360 Gunner - A 2D platformer to evaluate network latency compensation

A Master Thesis Report
Submitted to the faculty of
Worcester Polytechnic Institute
In partial fulfillment of the requirements for the
Degree of Masters
in
Interactive Media and Game Development
By:

Thanh Long X. Vu

Date:
December 6th 2019

Thesis Advisor:

_____

Professor Mark Claypool

Thesis readers:

_____

Professor Charles Roberts

_____

Professor Ralph Sutter

Abstract

Online gaming is rapidly growing as an entertainment choice, as it provides players with a high variety in genres, affordability, ubiquity and also real-time online interactions. However, slow networks or congestion can cause perceivable network latency and make players suffer from a degraded gameplay experience. Latency compensation techniques have been developed to combat the negative effects of network latency, but more understanding of latencies affects and latency compensations benefits are still needed. Our project studied the degradation of different game actions with latency and how player prediction - a classic latency compensation technique - affects gameplay in a 2D platformer. We designed and implemented an original 2D platformer with player prediction implemented for player movement actions, then invited players to play our game under different network and latency compensation conditions. Based on the subjective and objective data collected, we found that 2D platformers are sensitive to even modest amounts of network latency. Player prediction helped players have fewer deaths below 200ms of latency, but at 400ms and above its benefits were outweighed by its disadvantages to visual consistency.

TABLE OF CONTENTS

1. Introduction

In recent years, online games have become one of the most popular entertainment applications in the world. Innovations in Internet technology and smartphones continue to fuel the accessibility of online gaming. By the end of 2017, gamers spent on average 6.5 hours per week playing with others online and only 4.5 hours per week playing with others in person [1]. By 2020, the number of online console gamers in the United States is expected to grow to over 57 million, and the worldwide market for PC online games alone is predicted to reach a value of around 44.2 billion U.S. dollars [2].

More players are opting for online gaming as an entertainment choice because of its variety in genre, affordability, ubiquity, and also the opportunity for real-time interactions with other players around the world. While playing online games, players are not only satisfying their entertainment or serious purposes, but are also immersed in a virtual society. They can be having fun with friends who live far away or interacting with new people who they would hardly have a chance to talk to in real life. This real-time interaction between players despite geographical distance is an important feature of online gaming that has brought about its deserved attention [3].

Real-time player interaction in online games is achieved by two main types of game architectures: client-server architectures and peer-to-peer architectures [4]. In a client-server architecture, game data from players are centralized in the server, while in the peer-to-peer architecture every game controls its own state, sends data to and receives data from all other peers. Online game architectures can also be characterized as authoritative or non-authoritative. Authoritative means the server has the “master” copy of the game, and any actions by the players need to be received and authorized by the server. Non-authoritative means game clients are responsible for some parts of the game logic. The most popular online game architecture is authoritative server-client, which is used in well-known games such as Counter-Strike: Global Offensive (Valve Corporation, 2012) or Overwatch (Blizzard Entertainment, 2016) since it limits the ability of players to cheat, because only the server can make decisions on game-critical player actions. More specifically, this architecture makes use of

a single authoritative server to support the main logic inside the game. Every client connected to the server periodically receives data, then locally creates a representation of the game state. Through the network connection, the server executes the input commands received from the clients, applies the input and updates the game world, then sends back to the client any changes to game objects [5].

The authoritative client - server architecture provides benefits to online games, but there are also intrinsic challenges that must be overcome. When players take actions inside the game - for example moving an avatar - network messages containing the critical information about these actions take time to arrive at the game servers. The game servers then take time to process these messages and send back update messages to the clients. In a similar fashion, these update messages arrive back at the game clients after a round-trip delay. Under good network conditions, the total round-trip time is small enough that it is not noticeable for players. However, sometimes the connection between game clients and game servers is slow enough that the delay becomes perceivable. As a result, the game feels unresponsive, leading to disruptive gaming sessions or even the sense of real-time interactions between players being lost. In general, the pace of the game and the frequency of user actions largely determine how sensitive a game is to network latency.

There are known solutions to mitigate the effects of latency, which are called latency compensation techniques [5]. One example of these techniques is player prediction, where player actions are sent to the server and performed on the clients at the same time. This technique makes player actions feel immediately responsive, but can lead to inconsistencies that need to be fixed up later because the same player action might be resolved differently on the server and the player's client. In general, latency compensation techniques can help game systems ameliorate the negative effects of network latency by reducing perceived latency or making the game fair for players who have higher network latencies.

Many studies are being done on traditional latency compensation techniques, their effects in certain situations and also their improved versions. Newer techniques are also being developed in the everlasting battle with network latency. However, the effectiveness of latency compensation techniques covering the wide range of game types and game actions is still not fully understood. For game designers and developers to deliver the best Quality of Experience

(QoE) to players, more understanding is needed of how different latency compensation techniques can be used effectively for all types of games and all types of actions inside a game.

This project studies how different amounts of lags affect players with different types of game actions in a 2D platformer game, and the impact of the player prediction latency compensation technique on player performance and QoE under such conditions. I first designed and implemented an original 2D platform Contra-like game with a classic authoritative server-client networking architecture. I then implemented player prediction where the client predicts the result of player actions before contacting the authoritative server. After that, I conducted a user study of player experiences under different amounts of network latency, with and without the latency compensation technique. In each case, I measured the players' objective performance and asked them for their subjective opinions with short survey questions.

Results from analysis of 17 players shows that, on a range of network latencies from 0 ms to 800ms, when the latency compensation technique is enabled, players feel an increased responsiveness and generally improved gameplay experiences. However, player performance does not differ much from when the latency compensation is disabled since the latency compensation does not affect the game logic on the server side. Overall, 400ms of latency is the approximate threshold where the benefits of latency compensation starts to be outweighed by its disadvantages.

The remaining chapters of this thesis are organized as follow: Chapter 2 provides background concepts and related works; Chapter 3 describes my methodology in designing and implementing the game and also the latency compensation technique; Chapter 4 describes the user study and analyzes the collected results; Chapter 5 summarizes the conclusions and mentions possible future works.

2. Background & Related work

This section provides background and related works, including the Dragonfly engine, the concept of latency, latency compensation techniques, and previous research done on latency in games.

2.1. The Dragonfly engine

Dragonfly is a text-based game engine designed to teach students how to make a game engine as part of a university class [6]. Dragonfly uses the Simple and Fast Multimedia Library[1] (SFML) as its graphics engine to draw game objects and interfaces on the screen, as well as loading and playing music and sound effects. Although Dragonfly only supports ASCII graphics and is single-threaded, it fully supports the core functionalities of complete games such as sprite rendering and animations, collision detection, game world hierarchy, message logging or custom scripting.

We used a networked version of Dragonfly for the implementation of our program. In this version, Dragonfly had been extended to make use of TCP sockets to support client-server communication, including setting up network nodes as either servers or clients, marshalling and unmarshalling basic game object data and transmitting information between network nodes.

The art sprites used in Dragonfly games are made up of ASCII symbols rather than individually colored pixels, which makes the art pipeline different from what most game artists are used to. Most game artists use industry standard 3D software such as Zbrush, Maya, and Blender to make 3D objects for game assets, drawing software like Photoshop or Illustrator to make 2D game assets. Dragonfly, on the other hand, only supports sprite sheets in text files containing ASCII symbols with a very specific format which is shown in Figure 1. The header includes the number of frames in the sprite and its size and color. The body of the file include the frames with an “end” line after each frame. The footer is an “eof” line after all the frames are listed.

[1] https://www.sfml-dev.org/

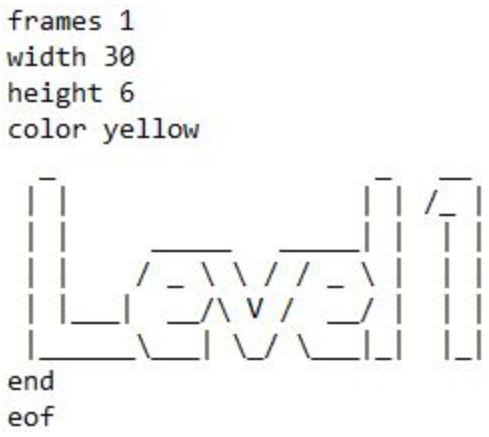

Figure 1. Example Dragonfly sprite

2.2. Latency

In online games, there are two sources of latency that could potentially have negative impacts on gameplay: local latency and network latency.

The first source, local latency, is comprised of latency from the local computer [7]. A small amount of local latency (4-8 milliseconds) comes from the polling of user input from input devices. Different kinds of monitors have different refresh rates and can add up to 66 milliseconds to the local latency. Additionally, a large portion of local latency comes from the processing delays of the game software and local graphics rendering, known in general as graphic latency [8]. In most cases, graphic latency comes from the CPU (central processing unit) and GPU (graphics processing unit). The game screen rapidly displays static images called frames, measured in frames per second (fps). Generally, the higher the frame rate, the smoother the picture is and the smoother the game plays. In contrast, if the frame rate is low - for example when the game software is taking too long for its calculations - the picture feedback appears as inconsistent or jittery animations, known as graphics lag. However, this is not the kind of latency that this thesis studies about.

The second source is network latency, which is caused by network connections between game servers and clients and often referred to as “ping” by gamers. In computer networking, latency is the time it takes for information to travel from a source computer to the server and back to the source computer [9]. This latency mainly comes from 3 sources, all of which add cumulatively to the total network latency experienced. The first source is propagation delay,

caused by the physical distance between the client and the server. The speed of light is the upper bound of how fast data can be transmitted through a particular medium. This puts a lower bound on latency between points on the Internet based on how far they are geographically. When this distance spans thousands of kilometers, the propagation delay can become noticeable. The second source is serialisation delay, which is the time taken to transmit an IP packet one bit at a time. This delay mostly depends on the speed of each network link (in bits per second), which is sometimes imposed on the connection by the Internet provider, and the length of the packet. The third source is queueing delay. When multiple packets arrive at the same router faster than it can process and transmit, they will be queued up and transmitted one after another. The queuing delay that a packet in the queue experiences comes from the serialisation delays of all the packets ahead of it in the queue.

When these delays add up to a value large enough, players are aware of the time difference between when commands are made and when the game responds. As a result, high network delay in games make the game feel unresponsive and can result in game objects being teleported or out of control. In multiple game genres where timing is crucial such as First Person Shooter (FPS), racing games or rhythm games, high network latency greatly degrades the gameplay experience because player actions are slowed down by this delay time. The game can even become unfair when players who have different amounts of latencies play together. For example, consider two players working together to defeat a level, one with really high network latency and the other without. After the level is completed, the server spawns some rewards for them to loot. The player with low latency will receive the update from the server first and will be able to net on the dropped loot faster. In general, high network latency causes online gaming to be less enjoyable.

In online games, the effect of latency depends upon each game action's precision and deadline [10]. Precision means the accuracy required for the players to complete the action successfully. For example, players need to be accurate if they want to shoot an opponent who is far away. Deadline means the amount of time players can have to achieve the final outcome of the action. For example, in a rhythm game, because songs with a slower tempo give players more time to press the required sequences of buttons, player actions have looser deadline than in faster songs. In general, even the smallest amount of latency can affect game actions that require high precision or have tight deadlines, while having little effect on game actions with low

precision and loose deadlines. A part of my study was to learn about how network delay affects different types of game actions in the chosen game genre.

2.3. Latency compensation techniques

Latency compensation techniques are methods for games to mitigate latency to reduce the effects of latency from the player's perspective. Two commonly used latency compensation techniques are player/opponent prediction and time manipulation techniques.

With player prediction, the client takes the user input, tries to predict the server's response to that input and renders the player actions immediately. After the client receives the authoritative response from the server, it fixes up any inconsistency between the server state and its predicted state. Player prediction can make the game appear immediately responsive because the results of the player actions are shown before the client communicates with the server. As a result, the game appears to be running with no network latency. On the other hand, with opponent prediction, the client tries to predict units that are not controlled by the player but by other players or the server. This approach works well when units in the game world are static or have constant movement because the predicted states of the units will be computed and used at each client without the client having to receive many state updates over the network. However, when units move unpredictably, like in a chaotic first-person shooter game, there may be a lot of inconsistencies between the clients' predicted states of game units and their actual states. As a result, the benefits of trying to predict other units' locations diminish.

Time manipulation is an approach to account for the difference in latency among clients. Messages from a client further away from the server take longer to arrive than those from a client closer to the server, which leads to unfairness in gameplay. Time delay is one technique of time manipulation. With time delay, the server delays client commands and/or world state updates for the client closer to it, so that this client has the same latency with the client further away from the server. A widely used time manipulation technique is time warp. With time warp, when the server processes a message from a client it rolls back the game world to the time when that message was sent and executes the message at that time. By using this technique, the server can still execute commands that come from clients with a high latency at the time that those commands were made in the game by the player.

Brun et al. (2006) described latency compensation techniques as "trading inconsistencies" [11]. For example, *player prediction* reduced the perceived response time of the program upon player input, but could lead to incorrect predictions resulting in these predictions having to be revoked and seen by players as local roll-backs on their machines. *Time-warp* let players aim their hit exactly at the opponents' positions being shown on their computers without having to lead the target based on their network latency. The server would resolve the hit correctly no matter how much latency the players had, but this process would delay hit registrations perceived by players and can lead to “shots-around-corners” where the hit confirmations from the server arrives at the clients of the hit victims after the victims have moved behind walls where they cannot be hit. These inconsistency trade-offs are crucial to understand for game developers and designers in order to deliver the best Quality of Experience (QoE) to online game players.

2.4. Related work

Previous works have proven that not all types of traditional games are equally latency-sensitive. In general, online games with the Avatar model – where players are represented in the game world by a single character – are more sensitive to network latency than those with the Omnipresent model – where players can interact with different aspects of the game world simultaneously [10]. More specifically, games with a first-person perspective, like racing games or first-person-shooters, are most susceptible to network latency. Games with a third-person perspective like sports games or role-playing games can be more tolerant towards network latencies up to 500 milliseconds. Games using the Omnipresent model such as real time strategy games are the least latency-sensitive. Researchers have studied the impact of network latency on the most popular genres like First Person Shooters (FPS) [13], Sports [14] or Real-time Strategies (RTS) [15]. However, little work has been done on 2D platformers, which also belong to the Avatar model and might suffer from even low amounts of network latency. Our work attempts to clarify the effects of network latency on 2D platformers by studying how player performance and QoE are affected in an original 2D platformer game under a range of network latency.

Many methods are being discovered to deal with the problem of network latency in traditional online games. Moller et al. (2019) showed that gamers could adapt to a constant delay by predicting and performing actions earlier in games with predictable parameters, and

proposed a network resource allocation scheme that gives more priority to games that are more difficult to adapt to [16]. Cooper et al. (2014) showed that a player's lag can have negative effects on the Quality of Experience (QoE) of other players within the cooperative group and suggested the reduction of the latency of the most lagged player in the group to improve the overall QoE for all players [17]. Dynamically adapting game parameters to network latency, such as modifying object sizes [18] or changing the pace of the game [19], was experimented by Kim et al. (2019) and Griwotz et al. (2018) and yielded promising results despite needing extensions to more complex tasks and games. Chang and Lee (2018) proposed an improved version of time-warp called Advanced Lag Compensation for multiplayer shooting games to greatly reduce the number of shots around corners by giving the victim a chance to appeal to the hit upon being shot at [20]. Graham and Savery proposed simplifying the expressions of latency compensation algorithms by making time an integral part of the programming model with timelines [21]. These timeline objects stored all past values and the time these values were inserted and are synchronized between network nodes automatically, thus enabling programmers to quickly access past states and predict future states. Although timelines provided many benefits, they required the whole shared objects to be sent over the network, therefore were not suitable for large data structures and needed more optimizations. All of these new methods proved promising in compensating for latency in certain cases, but they left even more space to fill in the gap of knowledge on how different latency compensation techniques would affect different types of games and actions in games, and how players would benefit in these situations. Our work puts player prediction - a classic latency compensation technique - into our original 2D platformer to study how effective the technique is in the 2D platformer genre.

Recently, more efforts are being put into studying the effects of network latency in cloud gaming. Cloud-based games keep the game content and computations on the server, enabling users to have lightweight game clients that are mostly responsible for sending user input to the servers, receiving and decoding graphical frames from the cloud. Chen et al. developed a model to predict how cloud-gaming friendly a game is based on the game pace and the frequency of user actions [22]. The work also showed that cloud-gaming latency affects different games in different ways. Claypool and Finkel (2014) showed that latency affects cloud-based games very similarly to traditional first-person avatar games, the most sensitive class of games [23]. Many researchers are finding new ways to combat network latency in these sensitive games. Bruzgiene et al. proposed a method surrounding equalizing the up and downlink delays in

real-time for all players to compensate for latency in cloud FPS games [24]. Chu et al. developed Outatime, a system that mask up latency effects for more lightweight cloud games by sending future possible frames to the clients ahead of time and quickly recovering from mispredictions [25]. These efforts, while aiming at a newer generation of online games, provided little help to traditional games, which accumulate to the majority of online games and game players on the current market. Our work helps provide a better understanding of how network latency affects an online 2D platformers with a classic authoritative server-client architecture.

3. Methodology

In order to evaluate the effects of network latency and the impact of the player prediction latency compensation technique in a networked 2D platformer game, we built an original platformer from scratch so that we could control all game actions and latency. We then invited players from the university to play the game cooperatively as pairs.

3.1. The game: Game design & Features

To control all the actions in the game, we built our test game from scratch using the Dragonfly game engine. Our primary inspiration was Contra (Konami, 1987), which is a well-known classic platformer. We also looked at more modern 2D platform shooting games such as Cuphead (Studio MDHR, 2017) and the Mega Man franchise (Capcom). These games are designed so as to not lose the player's interest even though all the levels were extremely difficult to beat. Our design goal was to achieve the same balance between high difficulty and satisfaction.

Our game features robot themed arts with ASCII graphics, as shown in Figure 2. Since SFML - the multimedia engine used by Dragonfly - supports drawing both color filled pixels as well as ASCII characters, we had the options of ASCII graphics, pixel graphics or some combinations of the two. After some prototypes, we opted to go for ASCII graphics since it is already supported by Dragonfly and graphics complexity is not central to our project. Due to the limitations in computational capabilities of the engine, we chose a relatively low resolution of 180x70 characters to ensure that the program can run smoothly on campus machines. With this resolution, it was much easier to achieve fine details with ASCII characters for hard-edged objects than for soft objects. Therefore, it was natural to go for a robot themed game.

Figure 2 depicts a screenshot of our game. The player controls the avatar - the Hero - on the left and move to the right to reach the end of the level. On the way, the player has to move at an agile pace through platforms to avoid the enemies coming from the right and their bullets. The player can use different kinds of weapons to kill the enemies in the level, including a Boss at the end. Because the player can attack 360 degrees around the avatar, we call our game "360 Gunner", or 360 for short.

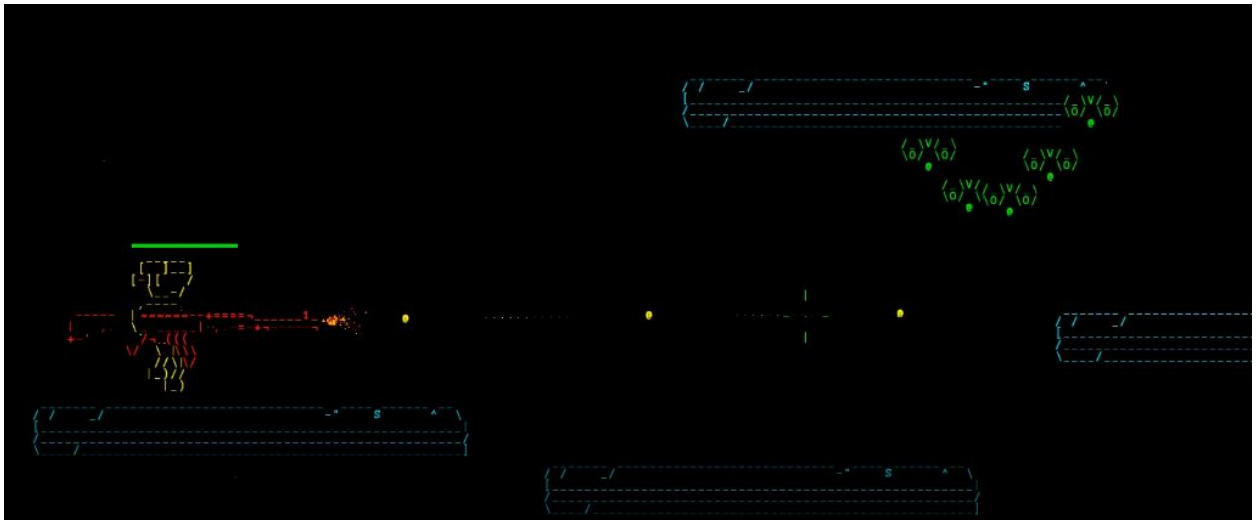

Figure 2. 360 Gunner screenshot

Our game features realistic shooting mechanics similar to those found in typical FPS games, but in 2D. For example, an assault rifle can fire light and fast bullets at a very high rate, but the recoil causes successive bullet spraying to have a continually decreasing accuracy. A sniper can be very inaccurate within the reticle area while unscoped, but has perfect precision while scope mode is on. Because we aim for a realistic feeling for the weapons, we choose realistic sound effects to fit these attacking mechanics.

There are 3 types of weapons in our game: AK-47 - an assault rifle type weapon, AWP - a sniper type weapon, and Grenade Launcher. Each weapon features different properties such as damage, projectile speed, accuracy, fire rate, area of effect or recoil. The first type of weapon, AK-47, represents a game action with medium level of deadline and precision. The second type of weapon, AWP, represents a game action with tight deadline and high precision, because the raycast is very small compared to the hit boxes of enemies. The third type of weapon, Grenade Launcher, represents a game action with loose deadline and low precision, because the projectile can cover a good area around its path. These weapon characteristics are crucial to examine when evaluating the effects of latency, and are explained in more detail later in our thesis.

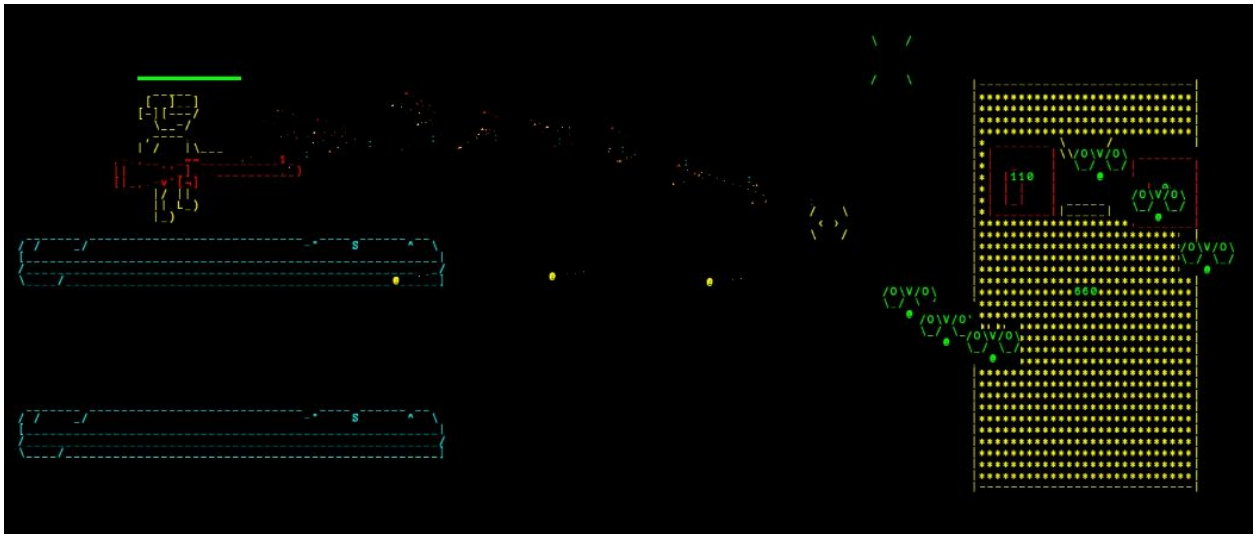

Figure 3. Boss fight screenshot

There are two levels in the current offline version of the game: a tutorial level and one playing level. In the tutorial level, players are guided to walk through the basic moving and shooting mechanics of the game through text messages on the screen. The level is completed when players have carried out all moving tasks and tried out all weapons. In the playing level, players make progress forward through a relatively small map and fight off waves of enemies before encountering a boss at the end of the level. Figure 3 depicts a scene in the boss fight where the player is launching a grenade with the Grenade Launcher at the red-eyed boss and the minions on the right. There are multiple levels of platforms for the player to use to avoid taking damage. The game pace in this playing level was tuned so that players have to be agile between platforms to attack and avoid getting hit at the same time. A fellow WPI student also helped us by composing two background music tracks, which fit the fast-paced gameplay and the retro aesthetics of the game very well for their funky feel and tempo. After the boss is defeated, the level is completed.

The game has 2 versions: an offline and an online version. The offline version includes the 2 levels as described above. The online version was used in our study of network latency and latency compensation. It contains only the playing level and is playable with either 1 or 2 players. In the latter, two players can connect to the server and play together remotely. In this case, the enemies health and damage are scaled up by 2 to keep the same level of difficulty.

3.2. The game: implementation

We started with an offline single-player version, then mitigated the whole program into an online version where 2 players could join a game and play cooperatively. We then implemented player prediction into the game as a functionality that can be toggled on and off by the server. The entire implementation was done with the Dragonfly engine in Microsoft Visual Studio using the C++ programming language.

The offline game includes several main components: the player character, the weapon system, projectiles, bots and bot behaviors and the levels system.

The player character class, the Hero, keeps track of player stats like movement speed, jump height or health. It also keeps track of all the weapons currently owned by the player. Additionally, it's responsible for receiving and processing user input, including moving, jumping up and falling down from platforms, crouching, shooting, reloading, switching between weapons and turning the sniper scope on and off. The class also carries out other game logic related to the hero such as taking damage on collision with enemies or enemy bullets.

```
Weapon::Weapon(std::string weaponName, WeaponType weaponType, Hero* owner, int bulletSpeed,
    int fireRate, int ammoLoadedMax, int ammoBackupMax, int dmg, bool affectedByGravity,
    float bulletWeight, float radiusOfEffect, float reloadDuration) {
```

Figure 4. Weapon object constructor signature

To achieve a highly customizable weapons system and more realistic attacking mechanics, the weapon class supports different weapon types with various properties as shown in Figure 4. The fireRate variable determines how fast the weapon can fire successive bullets. The ammoLoadedMax variable determines how many bullets a weapon can shoot before needing to be reloaded, also known as the clip size. AmmoBackupMax is the amount of backup ammo the weapon has. The bulletSpeed, affectedByGravity and bulletWeight variables together determine the path of the projectiles after the bullets are fired from the weapon. RadiusOfEffect decides whether the bullets will create an explosion on collision with targets, and how large the explosion will be. ReloadDuration dictates how long it takes for the reload action to be completed. All weapons operate under the same logic using these different parameters. After the fire button is pressed, the active weapon is called upon to fire a projectile at the reticle

position. If the ammo clip is not empty, an actual target is then calculated based on the accuracy of the weapon, then a projectile is created and launched from the weapon and the ammo count decreased. After firing a bullet, each weapon slightly decreases in accuracy for different degrees due to the weapon recoil and becomes stable again after a short time not firing any bullets. Different types of weapons also have distinct behaviors. For example, the sniper allows players to aim at enemies in scope mode, where the accuracy is absolutely true to the center of the reticle.

The projectiles are created whenever a player fires a weapon or an enemy bot attacks. Projectiles therefore have 2 types: HERO_BULLET or ENEMY_BULLET, and use the type to deal damage to the right entity on collision. Since the size of each bullet is usually as small as an ASCII character (unless it comes from a Grenade Launcher), there are cases where a bullet moves too fast such that it goes through the target without the collision being detected. This is a classic problem in game development that appears even in industry-standard engines. For example, in Figure 5, a bullet (B) on the left is moving to the right and towards the target (T). At the frame update right before B reaches T, B is at Position 1. If B is not moving too fast, a collision is detected on the next frame update because B is at Position 2 and partly overlapping with T. On the other hand, if B is moving too fast and ends up behind T on the next frame update (Position 3), the collision is overlooked by the engine. There are multiple ways to solve this problem such as decreasing the speed of the bullets or increasing the sizes of the bullets or the enemies, but these solutions place restrictions on the game design. Therefore, we implemented bullet trails as a simple solution. At every update, a bullet moves to the next position and creates a trail between this position and the position on its last frame with BulletTrail objects. These BulletTrail objects can be seen as little dots following the bullet (Figure 6). In case the bullet moves too fast, like from Position 1 to Position 3 in our example, BulletTrail objects will still collide with the target and inform the bullet to deal damage to the right entity. BulletTrail objects also provide an interesting visual effect showing the movement of the bullets.

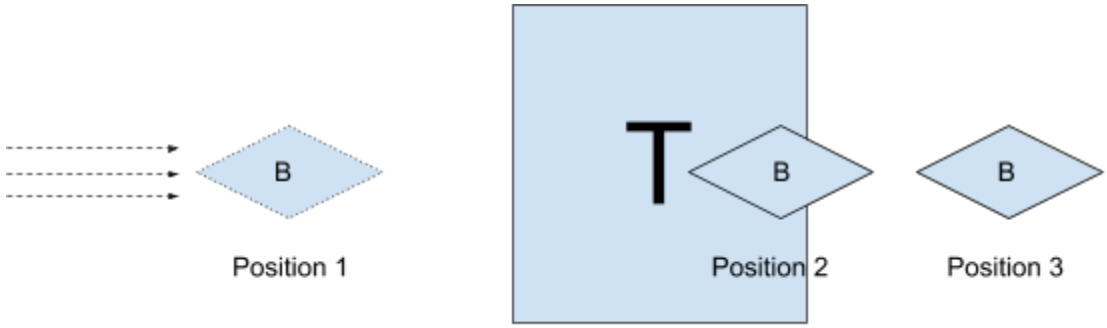

Figure 5. Collision undetected problem

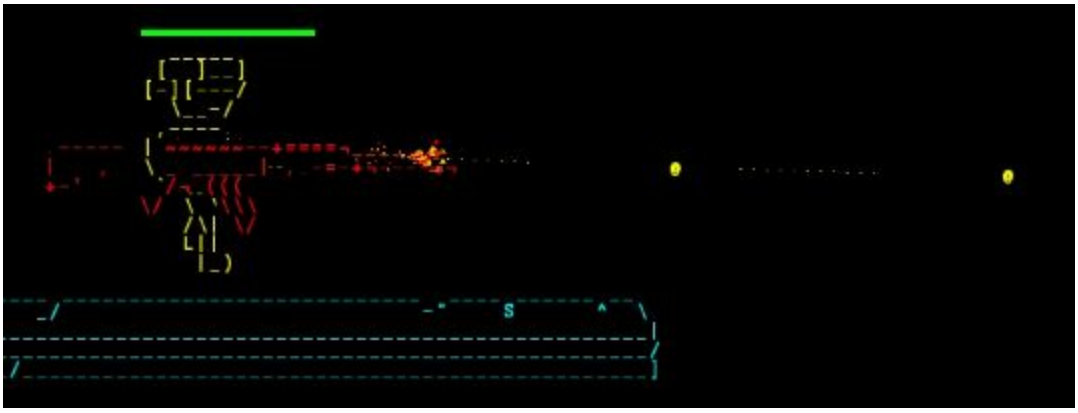

Figure 6. Bullet trails

Similar to weapons, enemies in 360 Gunner have multiple parameters such as health or damage. Each bot's behaviors are also determined by two properties: movement type and attack type. Each state update, the bot's EnemyMovement entity determines where the bot moves to, and its EnemyAttack entity determines if it's time to fire a bullet and where to attack. In the current version of 360, enemy bots can chase the hero, fly up and down, fly straight across the screen or fly in a sine wave path or a spring-like path. Bots can also attack the hero by firing fast bullets or slow double bullets. These behaviors are adequate for one playing level, and are easily extendable for further bot designs and implementations with the EnemyMovement and EnemyAttack entities. In addition to minions, a boss is a special type of enemy that needs to be defeated to get through a level. A boss has attached parts called WeakPoints that take critical damage when hit. A WeakPoint has its own total health and is destroyed when its health reaches 0 or the boss is defeated. For example, in the boss fight in Figure 3, the red eyes of the boss can be hit for 5 times the regular damage of a bullet. This allows for more interesting boss designs in the future.

The level system lets players select which level they want to play. After a level is selected, it initializes its own landscape and spawns the hero at the starting point. The level object keeps track of the hero's progress throughout the level and spawns correct waves of enemies accordingly. At the end of the level, the boss is spawned and the boss fight music is played. When the level concludes, a "Game Over" or "Level Completed" message is displayed, and all the game world objects are cleaned up.

Figure 7. Weapon and ammo display

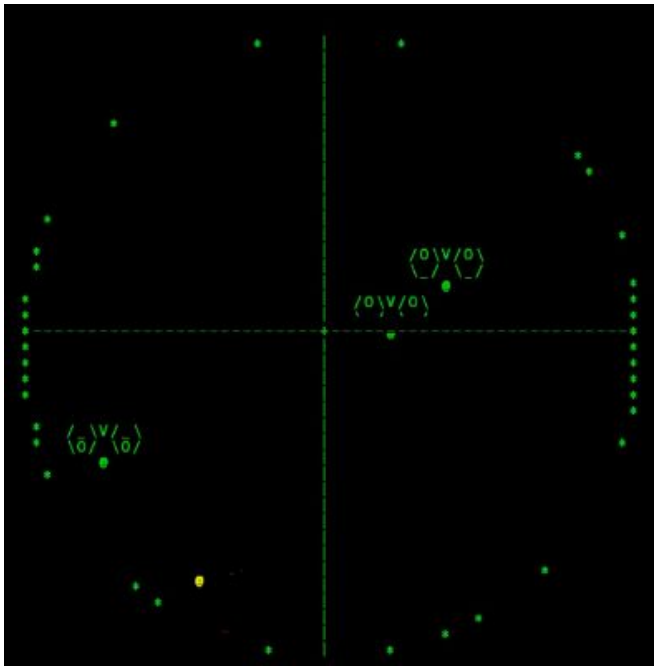

Figure 8. AWP Scope mode Reticle

Besides the main game components, there are also visual elements implemented into the game that are necessary for a shooting game. The Weapon and Ammo Display placed on the bottom left of the game screen uses information collected from the hero object to display the name of the current weapon activated and how much loaded and backup ammo the weapon has left. Figure 7 depicts this visual element where the AK47 is being used with 28 loaded bullets and 90 backup bullets. The Reticle has a custom look and also behaves differently for each type of weapon. The size of the Reticle helps demonstrate weapon characteristics. It's slightly enlarged when bullets are fired and weapons become unstable, and is shrunk back to normal when the weapon is not fired and becomes stable again. It also has a scope mode for the AWP, when players can focus on a very small point on the screen. Like depicted in Figure 8, when this scope mode is activated the player can only see what is currently inside the scope of the gun. However, the target of the hit is the exact center of the scope view, and the damage will be dealt instantly. Additionally, there are also multiple particle effects to make the game look better, such as enemy deaths or grenade explosions.

After the single-player offline game was fully implemented, we brought it to a game testing event on campus called AlphaFest. The game received attention from the public and

generated excitement throughout the whole event. Although the graphics style of the game was not suitable for a portion of playtesters, many players described the game as “appropriately frustrating” as it keeps them playing despite having many deaths back to back. We also received constructive feedback from the players that we took into account when mitigating the game into the online version. For example, most players did not like how hero jumping felt because it was too quick and heavy, so we made the jumping action of the hero smoother and lighter for a better game feel. Most players also felt like the Grenade Launcher was too difficult to use, so we slightly increased the bullet speed and also implemented a path preview for the projectile whenever the player is using this weapon.

We started implementing the online version of the game by using a client-server architecture with a thin client. The client’s only main duty is polling for mouse and keyboard input from the players and sending it to the server through network messages. The server, on the other hand, is responsible for handling user input, performing game logic such as spawning enemies, handling events like collisions, updating all game objects and sending back to clients all necessary changes to the game world every frame. Because two players can play cooperatively in the online version, there are potentially two identical heroes in the game world. This requires parts of the existing logic to be changed because certain subsystems in the single-player version, such as the reticle or the Ammo display (Figure 7), rely on the assumption that there is only 1 hero in the world, e.g. looking up the list of hero objects in the game world and selecting the first one in the list. Since in Dragonfly, all objects have unique object ID numbers unless enforced otherwise by the programmer, we used 20 and 40 to be the ID’s of the hero objects, and the ID’s for other objects start at 100. The first client who connects to the server uses 20 as their hero ID, and the second 40. Everytime a client connects to the server, the server checks if the client is player 1 or player 2, and sends a message to the newly connected client containing its hero ID number. With this information, the clients can display game elements correctly, apply synchronize messages from the server to their corresponding heroes and set the game view to follow the right hero. In addition to engine-specific information like an object’s position or acceleration, certain functionalities on the screen, such as the Ammo display or a hero’s health bar, also require the server and clients to exchange object synchronization messages containing implementation-specific information, such as the hero’s hit points or the remaining ammo of the weapons a hero has. This is achieved by overriding the default marshalling method of the necessary object classes on the server to create network

messages containing extra information, and overriding the default unmarshalling method of the same classes to read this extra information on the clients.

After mitigating the single-player game to the client-server architecture, two players can connect to a previously started up server through a simple command line specifying the IP address of the server's computer and play the game. When a client connects, they are immediately moved into the game world of the playing level. Their keyboard and mouse input are received by the game client and sent to the server for processing. At the server, these inputs are translated to the correct game action (moving the hero, firing the gun, etc.). Any updates to any objects (new positions or movements, new objects created, objects destroyed) are then sent back to the client. The client receives these updates and applies the received information to the correct game objects using the aforementioned object ID. This chain of actions repeats until the boss is killed.

To mitigate the effects of latency, we implemented player prediction for player movement, which included moving forward and backward, crouching, jumping and dropping down from a platform. The goal was to make the movement of the hero seem smooth and responsive even when there is network latency involved. When player prediction is enabled, in addition to sending player's keyboard input to the server, game clients also forward this input to their main hero object to execute. Therefore, movement code is executed similarly on the server and the clients. This also requires clients to be able to handle collision events between the heroes and platforms, a responsibility previously performed only on the server. This is because when heroes land on platforms on the client side, the clients should not have to wait for the server to send back important information such as the heroes' new velocity or their ability to jump being reset. Additionally, there is potentially no need for frequent update messages to be sent from the server for player movement, because the movement code is always executed similarly on the server and the clients. Therefore, the server, after receiving movement commands and executing the necessary code, can clear the update flags for the newly updated attributes of the heroes, such as position, velocity or acceleration, so that these attributes are not included in the network messages distributed to the clients later on. However, this only works if there is only one client connected to the server. For example, assume two players are playing at the same time. Player 1 issues a movement command and the code is executed on player 1's game client. The command is also sent to the server and executed on the server

similarly. If the server now clears the update flags and does not notify the clients about what happened to player 1's hero, the changes will not appear on player 2's client. Therefore, player 2 will have an incorrect representation of the game world. For this reason, we decide to send hero update messages whenever changes happen to the heroes on the server, and let the clients decide on their own when to use these update messages.

These hero-related update messages are used in several ways when received by a client. A client applies updates of the hero of the other client as soon as these updates are received, so that it has the closest representation of the game world to that on the server. On the other hand, when a client receives updates of its main hero, it does not apply the movement-related information in the message, because these changes are already present in the client since the movement code is already executed beforehand. Instead, the client uses this information only in cases where inconsistencies happen to fix up the hero's position, such as when the position of the hero on the client is too far away (55 spaces or more) from the position received from the server. Fix-ups also happen when the main hero takes damage from any source. In these instances, if the hero is not within a certain distance (more than 28 vertical spaces or 54 horizontal spaces) from the place where the damage happened, it's also snapped back to the position received from the server. We experimented with gradually fixing up the hero's position over time - moving the hero somewhere between its current position and the correct position every frame so that it keeps getting closer to the correct position. However, we found out through these experiments that the method does not work well for the genre because there are cases where the position of the hero is incorrect on the vertical axis. If the hero is on a platform and a positional fix-up containing a vertical component needs to be done, he will be placed somewhere below the platform and start falling down, which produces jittery effects. Therefore, snapping the hero to the fix-up position is selected for simplicity.

Besides hero movements, weapon switching is also similarly predicted on the client for a constant responsiveness with the movements. When the player hits the switch weapon button, the command is executed on the client and sent to the server for execution at the same time. After the hero on the server switches to the next weapon, the server sends back a synchronization message to the client confirming the hero's active weapon. If for some reason this active weapon is different from the active weapon on the client, the weapon will be changed to the correct one.

There are also other elements that are not synchronized between the clients and the server. For example, the Ammo Display and the Reticle, as described above, are only present on the clients and do not exist on the server. These elements use information acquired from the main hero of their clients to function properly. Additionally, particle effects are used when game elements on the clients receive meaningful updates from the server. For example, when an enemy object on the client receives a message from the server stating that it has been destroyed, it then triggers the enemy death visual effect. When a weapon receives an update from the server that its bullet count has decreased by 1, it triggers the firing particle effects. Not synchronizing these elements helps reduce network data as much as possible.

We also look to further reduce the network traffic between the server and the clients since our game has potentially many objects to synchronize. For example, when the player is spraying bullets from the AK47 at a wave of enemies, there are many bullets and bullet trail objects coming from both sides being synchronized over the network at the same time. This causes problems when the game world is too crowded because the engine is not provided enough frame time to handle network messages. The unhandled updates are then delayed until further frames, causing objects not receiving updates to appear frozen on the screen. To reduce network traffic, we stop the synchronizing of bullet trail objects, which account for a large part of all game objects in the world. Instead, each bullet on the client, after receiving updates from the server about its new position in the game world, creates bullet trail objects between its last position and this new position, the same way bullet trail objects are created on the server. However, bullet trails only generate collision events on the server. This way, both the functionality and the visual effect are preserved without taking up lots of network traffic. We also give all bullet trail objects the same object ID of 0 to maintain a consistent object ID assignment scheme for new game objects when they are created.

3.3. Experiment Design

The experiment is designed to answer these research questions:

(1) How does player prediction for movement change the players' gameplay experience under different network conditions?

The prediction of movement on the client should make the game feel more responsive even when there's a high amount of network latency, because player input is performed immediately on the screen. However, there is a tradeoff between responsiveness and consistency. Therefore, we want to assess whether player prediction helps make players' gaming experience better. Additionally, we want to study the latency threshold where the visual inconsistencies outweighs the benefits of player prediction.

(2) How are different types of game actions affected by different amounts of network latencies?

Not all game actions inside the game are equally affected by network latencies. Therefore, we want to study the characterization of the implemented types of weapons in terms of precision and deadline, and the degradation of different types of weapons as the amount of network latency increases. Generally, a game action with higher precision and tighter deadline will be more severely affected by the increase in latency. We want to assess whether this hypothesis is true in our case.

(3) Does player prediction for movement help improve player performance?

Player prediction makes movement feel more responsive, but introduces more visual inconsistencies to the screen. Furthermore, the logic on the server side stays the same whether player prediction is on or off. Therefore, we want to assess whether player performance separates much in both cases.

We used Clumsy ((https://jagt.github.io/clumsy/) on the server machine to inject symmetrical network latency to the connection between the server and the clients. There are 5 values of latency added: 0ms, 100ms, 200ms, 400ms and 800ms. The latency is symmetrical, meaning packets take an equal amount of time to travel from the server to the clients and from the clients to the server. This is simulated with Clumsy by giving both the inbound and outbound delay on the server machine the value of half the total latency wanted. This also gives all the clients connected to the server at the same time the same amount of lag. Each latency value was also used twice in playthroughs, once with no player prediction and once with player prediction turned on. Therefore, there were 10 playthroughs in total for players to play. Each playthrough had a 2-minute timer, so the level would end when the timer is up or the boss is killed. The values of network latency and player prediction were also shuffled to avoid players getting used to the delay in gameplay.

We created two surveys with Google Forms to collect data from the players. The first survey is used to collect player background information. This survey asks the players for their age and gender, how many hours they spend each week playing games and the genres of games they are most familiar with. The second survey is used to collect subjective data from players after each of the playthroughs of the game. It asks the player to rate, from 1 (low) to 5 (high), how much lag they experience, how responsive the game is, how many visual glitches they see, how much lag affects their gameplay, how challenging the level is and how much they enjoy the playthrough.

The experiment was conducted in the WPI Zoo Lab. The clients were installed on two Zoo Lab machines before the experiment started. Each of the two client computers is equipped with an Intel Core i7-8700k @ 3.7GHz, a GTX 1080 and 64GB of RAM. The server was run on a personal laptop equipped with an Intel Core i7-7700HQ @ 2.8GHz, a GTX 1070 and 12GB of RAM. Both of these setups are capable of running the game at the designated 30 frames per second. The frame rates are the same at the server and the game clients. While setting up the Lab to prepare for the experiment, the offline version of the game, which contains the tutorial level, was installed on the two client computers. The two client computers were chosen so that players would face opposite directions and could not see each other during gameplay. This was to minimize the distractions and influences players might get if they looked at each other's computer during the game. The laptop used to run the server was placed on the other side of the room so that players would not feel pressured from being watched by the experiment conductor.

We had 17 players from Worcester Polytechnic Institute participate in the experiments. Among the 17 players, 14 participated in the experiment in pairs. The other 3 played the online game solo. When they arrived at Zoo Lab for the experiment, they were first given a general consent form that contains information about the general purpose of the study and the procedures of the experiment. The experiment was also described to the participants as voluntary, risk-free and confidential about participants' personal information. Prior to each experiment session, the offline game was started up on the client computers. The background survey and the gameplay survey were also opened up so that players could easily access them. After the players read the consent form and signed it, they were directed to their computers to fill in the Player Background survey. After the survey was completed, we told both players to

learn the game's controls through the tutorial level in the offline game. The level was short and took no longer than 2 minutes to walk through for all players. Meanwhile, the server was started up on the server machine along with Clumsy. After both players had completed the tutorial level, they are guided to start the online game clients on their computers. When the clients connected to the server, players were in a start screen with a Ready button. Once both players hit the Ready button, the level started. Both players' objective gameplay measurements were recorded in log files as the players played the game. After the boss had been defeated or the 2 minute timer had ended, the level was complete and players were moved back to the start screen. Players then switched to the web browser tab to answer the gameplay survey questions. While players were filling out the survey, we adjusted the network latency on Clumsy and the latency compensation to prepare for the next playthrough. After the players had completed the gameplay survey, they switched back to the game clients and started playing the next playthrough. These chain of actions repeated until the players finished all the designated playthroughs. The order of latency and player prediction values for the games played is as followed:

- Game 1: 0ms -prediction on
- Game 2. 100ms - prediction off
- Game 3: 400ms - prediction on
- Game 4. 800ms - prediction on
- Game 5: 200ms - prediction off
- Game 6: 200ms - prediction on
- Game 7: 800ms - prediction off
- Game 8: 100ms - prediction on
- Game 9: 400ms - prediction off
- Game 10: 0ms - prediction off

4. Study results

In this chapter, we present the players' background information that we collected before starting each experiment session. We also summarize the subjective data from gameplay surveys and objective gameplay measurements from the log files collected after each playthrough. We then provide our analysis of the data collected to answer the research questions presented previously.

4.1 Player demographics

The following graphs are retrieved from the demographic information survey given to the user study participants. There are 17 players who participated in our study in total and 17 responses for each of the demographic questions.

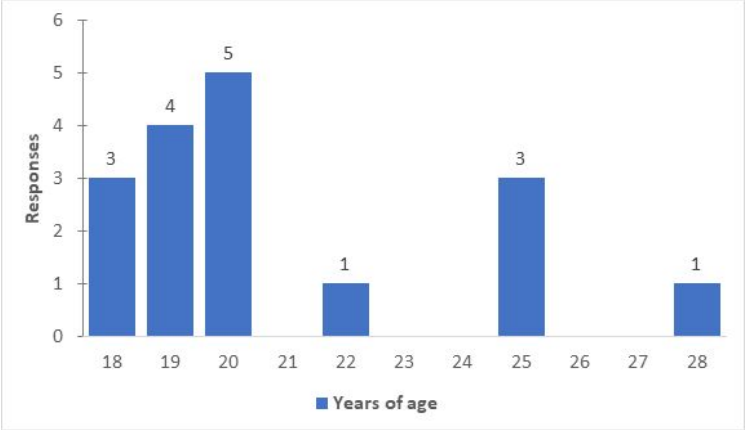

Figure 9. Player's age

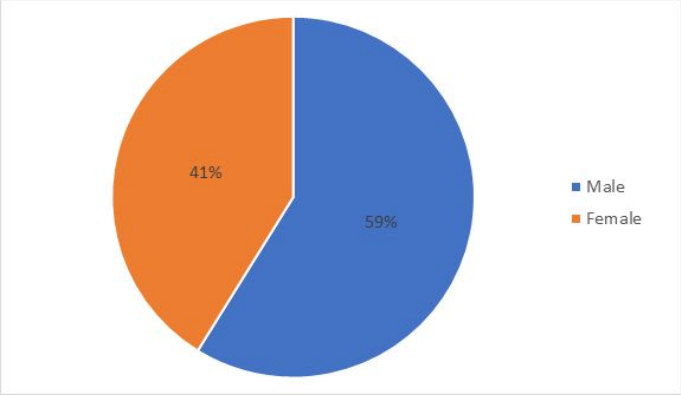

Figure 10. Player's gender

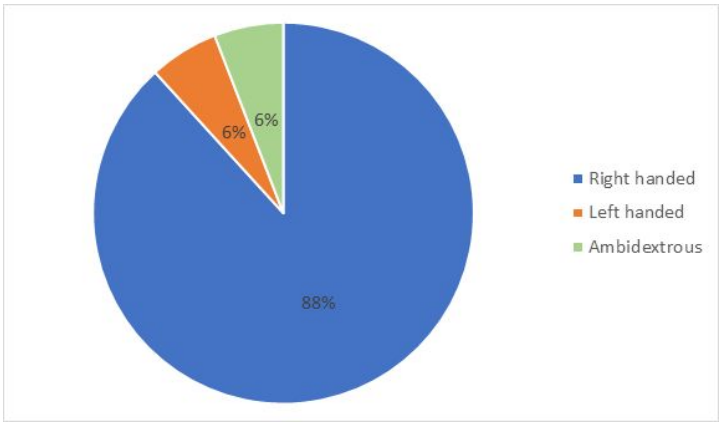

Figure 11. Player's handedness

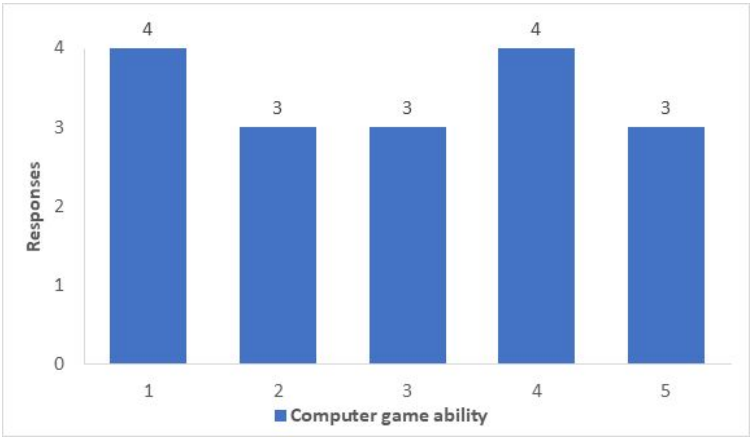

Figure 12. Player's self rating of computer game ability

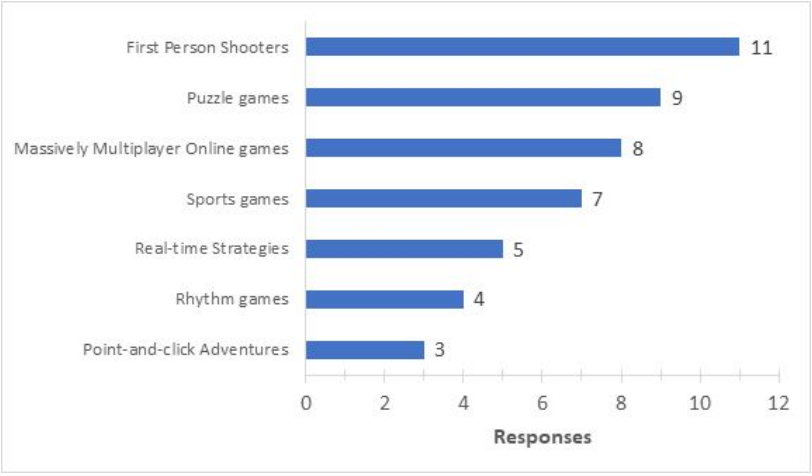

Figure 13. Player's familiarity with different game genres

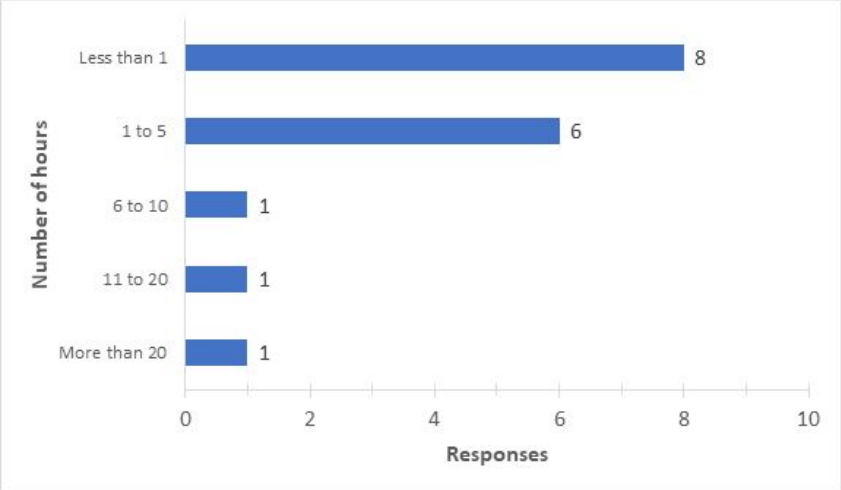

Figure 14. Player's time spent per week playing video games

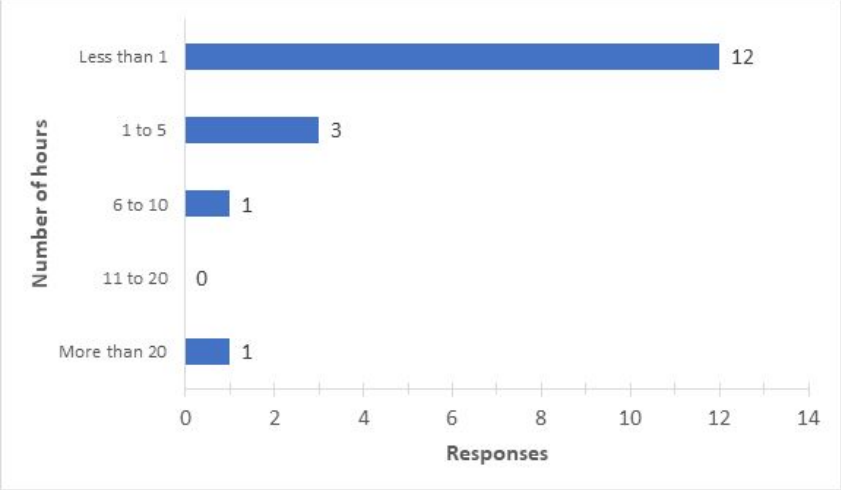

Figure 15. Player's time spent per week playing computer games with mouse and keyboard

Among the 17 players, 10 are males and 7 are females. Most of them are 18 to 22 years old with a few exceptions. The average age of all participants is 20.9 with a standard deviation of 2.9. Only 1 player is left handed while the rest are either right handed or ambidextrous. On a scale from 1 to 5 of computer gaming ability, players' self rating are fairly evenly distributed with a mean of 2.9 and a standard deviation of 1.4. Among all 17 players, 10 players rated themselves as 3 (average) or above. Most players are familiar with popular online game genres such as FPS or Real-time Strategies. Over half of the players spend above an hour per week playing games. However, over 70% of the players spend less than 1 hour a week playing computer games with a mouse and a keyboard. This suggests that players are more familiar with console or mobile games than computer games and the need for our initial practice time. The overall gaming experience of the players suggests that they should be able to get comfortable with the game's controls after some practice time but still be challenged by the gameplay.

4.2 Subjective results

There are 6 questions in the short gameplay survey that players take after each playthrough that ask them to rate different aspects of the gameplay experience on a Likert scale from 1 (low) to 5 (high). These graphs show the means of player answers with standard error bars versus the amounts of latency added. The orange trendlines with round markers show the players' ratings for playthroughs where player prediction is on, and the blue trendlines with squared markers show the players' ratings for playthroughs where player prediction is off.

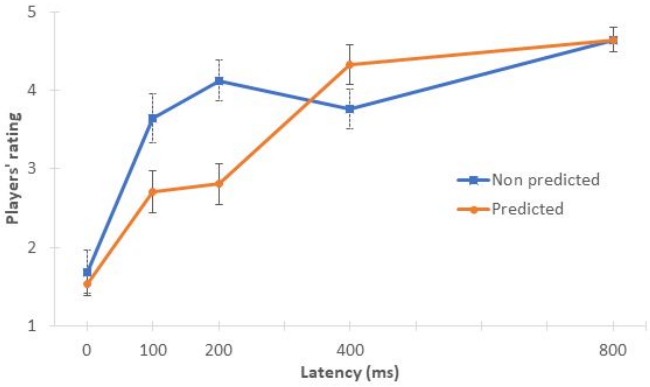

Figure 16. How much lag did you experience?

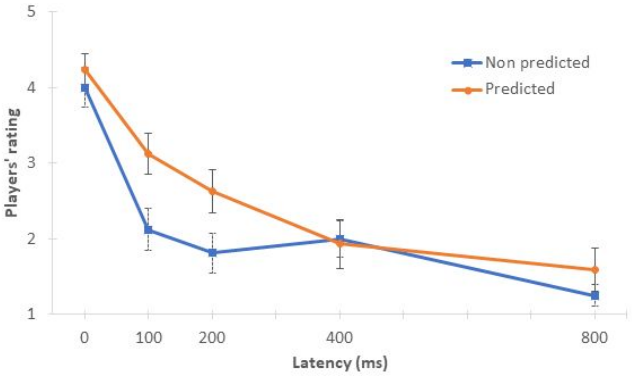

Figure 17. How responsive was the game?

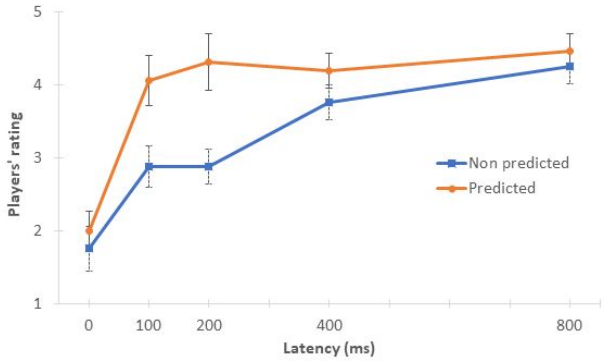

Figure 18. How many visual glitches did you see?

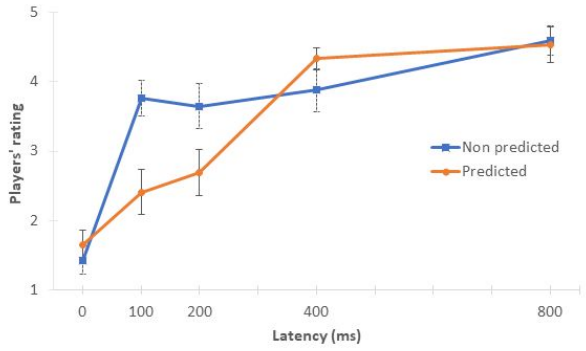

Figure 19. How much did lag affect your gameplay?

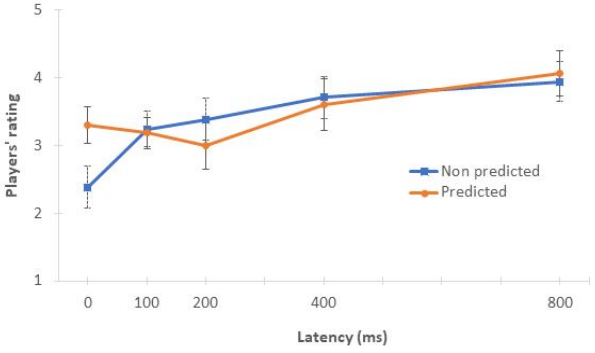

Figure 20. How challenging was the level?

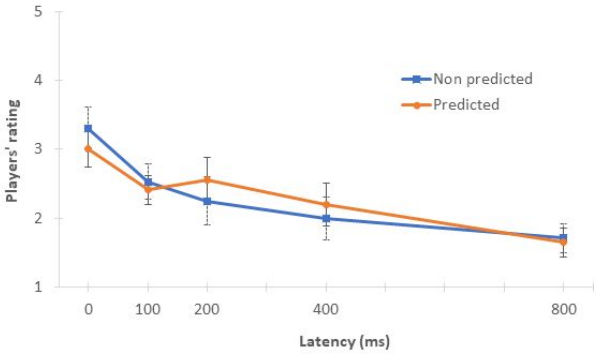

Figure 21. How fun was the level?

From the data shown in these graphs, 400ms seem to be the latency threshold where the benefits of player prediction diminishes. Below 400ms of network latency, when player prediction is on, players notice less lag, feel that the game is more responsive and less affected by the latency. On the other hand, from 400ms of network latency, the game starts to become very difficult to play with the delay, and the number of visual glitches becomes very high compared to the benefits of player prediction. The trade-off between responsiveness and visual consistency can be better seen in Figure 22. In almost all cases, the number of visual inconsistencies are higher when player prediction is on, but the game controls' responsiveness is improved, which is as expected. From Figure 20 and 21, although there does not seem to be a clear separation of the players' perception of how challenging and fun the level was between when player prediction is on and when it is off, the higher network latency is shown to have made the game harder to play and less fun, both of which translate to a degradation in gameplay experience.

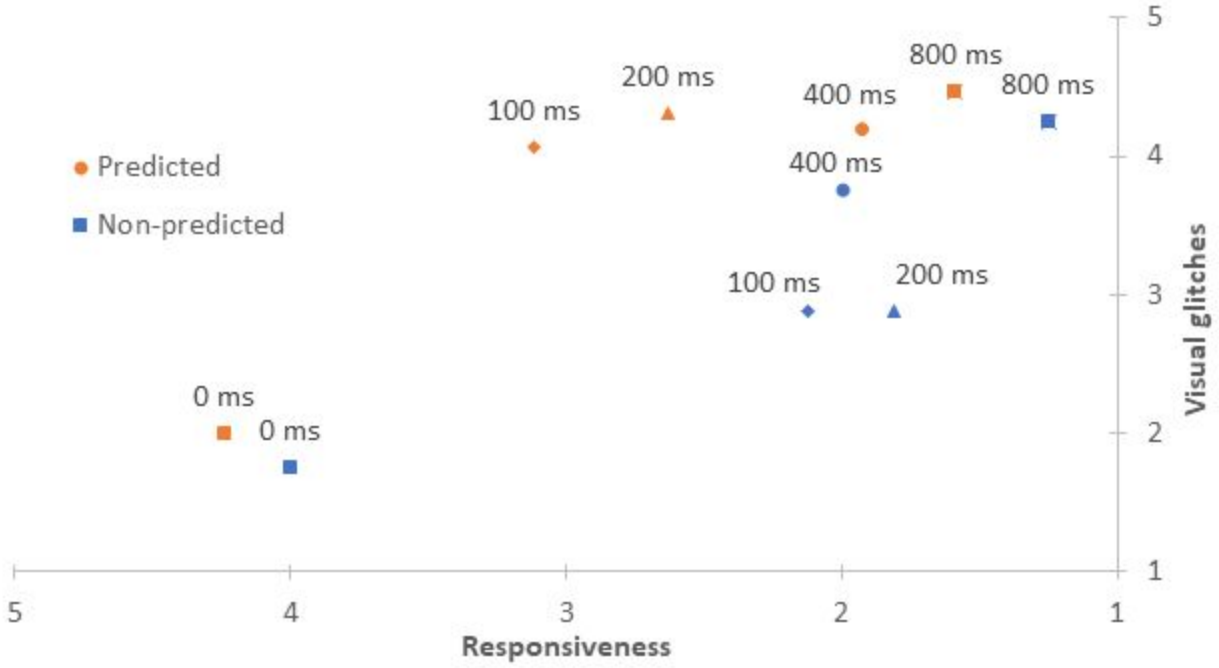

Figure 22. Trade-off between responsiveness and visual consistency

The subjective data collected from the study helps us answer the first study question about how player prediction for movement changes the players' gameplay experience under different network conditions. Generally, when player prediction is on, the network latency is less noticeable and the game controls feel more responsive. However, with player prediction, visual inconsistencies are more noticeable. This explains why player prediction in this case did not help players have more fun with their gaming experience. At Alpha testing, many players described the offline version of our game as appropriately frustrating, meaning it has a good balance between frustration and satisfaction which keeps players around. When the game is played online however, network latency and visual inconsistencies throw off the balance as the difficulty and frustration increase but the satisfaction does not. The challenge now is more due

to the latency rather than the game design and is not viewed as enjoyable. Therefore, players do not seem to enjoy the gameplay experience as much. Additionally, 400ms appears to be the approximate threshold where player prediction's benefits are outweighed by its disadvantages.

4.3 Objective results

Objective data is retrieved from the log files that are used by the game clients to record players' objective measurements. The following figures show the average objective measurements for all 17 players with standard error bars. The orange trendlines with round markers show the means of the data for games where player prediction is on, and the blue trendlines with squared markers show the means of the data for games where player prediction is off.

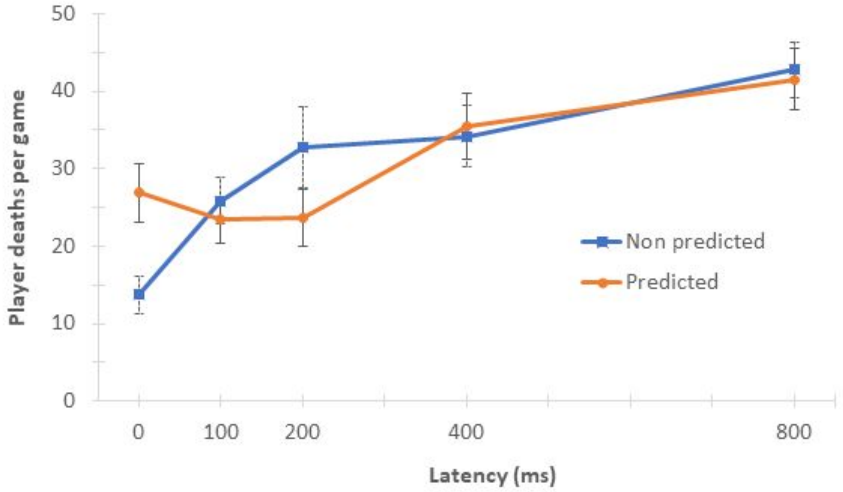

Figure 23. Player deaths per game

Figure 23 depicts the average number of deaths per game that players have. Below 400ms of latency, players generally die fewer times when player prediction is on. At 0ms is a special case where there are more deaths when player prediction is on. 0ms of latency with player prediction on was the first game that participants played, so some players died more times than they would have with more practice. 0ms lag with player prediction off, on the other hand, was the last game played, where the players have had the most practice after having played the same level 9 times. Furthermore, 100ms and 200ms seem to be the amounts of latencies where player prediction has a small positive effect in aiding players in the gameplay, especially at 200ms. Player deaths in 360 Gunner are caused by not being able to avoid enemies and enemy bullets or falling off from platforms. The immediate responsiveness from player prediction might have been the reason where players are able to position their heroes better, thus having fewer deaths. At 400ms and above however, player prediction provides no more benefits regarding player deaths.

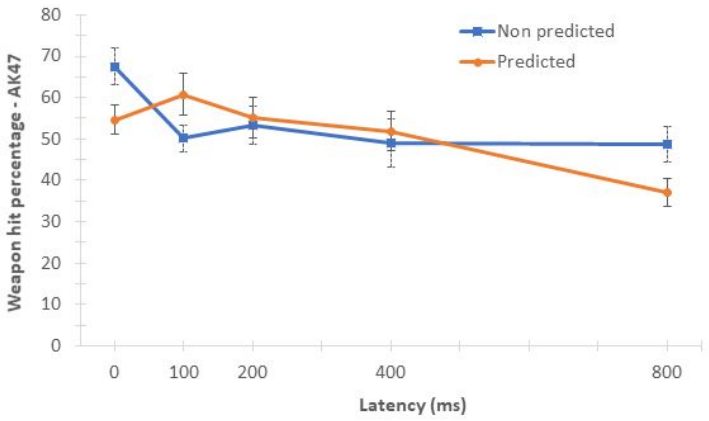

Figure 24. Weapon Hit percentage versus Latency - AK47

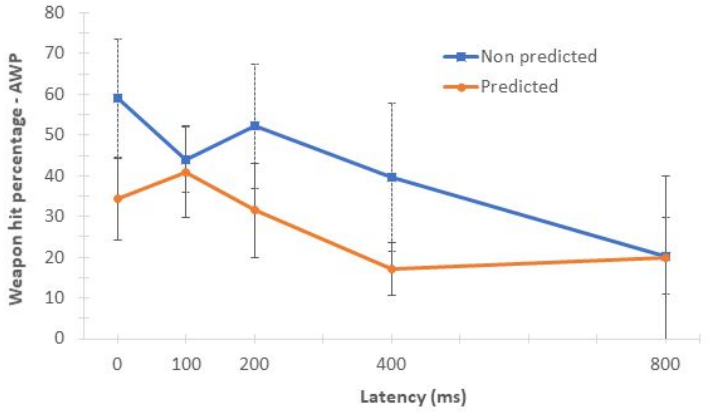

Figure 25. Weapon Hit percentage versus Latency - AWP

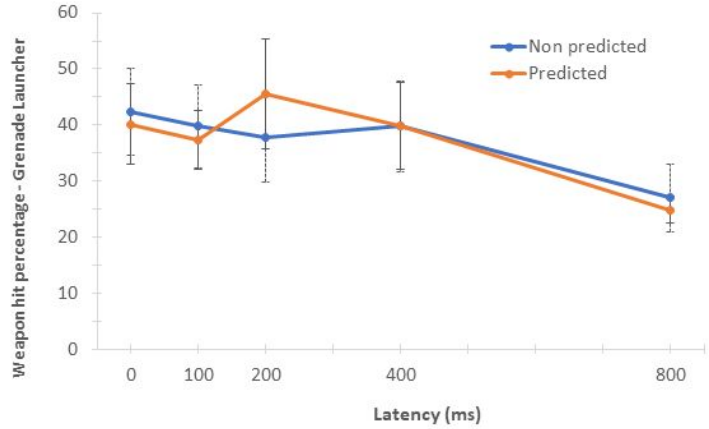

Figure 26. Weapon Hit percentage versus Latency - Grenade Launcher

To understand how the implemented types of weapons differ in terms of deadline and precision, we assume that the AK47, the most basic weapon in the game, is a game action with a medium level of deadline and precision. Compared to the AK47, the AWP have a much higher precision, because if a shot is fired at the same point on the screen, the AK47 will hit any enemy anywhere on the line of the bullet until the bullet leaves the screen, but the AWP will only hit an

enemy that's overlapping the raycast at the center of the Reticle. We consider the scenario where an enemy bot is moving straight towards the hero, and the player is aiming the Reticle at a point on the enemy's line of movement. With the AK47, the player can fire a shot at any point from when the enemy appears to when it passes the hero and the shot will hit the enemy. With the AWP, the player only has a brief window of time where the enemy fly passes the center of the Reticle to attack. Therefore, the AWP also have a much tighter deadline compared to the AK47. On the other hand, while the AK47 bullet is only a single ASCII character, the Grenade Launcher projectile is 3 characters tall and 5 characters wide. The Grenade Launcher bullet can cover a larger area around its path than the AK47 bullet, but flies slower. Therefore, the Grenade Launcher has a relatively similar if not slightly lower precision than the AK47. Additionally, since it has a lower projectile speed and lower fire rate compared to the AK47, the Grenade Launcher has a fairly looser deadline. Generally, if the AK47 represents a game action with a medium level of deadline and precision, the Grenade Launcher represents a game action with a slightly looser deadline and marginally lower precision, and the AWP represents a game action with the tightest deadline and the highest precision.

Figure 24, 25 and 26 show the players' performance with different types of weapons in the game. As expected, there is not a clear difference between the Predicted series and the Non-predicted series. It's also noticeable that the standard error for the AWP weapon is quite high. This is caused by the special controls of the weapon. Although every player has learned about the scope mode of the weapon while warming up with the tutorial level, it is not natural for a portion of the players to use it in a stressful situation such as the real level, especially when there is network latency involved. Therefore, some players achieve much higher accuracy with the weapon than others because they are more familiar with scoping in shooting games. This is an oversight that should have been taken into account when weapon statistics are recorded so that AWP statistics are separated between scoped hits and un-scoped hits.

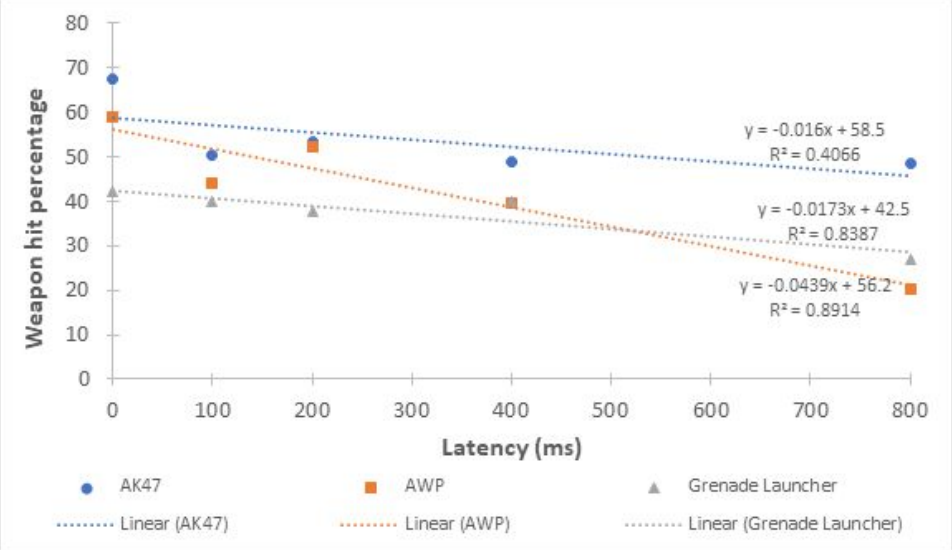

Figure 20. Weapons hit percentages when player prediction is off versus Latency added

Figure 20 shows the comparison of all weapons' hit percentages (without player prediction) as latency increases with linear regression lines. The performance degradation is the most severe with the AWP, with a linear regression slope of -0.439, while the AK47 and the Grenade Launcher have relatively similar slopes of -0.016 and -0.0173, respectively. Compared to the other 2 weapons, the AWP sniper has a much tighter deadline and higher precision. This supports our hypothesis that the tighter the deadline and the higher the precision required for a game action, the more sensitive the action is to network latency. It's also notable that although the Grenade Launcher has slightly looser deadline and lower precision than the AK47, it's still a more difficult weapon to use because of the projectile's low speed and curved path. This could be the reason why player performance for the Grenade Launcher degraded slightly more than for the AK47.

The objective data collected from all players' playthroughs help us answer the last two study questions: How are different types of game actions affected by different amounts of network latencies, and whether player prediction for movement helps improve player performance. The data supports the hypothesis that a weapon with tighter deadline and higher precision, such as the AWP, is more severely affected by network latency than weapons with looser deadlines and lower precision, such as the AK47 or the Grenade Launcher. An interesting finding about player performance suggests that player prediction does have a positive impact on the number of deaths that the players have per game, because player movement is closely related to the causes of player deaths in our game. On the other hand, attacking actions do not benefit clearly from the network latency compensation.

5. Conclusions

Online gaming is a popular entertainment choice and fueled by the advancements in Internet technology and mobile devices. Online games provide players with a high variety in genres, affordability, ubiquity and also real-time virtual interactions with other players despite geographical distance. Unfortunately, slow Internet connections, network congestion or a large geographical distance from servers to clients sometimes causes players to suffer from a degraded gameplay experience due to network latency. Latency compensation techniques can combat the negative effects of latency. However, more understanding of how latency affects different game actions and game genres is needed as well as the effectiveness of latency compensation.

Our work provides knowledge on the effects of network latency in the 2D platformer genre. We looked at how different types of game actions are affected by different amounts of network latencies. We also studied how the player prediction latency compensation technique helps ameliorate the negative effects of latency. We first implemented an original 2D platformer called 360 Gunner in order to control all game actions and latency compensation. We then invited players to play our game under different network conditions simulated with Clumsy. We collected players subjective and objective data in all cases, and provided our analysis of these data.

In our original fast-paced platformer game, 100ms of latency is noticeable by players. This suggests that depending on the game pace and the complication of game actions, 2D platformers can be as sensitive towards network latency as First Person Shooters, the most latency-sensitive game genre [10]. Furthermore, previous findings that game actions with tighter deadlines and higher precision are affected more severely by network latency [10] are supported by our study. Players' accuracy with the AWP - the weapon with the tightest deadline and highest precision among all 3 weapons in our game - degraded much more as latency increased than with the other 2 weapons. Additionally, latency compensation with player prediction of hero movement helps the game feel more responsive, but is a trade-off for visual inconsistencies. At 400ms of network latency, the disadvantages of the latency compensation technique outweighs its benefits. Future work could look at smaller ranges of latency values to find out a more precise point between 200ms and 400ms of network latency where player prediction's benefits are outweighed by its disadvantages to visual inconsistency. However, despite having little impact on attacking actions, by making the player movement more responsive, the latency compensation reduces the number of player deaths in our game, which is closely related to being able to move correctly through platforms to avoid taking damage or falling off. This suggests that player prediction, by making player actions more responsive on the client side, can help players perform better in certain cases. However, to make player prediction fully effective in making the player's gameplay experience better, further steps need to be taken to minimize the visual inconsistencies.

6. Future work

The immediate future work following our project includes fine-tuning the balance between difficulty/frustration and satisfaction of the game to better suit an online gaming scenario. Doing so will allow players to enjoy the gameplay better and researchers to have a better understanding of how player experience is improved with player prediction. This can be done by tuning bot parameters, finding ways to reduce visual inconsistencies, implementing smoother transitions when visual fix-ups are needed. The satisfaction in gameplay can also be increased with improved visual aspects, such as better sprite designs and animations, or even moving the project to an industry-proven game engine. Furthermore, adapting the prediction to specific game conditions, such as turning it on or off based on the position of heroes and enemies, can provide benefits such as reducing the number of visual glitches.

Our codebase provides a framework that is easily extendable for new types of weapons, enemies and level design. Future researchers can extend the game with new weapons, bots and levels and evaluate these new aspects similarly. Having an extended set of data will allow for clearer comparisons between different types of game actions and how these game actions are affected by latency. Other latency compensation techniques such as time-warp can also be implemented and evaluated through our game to figure out more effective ways of combating network latency in 2D platformers. Furthermore, we suggest more investigation into different types of mechanics from our realistic weapon scheme with other 2D platformer games. Doing so will allow researchers to determine the latency susceptibility of these games, which may differ from what our work has shown.

7. References

1. Lofgren, Krista. “2017 Video Game Trends and Statistics – Who’s Playing What and Why?” *Big Fish Games*, 5 Apr. 2017, www.bigfishgames.com/blog/2017-video-game-trends-and-statistics-whos-playing-what-and-why/.
2. Gough, Christina. *U.S. Online Gaming Industry - Statistics & Facts*. 7 Mar. 2019, https://www.statista.com/topics/1551/online-gaming/.
3. “Top 5 Reasons Why Online Gaming Has Become Popular.” *Connection Cafe*, 31 May 2018, https://www.connectioncafe.com/top-5-reasons-why-online-gaming-has-become-popular/ .
4. Bevilacqua, Fernando. “Building a Peer-to-Peer Multiplayer Networked Game.” *Envato Tuts Plus* , 12 Aug. 2013, https://gamedevelopment.tutsplus.com/tutorials/building-a-peer-to-peer-multiplayer-networked-game--gamedev-10074.
5. Bernier, Yahn W. “Latency Compensating Methods in Client/Server In-Game Protocol Design and Optimization.” *Valve Developer Community*, 2001, developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization.
6. Claypool, Mark. *Dragonfly - Program a Game Engine from Scratch*. Interactive Media and Game Development, Worcester Polytechnic Institute, 2014, https://dragonfly.wpi.edu/book/.
7. Raaen, Kjetil, and Andreas Petlund. “How Much Delay Is There Really in Current Games?” *Proceedings of the 6th ACM Multimedia Systems Conference on - MMSys 15*, 2015, doi:10.1145/2713168.2713188.
8. Yanqi, Shi. “How Do We Determine the Bottlenecks on Hardware According to Lagging in Game.” *Zhihu Focus*, Zhihu, 6 June 2018, zhuanlan.zhihu.com/p/35701078.
9. Armitage, Grenville, et al. *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*. John Wiley & Sons, 2006.
10. Claypool, Mark, and Kajal Claypool. “Latency and Player Actions in Online Games.” *Communications of the ACM*, vol. 49, no. 11, Jan. 2006, p. 40., doi:10.1145/1167838.1167860.

11. Brun, Jeremy, et al. “Managing Latency and Fairness in Networked Games.” *Communications of the ACM*, vol. 49, no. 11, Jan. 2006, p. 46., doi:10.1145/1167838.1167861.
12. “The state of online gaming - 2019.” *Limelight Networks*, 2019, https://www.limelight.com/resources/white-paper/state-of-online-gaming-2019/.
13. Li, Zhi, et al. “Lag Compensation for First-Person Shooter Games in Cloud Gaming.” *Lecture Notes in Computer Science Autonomous Control for a Reliable Internet of Services*, 2018, pp. 104–127., doi:10.1007/978-3-319-90415-3_5.
14. Nichols, James, and Mark Claypool. “The Effects of Latency on Online Madden NFL Football.” *Proceedings of the 14th International Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV 04*, 2004, doi:10.1145/1005847.1005879.
15. Claypool, Mark. “The Effect of Latency on User Performance in Real-Time Strategy Games.” *Computer Networks*, vol. 49, no. 1, 2005, pp. 52–70., doi:10.1016/j.comnet.2005.04.008.
16. Sabet, Saeed Shafiee, et al. “Towards the Impact of Gamers Adaptation to Delay Variation on Gaming Quality of Experience.” *2019 Eleventh International Conference on Quality of Multimedia Experience (QoMEX)*, 2019, doi:10.1109/qomex.2019.8743239.
17. Howard, Eben, et al. “Cascading Impact of Lag on Quality of Experience in Cooperative Multiplayer Games.” *2014 13th Annual Workshop on Network and Systems Support for Games*, 2014, doi:10.1109/netgames.2014.7008965.
18. Lee, Injung, et al. “Geometrically Compensating Effect of End-to-End Latency in Moving-Target Selection Games.” *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI 19*, 2019, doi:10.1145/3290605.3300790.
19. Sabet, Saeed Shafiee, et al. “Towards Applying Game Adaptation to Decrease the Impact of Delay on Quality of Experience.” *2018 IEEE International Symposium on Multimedia (ISM)*, 2018, doi:10.1109/ism.2018.00028.
20. Lee, Steven W. K., and Rocky K. C. Chang. “Enhancing the Experience of Multiplayer Shooter Games via Advanced Lag Compensation.” *Proceedings of the 9th ACM Multimedia Systems Conference on - MMSys 18*, 2018, doi:10.1145/3204949.3204971
21. Savery, Cheryl, and T. C. Nicholas Graham. “Timelines: Simplifying the Programming of Lag Compensation for the next Generation of Networked Games.” *Multimedia Systems*, vol. 19, no. 3, 13 June 2012, pp. 271–287., doi:10.1007/s00530-012-0271-3.

22. Lee, Yeng-Ting, et al. “Are All Games Equally Cloud-Gaming-Friendly? An Electromyographic Approach.” *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*, 2012, doi:10.1109/netgames.2012.6404025.
23. Claypool, Mark, and David Finkel. “The Effects of Latency on Player Performance in Cloud-Based Games.” *2014 13th Annual Workshop on Network and Systems Support for Games*, 2014, doi:10.1109/netgames.2014.7008964.
24. Li, Zhi, et al. “Lag Compensation for First-Person Shooter Games in Cloud Gaming.” *Lecture Notes in Computer Science Autonomous Control for a Reliable Internet of Services*, 2018, pp. 104–127., doi:10.1007/978-3-319-90415-3_5.
25. Lee, Kyungmin, et al. “Outatime: Using Speculation to Enable Low-Latency Continuous Interaction for Mobile Cloud Gaming.” May 2015, doi:10.1145/2742647.2742656.