Dynamic Feedback for Execution Errors of Fighting Game Motion Inputs

Aidan R. Goroway

March 17, 2024

Abstract

In competitive fighting games, mastery of your character's special moves is imperative to playing well. Very few fighting games if any will actually give the player feedback during the learning process, beyond telling the player when they are doing something right or wrong. I intend to create a tool that will provide the user with dynamic feedback on their mistakes with motion inputs in order to better communicate how they can rectify these mistakes.

Contents

List of Figures

List of Tables

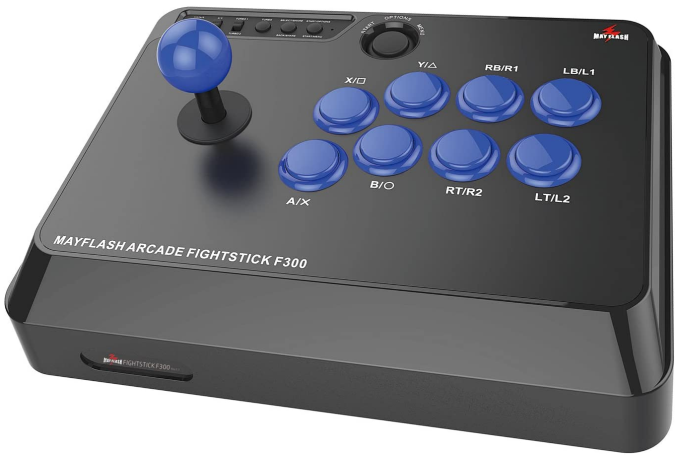


Figure 1: A fightstick. Note the many buttons and singular joystick.
Source: https://www.amazon.com/Mayflash-F300-Arcade-Joystick-Switch/dp/B019MFPLC0?th=1

1 Introduction

The traditional fighting game genre, a source of competitive entertainment and enjoyment for fans across the globe, has existed since the 1970s. What defines a traditional fighting game is not set in stone, but there are a few basic attributes that most of them have. It is worth noting that not all games are the same, so there is some variance in these qualifiers, depending on the game in question. Nearly all traditional fighting games pit one player against another, with each player controlling a character in a 2D environment, both visible on the same screen. The main objective of the game for each player is to lower the health of the opponent, until there is none left. Each player accomplishes that by hitting the other player's character with attack-moves, often just called "moves", from their own character. While the basic mechanics of a game's combat vary more than anything else, most traditional fighting games give the players the ability to move back and forth, jump, block, and of course, attack in many different ways. Originating as far back as the 1970s, then finally gaining mainstream success from arcades during the early 1990s, widespread distribution and popularity has led to several evolutions within the genre, and in return a sprawling tree of different fighting games to play. Modern fighting games allow the user a bevy of options on what controller to use, with one of the more widely adopted being the fightstick. An example of one can be seen in Figure [1]. While they vary according to make, all fightsticks contain a joystick (similar to the ones found on arcade cabinets), as well as an array of buttons. If you asked someone mostly unfamiliar with the genre (yet still aware of it) what they knew about fighting games, chances are they would say the *Hadouken*.

The Hadouken is a special move shared by Ryu and Ken (from Capcom's *Street Fighter* series) that allows either to throw a slow moving fireball across the screen at their opponent. Special moves are more powerful than normal moves, but are more difficult to execute. Normal moves can be used just by pressing

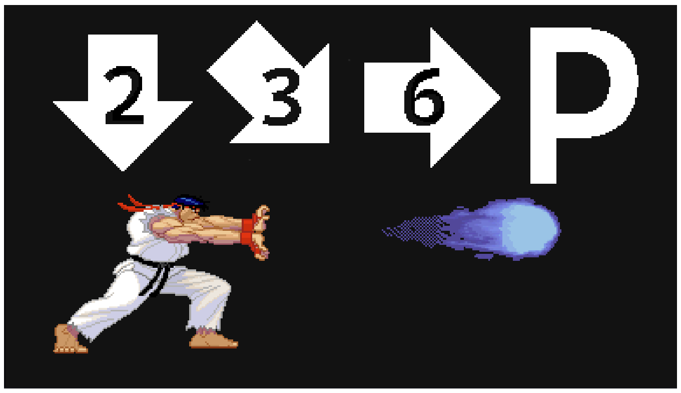
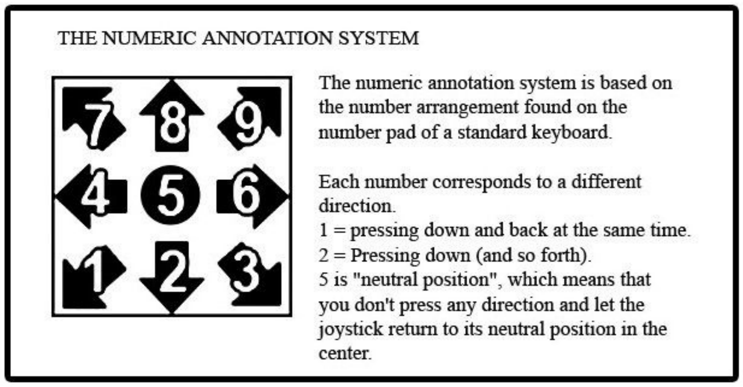


Figure 2: Ryu throwing the iconic Hadouken, one of his special moves. It is likely the most famous special move in fighting game history. 236P is the Numeric Annotation for the Hadouken.
Source (Image modified from): https://wiki.supercombo.gg/w/File:(ryuhdk).gif

a button on your controller by itself. In contrast, special moves typically require the player to move the joystick in a specified sequence before hitting a button, with the success of their special move depending on how accurate the player's stick movement was to the attack's specifications. This specification of the stick's movement can be referred to as a "motion input". More specifically, a motion input could be defined as a series of directional inputs performed with the joystick, followed by the press of one or more buttons that results in a special move [2]. Motion inputs are commonplace within the fighting game genre, and an impossibly large amount of them exist overall. Motion inputs are a staple of the fighting game genre, and every playable character has several. Ryu's Hadouken is likely the single most ubiquitous example of a motion input in video game history. Mastery of these inputs in a timely and consistent manner is an essential skill for anyone who wishes to achieve any level of success in a fighting game of their choosing. Mastery of these inputs is also a notable roadblock for most new players, and a cause of major frustration to them as well.

On a surface level, motion inputs are easy to understand, separated from their actual execution at breakneck pace during a match. In fighting games, motion inputs can follow many different styles of notation. The one I believe to be the most intuitive and the one I chose to use for this project is the Numeric Annotation System [3]. Because it resembles the keypad on the right side of a keyboard, it is sometimes also called Numpad Notation. When the joystick is in neutral position, untouched, it rests at "5". Moving the stick in any of the eight directions from neutral signals the corresponding number; 2 being down, 8 being up, 6 being forward (assuming the player's character faces right), and the like. Any alphabetical characters found in an input's annotation represents a button press. Compared to the static 1-9 of the stick, the specifics of a button press (almost always an attack) vary greatly across different games. The *Street Fighter* series uses 6 attack buttons, notated as LP (Light Punch), MP (Medium Punch), HP (Hard Punch), LK (Light Kick), MK


THE NUMERIC ANNOTATION SYSTEM

The numeric annotation system is based on the number arrangement found on the number pad of a standard keyboard.

Each number corresponds to a different direction.
1 = pressing down and back at the same time.
2 = Pressing down (and so forth).
5 is "neutral position", which means that you don't press any direction and let the joystick return to its neutral position in the center.

Figure 3: A representation of the Numeric Annotation System.
Source (Image modified from): https://wiki.supercombo.gg/w/File:NumpadNotation.jpg

(Medium Kick), and HK (Hard Kick). The *Guilty Gear* series uses P (Punch), K (Kick), S (Slash), H (Heavy Slash), and D (Dust). Every fighting game has its own variants on how attack buttons are notated.

Ryu's iconic Hadouken involves a quarter circle forwards, followed by a punch of any variety. A quarter circle starts downwards (2), passes through the intercardinal "3" position on its way forwards, ending in the forward position (6) with a punch (P). In Numeric Annotation, this reads as 236P. The button press (P in this example) is what signals the inputs as the execution of a *Hadouken*, any type of punch will do, be it LP, MP, or HP. Every motion input requires a button press at the end, in order for the game to distinguish the stick movements as execution of a special move, and not just the player's attempts to move their character. A more complex input would be Potemkin's *Potemkin Buster* from *GUILTY GEAR -STRIVE-*: 632146P. This input starts forwards (6), halfcircles back (63214), and then ends in front (6) with a P. Despite the fact that the player will cross over the neutral position (5), it is not listed in most inputs because it is usually just a means to an end. The neutral position just so happens to rest inbetween two more important positions. The exception to this is when an input only uses the neutral position, like 5K or 5D.

Another one of Potemkin's special moves is *Hammerfall*, a charge input, which operates on a different set of rules. Charge inputs require the player to hold a direction (usually downwards or backwards) for a set amount of time before performing the rest of the actions needed for the input. *Please note that the following usages of square brackets are not intended to be citations, despite sharing a similar style of notation to them.* The Hammerfall input, [4]6H, requires the player to hold backwards [4] for a short period of time, before moving the stick in the opposite direction with an H (6H). A charge input will always have the required hold direction in square brackets [ ].

Most new players will probably be able to perform a simple input consistently after only a few minutes of isolated practice. This practice typically comes from a part of the game called Training Mode or the

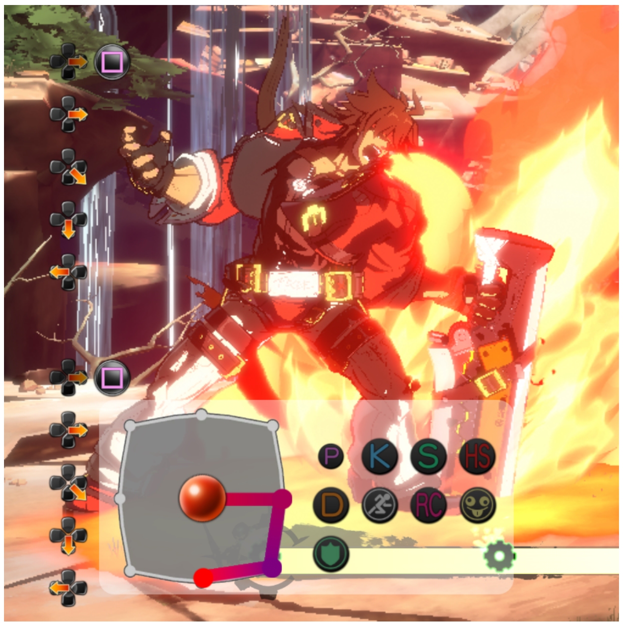


Figure 4: An example of what your previous inputs and stick position may look like. In this example, the player performs Sol Badguy's special move *Gunflame*, which happens to share the Hadouken's input: 236P. On the lefthand side are the player's previous inputs, with the ones closest to the top of the screen being the most recent. In the grey box is the current position of the stick, with the red lines representing the path the stick took to reach its current location, the neutral position.
Source: Image captured from GUILTY GEAR -STRIVE-

Training Room. In this mode, instead of another player, the opponent is faced with a still opponent, giving the player the opportunity to practice their motion inputs and other things for as long as they want. Outside of the training room, the luxury of isolated practice is not afforded, and execution speed is invaluable. When thrown into a real match, those same new players will likely be placed under pressure to input their moves faster, often leading to involuntary shortcutting. An example of this could be seen in the following scenario: your opponent approaches you, aiming to attack you. You, playing as Ryu, attempt to throw a Hadouken at them as soon as you notice their intent. In your rush to input the move as fast as possible, you don't actually push the stick in a downward direction, skipping to the rest of the move. The resulting input is 36P (instead of the intended 236P), which only registers the punch at the end, doing nothing to stop your opponent from beginning their assault on your character. The player is understandably confused, unable to fathom what they did incorrectly in the heat of the moment. This only leads to frustration. That frustration is what I aim to eliminate.

In the modern era of fighting games, it is somewhat common to allow the player to view their inputs during training mode [4], or when viewing saved footage of their previous matches, both standard features in the genre. While this is an excellent feature, it does not actually tell the player what they did wrong in that precise moment, only allowing them to view their mistakes as many times as they please. A tool that could provide immediate feedback on input mistakes, along with the necessary fix for them, would be a total game changer in the literal sense. I intend for the feedback to be given in training mode, so the faster it can be generated and shown to the player, the better. I myself still have difficulties with motion inputs in the various fighting games that I play on my own time. I noticed that while I could perform them somewhat consistently in training mode, my efforts often fell apart in actual combat. I could only reasonably guess what my mistakes were, and was never able to actively correct them in subsequent attempts within the same match. After this happened enough times, a question appeared in my head. **Could a tool that compared player motion inputs in fighting games to their ideal execution give helpful feedback on potential mistakes?**

2 Related Work

The act of comparing fighting game motion inputs to each other is not actually a new concept. The subject of handwriting comparison (as described in a paper by Mulgrew [3] that I'll go into more detail about later) is notably similar to the comparison of various input patterns. In the context of writing, the joystick could be considered analogous to any writing utensil. With that comparison in mind, a given motion input isn't very different to something like a simple japanese character, or some geometric shape. Taking the visualized path of the joystick would result in a series of straight lines. Examining an input's Numeric Annotation would provide a string of numbers and at least one letter. When performing an input, the visualized path of the joystick and the series of numbers and letters that it represents can both be compared against an ideal execution of the input.

Going over the visually based method of comparison first, there are several papers that specialize in comparing visual structures (be it geometrical shapes or handwritten characters) against each other. A research paper written by Mulgrew [3] in 2022 demonstrated the potential of various timings for giving feedback to students learning how to write Japanese Kanji characters. Despite building off of a background of Kanji recognition, the report was mainly focussed on the best time to provide students with feedback on their writing. While not statistically significant, it was determined that students who received immediate corrective feedback on each step of the process performed better and were overall less frustrated. Due to fundamental differences between fighting game inputs and Japanese Kanji characters, I will be giving corrective feedback after each input is concluded, rather than after each step. A Kanji character's steps are each integral to the final product, and you can practice each one individually to various degrees of success. In contrast, the "steps" of an input are too small to be practiced individually in the context of learning a single motion input, each step just being the movement of the stick into different positions.

A paper written by Joshi et al. [1] in 2004 also approaches the topic of alphabetical character recognition, though with much more emphasis on the technical details. Their paper was based on the Indian language of Tamil, which contains 156 commonly used characters, with many of them bearing notable similarity to each other. They found that there were schemes that could strongly and quickly identify Tamil characters (up to 32.6 characters a second with 95.9% accuracy.) One key difference between this study and my own is that Tamil characters are extremely curvy in comparison to the line-based structures that would form when stringing together the steps of a motion input. Another difference is that while there are nearly infinite ways to draw a character (which is what necessitates recognition algorithms in the first place), there are a finite number of input patterns that can be created. This is because of the finite nature of the inputs we are comparing; Annotation strings only consist of 1-9 and a few letters, and visual representations of the stick can only form simple linear structures. The reason for this is that there are only nine positions the stick can be in, so all inputs can be represented as lines. It is for these reasons that I cannot in confidence follow the procedures outlined in this paper.

Another paper, this one from 2002 by Moon et al. [2] makes a push for optimizing edge-based shape detection. The system they developed is best suited for geometric shapes which more closely resemble the structures formed by input patterns. While a similar approach to theirs could certainly be used with my own project, what I require is far less complicated. While I could use an algorithm created to detect line patterns formed by inputs, those lines are just formed by visually representing the Numeric Annotation of the inputs. An image recognition algorithm for a created visual representation is unnecessary if we can parse the same information out of the inputs themselves.

That is why an Edit Distance algorithm would seem to be the best choice for my project. Edit Distance is a type of algorithm that examines two strings and determines how dissimilar one is to the other. This is extremely relevant due to the fact that all inputs can be broken down into strings constructed of numbers and letters. There are multiple types of Edit Distance algorithms, but I believed the optimal choice to be the Levenshtein Distance algorithm. It can delete, insert, and substitute characters, acting as one of the more robust algorithms of the Edit Distance variety. Originally, I planned to make direct use of an Edit Distance algorithm, mainly a variant from a 2017 paper by Zhang et al. [4] that further improved it by reducing its space complexity. The reason I opted away from directly using an Edit Distance algorithm is because the measurable difference between two inputs is not relevant to my project. Being able to quantify the mistakes between the inputs, even being able to replace those mistakes in the player's input in order to recreate the ideal input, ultimately does not tell us anything about why they were made in the first place.

Despite the fact that I am not directly using an Edit Distance algorithm, the fact remains that it is still very relevant to my project conceptually. When an Edit Distance algorithm takes two strings for comparison, the actions it takes to get the “distance” between those strings can be separated into different cases. Specifically, the removal, addition, or replacement of each given character in one string needed for it to become identical to the other string. This key idea of separating the various characters of a string into different cases for further analysis and action is one that will become very relevant later. Specifically, relevant to determining what type of error an input has, and how it is given feedback.

3 Methodology

3.1 Approach

In order to properly gauge the success or failure of a player's input, it must be compared against an input that is entirely accurate. This is actually very easy to achieve; all we need to do is provide the tool with the Numeric Annotation of both the ideal input and the player's input. In the context of an experiment (which I will describe in more detail later), the ideal inputs would be prepared ahead of time on an automated list, and the player would be tasked with replicating them. In normal use, the user could provide the tool with the Numeric Annotation of the input they wished to compare theirs against. After each player input, the tool would provide feedback based on any discrepancies between the player input and the ideal input.

In terms of displaying information, I would have liked to have the tool give the [5] following: The ideal input (in Numeric Annotation), the player's input, a graphical method of communicating the difference between the two, and the feedback we are giving the player in text form. The was my original idea for what the tool would show the player, before I had started implementation. While most of these informational pieces made it into the final project, some did not, and a few things were added. I will go into more detail about the implementation in the next section.

I planned for the tool to be robust enough to handle all types of fighting game inputs. This would include the several standard patterns that are commonplace in the genre, in addition to less common ones. It is probably impossible to catalog every single input ever created, though essentially all inputs follow one of these patterns. Examples described thus far have included a quarter-circle, an input containing a half-circle, and a charge input. Generally most inputs use a combination of these distinct patterns, making their structure fairly predictable. There do exist other more niche inputs like 360 inputs (632147896), or the ”22” inputs that show up in French Bread's *Melty Blood* series of fighting games.

In an effort to encompass all possible types of inputs, there are several types of player input mistakes that may occur. A non-exhaustive list of just some of possible mistakes exemplified in Table [1] would include:

Mistake	Player Input	Ideal Input
1.	36P	236P
2.	632K	6323K
3.	632147896K	632146K
4.	632141236P	632146P
5.	214126P	2141236P
6.	23623K	236236K
7.	41236P	41236K
8.	46P	[4]6P

Table 1: A table showing examples of the different mistakes a player could make, and what the ideal version of their input would look like.

1. Missing the start of an input but correctly performing the rest of it.
2. Starting the input correctly but making a mistake in ending it.
3. Hitting an upward direction when moving the stick left or right.
4. Hitting a downward direction when moving the stick left or right.
5. Starting and ending the input in the right position, but skipping a step in the middle.
6. Performing the input motions correctly, but pressing the final "attack" button at the wrong time.
7. Performing the input motions correctly, but pressing the incorrect attack button.
8. Releasing the charge too early (in the case of charge inputs.)

Many of these input errors can be boiled down to missing a direction somewhere in the input, which will be determined by the existence of missing or additional "steps" in the input that can be detected and have feedback provided for.

Due to the fundamental differences between the characters of certain languages and motion inputs, the choice of when to provide feedback is considerably different from the Kanji program referenced in the Related Work section. That study found that feedback was most effectively given after each individual step of the process of writing any one Kanji character. This works because of the nature of Kanji; that being that each individual stroke has its own eccentricities, and there are several ways to fail each individual one. In fighting games, it is the opposite case. Every part of an input is equally important to the special move, and there is no way to "fail" a single step in isolation: you either put the stick in a position or you don't. Failure can only occur in the context of an input. It is because of this that the best time to provide the player with feedback is after the input is complete. In the vast majority of cases, an input ends with an attack, so the

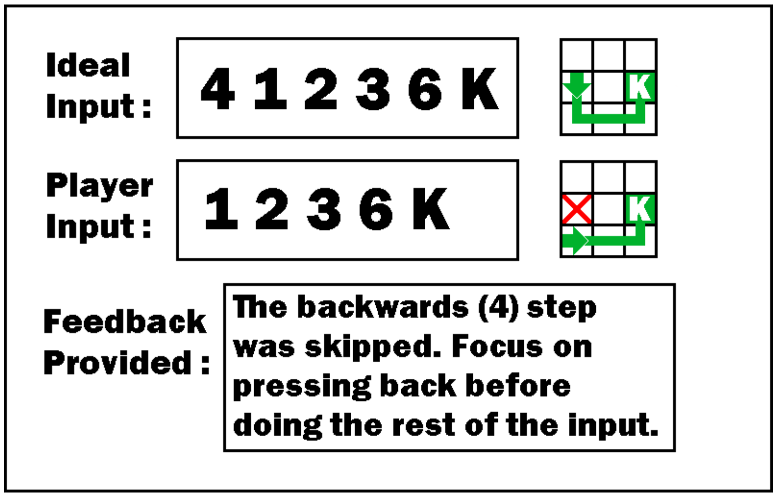

Ideal Input :
4 1 2 3 6 K
K
Player Input :
1 2 3 6 K
K
Feedback Provided :
The backwards (4) step was skipped. Focus on pressing back before doing the rest of the input.

Figure 5: A rough mock-up of what the provided feedback was planned to look like, early on in the project.

tool will be able to understand that a player has intended their input to come to an end, in the cases where mistakes mask their intent.

The feedback provided is the essence of the tool, and for each type of mistake possible, there can be several types of feedback to give to the player depending on the context. If the player clearly misses the start or end of the input, they would be told the specific direction they missed, and to focus specifically on rectifying that mistake. If they accidentally hit the up direction, the system will point out that they should avoid moving their control stick upwards. It is also possible to press the wrong button, which is easily communicated to the player in a similar fashion. Charge inputs operate on different rules, so the feedback for those would be different. In addition to any of the previously mentioned mistakes, it is also possible to fail a charge input by releasing the charge too early. In this case, the player would be told to wait longer before releasing the input. In essentially all cases, it is not possible to hold a charge input too long, so that is not an issue. Because the required charge length is different for each move with this step, more specific advice based on the timing cannot be provided without additional context from the player. Similarly, it is possible to press the right button at the wrong time. While it is possible to press it too late, the rules about this differ on a per-game basis, so it is not easily identified. It is much easier to identify an early button press because in most cases, if all other parts of an input are performed correctly, the button press will execute the special move. If an earlier part of the input is failed (or done in the wrong order), the button press will not execute the move. In this case, the player will most likely be told to focus on finishing the motion steps first before hitting the button. Regardless of specifics, feedback would be provided onscreen in the form of text.

The main reason it is even possible to give feedback on such a wide bevy of possible motion inputs is the

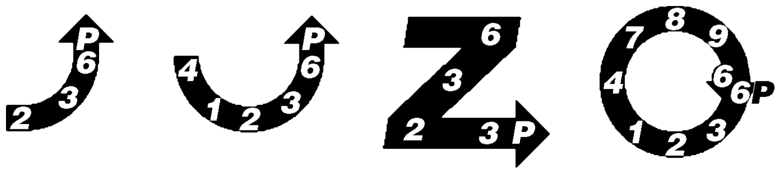


Figure 6: A few archetypes varying in commonality.
From left to right: quarter-circle forwards, half-circle forwards, Z-input, 360-input.
These archetypes assume the player character is facing right.

existence of the "archetype" [6]. An archetype is a term I have created myself, and it pertains to the makeup of an input. Specifically, an archetype could be defined as a common pattern of stick movement that can appear in a motion input. Due to the previously established limitations of the control stick, there is a finite number of reasonable and standardized patterns for motion inputs across the genre. The 236P example from earlier would be a forwards quarter-circle, because the shape the input traces going from 2 to 3 to 6 resembles a quarter of a circle, moving forwards (from the downwards position.) The 632146P example contains the numbers "63214", which would be known as a backwards half-circle, following a similar logic. The vast majority of all motion inputs contain at least one archetype. Performing an archetype will move the stick in such a way that it will always border the "gate" containing the control stick. The gate is the part of the fightstick that physically houses the stick and allows it to specifically only move in 8 directions from neutral. This is to make the performance of the archetype more consistent for the player. However, this also means that the player is given no leeway when performing an archetype. If the characters of an archetype performed by the player are not identical to the characters of an ideal input, it is not correct.

An issue that is showing up with increasing frequency is the fact that different fighting games (and sometimes different moves within the same game) have different rules on the subject of timing their steps. It may be impossible to create a tool that can be used in conjunction with every known fighting game without numerous issues popping up. My solution to this problem was to build a program to test the tool. The program is not a full fighting game itself. All it needs to achieve is allow the user to control a character that can perform special moves with motion inputs. Because I have designed this program from the ground up, I was able to design the timing rules of the tool around the specifics of the program. The program allows the player to perform any of the normal motions and actions any fighting game would allow, in order to let the player make any of the same mistakes that come with input failure during a real match. The program also includes the modern fighting game convention of allowing the player to see the current position of their stick. Originally, I also wanted to allow the player to see their previous input history as well, but I was unable implement that feature in the final project.

When creating the feedback tool and the testing program, I ultimately decided to merge them both into the same program. As I was deciding how I wanted to build my program, I had a few important considerations in mind. First, the program needed to be able to read information from a controller, specifically a fightstick. The program also needed to be able to display information similarly to a GUI (Graphical User Interface.) Based on these criteria, I decided to make the program in the Unity game engine, having it mimic a fighting game. A game engine is a program specifically designed for creating video games. The specifics of how I implemented my methodology into Unity will be discussed in the next section of this report.

It should be noted that there are some potential flaws and limitations with my approach. I have already mentioned that it is impossible to have an extensive catalog of all fighting game inputs aside from what's standard (or what I am personally knowledgeable about), which can potentially limit the use of the tool in cases of exceptions. There is also the matter of how robust the mistake detection can be. While inputs with one small mistake may be easily corrected, the farther the performed input is from the ideal input, the more complicated the mistake is, and the harder pinpointing the mistake to provide feedback becomes. I mentioned earlier that the atmosphere of the training room (where I intend this tool to be used) and a real match are completely different. In my personal experience, this difference has a tangible effect on the success rate of inputs, which could be seen as a potentially huge limitation to the tool's effectiveness. While there is technically nothing stopping a user from using the tool during a real match, the tool would be constantly picking up new inputs and generating irrelevant feedback, giving it no real effectiveness.

3.2 Implementation

As I stated in the last section, I opted to use the Unity game engine for my program. While Unity is a pre-established engine with its own functionality and features, it does still require large amounts of code, especially for larger projects. Unity specifically uses the programming language C# for its scripts. In this report I will not be going into specific detail about the code I have written, but instead describing the basic implementation, as to be applicable to multiple different engines and languages.

Before I can discuss the implementation at length, I must first detail some basic facts about the structure of a script meant to run on a game engine. An important fact pertaining to many game engines, including Unity, is the existence of the "game loop". The game loop, oddly enough, is not actually a loop in the same sense that a while loop is. It is closer to a feature, as it is built into most game engines. A game loop runs constantly while the game itself runs, only stopping when the game is no longer running. How fast the game loop actually runs is not relevant, but how exactly that speed affects the program in reality is important. Specifically for my project, each loop of the game loop is supposed to represent a frame. A frame is a unit of measurement, and for the purpose of this project, it is a sixtieth of a second: The vast majority of fighting games that exist run at 60 frames a second (fps.) Every frame, the game loop loops again and the game updates itself with the new information gleaned from the latest iteration of the loop, which may include the updating of variables and player controller inputs. This is reflected to the player by the screen changing as the game is played. In the case of Unity, there are a few built-in functions that interact with the game loop. Of most importance is the *Update* function, which is where most game functions meant to be looped are called. *FixedUpdate* also exists, and is reserved for functions that more specifically rely on running at an exact fixed rate (like timers.) The game loop's unique property of running near constantly has a few adverse effects on the way code is written, though that is mostly irrelevant to this report specifically. While the game loop covers a significant amount of the code, there are obviously cases where code is written outside of it. The code not written in the game loop is what comes first, and what I will start with.

At the start of most scripts will be the necessary library imports, followed by any initializations. Initialization plays a very important role in this project. Because the game loop is called on repeat, variables defined and initialized inside it are re-initialized every time the loop runs. In order to bypass this, you can initialize variables before the loop starts, outside of it. In order to make sure none of my variables were reinitialized by mistake, essentially all of my variables that were not intended to be strictly local in their scope are initialized in this way. This includes variables related to animations and inputs, as well as important fields to game processes like animators and renderers. The result of this way of initialization is that variables like strings can be continuously appended to across game loops without their content being lost. An important fact to note is that while this is not strictly required for all variables, I chose to implement most variables this way in order to avoid the unnecessary risk of initializing a variable in a place where the code may unintentionally be reached again. After the initializations, the first function is called. *Awake* is specifically designed to be called once on script startup, but the only functions within it are relevant game processes like the previously mentioned animators and asset renderers. Along with *Awake*, there is also the *Start* function. It behaves very similarly to *Awake*, being called once– this time before the first frame rather than on script startup. In practice this is almost identical to *Awake*. The only things to be found in Start are lines that enable some of the assets instantiated in *Awake*. The next function called is *Update*, which acts as the game loop. I will go over the functions called within *Update* later in this section. First, I need to describe what these functions are, and what purpose they serve.

The first function is one that determines what direction the player is "moving" in, according to the Numeric Annotation System. How *takeDirectionalInputs* works is that it takes an input from the player's controller, specifically the joystick. The joystick can be mapped to the horizontal and vertical axes. Each axis of the joystick can be measured from -1 to 1, a representation of the maximum value on both sides, with a middle of 0. While it is possible to have the measurement register a wide range of numbers within

Raw Input	Parsed Input	Ideal Input
555555522222333333336666P	5236P	236P
5555555556666666333222111144444K	563214K	63214K
55555555555P	5P	5P
5555554446666666K	546K	[4]6K

Table 2: A table showing a few examples of the differences between a raw input and what that input looks like after *parsePlayerInput* removes its duplicate characters.
An important detail to note is that the raw inputs would be much longer in practice. These examples are shortened for the sake of presentation.
The discrepancy between the lack of a 5 in most ideal inputs and the occurrence of it in the parsed input is explained by the handling of a different function, detailed later. How charge inputs (the bottom example) are handled will also be explained by a different function, detailed later.

the bounds of -1 to 1, representing small directional increments of the stick on an axis, I chose to accept a reading that is either -1, 0 or 1 only. By comparing the measurements of the joystick on both axes, it is possible to return a number representative of the player's current direction in accordance with the Numeric Annotation System; an integer 1 through 9.

Next is *takeButtonInputs*, a simple function to tell if the player hits a button. In order to simplify the experiment, my project only registers two distinct buttons: P and K. The function checks to see if either of the buttons on the controller are pressed, and returns the one that is, or an empty string otherwise.

The next function after that, *takeUnparssedInput*, uses the prior two functions in order to create a "raw" input. I have termed this input a raw one because it is messy, long, filled with imperfections, and generally does not resemble what an actual input would look like with Numeric Annotation. How it works is that the player's directional input (determined by the stick) and their button input (which is nothing until a button is pressed) are taken as parameters. As long as a button is not pressed, the player's input is updated with the number pertaining to their current stick position. The function returns nothing as long as there is no button pressed. When a button finally is pressed, the enormous raw input is returned for the next function.

The purpose of the next function, *parsePlayerInput*, is to take the raw input and condense it into a form that somewhat resembles a proper input. I call this process "parsing", as the function name makes reference to. To do this, all the function needs is the raw input. This is necessary because of how the previous function, *takeUnparssedInput*, works in a game loop. Because it is called in every single loop on repeat the entire time the program is running, the input it is being appended to constantly grows. The end result is that the raw input is huge, and often filled with repeat numbers [2]. *parsePlayerInput* takes that huge input and deletes the duplicates until there are no duplicates in a row.

The input returned from *parsePlayerInput* resembles the final input we will be comparing against the ideal input, with one key difference. This input still has the number 5 in it. As mentioned earlier in this report, most inputs generally don't include a 5, because it is treated as a "neutral" position. That is what *removeFives* accomplishes. This function takes the input we just parsed, now without its duplicates. It also takes the ideal input again. Like the last function, the ideal input is only required for the sake of checking a condition. Specifically, if the ideal input does happen to use a 5 in its notation. Inputs like 5P or 5K only use a neutral 5 in addition to the attack button, and so the 5 is not to be removed. If this is not the case, all instances of the number 5 are removed from the input, and it is finally ready for comparison.

There is also another discrepancy between some ideal and player inputs: the inclusion of square brackets, like [ ]. As stated earlier, these denote charge inputs. After *parsePlayerInput* and *removeFives*, charge inputs are specifically fed to *handleChargeInputs*. This function first checks to see if the direction meant to be charged has been held for long enough, referencing a helper-function called elsewhere, known as *chargeTimer*. *chargeTimer* itself is the function that keeps track of inputs having the proper charge time, which is represented by the correct direction being held for long enough. In my project, the adequate charge-time for an input is 30 frames, half a second. The second job of *handleChargeInputs* is to either "reformat" the player input or denote it as invalid for later. If the input is charged enough, the brackets present in its ideal form are added back to the input. Otherwise, the input is marked as invalid, specifically by not having enough charge.

The next function is one of the most important. The goal of *isInputValid* is to take the parsed input we have (most likely) just removed 5 from or added brackets to, and compare it to the ideal version of the input. The first thing the function does is mark down a list of the present "archetypes" (explained in detail in the previous section) for both the ideal input and the player input. Actually accomplishing this involves going through each input and checking for the specific occurrence of each archetype. The commonly used archetypes I looked for included both forwards and backwards quarter-circles (236 and 214) and half-circles (41236 and 63214), as well as Z-inputs (6323) and 360-inputs (632147896). While archetypes are often the biggest part of an input (both in terms of being the majority of an input's characters, and being the most important to its makeup), it is not always the only part. Motion inputs that contain more than one archetype usually need a way to bridge one archetype to the next, and some inputs with only one archetype still require additional steps that don't qualify as archetypes. Because these extra steps exist outside of the archetype classification, they are given more leeway.

A good example would be a previously used example, 632146P [7]. After the backwards half circle, the input requires the player to slide the stick from the back "4" position to the forward "6" position, before pressing the Punch button. Doing this would naturally bring the player across the neutral "5" position, but humans are not perfect. The absence of the gate or stick-border when performing this step makes it less consistent for real people to do perfectly, so additional unlisted steps are not considered automatic failures.

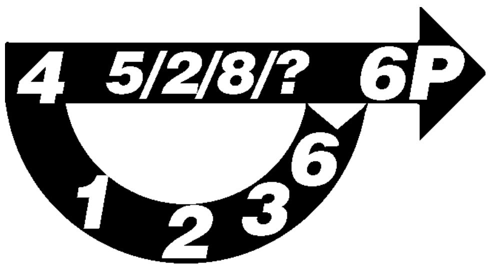


Figure 7: A more complex input that is made up of more than just its archetype.
Once the initial backwards halfcircle is completed, the player then needs to slide the stick to the other side of the gate before finishing the input. The **5/2/8/?** represents the additional leeway given to non-archetype segments of inputs.

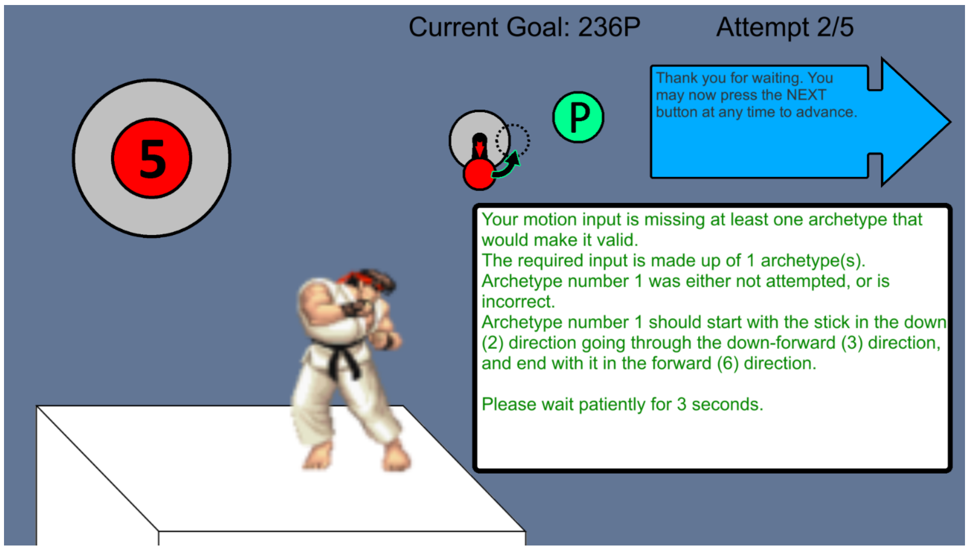
In my implementation, I allow at most two additional "steps" or directional inputs to be made when the expected next step of the motion input is outside of an archetype. If more than two extra steps are registered without performing the correct step, it is considered outside the bounds of proper leeway and deemed a failure. If a player's input contains all the archetypes as it should and completes all of the additional steps within the provided bounds, *isInputValid* marks the player's attempt as a pass.

The last major function is *dynamicFeedbackProvider*, and its purpose is to take the player's input and provide feedback if it is not deemed a pass. If the player input is correct, it does not give any feedback, as none is needed. If the player's input isn't correct, it provides dynamically produced feedback based on the context of your mistake. I have identified seven distinct cases for what may have caused a failure to occur. The information relating to these cases are stored in initialized strings that are appended to in the event that the distinct failure relating to one of them occurs in *IsInputValid*.

The simplest case is when the wrong attack button is pressed. Because our program only has two buttons, all it does in order to dynamically alter the feedback given is tell the player what button they should have hit instead of the one they did hit.

The next case pertains to charge inputs, and goes off if the player has not charged their move for long enough. The logistics of this are handled by *handleChargeInputs* and *chargeTimer*, as detailed earlier.

The next two cases refer to when the input goes outside the bounds of leeway by having too many extra steps. The only difference between the two cases is that the first checks for this mistake across all of the input, while the second case of too many steps specifically checks for it at the very end of the input, where the only remaining step was to press the correct attack button. These cases are able to dynamically tell the player if their stick was too high or too low (in reference to the y-axis.) This is possible because of an interesting quirk of the keypad which Numeric Annotation is based on. Compared to the middle row consisting of 4, 5, and 6, the numbers of the top and bottom rows respectively are greater and lesser. This means that the position of the stick when these cases are triggered can be used to tell the player what their



Thank you for waiting. You may now press the NEXT button at any time to advance.

Your motion input is missing at least one archetype that would make it valid.
The required input is made up of 1 archetype(s).
Archetype number 1 was either not attempted, or is incorrect.
Archetype number 1 should start with the stick in the down (2) direction going through the down-forward (3) direction, and end with it in the forward (6) direction.

Please wait patiently for 3 seconds.

Figure 8: A screenshot of what receiving feedback looks like in the final implementation.
Top-left: Visual representation of what position the stick is currently in.
Bottom-left: The player character.
Top-middle: The ideal input in Numeric Annotation, or ”Current Goal” as it is called in the experiment. A graphical representation if the Annotation is below it.
Top-right: The current attempt number and the status box below it. These will be explained in the Experiment Design section.
Bottom-right: The feedback textbox. The player’s input was ”23P” rather than ”236P”, requiring feedback to be provided.

mistake was, or more specifically where it was.

The last three cases all pertain to the archetypes of the player’s input. Specifically, if the player’s input has more archetypes than the ideal input, fewer archetypes, or the same amount with at least one archetype being incorrect. If there are too few archetypes, it is likely the player missed a step and the resulting input is lacking something. If there are too many archetypes, the player’s attempt was likely excessively large. Having the same amount of archetypes as the ideal input, but with them being wrong, probably means the player’s input registered a quarter circle instead of a half circle, implying they accidentally failed to complete a step before moving on to the next.

Once the function has finished running, it sends its dynamically generated contextually dependent feedback to the screen, as seen in Figure [8], so that the player may view it.

All of this information is just to give feedback to one input, but the program obviously needs to be able to handle more than just that. The purpose of *resetInputDetection* is to reset all of the variables pertinent to input detection and feedback provision. This reset will only occur if the player does not move for 3 full seconds (or 180 fps.) Because of the frame-sensitive nature of the function, it is the only function I call in *FixedUpdate* instead of *Update*. *FixedUpdate* is also where *chargeTimer* is called, as it also fulfills its job based on frame-count.

In my Approach section, I mentioned that not all of the planned features made it into the final implementation. Many of the content cuts were either causes by a shortage of time, or a change in how I wanted the experiment to be run. When designing my initial approach, the experiment was the part of this project with the least amount of planning put towards it. Ultimately, when compared to the initial mock-up design for feedback provision [5], the final implementation [8] has a few important differences. It was decided that the only way the player should be able to fix their mistakes was with feedback, so the Player Input in Numeric Annotation and its graphical representation were both removed from the final feedback. This is so the player can't directly compare their input and the ideal input, avoiding having to look at the feedback. In exchange, one new feature was added to the final implementation of the shown feedback: the ability to practice inputs. Practice Mode, as I call it, is very important to how the experiment was structured, so I will detail Practice Mode there.

4 Experiment

4.1 Experiment Design

While the creation of the program/simulation has been incredibly insightful on the process of fighting game motion inputs, it has not actually answered the question proposed at the end of the introduction. That is what the experimental portion of this project was for, which I held within the Crochet Lab of Union College's WOLD Center. The experiment followed the structure of a controlled experiment, featuring a distinct separation of control and experimental groups. Both groups were tasked with the same goal: to perform a series of motion inputs.

For the experiment, the program was pre-packaged with a series of motion inputs to perform already in a list. While the list was the same for every experiment participant, the order in which the inputs were given differed. Specifically, the first participant was given the inputs in the original order they were created and designed in, second participant had their list shifted to the right by one, and so on. The exact input orders are shown in Table [3]. The inputs required will each be represented on screen in both Numeric Annotation, as well as in a graphical format [9].

Before their experiment properly began, each participant was shown a video detailing how to go about taking the experiment. This video included the basic necessary knowledge of what a fighting game was, the participant's goal in this experiment, and how to progress through the experiment, as I will now detail.

The participant was asked to attempt a given input 5 times, regardless of if they perform the input

Participant #	Input Order	Exact Input List
1	Input #... 1, 2, 3, 4, 5, 6, 7, 8, 9, 10	5P, 5K, [4]6P, 22K, 236P, 41236K, 236236P, 2141236K, 632147896K, 632146P
2	Input #... 10, 1, 2, 3, 4, 5, 6, 7, 8, 9	632146P, 5P, 5K, [4]6P, 22K, 236P, 41236K, 236236P, 2141236K, 632147896K
3	Input #... 9, 10, 1, 2, 3, 4, 5, 6, 7, 8	632147896K, 632146P, 5P, 5K, [4]6P, 22K, 236P, 41236K, 236236P, 2141236K
4	Input #... 8, 9, 10, 1, 2, 3, 4, 5, 6, 7	2141236K, 632147896K, 632146P, 5P, 5K, [4]6P, 22K, 236P, 41236K, 236236P
5	Input #... 7, 8, 9, 10, 1, 2, 3, 4, 5, 6	236236P, 2141236K, 632147896K, 632146P, 5P, 5K, [4]6P, 22K, 236P, 41236K
...	...	...

Table 3: A table that displays the general pattern of the input order for participants of the experiment. While participant #1 received the inputs in their "intended" order from original conception, every other participant received a variation shifted to the right. Starting with participant 11, the order loops, so any participants numbered over 10 have orders that are repeats of earlier ones.

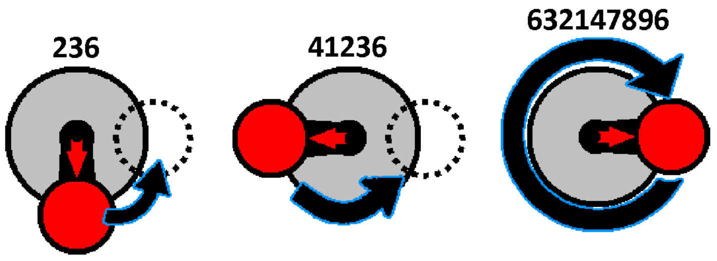


Figure 9: The graphical representation of a few motion inputs, as they appear in the program. The Numeric Annotation is listed above each.

correctly the first time or never at all. Before each input, the player is given access to Practice Mode, which I mentioned towards the end of my Implementation Section. In Practice Mode, the player is able to perform the input as many times as they want, for as long as they want, without actually using any of their 5 given attempts. For each practice attempt, they received feedback just as they would on a real attempt. Once the participant felt ready, they were able to begin their attempts in earnest. Between each attempt, there is a mandatory downtime period, where a complete 3 seconds of no button presses or stick movement must pass. This was done in order to incentive participants to look at their feedback. After the 3 seconds, the participant is free to begin their next attempt. This cycle of Practice Mode, followed by 5 attempts, repeats for each input until all 10 inputs are finished. For the experiment, each participant was given my personal Mayflash Universal Arcade Stick F500 V2. It is a very common entry-level fightstick, about as standard as you can get, and I can personally vouch for its quality. As stated earlier, participants were also be able to see the movement of their stick in real time on the screen, which is a standard feature of several modern fighting games.

So far, most of what I have described applies to both groups of the experiment, with one exception: how feedback is given. Specifically, participants of the experimental group will have access to the Dynamic Feedback Provider, while the control group will not. What this means in practice is that the control group is not given feedback. If an input of theirs is invalid, the only message they get is *"Your input was incorrect."* This also applies to their practice attempts as well. Other than that, there are no differences between the two groups. Even the input order pattern [3] is the same for each group. There is a reason I am choosing a between-subject design, as opposed to a within-subject design. The within-subject design approach for this experiment would go as follows; there would be only one group, with each participant receiving feedback on some moves and not others. I personally believe a direct comparison between feedback and no-feedback for every move will provide the most unbiased data on the success of the experiment, hence why I favor the between-subject design approach for the experiment.

I had two primary ways of collecting data for this experiment. The first was the program data. For each participant, the results results of their input attempts were recorded. For the 5 attempts of their 10 inputs (50 attempts in total), the following pieces of information were recorded:

1. Was the input attempt considered valid?
2. What was the player's exact input for that attempt?

Something very important to note is that practice attempts were not recorded at all by the program. Aside from the program data, I also had each participant fill out a Google Form to record a few basic pieces of information. The form asked participants the following:

1. What school year were they? (Freshmen, Sophomore, etc.)
2. What major were they?
3. How much experience did they have with video games? (On a 1-5 scale, with 1 never having played a video game before, and 5 being a strong investment into the hobby.)
4. If the participant had any video game experience whatsoever, how much of that experience was specifically in fighting games?

When designing the experiment, my initial aim was a sample size of ten people in each group, though I had planned for a potential fifteen in each when requesting a budget (to be discussed in a later subsection.) I was present at each experiment run personally, in order to make sure any questions the participant had were answered (if possible), as well as take notes on paper about anything I found interesting with a particular experiment or participant. My reasoning for this was because it occurred to me after the first experiment run that there was a lot of potential data not being tracked by either the program or the post-experiment form. When designing the experiment, my idea was that directly comparing the results of each group, with only one having access to the feedback tool, would allow me to directly observe if the feedback provided could be considered helpful, answering my initial research question.

I had several expectations for the results of the experiment. First, a general trend of increased input-success rates after the first one or two attempts for the experimental group. I considered it possible that the control group participants might also see higher success rates over time, if they had a good intuition for the fine motor control needed. Another consideration I had was that no correlation between higher success rates and the dynamic feedback tool would show at all, for a variety of reasons. Primarily among them, that the type of feedback a player requires might not be possible to automate, even with the feedback being given based on dynamic contexts. Despite my numerous expectations for what the real results may have been, I was still surprised by them. After a brief subsection on Budgetary Constraints, I will go into detail about what the results were.

4.1.1 Budgetary Constraints

For my experiment, I requested a budget of $300. Because I already personally owned the fightstick that participants would be using, I did not require any additional equipment. Instead, my budget was allocated towards participant payment. I couldn't estimate the exact amount of time each participant's trial would take at the time, aside from knowing it would take no greater than an hour. The hourly rate for experiment participation is $10. While my initial aim for total sample size was 20 people, I also accounted for a maxi-

X	Feedback Group	No Feedback Group	Both Groups
5/5 Gaming Experience	94.5% (4)	89% (6)	91.2% (10)
2/5 Gaming Experience	87.5% (3)	88.5% (4)	88% (7)
5/5 + 2/5 Gaming Experience	91.69% (7)	90.28% (10)	X

Table 4: A table comparing the **Input Success Rate** for different groups in the experiment. In parenthesises are the number of participants who fall into the category.
Note that not all participant data is in this table. The 3/5s and 4/5s were excluded so that I could focus on the opposite extremes of gaming experience in participants.

mum of 30. Paying 30 people $10 for an hour-long trial would total $300. In the end, I could not gather 30 participants, and the full $300 was not used. All of the unused budget will be returned to the SRG (Student Research Grant).

4.2 Experiment Results

The results of my experiment (organized in Table [4]) initially shocked me, before thinking about why they may have occurred. For my experiment, I was able to find 13 participants for the experimental group (receiving feedback), and 14 participants for the control group (receiving no feedback). The success rate for valid inputs was 91.69% and 90.28% for each group respectively. That means that both groups, regardless of if they received feedback or not, had a nearly identical rate of success. With a sample size of only 27 participants, the difference between 91.69% and 90.28% is almost nothing. Just to be sure I wasn't missing anything, I took a few other numbers into account. Among the responses to the Google Form that each participant filled out, there were 10 responses indicating a 5/5 level of video game experience, and 7 responses indicating a 2/5. No participants indicated a 1/5, with every single one having at least some level of experience playing video games. The total input success rate for the 10 5/5s was 91.2%, while it was 88% for the 2/5s. While the difference between the input success rates of the two groups is larger, it is by a negligible amount, and the sample size is even smaller; these combined sub-groups only totalling 17 participants out of the already small 27. To make absolutely sure I wasn't missing anything, I decided to check the results acquired by separating into both feedback vs no-feedback and 2/5 vs 5/5. For 5/5s, the 4 participants with feedback had a total input success rate of 94.5%, while the 6 participants without feedback had a total input success rate of 89%. For the 2/5s, the 3 feedback participants had a 87.5% input success rate, and the 4 participants given no feedback had an input success rate of 88.5%. The difference in success rates is still pretty small, and the sample size for these numbers are so small I think making claims based on them would lead to an illegitimate conclusion.

At first, I was unsure of how the results ended up this way. Upon thinking about how the experiment was designed, a few explanations occurred to me. The most prominent among them was the Practice Mode. The primary reasoning behind the addition of Practice Mode was so that participants without gaming experience or fighting game experience weren't immediately lost or overwhelmed when taking the experiment. It would serve as a way for them to get their bearings together. The problem with Practice Mode was twofold. First, most participants tended to practice their inputs until they were consistently good at them. Only after the participant was confident in their ability to perform an input correctly, did they actually advance past Practice Mode to start their attempts properly. This wouldn't be as big of an issue, if it weren't for the second problem. As I mentioned earlier, the program data only captured the 5 input attempts for each input. None of the practice attempt data was captured at all, so the vast majority of all input attempts were not factored into the final statistics.

Another big explanation with the results is that the program itself had several issues. Despite working on it for several months collectively, there were still issues I never caught, or didn't have time to fix before starting my experiment. The function dedicated to input validity, *isInputValid*, was primarily designed around the archetype system. The two "extra steps" of leeway given outside of archetypes were never accounted for in a vacuum, in scenarios where there were no archetypes in an input. Of the 10 inputs included in the experiment, 4 of them lacked an archetype. There were a few notable instances of participants inputting "2K" instead of "22K" as intended. The same thing happens with "4[6]P", where participants would input "[4]P" instead. While these shortcuts are not intended to exist, the program still considers them valid inputs. It is very much possible that even more issues with the program exist that may have also affected the experiment.

The last explanation I could think of is that the participants that were given feedback were disincentivized to read it. My initial idea when designing this project was to have feedback be as precise and detailed as possible. An unfortunate consequence of this design choice was that by the end of the project, the feedback given had become lengthy and cumbersome to read. An unfortunate blindspot in the data of my experiment is that I never asked the participants (who received it) how effective the feedback was. While the data itself holds no info on that, I was present for every experiment in-person. I noted that most participants would begin their next attempts as soon as the mandatory 3-second wait period was up, even when the feedback would have realistically taken more than 3 seconds to read. Without properly asking the participants it's impossible to confirm, but based on body language, I only recall two participants actually reading the feedback in a way where I could tell they were seriously considering it. It's very possible that the feedback, for the majority of all participants' inputs, was not read at all.

To me, all three of these explanations likely play a part in skewing the data, and rectifying the mistakes that caused them would be my primary candidates for hypothetical improvements to be made to this project.

5 Conclusion

5.1 Potential Improvements to this Project

Given another opportunity to redo this project and run another experiment, there are several improvements I would make. The first and most obvious change to the project would be a fully functioning program. The small issues with *isInputValid* would need to be fixed, and any other bugs in the code I am unaware of also must be dealt with.

After bug fixes, the next part of the project I would change would be Practice Mode. While I do still believe Practice Mode is overall a positive addition to the program, I think it was executed in a flawed manner. I would probably put a limit on how many practice attempts the participant is given; I think 4 or 5 attempts is a fair number. I would also have the program record the data of any practice attempts, which was a fairly large oversight during the experiment.

In the initial design for this project [5], the textual feedback was not the only feedback the player was to be given. They were also provided with their input's Numeric Annotation, plus a graphical representation. I would like to restore that feature in a future rendition of this project. I also think shortening the textual feedback would also go a long way. It's possible that 100% textual feedback was the wrong approach for this type of task. If the primary method of communicating mistakes in feedback was based on graphics and images, then maybe the text could occupy a different or smaller role.

5.2 Concluding Thoughts on this Project

In the end, despite an experiment marred by mistakes and results that don't answer my original research question, I do think I have reached a definitive conclusion; albeit a different one than I assumed I would reach. From the onset of this project, I carried several assumptions about the difficulty of learning a task like fighting game inputs. I had assumed that it would be very difficult for someone without experience to grasp, even among video game hobbyists. Outside of fighting games themselves, fighting game inputs are a somewhat unique task within the video game medium. There aren't a lot of things to compare them to, especially among more casual spheres of gaming. In spite of all of this, given just a short video primer on their task, my experiment participants were able to get a pretty solid grasp on the concept. Across the two distinct groups and across two distinct levels of gaming experience, the rate of success for inputs stays generally pretty high. The only reliable tool participants really had was Practice Mode, which they used to great effect. If I am to believe that the feedback wasn't really acknowledged by the participants, then the data acquired allows me to draw a different conclusion. While I do still believe a feedback tool could benefit players who make input mistakes, the conclusion I have come to is as follows: **People are generally much more resourceful and much more likely to intuit their mistakes on their own than I initially thought.**

References

[1] N. Joshi et al. "Comparison of elastic matching algorithms for online Tamil handwritten character recognition". In: *Ninth International Workshop on Frontiers in Handwriting Recognition*. IEEE, 2004, pp. 444–449. ISBN: 0-7695-2187-8. DOI: 10.1109/IWFHR.2004.30. URL: https://ieeexplore.ieee.org/abstract/document/1363951.

[2] H. Moon, R. Chellappa, and A. Rosenfeld. "Optimal edge-based shape detection". In: *IEEE Transactions on Image Processing* 11.11 (2002), pp. 1209–1227. ISSN: 1941-0042. DOI: 10.1109/TIP.2002.800896. URL: https://ieeexplore.ieee.org/abstract/document/1097757.

[3] P. Mulgrew. "Corrective Feedback Timing in Kanji Writing Instruction Apps". In: (2022). *Honors Thesis.* 2574, pp. 1–14. URL: https://digitalworks.union.edu/theses/2574/.

[4] S. Zhang, Y. Hu, and G. Bian. "Research on string similarity algorithm based on Levenshtein Distance". In: *2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*. IEEE, 2017, pp. 2247–2251. ISBN: 978-1-4673-8979-2. DOI: 10.1109/IAEAC.2017.8054419. URL: https://ieeexplore.ieee.org/abstract/document/8054419.