The University of Hong Kong
Faculty of Engineering
Department of Computer Science
Final Year Project (COMP 4801)
2017 – 2018

Project Title:
A Glimpse at the Future of Gaming: Haptic Feedback Devices in Multiplayer Virtual Reality Applications

Individual Report
Date: 15\textsuperscript{th} April, 2018

CHEUNG CASADO, Sheng Leung
3035123863

Advisor:
Dr. CHOI, Yi-King (Loretta)
Abstract:

The excitement surrounding the virtual reality industry is spreading across all sectors of society, with the highest projected market value expected to be in entertainment and gaming. The advancements in the visual, auditory, and kinesthetic departments of virtual reality have been impressive and are well known. However, the endeavors that attempt to incorporate the tactile perception have been undermined and do not seem to enjoy the same popularity.

There is a general lack of VR games that support haptic feedback, and almost none of these are of the multiplayer genre. The games that do provide haptic feedback are through basic hand-held controller vibrations. This project aims to develop Spellbound, a two-player virtual reality game tailored with strategic elements. The project will also include the development of two haptic suits to support more sophisticated haptic feedback and to highlight that integrating haptic feedback can result in an enjoyable gaming experience.

Though the team acknowledges the importance of the current focus of frontrunners in haptic device development, the team believes that integrating proven technologies such as the Microsoft Kinect and the Leap Motion device could save valuable development time, enabling more time to be spent on ensuring high-quality haptic feedback experiences.

This individual report is focused on my personal contributions and opinions related to: Network research and implementation; Leap Motion integration (namely gesture recognition and user interaction); and, preliminary haptic feedback testing. Encountered challenges throughout development, limitations, possible improvements, and future works are also discussed.
Acknowledgements

I would like to express our gratitude to Dr. Choi for allowing us to work together. Despite her very busy schedule, she has continually provided valuable advice whenever needed. Given the general lack of virtual reality development experience the team has, her support has been instrumental, informing the team about potential technical hurdles we may face during development.
# Table of Contents

List of Figures ................................................................................................................. 6
List of Tables ...................................................................................................................... 7
Abbreviations ..................................................................................................................... 8

1. Introduction .................................................................................................................. 9
   1.1 Motivation ............................................................................................................... 10
   1.2 Project Expectations ............................................................................................ 10
   1.3 Related Works & History ...................................................................................... 11

2. Methodology ............................................................................................................... 12
   2.1 Prerequisites: Hardware ....................................................................................... 12
      2.1.1 Virtual Reality Head-Mounted Devices (HMD) ........................................... 12
      2.1.2 Supporting Computer(s) .............................................................................. 12
      2.1.3 Leap Motion .................................................................................................. 12
      2.1.4 Microsoft Kinect .......................................................................................... 13
      2.1.5 Arduino Uno & Relevant Components ......................................................... 13
   2.2 Prerequisites: Software ......................................................................................... 13
      2.2.1 Unity ............................................................................................................. 13
      2.2.2 Blender ......................................................................................................... 14
      2.2.3 Image Editor .................................................................................................. 14
   2.3 Prerequisites: Scripting ......................................................................................... 14
      2.3.1 C# ................................................................................................................. 14
      2.3.2 C/C++ .......................................................................................................... 14
   2.4 System overview .................................................................................................. 15

2.5 Game development ................................................................................................. 16
   2.5.1 Game Mechanics ............................................................................................. 16

2.6 Haptic suit development ......................................................................................... 17

3. Personal Contributions .............................................................................................. 18
   3.1 Network Framework Research & Integration ...................................................... 18
      3.1.1 Why Choose PUN ........................................................................................ 18
      Issues with uNet ...................................................................................................... 18
      PUN & uLink (Authoritative, Non-Authoritative, Semi-Authoritative Servers) ...... 19
   3.2 Game Flow Realization & Use Case Design ....................................................... 20
      3.2.1 PUN Turn Manager ....................................................................................... 21
      3.2.2 Use Cases based on Client Choices ............................................................... 22
         Handling Scheme for Active Clients .................................................................. 23
         Handling Scheme for Inactive Clients ............................................................... 23
   3.3 Leap Motion Integration ....................................................................................... 23
      3.3.1 Hover UI Kit & Turn Interface Panel ............................................................. 23
      3.3.2 Gesture Recognition Implementation ........................................................... 28
         Why Custom Gestures? ...................................................................................... 28
         Attack Gesture ................................................................................................... 29
         Grab Gesture ....................................................................................................... 30
         Shield Gesture ..................................................................................................... 31

3.4 Preliminary Haptic Feedback Communication Testing ............................................. 32

4. Challenges, Limitations & Possible Improvements .................................................. 35
   4.1 Leap Motion Faulty Detection .............................................................................. 35
   4.2 Lack of Kinect Integration .................................................................................... 35
   4.3 Reworked Game Mechanics ................................................................................ 35
   4.4 Microcontroller Electronics & Wireless Communication ...................................... 36
   4.5 Handling Inactive Game Clients .......................................................................... 36
4.6 Gesture Flexibility & Maintainability ................................................................. 37

5. Future Works ........................................................................................................... 38
  5.1 VR Performance Overview for Unity Network Solutions ................................. 38
  5.2 More Sophisticated Hardware Components..................................................... 38

6. Conclusion ............................................................................................................... 39

7. References ............................................................................................................ 40

8. Appendices ............................................................................................................ 43
  8.1 Project Schedule ................................................................................................. 43
List of Figures

▪ Figure 1 Architecture Overview of System for Spellbound, p.15
▪ Figure 2.1 The different Phases in Spellbound, p.16
▪ Figure 2.2 Rock-paper-scissors nature of the game mechanics in Spellbound, p.16
▪ Figure 3 Haptic Feedback Outline (Front-View), p.19
▪ Figure 4 Example Implementation of OnTurnCompleted() Callback Method, p.21
▪ Figure 5 In-Game Timer Using OnTurnCompleted() Callback Method, p.21
▪ Figure 6.1.1 Turn Interface Panel UI Realization In-Game, p.24
▪ Figure 6.1.2 Turn Interface Panel Highlight, p.24
▪ Figure 6.2.1 Turn Interface Panel Case of Attack Option, p.24
▪ Figure 6.2.2 Lane Selection Menu, p.24
▪ Figure 6.3 Heads-up Display, p.26
▪ Figure 6.4 Birdseye view of the lanes within the in-game environment, p.26
▪ Figure 6.5 Green Lane Highlighting (Designating Lane Occupied by Player), p.27
▪ Figure 6.6.1 Red Lane Highlighting (Designating a Lane), p.27
▪ Figure 6.6.2 Red Highlight Resets When Choosing Move (same applies for defend), p.27
▪ Figure 7 Performance Phase (Birds Eye View), p.28
▪ Figure 8 Method for recognizing gestures prior to API v3.0 (C#), p.28
▪ Figure 9.1 Attack Gesture, p.29
▪ Figure 9.2 Attack Gesture Script in Unity Editor, p.29
▪ Figure 9.3 TipPosition Vector from Finger class from Leap Documentation, p.30
▪ Figure 10.1 Grab Gesture Script in Unity Editor, p.30
▪ Figure 10.2 Grab Projectile Activated by player on the right, p.31
▪ Figure 10.3 Grab Projectile Follows Hand Movements, p.31
▪ Figure 11.1 Shield Gesture Script in Unity Editor, p.31
▪ Figure 11.2 Shield Gesture Recognition Code, p.32
▪ Figure 11.3 Leap Axes for HMD in Unity, p.32
▪ Figure 11.4 Depiction of PalmNormal variable from Leap Documentation, p.32
▪ Figure 12.1 Excerpt from Sending.cs, p.33
▪ Figure 12.2 Excerpt from Arduino sketch of the old method, p.33
▪ Figure 12.3 How Uniduino Improved our Development, p.34
▪ Figure 13 Hand Model Extended Finger Bug, p.36
▪ Figure 14 Leap Motion Axes In Table-Top Configuration for Unity, p.38
List of Tables

- Table 1.1 Use Cases for Resolution of Performance Phase, p.22
- Table 1.2 Use Cases for Failed Gesture Scenarios, p.37
Abbreviations
3D: Three-dimensional
API: Application Programming Interface
ARAIG: As Real As It Gets™
GB: Gigabyte
HMD: Head-Mounted Device
HUD: Heads-up-display
IDE: Integrated Development Environment
PUN: Photon Unity Network
RAM: Random Access Memory
SDK: Software Development Kit
TGV: Tactical Gaming Vest
UI: User Interface
VR: Virtual Reality
VAMR: Virtual, Augmented, and Mixed Reality Applications
1. Introduction

Multiple market projections forecast that the virtual, augmented, and mixed reality applications (VAMR applications) field will become the next multibillion-dollar industry within a decade from now [1, 2, 3]. The advent of applications such as Pokémon Go! [4], hardware such as the TrueDepth camera of the iPhone X [5], and the multiple growing communities of Oculus, HTC Vive, and Microsoft Hololens users, add to the credibility of such predictions [6, 7]. While VAMR software has many practical applications, VR applications are expected to become a big part of the entertainment and gaming industry [1].

The most popular VR device that enables current VR gaming is the head-mounted display (HMD). HMDs are often coupled with a headset and handheld controllers to deliver a visually, auditory, and kinesthetically enjoyable experience. However, haptic devices could improve the experience even more. Haptic devices specialize in simulating the perception of touch. Outside of medical applications [9], the most common application of providing haptic feedback is through handheld controllers that vibrate. Compared to HMDs, other efforts involving the realization of the touch perception in gaming have not received the same degree of public recognition, and this is likely due to a difference in financial backing. For example, in 2014, Facebook acquired Oculus VR for $2 billion, Google Cardboard was introduced, and Sony PlayStation VR was announced [10]. Until investors on a similar scale become involved, it is unlikely that haptic devices will receive the same popularity [11]. Nevertheless, the team believes that there are merits to enabling haptic feedback in gaming, and that the incentives to such merits can be brought into popularity more quickly by integrating existing technologies to expedite development.

The structure of this paper is as follows. This section is the introduction. Section 1.1 states the motivation driving this project. Section 1.2 defines the project scope, objectives, and deliverables. Section 1.3 describes relevant efforts in haptic device research. Section 2 is the methodology. Sections 2.1, 2.2, and 2.3 introduce the required equipment and technologies. Section 2.4 covers a system overview of the project idea. Section 2.5 specifies the game mechanics for Spellbound. Section 2.6, describes the design of the haptic suit. Section 3 details my contributions to the project. Section 3.1 focuses on network research integration. Section 3.2 focuses on the game flow realization and use case design. Section 3.3 covers Leap Motion integration with an emphasis on user interactivity, and gesture recognition. Section 3.4 details the specifics of the preliminary haptic device testing. Section 4 reports the challenges encountered for the project. Section 5 suggests some ideas this project inspires. Lastly, Section 6 is the conclusion.
1.1. Motivation

There are two issues with the existing efforts in haptic feedback for VR games: Firstly, there is a difference in focus. Most commercially available options for haptic devices do not offer support for VR games, and those that claim to do so are still under development [20]. Secondly, as of the time of this writing, search results from popular game portals such as Steam, among other online resources [21], indicate that there is a general lack of multiplayer VR games that provide haptic feedback.

The current focus for frontrunners in haptic VR devices is on providing the tactile experience as a whole package to maximize the commercial value of the technology. Consequently, not only are developers implementing the haptic sensation, but also valuable development time is being consumed building technologies to layer atop the haptic feedback functionality (e.g. independent position tracking, and gesture recognition) when there already are existing technologies that provide said services.

Although creating such APIs provide some long-term benefits in terms of the standardization in the future development process, it is also important to expose the haptic-enabled VR experience to the public quickly to explore potential short-term unknowns. For example: medical concerns analogous to those caused by VR HMDs could arise [22]; alternatively, the public may simply disapprove of the device design or find the gaming experience too cumbersome. In the case of multiplayer VR games that aim to include a haptic component, it is especially important to clarify such prospects since they are few in number.

Thus, it is convenient and faster to integrate proven technologies such as Leap Motion and Microsoft Kinect to investigate such possibilities.

1.2. Project Expectations

This project has three objectives. First, to develop Spellbound a two-player virtual reality game that contains strategic elements (see Section 2.5). The original design decision considered adding multiple characters for the player to experiment with will be beneficial to the gaming experience, since the scope of the project did not include story, and quest components. However, this was discarded to enable resources to be redirected toward ensuring proper haptics. Second, to build two haptic suits. The types of haptic feedback to be provided shall be mechanical in nature. The team tested vibration intensities, frequencies, and locations to best emulate different types of tactile sensations. The project does not include the audio-to-mechanical conversion features (i.e. services provided by Woojer and KOR-FX, see Section 1.4). Third, to integrate Leap Motion and Kinect technologies to enable capture of body
movement, gesture, orientation, and position as input for the haptic feedback suit or the game application. Therefore, the two key deliverables will be the game software of *Spellbound* and two haptic suits. The aim is to create an enjoyable gaming experience that is complemented by the haptic feedback.

1.3. Related Works & History

In 2010, students at the University of Pennsylvania developed the Tactile Gaming Vest (TGV), which enables users to play *Half-Life 2* and feel haptic feedback similar to being shot, or slashed. Though reportedly, the vest only responded to receiving game events. No feedback would be provided if the game event originated from the user of the vest [12]. A Kickstarter effort in 2013 titled As Real As It Gets (ARAIG) attempted to expand on the same concept but in addition to mechanically generated haptic feedback, the ARAIG product aimed to also use electrical muscle stimulation to emulate certain touch perceptions. The ARAIG group also recorded a demo in which their vest was capable for providing haptic feedback for a game event that originated from the wearer of the vest (e.g. shot recoil) [13]. However, neither vest was integrated to VR equipment as users played the video game using a desktop computer [11,12].

The Aura Interactor was one of the earlier haptic gaming devices. Commercialized in 1994, it was the first of its kind to use electromagnetic actuators to convert bass sound waves into vibrations [14]. This idea of adding the haptic sensation due strictly to auditory game events is experiencing a reemergence. Woojer, a 2017 startup, is currently manufacturing a vest with improved technology along with a SDK for developers to provide a service similar to what the Aura Interactor achieved [15]. The competitor to Woojer is KOR-FX, which originated from a successful Kickstarter campaign in 2014. The KOR-FX vest has already been tested with recent games like *Resident Evil 7* and is commercially available [16].

The Hardlight VR Suit is under development by NullSpace VR Inc. and aims to provide both vibratory feedback similar to the TGV and the audio-generated vibratory sensations akin to the Woojer and KOR-FX products [17]. Contrary to Hardlight VR, which is targeted to be easily affordable and for home use, companies like The Void have opened a handful of theme-parks that provide a similar experience, albeit more costly, by using their product, the Rapture VR set: a combo of a VR head-mounted display (HMD) and a haptic vest [18]. Lastly, the Teslasuit, currently under development by Tesla Studios, is one of the most ambitious projects in haptic feedback VR devices. It aims to be the first haptic feedback solution that is full-body, provides an array of haptic feedback sensations, has motion capture and positioning systems built-in, is thermo-regulated, and is capable of multiplayer [19].
2. Methodology

The development was divided into two phases. There are three major milestones for the first phase: implementing the game prototype with core game mechanics; building the prototype for the haptic suit; and lastly, realizing the basic character or game object models. The focus of phase one is to have a framework with which the team can experiment with the game idea from the perspective of a single player. The priority during phase two is the network component that will enable multiplayer gameplay. Our development progress prioritized game functionality ahead of game aesthetics.

2.1. Prerequisites: Hardware

This subsection introduces the equipment used for the project and provides the reasoning behind their selection as compared to the other possible alternatives.

2.1.1. Virtual Reality Head-Mounted Devices (HMD)

Choosing the correct VR hardware is paramount to realize the development of our game. The two main contenders are the Oculus Rift, and the HTC Vive. While the spec comparison is close, the key difference is in the tracking systems. The Oculus uses a Constellation tracking camera, whereas the Vive uses a lighthouse laser tracking system. The issue with the Constellation tracking camera is that the maximum tracking space is eight by eight, any larger and occlusion may occur. In other words, the equipment simply will not be able to accurately track objects outside of said range. By comparison, the lighthouse laser system is capable of a maximum tracking space of 15 by 15 feet [23]. Since the game mechanics of Spellbound favor a larger playing area for the users, and since there is no software compatibility advantages to choosing one headset brand over the other, the Vive was selected simply due to its larger tracking area.

2.1.2. Supporting Computer(s)

While the Vive provides impressive performance, it also has hefty demands to maintain 90fps gameplay. The specific requirements are computer running on Windows 7.1 or later with either an Nvidia GTX 1060, or AMD RX 480 graphics card, and an AMD FX 8350 or an Intel i5-4590 processor with at least 4GB of RAM. The team estimated that running two instances of our client software locally would be too demanding to effectively support the multiplayer experience on the work stations available to the team [23]. As such two computers are used. One for each client instance.

2.1.3. Leap Motion

The Leap Motion device is a renowned vision-based gesture-tracking device that uses an infrared pattern projection to define a hover zone. Within this area the technology can recognize
the hands of the user. Furthermore, while the device has some gestures that it can identify by default, it is also possible to program unique gestures [24].

Another incentive to use leap motion is its modest dimensions and weight, so it can be easily attached to the Vive HMD without hindering the playing experience.

2.1.4. Microsoft Kinect

The Kinect is a 3D motion-sensing camera, which also uses infrared technology. Though the Leap Motion device is capable of motion detection, it is highly specialized for hand tracking, motion and gestures. By comparison, the Kinect can do the same for the entire body (i.e. skeleton tracking), which can help with position capture and tracking to help realize the game mechanics [25]. However, there have been studies regarding the positional accuracy and performance of the Kinect [9].

The Kinect SDK has support for Unity, and Unity also provides packages to aid in general development [25].

Unfortunately, Kinect integration was discarded to redirect efforts into proper HTC Vive integration. The repercussions of this decision is revisited in the limitations section.

2.1.5. Arduino Uno & Relevant Components

The Arduino Uno is a single-board microcontroller. Among the options for microcontrollers, the Arduino Uno is well supported by Unity in the form of the plugin Uniduino [26, 27]. Also, the Arduino products were the only option whose language was easiest for the team to learn. The documentation regarding Arduinos is extensive, but very informative, so the team expects it will likely to simplify the coding process.

Along with the microcontroller will be the auxiliary wiring and other pieces that will recreate the haptic sensations upon signaling. As stated in Section 1.3, the team focused on recreating mechanically induced haptic sensations. To accomplish this vision, the necessary transducers, actuators, or vibrators will also be required.

2.2. Prerequisites: Software

This subsection introduces the software used in this project and provides the reasoning behind their selection as compared to the other possible alternatives.

2.2.1. Unity

Unity is a free game engine that supports an array of VR plug-ins. Not only is the engine powerful, but also its development environment is very versatile. Animations, audio, game objects, the user interface, etc. Unity provides a multitude of tools to tailor and facilitate development of these elements and more [28].
We are also more familiar with Unity development, its documentation, and its community compared to other well reputed game engines such as the Unreal Engine.

2.2.2. Blender

Blender is an open source program for 3D content-creation that supports VR rendering. It has multiple modules for rigging and animating. In graphics, rigging is the umbrella term for techniques that make the digital sculpture move. In other words, it is like defining the skeleton, muscles and skin of an entity; and, how the movements relate to one another. The main incentive for using Blender is that there are free-licensed projects that have been tested in Unity, enabling us to spend less time designing and focusing more on the game mechanics implementation, and haptic suit realization [29].

2.2.3. Image Editor

Adobe Photoshop and Gimp are the image editing applications we may use when creating game assets, if the asset requires more specialized editing than what is provided by the Unity development environment.

2.3. Prerequisites: Scripting

This subsection provides the reasoning behind the selection of scripting languages as compared to the other possible alternatives.

2.3.1. C#

The programming languages supported by Unity are C# and JavaScript. The popular opinion on the Unity forums is that it is considered better practice to use C# as the language reinforces good coding principles. There have also been some tests that suggest the engine performs better on C# scripting. The official documentation is generally better supported for C# than JavaScript as well [28].

2.3.2. C/C++

Technical prowess in C/C++ is required since the Arduino language is wrapped around C/C++. That is, the methods called in the Arduino IDE are essentially C/C++ functions [26].
2.4. System Overview

A summary of the system setup can be seen in Figure 1. Under the motion capture block, the three devices from top to bottom are: the Microsoft Kinect, the Leap Motion, and the HTC Vive.

The Vive image features the Vive HMD on the center, the two motion controllers on the front, and the two lighthouse sensors (as described in Section 2.1.1). It is worth noting that we are not using the motion controllers in their traditional sense, but instead as part of the haptic suit to take advantage of their inherent position tracking ability for both game logic, and animation purposes (see Section 2.6, Design). By extension, the original motivation behind using the Kinect was because we could use it for position tracking of the entire body. However, the motion controllers for the Vive already fulfill this need. Therefore, the Kinect has been dotted and outlined in red, as it was discarded from the development process.

The directed arrows represent the expected input-output relationship. The position and orientation information of the hands is delegated to Unity via the Leap Motion SDK. The 3D rendering data is sent to the Vive HMD to produce the game visuals. In turn, the HMD also sends position and orientation information, so the engine can produce the correct perspective. The unidirectional arrow on the Vive image represents the tracking data from the motion controllers as mentioned on the paragraph above. Lastly, the engine should signal when the suit should trigger the tactile feedback.
2.5. Game Mechanics

After two players are successfully paired, they will be placed on opposing sides of the middle of one of five lanes on the virtual gameplay environment. There are three game parameters: health; move ammo, which allows the user to move for a turn; and, shield energy, which allows the user to shield for a turn. The game would proceed in four phases (see Figure 2.1). The purpose of each phase is described below.

Select phase: During the Select Phase, players have a limited time to interact with a virtual menu to choose their desired action for this turn. The actions include: move, attack, grab, and defend. For actions other than defend, the player must also target a lane. For example, if the player selects move, then the player must also choose which lane to switch to on their side of the field. Alternatively, if the player chooses to attack, then the player must select the lane on which the opponent may be in.

Evaluation phase: During the Evaluation Phase, the game decides what will occur during the Performance Phase (see Figure 2.2). The relationship between attack, grab, and defend is similar to rock-paper-scissors, but only resolves if the selected actions involve the same lane.

Performance phase: During the Performance Phase, the player must execute his action via gestures within a limited time span. For example, if a player chooses attack and it is the correct lane, then during this phase the player would perform a gesture to damage the opponent. It is the interactions during this exchange that will signal the tactile feedback on the respective haptic suit. Alternatively, if the player had chosen to move lanes, then the player model would be teleported to a different lane in the virtual space.

Resolution phase: During this phase additional game events that relate to resetting the game logic occur.

Winning conditions: The primary winning condition in the game is to deplete the health of the opponent. Alternatively, after a specific amount of turns has passed, the game ends, and the winner is determined by the percentage of health remaining.
2.6. Haptic Suit Design

The design for the haptic suit is expected to be able to produce haptic feedback at 11 different points (see Figure 4). The locations of the motors that will generate the haptic sensations are denoted by the red dots.

The different color regions indicate that the motors within that color are to be triggered when a player attacks that area. Since the HTC HMD physically obstructs the head region, there will be no haptic feedback on the head region. Moreover, all haptic feedback will be induced from the front of the body since the players always face one another during game interactions.

To help with tracking the position of a player in the game, the motion controllers of the HTC Vive, denoted by the black triangles, will also be attached to the suit.

Lastly, the blue square outlined in white is the expected location of the microcontroller. Since all motors and power supply will have to be wired to the microcontroller, it needs to be in a centralized position.
3. Personal Contributions

This chapter outlines my individual contributions to the project, namely: network framework research and integration, realization of game flow based on the game mechanics and use case design, Leap Motion integration (Hover UI Kit integration, and gesture recognition), and preliminary haptic feedback testing. Each subsection will not only describe the contribution, but also share the rationale behind the engineering choices, or technologies among other technical opinions and remarks. Potential improvements to the contributions detailed here are discussed in the Challenges Encountered & Potential Improvements section.

3.1. Network Framework Research & Integration

The lack of team experience in networking using Unity was one of the main concerns during the early phases of the project. It was my responsibility to conduct most of the research regarding networking frameworks. Ultimately, the Photon Unity Network (PUN) was determined to best suit the needs of our project idea.

3.1.1. Why Choose PUN?

There are three renowned solutions to realize the networking of a Unity project: (1) Unity Networking (uNet); UnityPark Suite (uLink); and, Photon Unity Network (PUN). Unity Networking is the native library that accompanies the Unity Engine, whereas uLink, and PUN are technologies that leverage the built-in library.

**Issues with uNet**

As uNet is the native library, it does provide the highest degree of control over our design. However, a few base classes provided in uNet are notoriously reported to be faulty, or poorly optimized. For example, the have been cases of the NetworkLobbyManager class not synchronizing to clients properly if any changes are made to the server before instantiating objects [30]. Granted, the project idea required very simple lobbying and the team expected that the server settings would rarely change, therefore this was not the main concern. The more severe issue was with the SyncVar attribute, the SyncList class, and, most importantly, the NetworkTransform class.

The SyncVar attribute, and the SyncList class have been reported as inconsistent or unreliable, and the popular opinion across Unity forums is to write additional code that either handle said targeted functionality or check the integrity of the data involved. Otherwise, state synchronization across the network would be compromised. This additional step could have been entertained by the team had there been no issue with the NetworkTransform class, which was not the case [31].
The NetworkTransform class interpolates under strict conditions, namely, it will interpolate for Character Controller, and RigidBody classes, but if one wishes to synchronize a Transform class, it does no interpolation. Considering that the Leap Motion Integration will require synching of transforms, this would mean that if uNet were chosen as our network framework, a custom NetworkTransform class would have to be made that can compensate the resulting jittering and teleporting behavior.

Moreover, Unity treats rigidbodies that are flagged as isKinematic as only editable by their Transform class (and not Unity’s Physics Engine), therefore the interpolation problem would also present itself when wanting to sync kinematic rigidbodies as well as transforms. This complicates, or at the very least limits the realization of projectiles and projectile collisions, which are crucial in our gameplay. Furthermore, even when NetworkTransform interpolated as intended, the computations to calculate for correction were also not optimized, resulting in latency increases even when testing clients on locally.

The team decided that resolving said concerns would take away resources from the development of gameplay and haptics, which was the main interest of our project. Nevertheless, it is important to note that there is a recent effort named uNet High-Level API Pro [32], which is aimed at resolving these exact same issues outlined, but by the time this was discovered, the team was already into development and could not afford to port over to a different network framework.

**PUN and uLink (Authoritative, Non-Authoritative, Semi-Authoritative Servers)**

PUN and uLink do not suffer from the previously issues discussed about uNet to the same extent. Both options provide functionality that simplifies many network concerns such as match-making, lobbying, state synchronization, and latency handling. Both services also come with prebaked templates that accommodate to FPS, and massive multiplayer online (MMO) applications.

uLink provides a more impressive architecture with built-in player movement prediction, to further optimize performance, and the uZone extension to facilitate network segregation and culling. The most impressive feature of uLink is that the game server logic has a good understanding of the physics engine Unity uses, enabling the design of an authoritative scheme for an application [33].

In an authoritative scheme, clients must repeatedly request permission from the server. In turn, the server has the responsibility of processing the request. This includes validating the request and replying with corrective measures whenever necessary. Remote procedural calls (RPCs) between clients is forbidden. In effect, all client interaction that is perceived to be
output locally results from a server reply. This is only possible because the server infrastructure is capable and optimized for simulating scene geometry, physics, and collisions from Unity. Otherwise, the overhead from the server-side processing would be too significant. Despite its advantages, the uLink license is outside our project budget, with the free-trial license lasting only 30 days. We deemed uLink as a technology clearly oriented for projects with commercialization in mind. As this is not the focus of our project we deferred to PUN, which is a more economical alternative.

PUN is a Unity client plugin that interfaces the main code base of Photon the Load Balancing API as software as a service (SaaS) via the Photon Cloud. Unlike uLink, the PUN does not include simulation of Unity physics on its server side. Most of the game logic is executed in the client-side. This allows for a non-authoritative scheme, or a semi-authoritative scheme [34, 35].

A non-authoritative server is one that proxies between all connected clients. Such servers do not know about the game logic and focus on relaying messages sent by clients. The concern with the non-authoritative approach is that it is insecure. Clients can easily cheat by changing the game logic on the client side. In return, the server requires much less CPU and memory resources due to the increased workload on the client-side.

Semi-authoritative is a hybrid between the two former approaches. The client is given some authority over certain parts of the game logic. For example, the client reports to the server when a client is hit locally to the server, and the server updates the relevant client parameters.

Ultimately, the team agreed to adopt a non-authoritative scheme. As our application is targeted for demonstrative and non-commercial purposes, the security concerns that could arise from client-side exploitation can be prioritized as a milestone for later iterations. In doing so, the team can assume that the clients can sort out physics locally, avoiding the need for implementing a physics library, or learning an additional third-party physics API on top of the standard Unity engine physics API. However, one of the future milestones for a project like this is to scale to a semi-authoritative and eventually fully authoritative scheme. The server should also have more responsibilities rather than simply being a proxy because the VR elements in Spellbound can be locally demanding.

3.2. Game Flow Realization & Use Case Design

Spellbound is better understood as a turn-based game due to its strategic elements with additional action elements based on the outcome of the strategic choices made by the player. This subsection outlines the details of how the game flow was implemented, using PUN Turn
Manager, and the use case design for the potential outcomes from the evaluation phase during the performance phase.

### 3.2.1. PUN Turn Manager

PUN Turn Manager is a generic component class provided in the PUN documentation that facilitates implementation of turn-based games. The game manager of a game instance in *Spellbound* adds this script component. The main responsibility of the class is to maintain data related to the occurrence of a turn in a Photon room, which is how PUN organizes its lobbying. Our application only uses one room, tailored for two players.

```csharp
public void OnTurnCompleted (int turn)
{
    ToggleTurnInterfacePanel (false, false);
    Debug.Log ("OnTurnCompleted: " + turn);
    this.localTurnSelection.turnAction = localPlayerManager.getAction ();
    this.localTurnSelection.laneSelected = localPlayerManager.getActionLane ();
    this.localTurnSelection.laneOccupied = localPlayerManager.getLaneNumber ();

    this.remoteTurnSelection.turnAction = remotePlayerManager.getAction ();
    this.remoteTurnSelection.laneSelected = remotePlayerManager.getActionLane ();
    this.remoteTurnSelection.laneOccupied = remotePlayerManager.getLaneNumber ();

    this.EvaluateSelections ();
    this.OnEndTurn ();
}
```

**Figure 4** Example Implementation of OnTurnCompleted() Callback Method

The PUNTurnManager class also provides useful callback methods through the IPunTurnManagerCallbacks interface, such as OnTurnBegins(), OnPlayerFinished(), OnTurnComplete(), and more. These callbacks are used to realize the phases described in the game mechanics section by understanding each turn as both players having made their choices during selection phase (see Figure 4). After a turn is completed, the code gathers the data from the client choices, including: the turn action (i.e. whether attack, grab, or defend was chosen); the lane that the client designated; and, the lane that the client currently occupies. Afterwards, such information is used in the EvaluateSelections() method (described in the following subsection) before calling the OnEndTurn() method, which performs logic resetting, and other validation and cleanup associated with the end of a game turn.

**Figure 5** In-Game Timer Using Server Time Stamp

Aside from enabling the implementation of the game loop with its callbacks, the PUNTurnManager base class also aids with part of the UI implementation, namely with the game clock (see Figure 5). The implementation for the “turn” property uses the PhotonNetwork
(i.e. the main class used in PUN) server timestamp. Since the turns are relayed between players, this ensures that there is no disparity or offset in the game timer between the two clients using the application.

3.2.2. Use Cases based on Client Choices

<table>
<thead>
<tr>
<th>Case</th>
<th>Player 1 (P1)</th>
<th>Result</th>
<th>Player 2 (P2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attack 1</td>
<td>P1 correct</td>
<td>P2 correct lane</td>
<td>Damage reflected to P1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P2 shield energy decreased</td>
</tr>
<tr>
<td></td>
<td>P1 incorrect</td>
<td></td>
<td>P1 damaged</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P2 damaged</td>
</tr>
<tr>
<td>2</td>
<td>Attack</td>
<td>P1 correct</td>
<td>P2 receives extra damage</td>
</tr>
<tr>
<td></td>
<td>Grab</td>
<td>P1 incorrect</td>
<td>P1 slightly damaged</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P2 absorbs P1 shield energy</td>
</tr>
<tr>
<td>3</td>
<td>Attack</td>
<td>P1 incorrect</td>
<td>P1 &amp; P2 slightly damaged</td>
</tr>
<tr>
<td></td>
<td>Attack</td>
<td>P1 correct</td>
<td>P1 &amp; P2 shield energy decreased</td>
</tr>
<tr>
<td>4</td>
<td>Attack</td>
<td>P1 correct</td>
<td>P2 receives extra damage</td>
</tr>
<tr>
<td></td>
<td>Move</td>
<td>P1 incorrect</td>
<td>P2 move ammo decreased</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>P2 shield energy decreased</td>
</tr>
<tr>
<td>5</td>
<td>Defend</td>
<td>P1 correct</td>
<td>P1 &amp; P2 shield energy decreased</td>
</tr>
<tr>
<td></td>
<td>Move</td>
<td>P1 incorrect</td>
<td>P1 &amp; P2 shield energy decreased</td>
</tr>
<tr>
<td>6</td>
<td>Defend</td>
<td>P1 correct</td>
<td>P2 absorbs P1 shield energy</td>
</tr>
<tr>
<td></td>
<td>Grab</td>
<td>P1 incorrect</td>
<td>P1 slightly damaged</td>
</tr>
<tr>
<td>7</td>
<td>Defend</td>
<td>P1 correct</td>
<td>P1 &amp; P2 slightly damaged</td>
</tr>
<tr>
<td></td>
<td>Move</td>
<td>P1 incorrect</td>
<td>P2 absorb P2 shield energy</td>
</tr>
<tr>
<td>8</td>
<td>Grab</td>
<td>P1 correct</td>
<td>P1 &amp; P2 slightly damaged</td>
</tr>
<tr>
<td></td>
<td>Move</td>
<td>P1 incorrect</td>
<td>P1 &amp; P2 slightly damaged</td>
</tr>
<tr>
<td>9</td>
<td>Grab</td>
<td>P1 correct</td>
<td>P2 slightly damaged</td>
</tr>
<tr>
<td></td>
<td>Move</td>
<td>P1 incorrect</td>
<td>P2 move ammo decreased</td>
</tr>
<tr>
<td>10</td>
<td>Move</td>
<td>P1 correct</td>
<td>P2 &amp; P2 shield energy decreased</td>
</tr>
<tr>
<td></td>
<td>Move</td>
<td>P1 incorrect</td>
<td>P2 &amp; P2 shield energy decreased</td>
</tr>
</tbody>
</table>

Table 1.1 Use Cases for Resolution of Performance Phase

Handling Scheme for Active Clients

Table 1.1 summarizes the expected resolution to the possible use cases of the performance phase. This table assumes that the users have successfully executed the gestures that are prompted during the performance phase. For example, the second column of case one describes
the case when player one chooses to attack and player two chooses to defend. This then results in four cases depending on the correctness of the lanes designated by either player.

The table also clarifies that the user does not need to designate a lane when defending. That is why in case five to seven, as indicated by the strikethrough text, checking for the correctness of player one is unnecessary. The same is the case for the move action since the lane that is designated during the select phase moves the user and does not concern the opponent. This is evidenced in cases four, seven, nine, and ten as the outcome is independent of player two (i.e. the outcome is the same across columns).

Lastly, the descriptions that are underlined and italicized indicate the cases that should trigger the haptic feedback. There are five in total: slightly damaged, damaged, extra damaged, reflected damage, and, shield energy absorption. The first four can be targeted to a specific body part, whereas the last type is felt throughout all haptic feedback points (see Section 2.6).

Handling Scheme For Inactive Clients

The current iteration of Spellbound assumes that both clients always have made a choice by the end of selection phase, which is denoted by the end of the in-game timer. To enable this, client choices are generated randomly if the client is inactive. These randomly generated choices follow the game logic (i.e. if the randomly generated turn action is to defend, then the player should not be targeting a designated lane).

3.3. Leap Motion Integration

Leap Motion was used to achieve two aspects of client interactivity in Spellbound. Firstly, in tandem with the Hover Kit UI library for the realization of the UI displayed during the selection phase. Secondly, for the realization of gesture-enabled functionality (i.e. attack, defend, and grab gestures).

3.3.1. Hover UI Kit & The Turn Interface Panel

The Hover UI Kit (hereon, HoverUI) is a tool made to simplify UI interactivity for VR and AR applications [36]. The core mechanism behind HoverUI is a set of 3D cursors that are mapped to the Leap Motion’s hands (hereon, the Leap Hands). The library then provides some prefabs for UI items that are fully customizable and tailored to interact with these 3D cursors.

HoverUI is not much different to the LeapInputModule which is already incorporated within the Unity.InputModules namespace. In fact, it provides less functionality. For example, the LeapInputModule has a few methods particularly designed for handling pinch events. However, the reason HoverUI was chosen over LeapInputModule was the superior documentation.
Figure 6.1.1 Turn Interface Panel UI Realization  Figure 6.1.2 Turn Interface Panel Highlight In-Game

Figure 6.1.1 shows the TurnInterfacePanel prefab made using HoverUI. This is displayed during the selection phase. The user can interact with the menu by pointing in the general direction of the item. There is visual feedback in the form of a highlight to inform the user that the item is being interacted with. Light blue indicates the item which is currently being selected, and an item is indicated as selected when the item is filled with a green highlight. Figure 6.1.2 for example, demonstrates the case of the client using his right hand to select to defend.

Recall that choosing to defend does not require the client to specify a lane for the turn, but the remaining three options do. Figure 6.2.1 depicts the case of the user choosing to attack. Upon successful selection, the TurnInterfacePanel transitions into the lane selection menu (see Figure 6.2.2). This second menu consists of four items, the three arrows to the left are used to designate the left, middle, and right lane, while the last item is a back button, which returns them to the main menu of the TurnInterfacePanel (i.e. Figure 6.1.1).

The TurnInterfacePanel is associated to the game logic and this is apparent to the client through the heads-up display (HUD) (see Figure 6.3). Unlike the TurnInterfacePanel, the HUD is not expected to be able to interact with the Leap Hands of the user, therefore, it is not made using the items provided by HoverUI. However, whenever the user interacts with the TurnInterfacePanel, the HUD provides additional information for clarity. The blue circle on the upper left describes the action selected through the TurnInterfacePanel. In this case it is attack, with the targeted lane being the left lane. The upper right image describes the lane currently occupied by the user. The green bar in between is a representation of the time
remaining. The red bar on the lower left represent player health. The blue bar represents shield energy, which is expended when defend is chosen as the action for the turn. On the lower right is an indicator of move ammo, which is expended when move is chosen as an action for the turn. In this case, it indicates four move ammo remain from a maximum of seven.

![Figure 6.3 Heads-up Display](image)

![Figure 6.4 Birdseye view of the lanes within the in-game environment](image)

The TurnInterface panel is also related to the lanes in the in-game environment (see Figure 6.4). This current iteration of Spellbound conceptually defines the lanes as left, middle, and right. However, the implementation considers the lanes as six GameObjects. If the player is assigned the lower three lanes, then those are the lanes which the player can occupy while the opposing three lanes are those that the player can designate (to attack, or grab).

This implementation provides some clarity because it enables for further user feedback in the form of lane highlighting. The current lane occupied is signaled with a green highlight (see Figure 6.5), whereas the whereas designated, or targeted lanes are colored red (see Figure 6.6.1). Figure 6.6.2, show that the highlighting logic associated with the TurnInterfacePanel
abides by the game logic. Lanes change color when targeted, and the color resets if the option chosen does not involve targeting an opposing lane (i.e. when defend, or move are chosen).

Figure 6.5 Green Lane Highlighting (Designating Lane Occupied by Player)

Figure 6.6.1 Red Lane Highlighting (Designating a Lane)

Figure 6.6.2 Red Highlight Resets When Choosing Move (same applies for defend)
3.3.2. Gesture Recognition Implementation

The events that trigger haptic feedback occur in the performance phase of the game, which is arranged as shown in Figure 7. The two players are in an elevated platform and must engage each other using gestures. The user is enabled the use of a gesture associated with the action chosen for this particular turn. For example, if player A chose to attack while player B chose to defend, then player A will only be enable the use of the attack gesture while player B will only be able to use the defend gestures. After conducting some research, I was given the responsibility of implementing custom gestures.

Why Custom Gestures?

An important note regarding Leap gestures is a change regarding the Orion SDK for C#. The most recent version as of the time of this writing is v3.2, and as of v3.0 the Gesture class has been deprecated [37, 38]. The earlier iterations of the API simplified the incorporation of gestures by using built-in default gestures (see Figure 8). The Gestures would be stored within an instance of the Frame class, from which gestures can be extracted by simply calling the Gestures() constructor. This returns an enumerable that can be compared with the built-in gestures(). However, the documentation warns of its unreliability, and leap developers have openly

Figure 7 Performance Phase (Birds Eye View)

Figure 8 Method for recognizing gestures prior to API v3.0 (C#)
announced in their official forum that support for the Gestures class has stopped and they will be shifting away from abstract gestures and onto more meaningful gestures (namely pinching, and grabbing motions). The consensus for custom gestures is to use the lower-level tracking data directly to define custom gestures and tie such gestures to Unity events. The attack, shield, and grab gestures were developed with this paradigm in mind.

**Attack Gesture**

[Figure 9.1 Attack Gesture]

The attack gesture is the simplest of the custom gestures. It is defined as extending the thumb, with the option of also extending the index and, or middle finger. No other configuration will trigger the attack gesture event OnAttack (see Figure 9.1). An extended finger is defined as bent and curled toward the palm. The frame data required to check for this state is available, but the same functionality is already provided through the isExtended() method provided in the Leap core. This method is the foundation of achieving this gesture.

The script has been coded in a flexible manner such that it is comfortably editable with the Unity inspector (see Figure 9.2). The variables under the Finger States heading can be edited with the drop-down menu to the right to alter the logic required for triggering the OnAttack event. For additional prerequisites to the logic, it is also possible to specify a maximum and minimum extended finger count. Otherwise this value is automatically updated depending on the choices selected on the drop-down menus.

The inspector also specifies the details of the OnAttack event. This, as with the custom events of the other two gestures pass the IHandModel structure to realize projectiles. The IHandModel enables access to the Finger class, which represents the fingers of the Leap Hands. As shown in the middle picture of Figure 9.1, the projectile is shot precisely out from the tip of the index finger. This is could be enabled using the TipPosition vector available in the Finger
class, which is calculated every frame. Figure 9.3 depicts these vectors as red arrows. However, we found the StabilizedTipPosition vector to be more fitting because it stores a calculated average of the position vector. This means that the projectile directory would ultimately stutter less and appear more consistent.

![Figure 9.3 TipPosition Vector from Finger class from Leap Documentation [37]](image)

**Grab Gesture**

The grab gesture uses similar logic to the attack gesture, also checking for finger extension. However, it stores the finger states twice and compares them to one another and only triggers when both the prior and current conditions are met. Figure 10.1 shows the conditions for our current gesture definition. If the finger state prior has all fingers extended, and the second finger state afterwards has all fingers closed, then the OnGrab custom event is fired.

As with the attack script, the required finger states can be edited through the inspector if the gesture definition needs to be changed. The key condition for using this script to define gestures is that the event only triggers when both finger states are detected in succession, compared to the attack script which triggered when only a single detection of a set finger extension configuration was detected.

The data used to spawn the grab projectile is the PalmPosition position, which is a vector that stores the center of the palm of the Leap Hands relative to the location of the Leap Hand Controller. At this point the target of the projectile is retrieved, and the projectile is parented to the Transform component of the Leap Hand that triggered this gesture in the Worldspace coordinate system. In doing so, the effect depicted by Figure 10.2 is achieved. The particle
effect originates from the target and go toward the player. Since the effect is tied to the hand of the player it follows the exact hand position (see Figure 10.3).

![Figure 10.2 Grab Projectile Activated by player on the right](image)

**Figure 10.2** Grab Projectile Activated by player on the right

**Shield Gesture**

The shield gesture also relies on detecting all extended fingers, but it also has two additional requirements. Firstly, the palm orientation must be facing the horizon in the virtual space, or more exactly, the face of the palms must be within an angular offset from the horizon. Secondly, there must be a net velocity that exceeds velocity and displacement thresholds in the forward direction of the player. Effectively, the script triggers the OnShield custom event when a pushing motion is detected with all fingers extended.

![Figure 10.3 Grab Projectile Follows Hand Movements](image)

**Figure 10.3** Grab Projectile Follows Hand Movements

![Figure 11.1 Shield Gesture Script in Unity Editor](image)

**Figure 11.1** Shield Gesture Script in Unity Editor

Figure 11.1 shows the customizability of the shield gesture script. The pointing type variable can be edited to relative to horizon, to the main camera, to the worldspace, or toward a specific target. The angular offset is determined by the On Angle, and Off Angle variables. That is, the gesture is valid if it is within On Angle degrees, and invalid if it is greater than or equal to Off Angle degrees. Editing the prerequisite velocity threshold, and extended finger configuration is also possible.
Of note, one of the prerequisites not immediately evident on the Unity inspector for this script is that the pushing gesture must be relative to the forward direction of the player. The variables related to this constraint are the velocity and displacement thresholds. Figure 11.2 describes the constraint in detail. The script captures the z-coordinate of the StabilizedPalmPosition, which behaves similarly to and derives the same benefits as the StabilizedTipPosition variable that was previously discussed, during two instances and calculates the difference.

Figure 11.2 Shield Gesture Recognition Code

The Leap API provides relative to the coordinate system, but that the axes of this coordinate system change depending on how the project is configured. The axes for HMD device configuration in Unity is shown in Figure 11.3. Consequently, to define a pushing motion, the DifferenceThreshold must be negative, as evidenced in the conditional statement of Figure 11.2.

Figure 11.3 Leap Axes for HMD in Unity [37]

Tangentially, Figure 11.2 also clarifies the other constraints of the shield gesture definition. angleTo represents the angular offset relative to the intended direction of the gesture (in this
case relative to the horizon). The variable named normal is based on the PalmNormal provided by the API, whose definition is depicted by the red arrow perpendicular to the palm as shown in Figure 11.4. palmVelocityMagnitude estimates the velocity the palm is moving at and that is compared to the intended velocity threshold. Lastly, fingerState is evaluated on a separate coroutine and represents whether the intended extended finger configuration is met.

One minor point to clarify is whether checking for the z-component of the velocity more optimal than checking for the magnitude of the vector when checking the velocity threshold. While this suggestion may seem as an optimization conceptually because the operation for calculating magnitude requires the consideration of variables in three axes compared to simply checking the z-coordinate differential, it does not apply in this case. By the time the information arrives using the GetLeapHand() method (see Figure 11.2), the magnitude calculation cost is already inherent because magnitude is stored as a property, therefore there is no optimization in not using the magnitude measure.

3.4. Preliminary Haptic Feedback Communication Testing

During the early phases of development, the team struggled to find the necessary motor components to emulate the haptic feedback. Nevertheless, I took the responsibility to perform

![Excerpt from Arduino sketch of the old method](image_url)

![Excerpt from Sending.cs](image_url)
some preliminary testing for the intended form of communication using LED lights. The same method was applied to the motor hardware successfully.

This method uses the SerialPort class from the System.IO.Ports namespace (see Figure 12.1). Note that this namespace must be explicitly enabled on Unity by adjusting the API Compatibility level to .NET 2.0. Lines 13 to 31 define a basic function, OpenConnection(), which is able to establish a serial connection between Unity and the microcontroller.

Line 33 and onwards contain the functions that send a character through the port to the microcontroller for interpretation (see Figure 12.2). Prior to running the application on Unity, the Arduino sketch needs to be uploaded to the microcontroller using the Arduino IDE. This file specifies the port from which to expect communications from, the pins that are connected to the hardware to be activated, and under what circumstances to activate them. The beginning lines of the file designate pin 11, 12, and 13 to be for the left, right, and middle torso motors respectively. Then, the script instructs the microcontroller to compare the bytes read from port 9600. Depending on what character the bytes correspond to, then one of the three motors is turned on while the other two are turned off.

**Figure 12.3 How Uniduino Improved our Development**

The previous procedure split development between the Unity environment and the Arduino IDE. Each time a change was made on the microcontroller, the Arduino file would have to be uploaded. Moreover, the team noticed that it became much more difficult to support more sophisticated microcontroller interactions without sacrificing performance and proper code
encapsulation. The logic within the setup() and loop() functions would grow extensive and some lines would be redundant for some use cases. The Uniduino plugin resolved this situation.

The Uniduino plugin improves development because it centralizes all coding on the Unity platform. The plugin enables the microcontroller to be configured on a script-by-script basis, which facilitates proper code encapsulation. Lines 7 to 9 and 17 to 21 (see Figure 12.3) configures the pins on the microcontroller similar to the previous Arduino sketch with pin 11, 12, and 13 corresponding to torso motor left, right and middle respectively. However, it is possible to specialize the script and create a more sophisticated behavior. In this case, the script activates each torso motor in a sequence. No redundant or extensive code needs to be uploaded to the microcontroller before running the game application. Instead, only the code that needs to be run is uploaded and executed.
4. Challenges, Limitations & Possible Improvements

There were three issues encountered during development of which all could have been avoided with more preemptive planning.

4.1. Leap Motion Faulty Detection

During the early phases of development, there were instances during which the Leap device malfunctioned, displaying some faulty detection (see Figure 13). In this image, the presumed finger state contradicts the digital hand model. The yellow-colored hand has red and green circles at the finger tips. Green represents the active state, in which the finger is detected as extended. Conversely, red implies that the engine understands the finger as not extended. Though the yellow-colored hand is fully extended, only two fingers are detected as extended. Clearly, this produces a logical contradiction.

The team has determined this to be a hardware complication, because this behavior varied depending on how often Leap Motion is recalibrated and the amount of light in the environment. Regardless, the team could not manage to recreate this anomaly.

4.2. Lack of Kinect Integration

The team replaced the Kinect with the HTC motion controllers as delegates to track user position. When the complication with the Leap Motion controller arose, too much time was allotted to investigating its cause that the team had overlooked the possibility of using the Kinect device as a second point of reference to reconfirm user gestures. Realizing this earlier would have expedited development significantly. However, the team has decided not to integrate Kinect into the project at this stage as devoting study efforts for the Kinect library was deemed highly inconvenient. There were also concerns about potential performance drops if the Kinect were to have been used in this manner. The time necessary to validate this through research would be better suited for completing the current milestones for the team.

4.3. Reworked Game Mechanics

The original game mechanics overlooked a few use cases. After the use cases were clarified, the mechanics had to be reworked to include the concept of movement ammo, and shield energy. Prior to this point, the team had already completed the game UI and scripting
relationship of the character parameters. While this issue was not as serious as the former two, it still caused the development priorities to shift, aggravating project organization.

### 4.4. Microcontroller Electronics & Wireless Communication

The team is generally lacking in extensive knowledge regarding electronics. This complicates the acquisition for the auxiliary components for the microcontroller. The team has found additional help through personal connections to facilitate this and concerns regarding the efficient wiring and estimates of the necessary battery power.

After completing the design for the haptic suit, the team concluded that the haptic device wiring would be too cluttered and may result in a worse user experience. To mitigate this, the team sought to make the communication between the microcontroller and Unity wireless. This required additional chips to merge with the microcontroller, which complicated the hardware electronics even further. Moreover, the team found that this could introduce very low-level complications within the Uniduino plugin, which was already being used. Granted, this is a time loss that the team could not afford. The team has decided to follow-through with the current suit design first and remove some feedback points if the wiring is indeed convoluted enough to detriment user experience.

### 4.5. Handling Inactive Game Clients

<table>
<thead>
<tr>
<th>Use cases for Failed/Idle State</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attack versus Idle</strong></td>
<td>Attack successful regardless of lane chosen</td>
</tr>
<tr>
<td></td>
<td>Attacker wins after multiple consecutive idle states</td>
</tr>
<tr>
<td><strong>Defend versus Idle</strong></td>
<td>No defend energy deducted</td>
</tr>
<tr>
<td></td>
<td>Defender wins after multiple consecutive idle states</td>
</tr>
<tr>
<td><strong>Grab versus Idle</strong></td>
<td>Grab successful regardless of lane chosen</td>
</tr>
<tr>
<td></td>
<td>Grabber wins after multiple consecutive idle states</td>
</tr>
<tr>
<td><strong>Idle versus Idle</strong></td>
<td>Turn skipped, match cancelled after multiple consecutive idle states</td>
</tr>
</tbody>
</table>

**Table 1.2 Use Cases for Failed Gesture Scenarios**

Generating random choices whenever a client does not do anything during the selection phase is effectively no different than making *Spellbound* a game of chance against a random number generator. *Spellbound* is meant to be an engaging activity between the clients, it is not in the spirit of the application to allow clients to effectively play the game while idle. This could be improved by employing a punishment system that detects for the idle state, meaning that an alternative win condition could be introduced after three consecutive failed status has been detected for the same player (see Table 1.2). The failed, or idle state occurs when a player fails to perform the prompted gestures during the select or the performance phase. Notice that for the cases in which one of the two players has selected a valid action (attack, defend, or
grab), the mechanics place that player at an advantage by either automatically deeming the action successful regardless of the lane, or relieving the player of the cost for defending.

The specific number of lanes, turns, and consecutive idle state occurrences before terminating the game would have to be determined after play testing. Nevertheless, incorporating these adjustments can expedite game match duration, and enforce clients to actively participate in the game as intended.

4.6. Gesture Flexibility & Maintainability

The gesture scripts can be customized to a fair degree, and even more so because the scripts were designed to enable editing of the logic through the Unity inspector. Nevertheless, their flexibility and maintainability could be improved even further.

Consider if this project needed to switch to the table-top configuration (see Figure 14). In this setting the green light on the lower right faces the user. The attack and grab gesture scripts are entirely functionally transferable. However, the code for the shield gesture no longer applies as intended. Recall that one of the constraints for the shield gesture is for the pushing motion to be relative to the horizon. When the configuration is changed to table-top, the vector chosen to validate the palm position relative to the horizon becomes the y-axis. This means that the gesture triggers an event whenever a palm faces the upward y-axis followed by a quick movement in the z-axis. The gesture definition changes from a push motion to one analogous to serving a platter.

Figure 14 Leap Motion Axes In Table-Top Configuration for Unity [37]
5. Future Works

This project could inspire future expansions from a network perspective, and a hardware aspect.

5.1. VR Performance Overview for Unity Network Solutions

Provided the comparison between the three most popular network solutions for Unity is close, each with their respective advantages, an interesting study could be benchmarking the VR performance of each framework. One limitation of the Photon Cloud technology which this project uses is that it is not specialized for relaying authoritative messages, specially when said messages involve Unity collisions and physics. The Photon Server service is an alternative designed to enable the deployment of authoritative servers, in exchange for costlier scalability-related payments. Therefore, such benchmarking research could prove to be valuable to enable more educated decision-making for game design and its associated economic considerations.

5.2. More Sophisticated Hardware Components

The lack of hardware proficiency in the team limited the creative options by which to expand the motivation of this project, and that is to make haptic feedback meaningful. HTC recently released a new model of the Vive featuring wireless capability [39]. A step in the right direction would be developing a wireless module for microcontroller communication with Unity as is in the spirit of this trend. Similarly, instead of focusing on a suit, a concentrated effort on a particular haptic sensation could also be interesting. For example, a Leap Motion interface with a set of gloves that provide more localized tactile feedback.
6. Conclusion

This individual report focused on my contributions to the team for this final year project. Three network solutions were considered as the foundation for Spellbound. The limitations of uNet eliminated it from contention. Interestingly, the concerns raised during the selection process were found to be valid as the team later discovered the HLAIP Pro project, which aims at fixing such concerns. Though uLink provides an impressive service, it was not chosen due to budget considerations. Due to its rapid deployment and convenience enabled by the Photon Cloud service, PUN enabled the team to quickly adopt it to test the Leap Motion integration. The team decided to adopt a non-authoritative server, as the nature of the Photon Cloud obstructs the semi-authoritative or authoritative server schemes.

An additional benefit to using PUN was the PUNTurnManager class which facilitated the implementation of the game logic through its useful callback methods. It also aided with synchronizing in-game time between players. The specific use cases of the game mechanics were also outlined. In the case of inactive clients, the current iteration of Spellbound generates random choices for the idle client.

The Leap Motion Input Module was considered to realize UI elements, but the team preferred the Hover UI Kit documentation. HoverUI was used to make the TurnInterfacePanel, which is the main menu users interact with in-game. The TurnInterfacePanel provides multiple visual cues on itself, the HUD, and the lanes within the in-game environment to make the game more intuitive. With the recent deprecation of the Leap Gestures class, custom gestures had to be employed. The details of the three custom gestures: attack (thumbs up, or gun configuration), grab (grasping motion), and defend (forward push) were covered.

Preliminary testing on the LED lights proved to be successful and the same method was applied when the team procured the motor hardware for the haptics. The Uniduino plugin facilitated project organization in this respect.

Challenges, and limitations of these contributions also emphasized how they could be improved and inspire future works. In terms of networking, comprehensive VR performance benchmarking of network solutions could lead to more educated engineering choices. In terms of gameplay, enabling a punishment system over randomly generated moves could handle inactive clients. As for Leap Motion, it calls into action more extensive studies regarding the its detection capabilities by device version; the need to maintain flexible and maintainable gesture design to handle different Leap device arrangements; and perhaps, a framework which uses Microsoft Kinect to reconfirm Leap Gestures. Lastly, investigations into wireless microcontroller communications with Unity could result in even better user experience.
7. References


### 8. Appendices

#### 8.1. Project Schedule

<table>
<thead>
<tr>
<th></th>
<th>October</th>
<th>November</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 1</td>
<td>1&lt;sup&gt;st&lt;/sup&gt; – 7&lt;sup&gt;th&lt;/sup&gt;</td>
<td>1&lt;sup&gt;st&lt;/sup&gt; – 4&lt;sup&gt;th&lt;/sup&gt;</td>
</tr>
</tbody>
</table>
|          | ● Finalize design choices for the:  
  ○ Haptic device  
  ○ UI of each game phase  
  ● Inspect playing field size  
  ● Equipment acquisition/booking  | ● Game mechanics  
  ● Game object modeling for haptic feedback device onto player model  |
|          |                           |                           |
| Week 2   | 8<sup>th</sup> – 14<sup>th</sup> | 5<sup>th</sup> – 11<sup>th</sup> |
|          | ● Motion capture environment setup/integration  
  ● Character design and basic player model  | ● Game mechanics  
  ● Game object modeling for haptic feedback device onto player model  |
|          |                           |                           |
| Week 3   | 15<sup>th</sup> – 21<sup>st</sup> | 12<sup>th</sup> – 18<sup>th</sup> |
|          | ● Character design  
  ● Map design  
  ● Game flow implementation  
  ○ Basic game object layout for different game flow phases  
  ○ Basic UI  
  ● Arduino setup  | ● Prototype testing & refinement (single player use case)  
  ● Haptic suit prototype  |
|          |                           |                           |
| Week 4   | 22<sup>nd</sup> – 28<sup>th</sup> | 19<sup>nd</sup> – 25<sup>th</sup> |
|          | ● Game flow implementation  
  ○ Character select  
  ○ Map select  
  ● Game mechanics  
  ○ Default & Custom Gestures  
  ○ Movement  
  ○ Health  
  ○ Game clock  
  ○ Winning conditions  
  ○ Aim & Collision  | ● Prototype testing & refinement (single player use case)  |
<p>| | | |
|          |                           |                           |
| Week 4.5 | 29&lt;sup&gt;th&lt;/sup&gt; – 31&lt;sup&gt;st&lt;/sup&gt; |                           |</p>
<table>
<thead>
<tr>
<th>Week 4.5</th>
<th>26&lt;sup&gt;th&lt;/sup&gt; – 30&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Haptic suit prototype</strong></td>
</tr>
<tr>
<td></td>
<td>• Networking research</td>
</tr>
<tr>
<td></td>
<td>• <strong>Haptic suit prototype</strong></td>
</tr>
<tr>
<td></td>
<td>• Process report writing</td>
</tr>
</tbody>
</table>

**December**

Presentation preparation over holidays

**January**

<table>
<thead>
<tr>
<th>Week 1</th>
<th>1&lt;sup&gt;st&lt;/sup&gt; – 6&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Networking</td>
</tr>
<tr>
<td></td>
<td>• Modeling</td>
</tr>
<tr>
<td></td>
<td>• Animations &amp; Audio</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 2</th>
<th>7&lt;sup&gt;th&lt;/sup&gt; – 13&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Interim Presentation</td>
</tr>
<tr>
<td></td>
<td>• Modeling</td>
</tr>
<tr>
<td></td>
<td>• Animations &amp; Audio</td>
</tr>
<tr>
<td></td>
<td>• Report writing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 3</th>
<th>14&lt;sup&gt;th&lt;/sup&gt; – 20&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Networking</td>
</tr>
<tr>
<td></td>
<td>• Modeling</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 4</th>
<th>21&lt;sup&gt;st&lt;/sup&gt; – 27&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Interim Report</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 4.5</th>
<th>28&lt;sup&gt;th&lt;/sup&gt; – 31&lt;sup&gt;st&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Process report writing</td>
</tr>
</tbody>
</table>

**February**

<table>
<thead>
<tr>
<th>Week 1</th>
<th>1&lt;sup&gt;st&lt;/sup&gt; – 3&lt;sup&gt;rd&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Two-player game mechanics testing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 2</th>
<th>4&lt;sup&gt;th&lt;/sup&gt; – 10&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Determine project process, determine whether AI component is feasible for single player experience</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 3</th>
<th>11&lt;sup&gt;th&lt;/sup&gt; – 17&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 4</th>
<th>18&lt;sup&gt;th&lt;/sup&gt; – 24&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 4.5</th>
<th>25&lt;sup&gt;th&lt;/sup&gt; – 28&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Process report writing</td>
</tr>
</tbody>
</table>

**March**

<table>
<thead>
<tr>
<th>Week 1</th>
<th>1&lt;sup&gt;st&lt;/sup&gt; – 3&lt;sup&gt;rd&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• <strong>Debugging and Testing</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 2</th>
<th>4&lt;sup&gt;th&lt;/sup&gt; – 10&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• <strong>Debugging and Testing</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 3</th>
<th>11&lt;sup&gt;th&lt;/sup&gt; – 17&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• <strong>Debugging and Testing</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 4</th>
<th>18&lt;sup&gt;nd&lt;/sup&gt; – 24&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• <strong>Debugging and Testing</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 4.5</th>
<th>25&lt;sup&gt;th&lt;/sup&gt; – 31&lt;sup&gt;st&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Process report writing</td>
</tr>
</tbody>
</table>

**April**

<table>
<thead>
<tr>
<th>Week 1</th>
<th>1&lt;sup&gt;st&lt;/sup&gt; – 7&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Report writing</td>
</tr>
<tr>
<td></td>
<td>• Presentation preparation</td>
</tr>
<tr>
<td></td>
<td>• <strong>Debugging and Testing</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 2</th>
<th>8&lt;sup&gt;th&lt;/sup&gt; – 14&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Report writing</td>
</tr>
<tr>
<td></td>
<td>• Presentation preparation</td>
</tr>
<tr>
<td></td>
<td>• Final report submission</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 3</th>
<th>15&lt;sup&gt;th&lt;/sup&gt; – 21&lt;sup&gt;st&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Report writing</td>
</tr>
<tr>
<td></td>
<td>• Presentation preparation</td>
</tr>
<tr>
<td></td>
<td>• Final report submission</td>
</tr>
<tr>
<td></td>
<td>• Final presentation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 4</th>
<th>22&lt;sup&gt;nd&lt;/sup&gt; – 28&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Final project evaluation and review</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Week 4.5</th>
<th>29&lt;sup&gt;th&lt;/sup&gt; – 30&lt;sup&gt;th&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Final project evaluation and review</td>
</tr>
<tr>
<td>May</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>2\textsuperscript{nd}</td>
<td>Final year project exhibition</td>
</tr>
<tr>
<td>30\textsuperscript{th}</td>
<td>Final year project competition</td>
</tr>
</tbody>
</table>