
                time_start = ticks;
                P1_start = P1_y;
        }

        if(current_p1_slope == 0){
                counter++;

                if(counter == 10){ //wait 10 milliseconds to register a stop
                        start_flag = 0; //reset the start flag
                        paddle_movement++;
                        //printf("The paddle stopped moving: %d \n", paddle_movement);
                        //fprintf(file, "The paddle stopped moving: %d \n", paddle_movement);

                        time_stop = ticks-10; //subtract 10 milliseconds to compensate for dwell time
                        time_delta = time_stop-time_start;
                        MT += time_delta;

                        P1_stop = P1_y;
                        P1_delta = P1_start - P1_stop;
                        A += P1_delta;
                }

        }else{
                //reset counter
                counter = 0;
        }

        //Update positions
        previous_p1y = current_p1y;
        previous_previous_p1_slope = previous_p1_slope;
        previous_p1_slope = current_p1_slope;

    //P2 Controls
    if (two_player)
    {
        P2_y = mouse_y;

    }
}
```

```
//Game output
void output()
{
   //Clear screen from previous tick
   clear_bitmap(buffer);

   //Draw field
   int i;
   for (i=8; i<SCREEN_H; i+=64)
   {
         rectfill(buffer, SCREEN_W/2 - 1, i, SCREEN_W/2 + 1, i+32, C_WHITE);
   }

   //Draw Paddles
   rectfill(buffer, 31, P1_y - paddle_size/2, 31+(paddle_width*4-1), P1_y + paddle_size/2, C_WHITE);
   rectfill(buffer, SCREEN_W - 31, P2_y - paddle_size/2, SCREEN_W - 31 - (paddle_width*4-1), P2_y +
paddle_size/2, C_WHITE);

   //Draw Ball
   circlefill(buffer, ball_x, ball_y, BALL_SIZE/2, C_WHITE);
   circlefill(buffer, ball_x2, ball_y2, BALL_SIZE/2, C_WHITE);

   //Draw Score
   extern FONT *font;
   textprintf_ex(buffer,font, 64, 4,C_WHITE, -1, "0");
   textprintf_ex(buffer,font, SCREEN_W-64, 4,C_WHITE, -1, "0");

   textprintf_ex(buffer, font, 64, SCREEN_H/2+DISTANCEBETWEENBALLS,C_WHITE, -1, "%d",
score_P2);
   textprintf_ex(buffer, font, 64, SCREEN_H/2-DISTANCEBETWEENBALLS,C_WHITE, -1, "%d",
score_P1);

   //Debugger


   masked_blit(buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);

}

//Game loop processing
void game_loop()
{
   input();

   //check for "restart" and "end" signals
   if (end_game) return;

   if(ball_hspeed < 0){ //ball is going left
         if(ball_x >= SCREEN_W/2 && ball_x <SCREEN_W/2 + abs(ball_hspeed)){
               center_time = ticks;
               fprintf(stderr, "center time %d, ", center_time);
         }
   }
```

```
	//Check if the paddle contains the top target
		if (ball_y >= P1_y - paddle_size/2 && ball_y <= P1_y + paddle_size/2){

			//paddle_flag = is the the active target. O is the top ball, 1 is the bottom ball
			if(start_flag == 0 && paddle_flag == 0){ //if the paddle stopped and the paddle is on the
top

				fprintf(stderr, "The ball paddle has stopped in the top on the top ball \n");
				score_P1++;
				paddle_flag = 1;

				//filefitt

				fprintf(filefitt, "%d, %d, ", P1_y, ball_y); //center of paddle and ball position
				fprintf(filefitt, "%d, ", MT); //Movement Time
				fprintf(filefitt, "%d \n", abs(A)); //Amplitude

				//reset movement time and amplitude
				MT= 0;
				A = 0;
			}
		}

	//Check if the paddle contains the bottom target
	if (ball_y2 >= P1_y - paddle_size/2 && ball_y2 <= P1_y + paddle_size/2){
			if(start_flag == 0 && paddle_flag == 1){

				fprintf(stderr, "The ball paddle has stopped in the bottom on the top ball \n");
				score_P2++;

				paddle_flag = 0;

				fprintf(filefitt, "%d, %d, ", P1_y, ball_y2); //center of paddle and ball position
				fprintf(filefitt, "%d, ", MT); //Movement Time
				fprintf(filefitt, "%d \n", abs(A)); //Amplitude

				//reset movement time and amplitude
				MT= 0;
				A = 0;
			}
		}
	}

	//Right Player Paddle
	if (ball_x >= SCREEN_W-(31+paddle_width*2-1)-abs(ball_hspeed) && ball_x <= SCREEN_W-31)
	{
		//play_sample(bump_bleep,128,128,1000,0);
		if (ball_y >= P2_y - paddle_size/2 && ball_y <= P2_y + paddle_size/2)
		{

			ball_vspeed = (ball_y-P2_y)/ball_speed_modifier;
			ball_hspeed = -ball_hspeed*1.1; //ball speed increase -- collision detection error occurs

			//printf("Paddle 2 Ball Position: ball_x, ball_y, %d, %d \n", ball_x, ball_y);
			//printf("Speed: ball_vspeed, ball_hspeed %f %f \n", ball_vspeed,ball_hspeed);
```

```
                    //fprintf(file, "Paddle 2 Ball Position: ball_x, ball_y, %d, %d \n", ball_x, ball_y);
                    //fprintf(file, "Speed: ball_vspeed, ball_hspeed %f %f \n", ball_vspeed,ball_hspeed);

                    rally_length++;

                    //right_player_collision_time = ticks;
                    //fprintf(filepong, "\t Right Player Paddle Collision time %d \n", ticks);
            }
    }

    //point marking!
    if (ball_x <= 0)
    {
            play_sample(match_bleep,128,128,1000,0);
            score_P2++;
            ball_x = SCREEN_W/2;
            ball_y = SCREEN_H/2;
            ball_hspeed = -1*ball_speed;
            ball_vspeed = 0.0;
            //printf("Point Player 2, Rally #, Rally Length,  %d, %d \n", rally_number, rally_length);
            fprintf(filepong, "Point Player 2, Rally # %d, Rally Length %d, Game Score %d %d, ",
rally_number, rally_length, score_P1, score_P2);
            //fprintf(filepong, "Center of Paddle Position %d, Center of Ball position %d \n", P1_y, ball_y);
            rally_number++; rally_length = 0;
            random_direction = 0;
            //right_player_collision_time = 0;
            center_time = ticks;
            fprintf(stderr, "center time %d, ", center_time);

            if(score_P2 == 11){
                    end_game = 1;
                    return;
            }
    }

    if (ball_x >= SCREEN_W - 4)
    {
            play_sample(match_bleep,128,128,1000,0);
            score_P1++;
            ball_x = SCREEN_W/2;
            ball_y = SCREEN_H/2;
            ball_hspeed = 1*ball_speed;
            ball_vspeed = 0.0;

            fprintf(filepong, "Point Player 1, Rally # %d, Rally Length %d, Game Score:%d %d \n",
rally_number, rally_length, score_P1, score_P2);
            rally_number++; rally_length = 0;
            random_direction = 0;
    }

    output();
}

//Game End Functions
void finish()
{
```

```
    remove_keyboard();
    remove_timer();
    //destroy_sample(bump_bleep);
    //destroy_sample(match_bleep);
    remove_sound();
    fclose(filefitt);
    fclose(filepong);
}
```

Appendix D – Code for Pong

```
#include <engine.h>

#define BALL_SIZE (4) //4
#define VERSION "1.0"

int end_game, timer;
BITMAP* buffer;
SAMPLE *bump_bleep, *match_bleep;
FILE *filefitt, *filepong;

char allegro_error[ALLEGRO_ERROR_SIZE];

//Game Vars
static int P1_y, //Center position of Player 1
   P2_y, //Center position of Player 2
   ball_x, ball_y, //Center position of ball
            random_direction, AI,
     score_P1, score_P2,
   time_start, time_stop, time_delta, center_time, //right_player_collision_time,
   MT, A,
   P1_start, P1_stop, P1_delta,
   rally_number, rally_length, paddle_movement, current_p1y, previous_p1y, current_p1_slope,
previous_p1_slope, previous_previous_p1_slope,                start_flag, counter, // In game
statistics
   ball_speed_modifier;

static float ball_hspeed,
   ball_vspeed;

//Game Options
static int paddle_size,
   paddle_width,
   paddle_speed,
   ball_speed,
   two_player;
static float speed;


//Initialization Functions
void init(int argc, char** argv)
{
   filefitt = fopen("FittS4Paddle24.txt", "w");
   filepong = fopen("PongS4Paddle24.txt", "w");

   end_game = 1; //On any error, get out!
   if (allegro_init() != 0)
   {
            fprintf(stderr,"ALLEGRO ABORTED!\n");
            return;
   }

   char x;
   int width = 1280,
```

```
        height = 960;

    speed = 2.0;
    paddle_width = 1;
    paddle_speed = 3;
    two_player = 0; //0 Set to 1 if you want 2 player, set to 0 for 1 player

    rally_number = 1;
    rally_length = 0;

    //Variables to Alter
    ball_speed = 3; //3
    AI = 6;    //Change this value to 4, 6, 8 for increasing difficulty of AI
    ball_speed_modifier = 8; //number of regions on paddle ie how much angle the ball gets of the
paddle. The lower the number the more angle (4,6,8) hardest --> easiest
    paddle_size = 24;

    while((x = getopt(argc,argv,"w:h:s:p:2H")) != -1)
    {
        switch(x)
        {
            case 'w': //grid width
                sscanf(optarg,"%d",&width);
                break;
            case 'h': //grid height
                sscanf(optarg,"%d",&height);
                break;
            case 's': //game speed
                sscanf(optarg,"%f",&speed);
                break;
            case 'p': //paddle size
                sscanf(optarg,"%d", &paddle_size);
                break;
            case '2': //2P mode activated
                two_player = 1;
                break;
            case 'H':
            default:
                printf("Usage: ./pong-%s [-w <width>] [-h <height>] [-s <speed>] [-p
<paddle_size>] [-2] [-H]\n", VERSION);
                printf("\t-w W: defines the window width to W (default: 640)\n");
                printf("\t-h H: defines the window width to H (default: 480)\n");
                printf("\t-s S: defines the game speed to factor S (default: 1.0)\n");
                printf("\t-p P: defines the size of the paddle to O (default: 64)\n");
                printf("\t-2: enables 2-player mode\n");
                printf("\t-H: prints this help.\n");
                return;
        }
    }
    //initialize game components
    config(width, height);

    start();
}

//Game Configuration
```

```
//Based upon comandline expressions
void config(int w, int h)
{
   //Keyboard
   //No key repeating: only key presses!
   if (install_keyboard() < 0)
   {
         fprintf(stderr, "Trying to configure keyboard, couldn't: %s", allegro_error);
         return;
   }
   fprintf(stderr, "Keyboard set up succesfully!\n");

   //Sound
   //Yay, Sound! \o/
   if (install_sound(DIGI_AUTODETECT, MIDI_NONE, NULL) < 0)
   {
         fprintf(stderr, "Trying to configure sound, couldn't: %s", allegro_error);
   }
   else
   {
         fprintf(stderr, "Sound card set up succesfully!\n");
   }
   bump_bleep = load_wav("snd/beep.wav");
   match_bleep = load_wav("snd/deedoop.wav");

   init_gfx(w, h);
}

//Initialize graphics
void init_gfx(int w, int h)
{
   set_color_depth(24);
   if (set_gfx_mode(GFX_AUTODETECT_WINDOWED, w, h, 640, 480) != 0)
         printf("Error setting graphic mode\n");
   buffer = create_bitmap(SCREEN_W, SCREEN_H);
}

//Start the game
void start()
{
   //In Game Statistics
   previous_p1y = SCREEN_H/2;
   paddle_movement = 0;
   counter = 0;
   previous_p1_slope = 0;
   previous_previous_p1_slope = 0;
   start_flag = 0;

   //Initiallize
   P1_y = ball_y = SCREEN_H/2;
   P2_y = SCREEN_H/2 + paddle_size/AI;
   score_P1 = score_P2 = 0;
   ball_x = SCREEN_W/2;
   time_start = 0;
   time_stop = 0;
   time_delta = time_stop-time_start;
```

```
   center_time = 0;
   MT = 0; A = 0;
   P1_start = 0;
   P1_stop = 0;
   P1_delta = abs(P1_start - P1_stop);

   //Mouse Control
   install_mouse();
   scare_mouse();

   srand(time(NULL));

   int possession = 2*(rand()%2)-1;

   fprintf(filepong, "The first ball is %d, (-1 for left player, 1 for right player) \n", possession);
   fprintf(filepong, "Width of Paddle %d, Number of Regions on Paddle %d, ball speed %d \n",
paddle_size, ball_speed_modifier, ball_speed);

   ball_hspeed = possession*ball_speed;
   ball_vspeed = 0.0;

   //Timer
   install_timer();
   install_int(tick, 1);  //1 millisecond per tick

   fprintf(stderr, "Timer routines set up succesfully!\n");

   end_game = 0;
}

//Game Loop Functions
void tick()
{
   ticks++;
}

//User input
void input(){
   if (keypressed())    {
            readkey();
            //Quit on Q
            if (key[KEY_Q]){
                     end_game = 1;
                     return;
            }
   }

   //P1 Controls
            if(mouse_x > paddle_size/2 && mouse_x < SCREEN_H - paddle_size/2){
                     P1_y = mouse_x;    //Use the vertical mouse input to control height [april 18th
2011]
            }

            //Movement Tracking
            current_p1y = mouse_x;
```

```
        current_p1_slope = current_p1y-previous_p1y;

        if(start_flag == 0 && current_p1_slope != 0) { // if the start_flag is off (ie. 0) and the
paddle is moving
                start_flag = 1; //The paddle is moving
                time_start = ticks;
                P1_start = P1_y;

                fprintf(stderr, "Start Has Happened \n");
        }

        if(current_p1_slope == 0){
                counter++;

                if(counter == 10){ //wait 10 milliseconds to register a stop
                        start_flag = 0; //reset the start flag
                        paddle_movement++;

                        time_stop = ticks-10; //to compensate for dwell time
                        time_delta = time_stop-time_start;
                        MT += time_delta;

                        P1_stop = P1_y;
                        P1_delta = P1_start - P1_stop;
                        A += P1_delta;

                        fprintf(stderr, "Stop has happenned \n");
                }
        }else{
                //reset counter
                counter = 0;
        }

        //Update positions
        previous_p1y = current_p1y;
        previous_previous_p1_slope = previous_p1_slope;
        previous_p1_slope = current_p1_slope;

   //P2 Controls
   if (two_player)
   {
        P2_y = mouse_y;

   }
}

//Game output
void output(){
   //Clear screen from previous tick
   clear_bitmap(buffer);

   //Draw field
   int i;
   for (i=8; i<SCREEN_H; i+=64)
   {
        rectfill(buffer, SCREEN_W/2 - 1, i, SCREEN_W/2 + 1, i+32, C_WHITE);
```

```
    }

    //Draw Paddles
    rectfill(buffer, 31, P1_y - paddle_size/2, 31+(paddle_width*4-1), P1_y + paddle_size/2,
C_WHITE);
    rectfill(buffer, SCREEN_W - 31, P2_y - paddle_size/2, SCREEN_W - 31 - (paddle_width*4-1),
P2_y + paddle_size/2, C_WHITE);

    //Draw Ball
    circlefill(buffer, ball_x, ball_y, BALL_SIZE/2, C_WHITE);

    //Draw Score
    extern FONT *font;
    textprintf_ex(buffer,font, 64, 4,C_WHITE, -1, "%d", score_P1);
    textprintf_ex(buffer,font, SCREEN_W-64, 4,C_WHITE, -1, "%d", score_P2);

    //Debugger
    masked_blit(buffer, screen, 0, 0, 0, 0, SCREEN_W, SCREEN_H);
}

//Game loop processing
void game_loop(){
    input();

    //check for "restart" and "end" signals
    if (end_game) return;

    if(ball_hspeed < 0){ //ball is going left
            if(ball_x >= SCREEN_W/2 && ball_x <SCREEN_W/2 + abs(ball_hspeed)){
                    center_time = ticks;
                    fprintf(stderr, "center time %d, ", center_time);
            }
    }

    //Check collision with walls
    if (ball_y <= BALL_SIZE || ball_y >= SCREEN_H - BALL_SIZE){
            ball_vspeed = -1*ball_vspeed;
    }

    //collision with paddles
    //Left Player Paddle
    if((ball_x <= (31+paddle_width*2-1+abs(ball_hspeed)) && ball_x >= 31)){
            //if the ball is between the top and bottom of the paddle
            if (ball_y >= P1_y - paddle_size/2 && ball_y <= P1_y + paddle_size/2)
            {
                    ball_vspeed = (ball_y-P1_y)/ball_speed_modifier;
                    ball_hspeed = -ball_hspeed*1.1;
                    random_direction += (rand() % 2)*2*paddle_size/AI - paddle_size/AI; // For
AI: To prevent equilibrium
                    rally_length++;

                    if(start_flag == 1) {

                            fprintf(stderr, "Ball Has crossed the plane wall Paddle Still Moving
\n");
                            start_flag = 0; //The paddle has stopped
```

```
                        time_stop = ticks-10;
                        time_delta = time_stop-time_start;
                        MT += time_delta; // need to print this in paddle collision, reset MT =
0;

                        P1_stop = P1_y;
                        P1_delta = P1_start - P1_stop;
                        A += P1_delta;
                }
        }

                fprintf(filefitt, "%d, ", ticks); //left paddle collision time
                fprintf(filefitt, "%d, ", center_time); //ball heading left and crossing the center
                fprintf(filefitt, "%d, %d, ", P1_y, ball_y); //center of paddle and ball position
                fprintf(filefitt, "%d, ", MT); //Movement Time
                fprintf(filefitt, "%d \n", abs(A)); //Amplitude

                //reset movement time and amplitude
                MT= 0;
                A = 0;

   }

   //Right Player Paddle
   if (ball_x >= SCREEN_W-(31+paddle_width*2-1)-abs(ball_hspeed) && ball_x <=
SCREEN_W-31)
   {
        //play_sample(bump_bleep,128,128,1000,0);
        if (ball_y >= P2_y - paddle_size/2 && ball_y <= P2_y + paddle_size/2)
        {

                ball_vspeed = (ball_y-P2_y)/ball_speed_modifier;
                ball_hspeed = -ball_hspeed*1.1; //ball speed increase -- collision detection error
occurs

                rally_length++;

        }
   }

   //point marking!
   if (ball_x <= 0)
   {
        play_sample(match_bleep,128,128,1000,0);
        score_P2++;
        ball_x = SCREEN_W/2;
        ball_y = SCREEN_H/2;
        ball_hspeed = -1*ball_speed;
        ball_vspeed = 0.0;
        fprintf(filepong, " 2,  %d,  %d,  %d, %d \n ", rally_number, rally_length, score_P1,
score_P2);
        rally_number++; rally_length = 0;
        random_direction = 0;
        center_time = ticks;
        fprintf(stderr, "center time %d, ", center_time);
```

```
            if(start_flag == 1){
                start_flag == 0; //If the paddle is moving when you get scored on.
                fprintf(stderr, "Got scored on, reset the paddle to not moving");
            }

            if(score_P2 == 11){
                end_game = 1;
                return;
            }
    }

    if (ball_x >= SCREEN_W - 4){
            play_sample(match_bleep,128,128,1000,0);
            score_P1++;
            ball_x = SCREEN_W/2;
            ball_y = SCREEN_H/2;
            ball_hspeed = 1*ball_speed;
            ball_vspeed = 0.0;

            fprintf(filepong, "1, %d,  %d, %d, %d \n", rally_number, rally_length, score_P1,
score_P2);
            rally_number++; rally_length = 0;
            random_direction = 0;

            if(start_flag == 1){
                start_flag == 0; //If the paddle is moving when you get scored on.
                fprintf(stderr, "You Scored, reset the paddle to not moving");
            }
    }

    if (!two_player)
    {
            if(random_direction > 0){
                if (ball_y > paddle_size/2-1  && ball_y < SCREEN_H - paddle_size/2
+abs(random_direction))
                {
                    P2_y = ball_y + random_direction+ paddle_size/AI;
                }
            }else{
                if (ball_y > paddle_size/2-1 + abs(random_direction)  && ball_y <
SCREEN_H - paddle_size/2)
                {
                    P2_y = ball_y + random_direction+ paddle_size/AI;
                }
            }
    }

    //Move ball
    ball_x += (int) ball_hspeed; //cast float to int to stop
    ball_y += (int) ball_vspeed;

    output();
}

//Game End Functions
```

```
void finish()
{
   remove_keyboard();
   remove_timer();
   //destroy_sample(bump_bleep);
   //destroy_sample(match_bleep);
   remove_sound();
   fclose(filefitt);
   fclose(filepong);
}
```