

Interactive Media and Game Development Master's Project

Drizzle: Design and Implementation of a Lightweight Cloud Game Engine with Latency Compensation

Jiawei Sun

December 2017

Thesis Advisor: Professor Mark Claypool

Committee members: Professor Lane Harrison

Professor Jeffrey P Kesselman

Abstract:

With the rapid development of the Internet, cloud gaming has increasingly gained attention. Cloud gaming is a new type of cloud service that allows a game to run on the cloud servers, and players interact with the game remotely on their own light-weight clients. There are many potential benefits for both players and game developers to deploy a game on a cloud server, such as reducing the need for clients to update the game, easing development of cross-device games and helping prevent software piracy. In this work, I developed a cloud game engine, Drizzle, with a time warp algorithm for latency compensation, and implemented a new transmission method that reduces the network bandwidth. Using Drizzle, I also developed a simple cloud game to evaluate the functionality and performance. Experiments with this game in a controlled laboratory environment provide objective measurements of game performance and subjective measurements of user performance. Analysis of the results shows Drizzle with time warp did not reduce noticeable latency but helped players get higher game scores compared to Drizzle without time warp. Moreover, Drizzle reduced network bitrates compared to some conventional cloud transmission methods.

Keywords: Drizzle; cloud; game engine; time warp; drawing commands

Table of Contents

1. Introduction.....	1
2. Background & Related work	3
3. Methodology.....	8
3.1 Dragonfly Implementation.....	8
3.2 Network Implementation	9
3.2.1 Network Architecture.....	9
3.2.2 Network Packet Structure	10
3.2.3 Server Implementation.....	12
3.2.4 Client Implementation	13
3.3 Time Warp	13
3.3.1 Time Warp Algorithm.....	13
3.3.2 Time Warp Implementation	14
3.4 Cloud Saucer Shoot Game Implementation.....	17
3.4.1 Game Object Design & Implementation.....	18
3.4.2 Score System.....	19
4. Evaluation	22
4.1 Subjective.....	22
4.1.1 Survey design.....	22
4.1.2 Set up environment	24
4.1.3 Hardware environment.....	26
4.1.4 Survey procedure	26
4.1.5 Survey result	27
4.2 Objective	31
5. Conclusion	36
6. References.....	38

1. Introduction

With the rapid development of the Internet, online games have become increasingly popular. Graphics-intensive online games often have onerous hardware requirements for players, such as high-end CPUs and GPUs and considerable memory capacity. To minimize equipment requirements for players, cloud gaming has been proposed. Cloud gaming is a new type of cloud service that allows a game to run on the cloud servers, and players play the game remotely on their own light-weight clients [1]. The basic idea is to distribute the heaviest work to the server while the client only needs relatively less powerful hardware.

Cloud gaming can bring additional benefits to the players [2]. First, players do not need to download the potentially large game to their personal computers. Some game clients are several gigabytes. For example, Blizzard's World of Warcraft client needs a 40 GB download. If World of Warcraft were a cloud game, the player could play the game as soon as the computer was connected since the game is on the server. Second, players can play games from any platform without having game specific hardware. Third, since the game images are rendered on the server, players experience high quality graphics even with a low power computer. For game developers, cloud games also have advantages. When game developers update a cloud game, they only need to update the server. Game developers only need to develop one version of a game for the cloud server, which allows them to focus on the gameplay, content and reduce the development costs. Plus, cloud games can help protect developers' copyrights since game does not run on the players local machine.

While cloud gaming is promising, it is still in its infancy with several challenges to address [1].

For example, conventional cloud games need high network bandwidth because they need to download game images as video from the game server in real-time. If there are interruptions in the network, the game video quality may decrease and thus degrade the player experience. Another problem is network latency. Due to the physical distance between game server and client, latency cannot be avoided, and for some fast-paced games, e.g. First Person Shooter (FPS) games, latencies over 200ms may be unbearable for the player [3].

Time warp is a technique which is designed to mitigate the effects of latency in network games [8]. The idea with time warp is to roll back and then revise the game state according to the network latency when the player provided input.

In this project, I extended the Dragonfly game engine to a cloud game engine (Drizzle) with two main differences [4]. I added time warp latency compensation that saves checkpoints of each game state on the game server. After receiving the player's command, the server rolls the game world back to the time when the player sent the command and adjusts the game world. To reduce the bandwidth requirement of traditional cloud gaming, the server sends basic graphics commands (the characters to draw) to the thin client.

For evaluation, I induced network latency and evaluated the user experience, memory usage and CPU load comparing Drizzle with time warp and Drizzle without time warp. I also analyzed the network bandwidth, comparing sending graphics commands, sending JPEG images and sending compressed video.

The rest of this thesis is organized as follows: Chapter 2 provides background and related work; Chapter 3 describes our methodology to design and implement Drizzle; Chapter 4 evaluates Drizzle; and Chapter 5 summarizes our conclusions and present possible future work.

2. Background & Related work

Mark Claypool designed and developed the text-based Dragonfly game engine [4]. Dragonfly is mainly designed to teach students how to make a game engine. It uses Simple and Fast Multimedia Library (SFML) as the graphics output library to draw game objects and user interfaces on the screen. Although Dragonfly is a text-based game engine and does not support full graphics, it does have core functions for supporting a whole game. I implemented the Drizzle cloud game engine by extending the Dragonfly engine.

There are already some cloud game engines available for use, such as GamingAnywhere (GA) [5], and GeForce Now [6]. GA is an open source cloud game engine platform. GA currently supports Windows, Linux, OS X and Android. My project implements latency compensation (time warp) technology in a cloud game engine while there is no latency compensation technology in GA. NVIDIA released GeForce Now, which is a cloud-based streaming service exclusively for owners of NVIDIA's Shield family of Android-based gaming devices, which includes the Shield Portable, Shield Tablet, and Shield Android TV. After becoming a member of GeForce Now, players can play streaming games on their own TVs without any installation or configuration. GeForce Now can run games at 1080p game and 60 fps. While promising, GeForce Now is not open source so we cannot extend its functions to evaluate latency compensation technologies.

To improve the quality of the user experience, research papers and technologies have attempt to address current problems with cloud games.

There are primary three ways to do latency compensation [1]: movement prediction, time warp and action priorities.

There are two types of movement prediction: player prediction and opponent prediction [7]. Player prediction is a way that the client can predict the server response, allowing the game client to respond to user input and render actions before getting authorized responses from the server. For example, if a player is flying a plane at a certain velocity and orientation, the client can calculate its path without information from the server. Once the authorized responses from the server are received, the client can re-compute any needed changes. Another kind of prediction is opponent prediction. The client predicts and renders the opponents' movements and every client calculates the difference between the opponents' actual movements and the predicted movements. If the difference is greater than a certain threshold, then the client receives the opponents' new movement from the server. That also reduces the network load but requires clients to do more calculations. Prediction is not suitable for cloud games since the client does not have game state.

Another way to reduce the effect of latency is to provide priority transmissions for the player's actions [8]. The basic idea is to pre-determine the action priority and send the action to the server according to a pre-set priority. In a game, some actions do not need to be sent as soon as they are created, while others need to be sent immediately. For example, if a player is shooting an enemy, the shoot command could be sent first to mitigate the effect of latency while the movement changes may not be as important. Actions are taxonomized along three axes: deadline, precision and impact. For example, if the action's transmission's deadline is soon plus needs a high precision and has high impact on the game, then this transmission would be sent to the server with a high priority. This technique requires mediation from the game. Game developers need to consider the effects of all actions and then set up the priority. Thus, this solution does not generalize to an engine alone, requiring client support, making it difficult in a cloud system.

Another technique to reduce the effect of latency is time warp [7]. Basically, the idea is to “roll back” game time on the server and make the corresponding updates to the game world according to when the actions occurred on the client. As an example, assume a client makes an action at t_1 and the network latency is t_2 . When this action has been transmitted to the server, the current time on the server will be t_3 (t_1+t_2). Then, the server rolls back time by t_2 to t_1 (which is exactly the time that player made the action), executes the command at t_1 , then updates game world to t_3 , sending the latest game state to the client.

Figure 1 shows a time warp algorithm flow chart. After the server receives packets from the client, it extracts the user input information and then calculates the time difference. Then the server rolls back all events to the elapsed time and executes the player action. After this, the server updates the game world and sends the latest game state back to the client.

Time warp can improve the accuracy of players’ commands with latency. For example, in a First-Person Shooter (FPS) game, if the player shoots right at the target, the server can resolve the shot no matter how much latency there is between the server and client. This method has the potential to make players feel as if they are playing a local game, but it may lead to some inconsistencies. Suppose a player places the crosshairs of a gun on an opponent and fires. The server, using time warp, will ultimately determine this is a “hit”. However, in the meantime, because of client-server latency, the opponent may have moved, perhaps even around a corner and out of sight. When the server warps time back to when the shot was fired and determines the opponent was shot, it will seem to the opponent that the bullets “bent” around the corner. In addition, time warp also requires the server to store the previous game states and actions which have happened before.

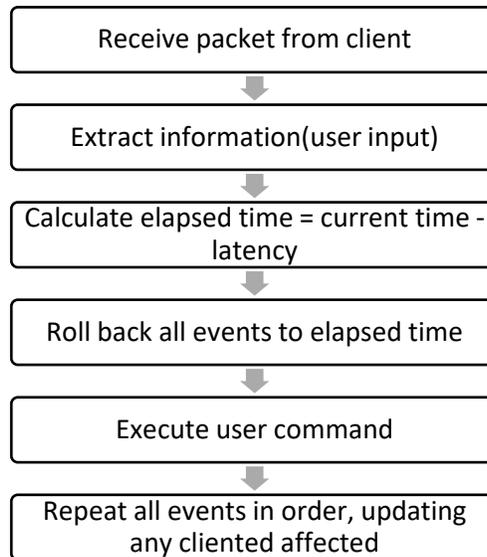


Figure 1. The time warp algorithm flow chart (server side)

To reduce bandwidth in a cloud game, de Winter put forward a new hybrid architecture model for a cloud game engine [9]. Their architecture sends graphics commands instead of full video frames. When the client is not able to deal with the graphics commands in real time, then the server sends the video frames directly to the client, and in this situation, the client decodes the video frames and shows them on the screen. When the client has the ability to process with the graphics, the server sends the graphics commands to the client, and the client uses these commands to draw and render locally. This hybrid architecture can save network bandwidth and utilize the client's ability of graphic processing. The classic X-Window system adopts a similar architecture. However, since X-Window was designed to run a whole desktop on the thin client, it emphasizes switching context between multiple tasks and windows, making game performance suffer [13].

Thus, Drizzle implements time warp in a single thread game engine. Drizzle also reduces

bandwidth by sending the text drawing commands to the client allowing the client to decode these commands to construct and then display the game frames.

3. Methodology

This section describes the methods used to implement Drizzle, a cloud game engine with time warp.

Section 3.1 describes the core game engine based on Dragonfly.

Section 3.2 demonstrates how I extended Dragonfly with network.

Section 3.3 describes how I designed and implemented time warp.

Section 3.4 introduces the design and implementation of a cloud saucer shoot game used for evaluation.

3.1 Dragonfly Implementation

The Dragonfly game engine is mainly designed to teach students how to make a game engine [4]. It uses the Simple and Fast Multimedia Library (SFML) as the graphics output library to draw game objects and the user interfaces on the screen. I followed the design by professor Claypool and implemented the base version of the Dragonfly game engine. In this base version, I implemented the Log Manager, Resource Manager, Game Manager, Graphics Manager, Input Manager and World Manager using C++ and SFML.

- The Game Manager is responsible for running the whole game loop. I set up the frame rate (frames per second) to 30, so the game loop will run every 33ms.
- The World Manager is responsible for storing and managing all game objects and events.
- The Log Manager is responsible for recording the game status and outputting the debug information into a log file.
- The Resource Manager is responsible for loading the sprites into the game.

- The Graphics Manager is responsible for drawing characters, drawing strings and sprite frames.
- The Input Manager is responsible for taking player keyboard and mouse input and generating an event to the World Manager for player input.

3.2 Network Implementation

To extend the base version of Dragonfly to the Drizzle, network functions are needed to communicate between client and server. This section describes the design and implementation of the network architecture for Dragonfly.

3.2.1 Network Architecture

The network architecture follows a traditional Client-Server model for cloud games. All computations are in the server and the client is only responsible for receiving and showing the frames to the player. Thus, it requires a reliable network connection which can guarantee that the client will correctively receive all the packets in order from the server. I used TCP which has a built-in algorithm to handle packet loss and out of order delivery. Figure 3.1 shows a basic TCP/IP connection [12]:

- 1 The server creates the listener socket that is waiting for remote clients to connect.
- 2 The client issues the `connect()` socket function to start the TCP handshake (SYN, SYN/ACK, ACK). The server issues the `accept()` socket function to accept the connection request.
- 3 The client and server issue the `receive()` and the `send()` socket functions to exchange data over the socket.

- 4 Either the server or the client decides to close the socket. This causes the TCP closure sequence (FINs and ACKs) to occur.

When the network condition degrades and leads to some loss or disorder, TCP retransmits lost packets.

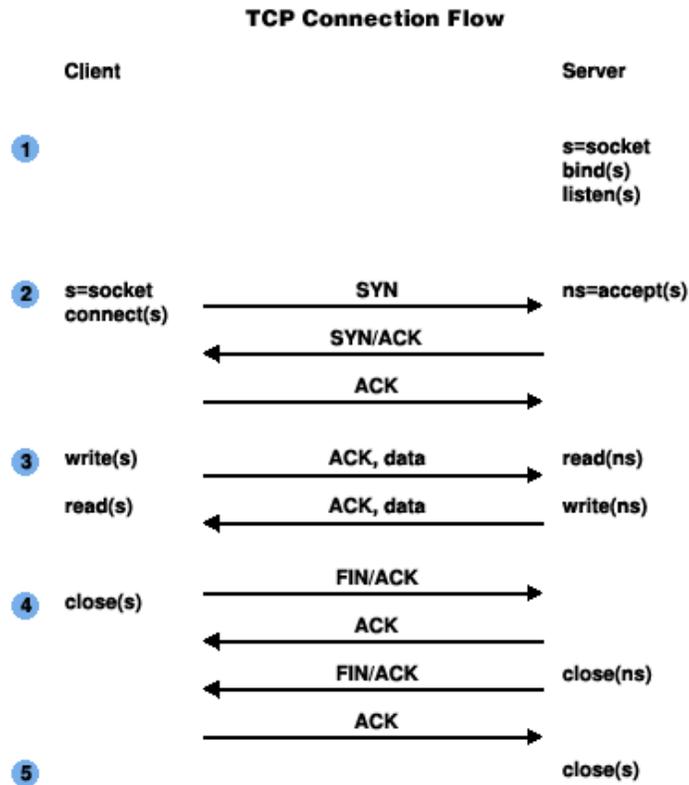


Figure 3.1. A Typical TCP Client-Server Connection

3.2.2 Network Packet Structure

In a cloud game engine, the server is required to send the frames to the client while the client is required to gather the player's input and the send them back to the server. To reduce the network bandwidth when transferring data, Drizzle sends the drawing commands instead of a whole image from the server to client. Because all drawing strings and drawing frame functions can be divided into different drawing characters, the client only needs to receive the frame characters. Thus, the drawing command packet structures consist of the parameters of the

drawCh(). The client is able to draw the corresponding character after receiving these packets.

Figure 3.2 shows the structure of the drawing packet. *Len* is for recording the length of the packet. *Type* decides whether this packet contains a character or a string. *X* and *y* are for locating the character. *Content* is the character to be drawn. *Color* is the character's color, *Time counter* is an unsigned integer which will be used in the time warp algorithm.

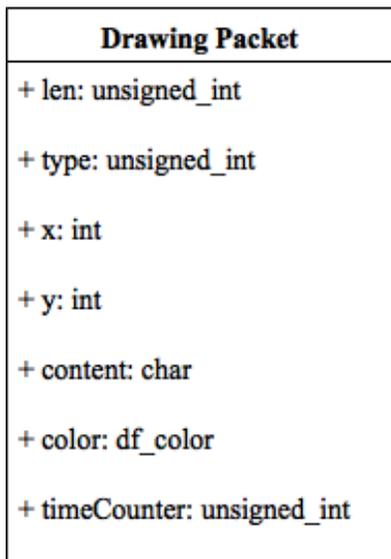


Figure 3.2. Drawing Packet Structure

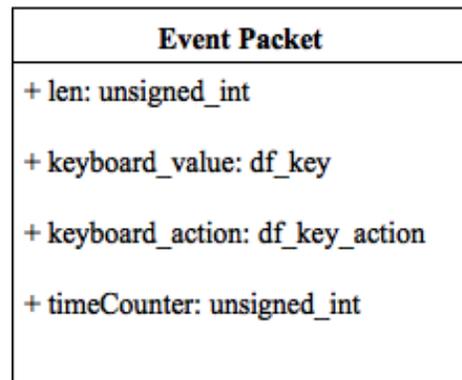


Figure 3.3. Event Packet

The client packs the player's inputs and sends them back to the server. The client event packet consists of what an input event needs: the keyboard actions (whether the player is clicking on a key, holding down a key or has just released the key) and the keyboard values. Figure 3.3 shows the structure of the event packet. *Len* is the packet length, *keyboard_value* is the key, *keyboard_action* is whether player presses holds or releases the keyboard, and *time counter* is used in the time warp algorithm.

3.2.3 Server Implementation

To support all network connection functions, I designed and implemented a network manager in Drizzle. The network manager on the server side is responsible for listening to all the connections, sending and receiving data.

The network manager was implemented with Winsock in C++. It has `startup()`, `accept()`, `send()`, and `receive()` functions. The `startup()` is responsible for starting the manager super class and initializing the member attributes. After calling `startup()`, the `accept()` must be called before all other network functions. In the `accept()`, the network manager binds the server IP address and a specific port number with the socket. Then, it sets the socket to non-blocking mode which means `receive()` returns a value immediately even if there is no data in the socket buffer. In the `send()`, the drawing commands structure is sent together with its size, so that when a client receives a packet, it can check its size to know whether the entire packet has arrived. In the `receive()`, the server gets the player's input packet and its size from the client.

In addition to implementing a network manager, I modified the draw function on the server. In the base version of Dragonfly, when the game manager calls `update()`, all objects in the game world call their `draw()` functions. In Drizzle, the drawing packets are instead sent to the client. I modified the `drawCh()` instead of calling SFML functions, I packed all the parameters that `drawCh()` needs into one struct, and sent it to the client by calling the network manager's `send()` function.

The final step of implementing the server is to add a function that supports receiving the player's input data from the client, pack it an event and then pass it to the game manager to handle it. I created a function called `checkCommand()` in the network manager. It first calls `ioctl()` to return how many bytes of data are in the socket buffer. If the return value is greater than the size

of the input packet, then the server tries to receive the packet and check whether packet size is as same as the input packet or not. According to the result, it decides whether to call receive() or not.

3.2.4 Client Implementation

The client implementation is quite similar to the server. However, the client network manager does not use the accept(), instead using a connect(). The connect() calls connect() of Winsock with the server IP address and the specific port number and then sets the socket into non-blocking mode. Send() and receive() are similar to the server side.

I modified the client input manager so that every time the player presses a key it produces an event packet and then calls the send() of the network manager on the server. I also added a new function named checkGraphics() to receive the graphics packet from the server. The work flow of this function is similar to checkCommand() in the server. It verifies that the socket buffer size is equal to the packet size and then calls receive() to extract the drawCh() parameters, finally calling drawCh() with the corresponding parameters to draw the frame for the player.

3.3 Time Warp

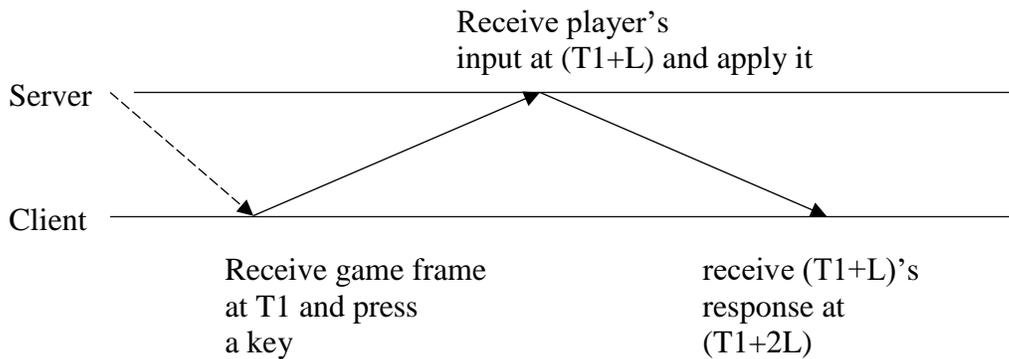
I designed and implemented a time warp algorithm into Drizzle. This section introduces, analyzes and demonstrates how.

3.3.1 Time Warp Algorithm

The Time warp algorithm tries to mitigate the effect of latency. Figure 3.4 shows a comparison of traditional client-server communication and one with time warp: Assume the player currently has latency L . When the player presses a key on the client at time $T1$, it takes L to send it to the server, so the server receives the message at time $(T1+L)$. Instead of just

applying the event at time $(T1+L)$, the time warp algorithm has the server roll the game world back to $T1$ then apply the event. The game world is then rolled forward to the current time. So, when the player at the client receives feedback after pressing a key, the response happens at the previous game time.

Without time warp



With Time Warp

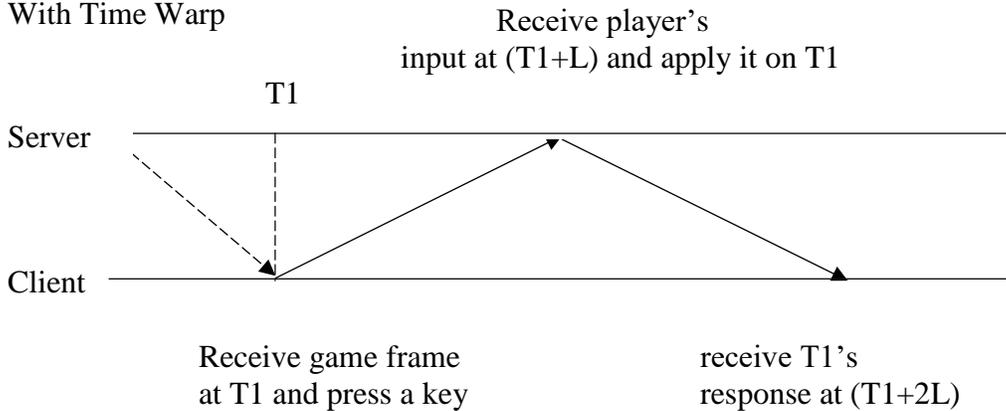


Figure 3.4. Comparison of responsiveness with and without time warp

3.3.2 Time Warp Implementation

According to the description about time warp above, it is important to measure the network latency is so that the server can roll back the game world time. So I added a time counter to record the time stamp of the game loop. When the server's game loop starts, the time

counter is set to 0. After that, every game loop, the time counter increases by 1. This time counter is sent as an attribute with the graphics packet, so the client can know the current time counter and associate it with the event packet sent back to server. In this way, the server can calculate the difference between the time counter in the event packet and the current time counter in the game manager to know how long the game needs to roll back.

Since the server needs to roll back the game world, saving previous game world states is essential on the server. In the base version of Dragonfly, all game objects are stored into an object list in the world manager. However, in Drizzle, simply copying the object list will not work. That is because the elements in the object list are only the pointers to the actual objects, so copying the list will only copy the pointer values. When we change objects by using these pointers, the change will be applied to all the objects in the previous game world. To solve this problem, Drizzle uses a deep copy which allocates new memory and copies every object attributes rather than just its pointer. Considering the object class as the base class of the game object, game programmers need to write their own object class inherited from it, using a virtual `copy()` added into the object class. Every object which is inherited from the base object is required to override this function. Within this new `copy()`, the game object needs to return its own type of constructor. When Drizzle iterates through the object list, it calls every element's `copy()` to make the deep copy.

After the above preparations, I implemented a time warp manager which is able to store up to 300 previous game worlds. This list was implemented by a loop array, so the server could automatically overwrite the game world without worrying about the memory leak. This list was responsible for fetching the previous game world according to the time counter. For example, when the server is required to fetch the previous game world with a time counter of 483, it will

mod this time counter by 300, so it will get $483\%300 = 183$. The server can simply fetch the previous game world by using index 183.

The Drizzle time warp manager also offers functions to deal with setting and getting the game world associated with the time counter. When the server detects the time difference between the client and itself, the time warp manager fetches the corresponding previous game world, re-applies the event into this previous game world, and then fast-forwards until the applied world has the same time counter as the current time counter in the game manager. Figure 3.5 shows how it works.

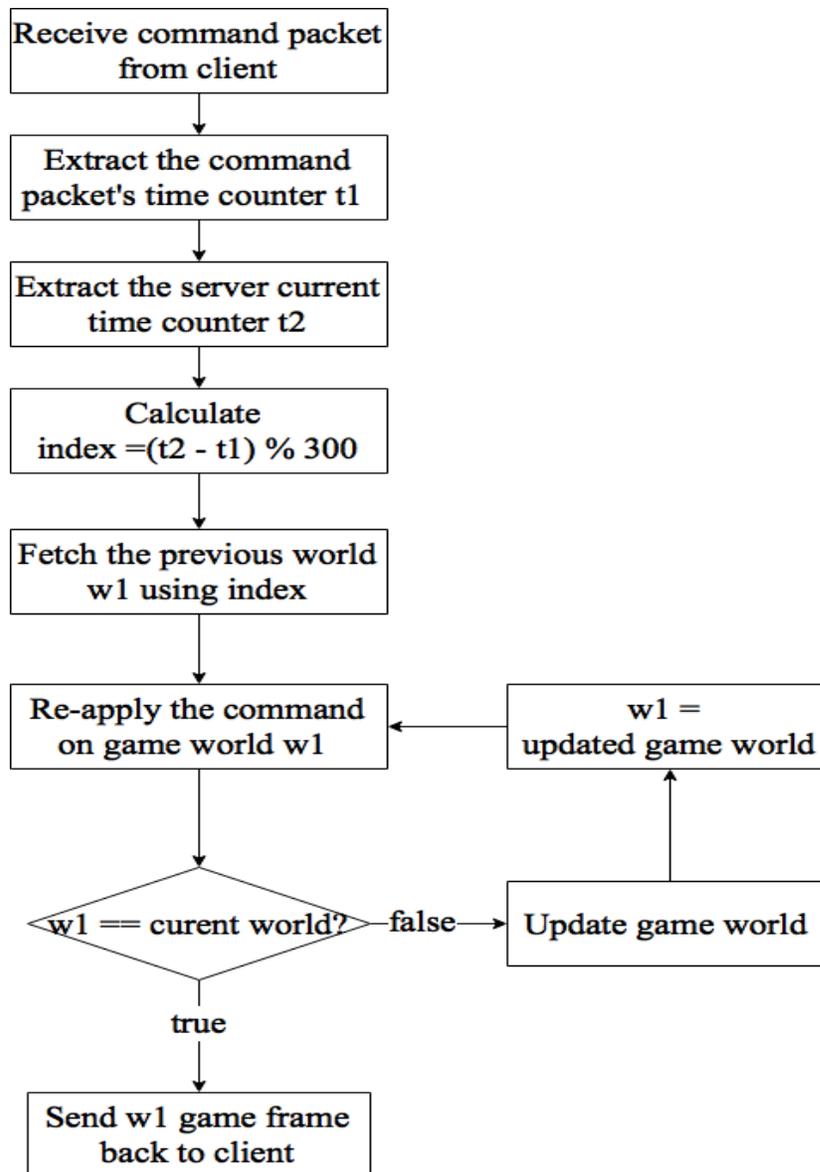


Figure 3.5. Time warp algorithm flow

3.4 Cloud Saucer Shoot Game Implementation

After implementing Drizzle, I also implemented a simple saucer shoot game using this game engine. The main goals for doing this: 1) Test whether this game engine is functional. I need to design a basic game using Drizzle to check all the managers and network functions

function properly. 2) Provide a game suitable for the user survey. I need a cloud game using Drizzle to evaluate whether time warp mitigates latency or not.

3.4.1 Game Object Design & Implementation

In this basic Saucer Shoot game, there are three different types of objects: The Ship, the Saucer and the Bullet. The Ship is the Hero in this game, and the player can use up, down, left and right arrows to control its movements. The Ship also has health which is reduced when hitting a Saucer. The Saucer is the enemy in this game. When the Saucer hits the Ship, it is destroyed. The Bullet is produced by the Ship when player press the space bar. When the Bullet hits the Saucer, both Saucer and Bullet are destroyed and the score for the player increases by 1.

The Ship class is inherited from the base Object class, and it needs to define two functions: `copy()` and `eventHandler()`. For `copy()`, as stated before, the Ship class needs to return a Ship type constructor so that we can use it for the deep copy. For the `eventHandler()`, it is important for the user to control the Ship. In this function, the parameter is an event pointer that is passed from the World Manager. The event pointer has a string attribute called `type` that I used to distinguish the type of the event and then according to the type to execute code to respond to keyboard input.

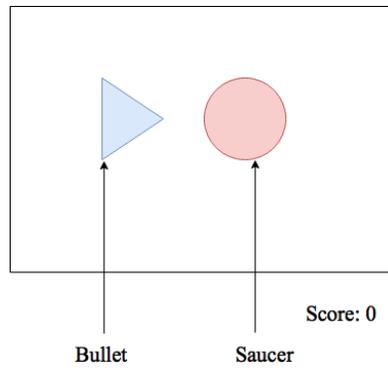
The Bullet class and Saucer class are similar to the Ship class. They all need to overwrite the `copy()` and `eventHandler()`. The main difference is in the `eventHandler()`. In the Bullet class, when there is a collision event, I iterate through the collision list to find if there any collisions related to the Bullet. If there are, it destroys the Bullet instance and increase the final score. In the Saucer's `eventHandler()`, besides the collision event, I implemented an enemy random generation algorithm. When the Saucer is hit or goes out of the screen, Saucer Shoot generates

1~3 enemies with random locations (locations are guaranteed to have no collision with other Saucers).

3.4.2 Score System

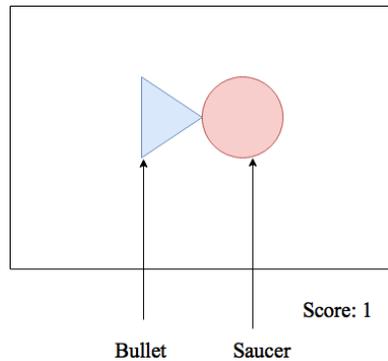
The conventional way of implementing the scoring system used to add a static variable in the Saucer class. When a Saucer instance is destroyed by a Bullet, the score is incremented. However, with the time warp algorithm, this method does not work so well. Because the server rolls back everything to process the time warp algorithm, one Saucer instance may be destroyed and re-generated many times. Figure 3.6 shows how this happens. When the Bullet approaches near to the Saucer, it will hit the Saucer in the next few frames, at that time a player press the up key which will move the Ship up. The server will not receive the up movement command before the Bullet hit the Saucer because of the latency. When it receives the command the collision has happened, the server rolls back the game world to the one in which collision has not happened yet and re-runs the game world. During the re-run period, the collision happens again. This scenario leads to the wrong score if we use the conventional static variable method.

Client at T_1 : Bullet is about to hit the Saucer, player presses a left button

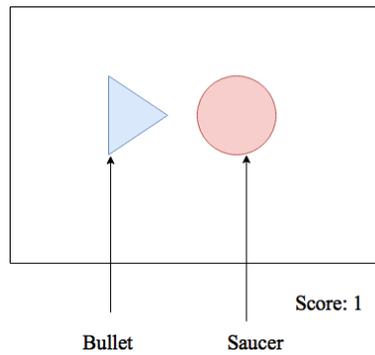


Client takes L latency to the server

Server at (T_1+L) : Bullet is about to hit the Saucer, player presses a left button



Server rolls the game world back, but the score does not change.



Server re-applies the game world, detects another collision and added score by 1.

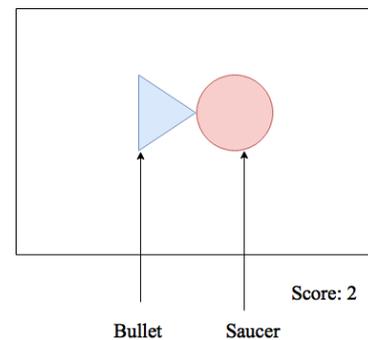


Figure 3.6: Demonstrate why static score is inaccurate with time warp

To solve this problem, Saucer Shoot implements the score as an Object that is managed by time warp. In this way, no matter how many times the server rolls back, it has the ability to keep track of the score. In the above situation, when the server rolled the game world back to before the collision, the score is rolled back to the value before the collision.

4. Evaluation

After implementing Drizzle and a cloud saucer shoot game, I proceeded to evaluate the performance. The evaluation was from two aspects, subjective and objective. For subjective, I designed a user study and invited approximately 30 people to play the game. A survey provides the basic demographics with thoughts on how the time warp algorithm performed. I will discuss the detailed survey in the following section. For objective, I measured the bandwidth of transmitting drawing commands instead of the image or video. I also analyzed the accuracy and consistence of using time warp. In addition, I analyzed the memory and process overhead of Drizzle with time warp.

4.1 Subjective

I designed a user study to evaluate time warp performance. I will describe the study content, survey environment and procedure in the following subsections.

4.1.1 Survey design

To evaluate how time warp performs with latency, I designed a survey. The survey was implemented on *Qualtrics* [<https://wpi.qualtrics.com>]. In the first part, I asked people about their age, gender and the frequency of playing online games. Figure 4.1 shows the demographics questions.

People would then play 10 game sessions. After playing each session, he or she was required to rate the responsiveness and graphics consistency from 0 to 5. Figure 4.2 shows the

second part of the survey content. Responsiveness refers to how well the game responds to the player's input. Graphics consistency refers to how well the graphics perform.



WPI

What is your test ID?

How often do you play online games?

- Everyday
- Once per week
- Once per month
- Never

Gender?

- Male
- Female
- other
- prefer not to answer

Age?

- <20
- 20~30
- >30

Figure 4.1 Demographics survey questions



WPI

Rate the responsiveness of the previous game session (Session 1/10)

0 (unresponsive)	1	2	3	4	5 (responsive)
<input type="radio"/>					

Rate the visual consistency

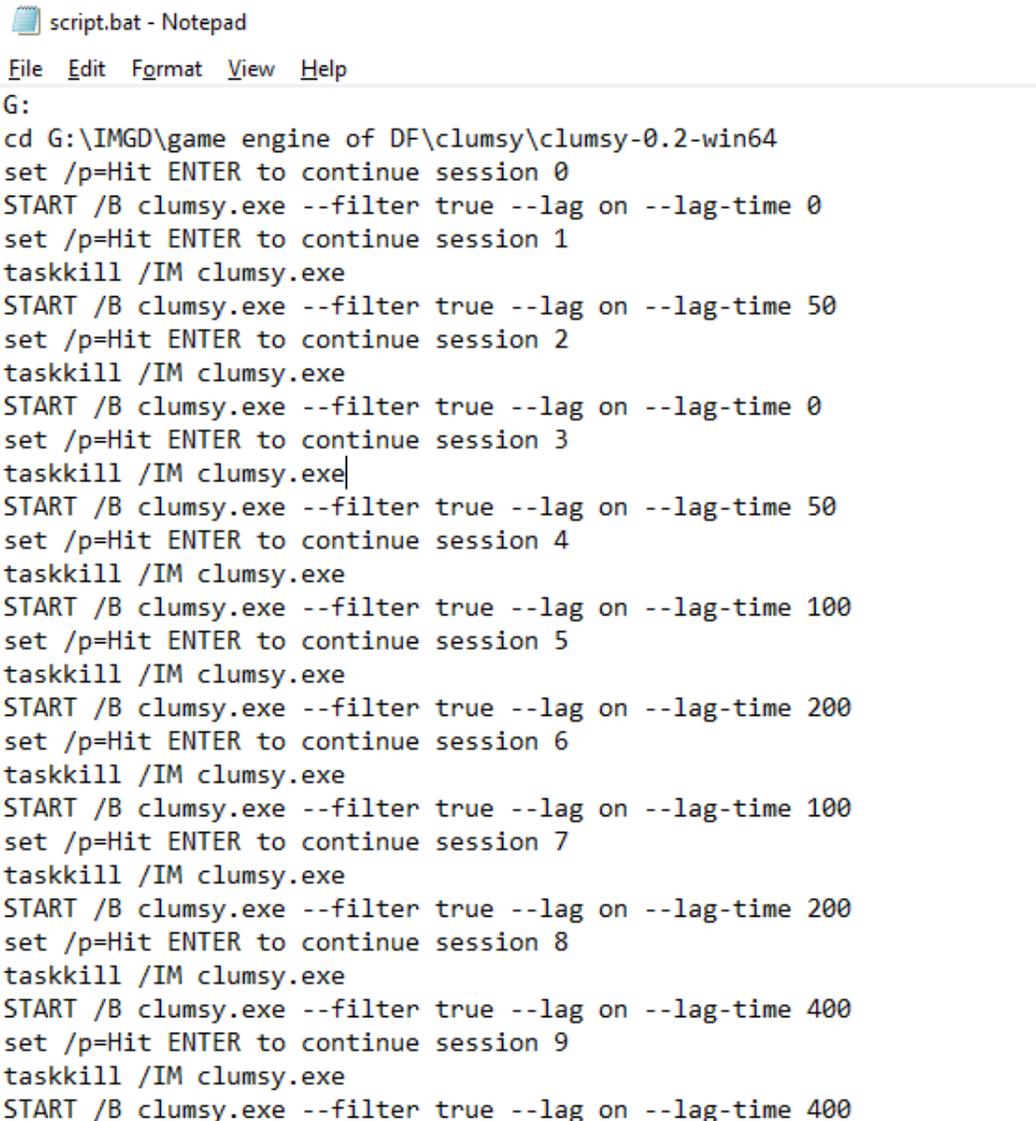
0 (inconsistent)	1	2	3	4	5 (consistent)
<input type="radio"/>					

Figure 4.2. Responsiveness and consistency survey questions

4.1.2 Set up environment

As for 10 similar game sessions, the differences between them are. 1) they are under different latencies. The latencies are 0, 100, 200, 400, and 800ms. Clumsy [<https://jagt.github.io/clumsy/>] is used to simulate the different latencies. 2) 5 sessions are with time warp on, while the other 5 game sessions are with time warp off. I set up a batch file to help me easily control these configurations (see Figure 4.3). I manually shuffled the order so that when playing with the game session, players did not know the latency of the game session or whether time warp is on or off. Figure 4.4 shows a screenshot of one game session. First, I described the game play to the respondent. In detail, I introduced a 15 seconds game play and

then told users how to move the Hero and how to shoot a Bullet . I also told them to try to hit all Saucers to get a higher score.



```
script.bat - Notepad
File Edit Format View Help
G:
cd G:\IMGD\game engine of DF\clumsy\clumsy-0.2-win64
set /p=Hit ENTER to continue session 0
START /B clumsy.exe --filter true --lag on --lag-time 0
set /p=Hit ENTER to continue session 1
taskkill /IM clumsy.exe
START /B clumsy.exe --filter true --lag on --lag-time 50
set /p=Hit ENTER to continue session 2
taskkill /IM clumsy.exe
START /B clumsy.exe --filter true --lag on --lag-time 0
set /p=Hit ENTER to continue session 3
taskkill /IM clumsy.exe
START /B clumsy.exe --filter true --lag on --lag-time 50
set /p=Hit ENTER to continue session 4
taskkill /IM clumsy.exe
START /B clumsy.exe --filter true --lag on --lag-time 100
set /p=Hit ENTER to continue session 5
taskkill /IM clumsy.exe
START /B clumsy.exe --filter true --lag on --lag-time 200
set /p=Hit ENTER to continue session 6
taskkill /IM clumsy.exe
START /B clumsy.exe --filter true --lag on --lag-time 100
set /p=Hit ENTER to continue session 7
taskkill /IM clumsy.exe
START /B clumsy.exe --filter true --lag on --lag-time 200
set /p=Hit ENTER to continue session 8
taskkill /IM clumsy.exe
START /B clumsy.exe --filter true --lag on --lag-time 400
set /p=Hit ENTER to continue session 9
taskkill /IM clumsy.exe
START /B clumsy.exe --filter true --lag on --lag-time 400
```

Figure 4.3. Screenshot of the configuration batch file

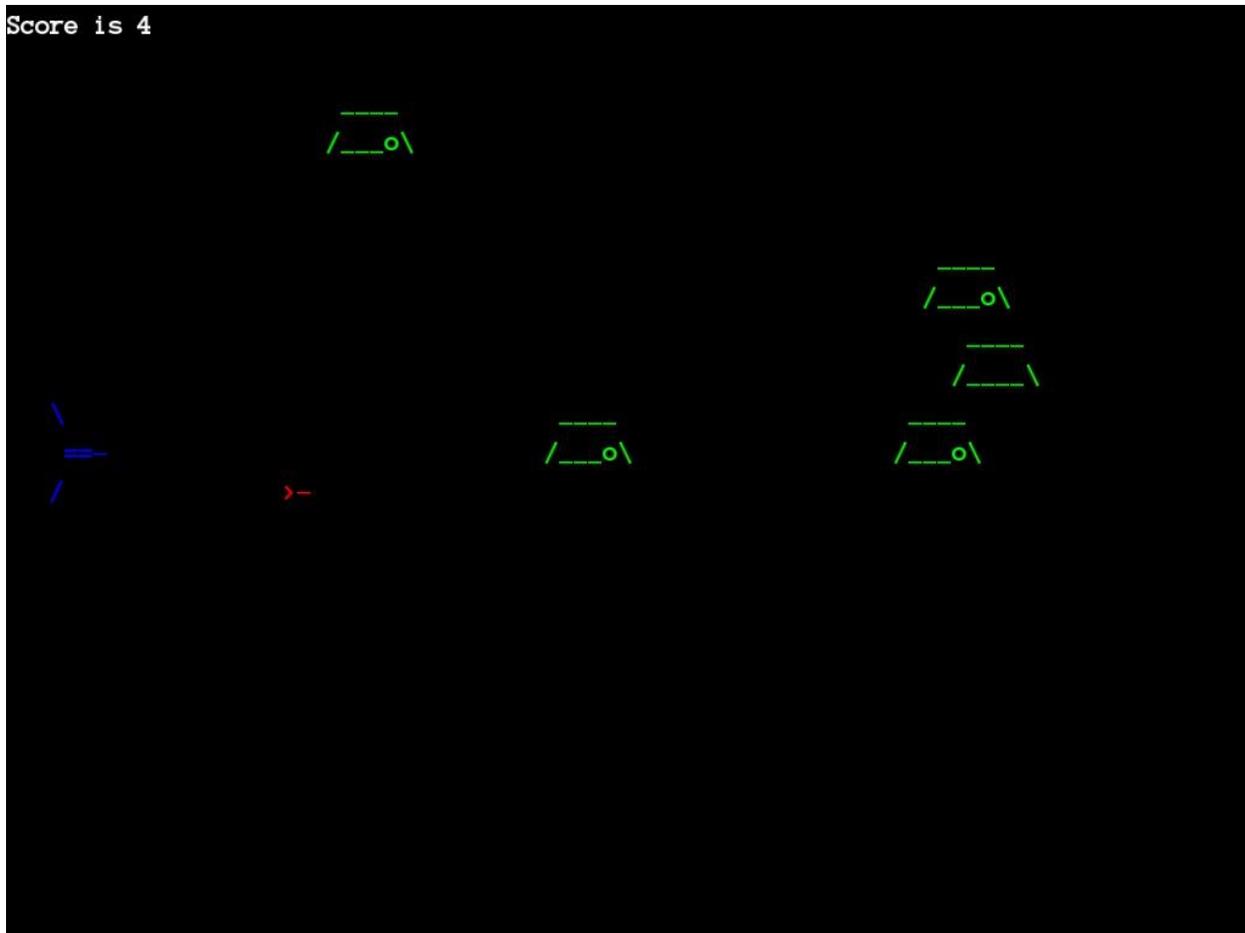


Figure 4.4. Screenshot of the cloud saucer shoot game

4.1.3 Hardware environment

I set up the server and the client both on my laptop and put it into Zoo Lab. My laptop is equipped with a 14” display, a Core i7 CPU, 8GB memory and the Windows 10 Pro operating system. I invited people to play the game session and answer the survey. I did not engage in the study except to provide the basic introduction and instruction for the survey.

4.1.4 Survey procedure

Pre-test: Setup environment in Zoo lab.

Step 1: User provides demographics information.

Step 2: Batch file started which configures network latency with Clumsy, starts server, and configures time warp on/off.

Step 3: User starts client and plays game session.

Step 4: User responds to survey of responsiveness and consistency.

Repeat steps 2-3 for 10 game sessions.

Repeat steps 1-4 for 30 users.

4.1.5 Survey result

I had 30 users to participate the survey. Their age gender and frequency of playing online games are shown in the Figure 4.5. 25 are males and 5 are females, and all respondents are between 20 to 30 years old. Almost half of the respondents play online games every day.

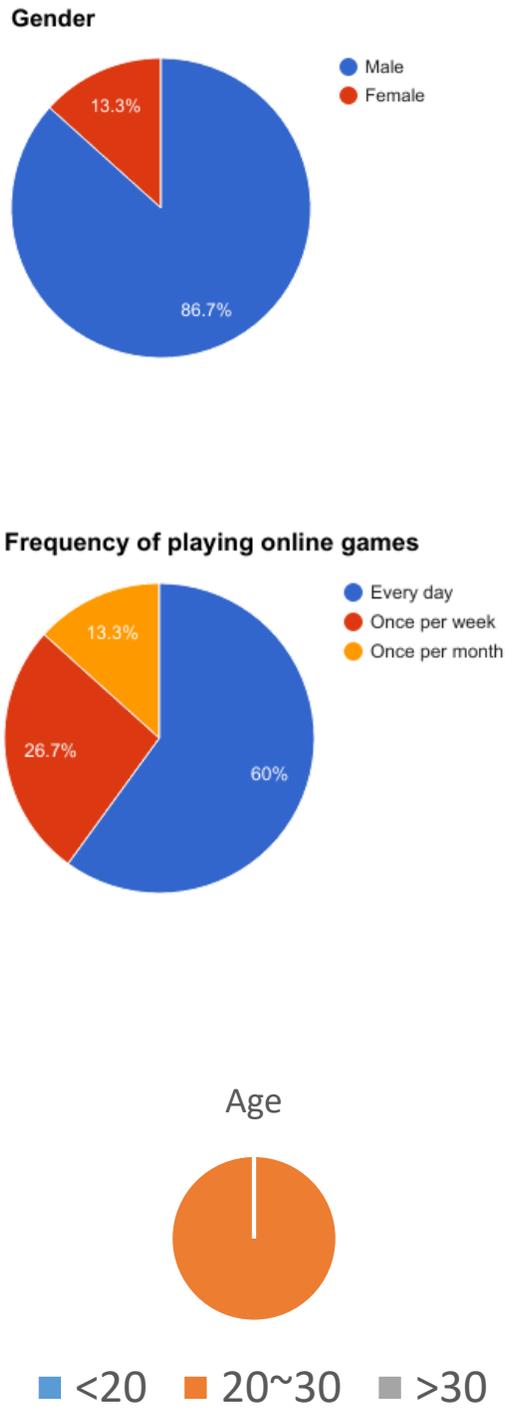


Figure 4.5. Demographics result.

Figure 4.6 shows the relation of player's score with latency. The x-axis is the different added latencies and the y-axis is the player's score. The points in the figure are the average scores of all 30 players with standard error bars. The red trend line is the game sessions with the time warp algorithm on and the blue trend line is the game sessions with time warp algorithm off. As we can see from the figure, players performed worse when the latency became higher. When the latency was in the range of 100ms and 400ms, players had higher scores with the time warp algorithm than without time warp.

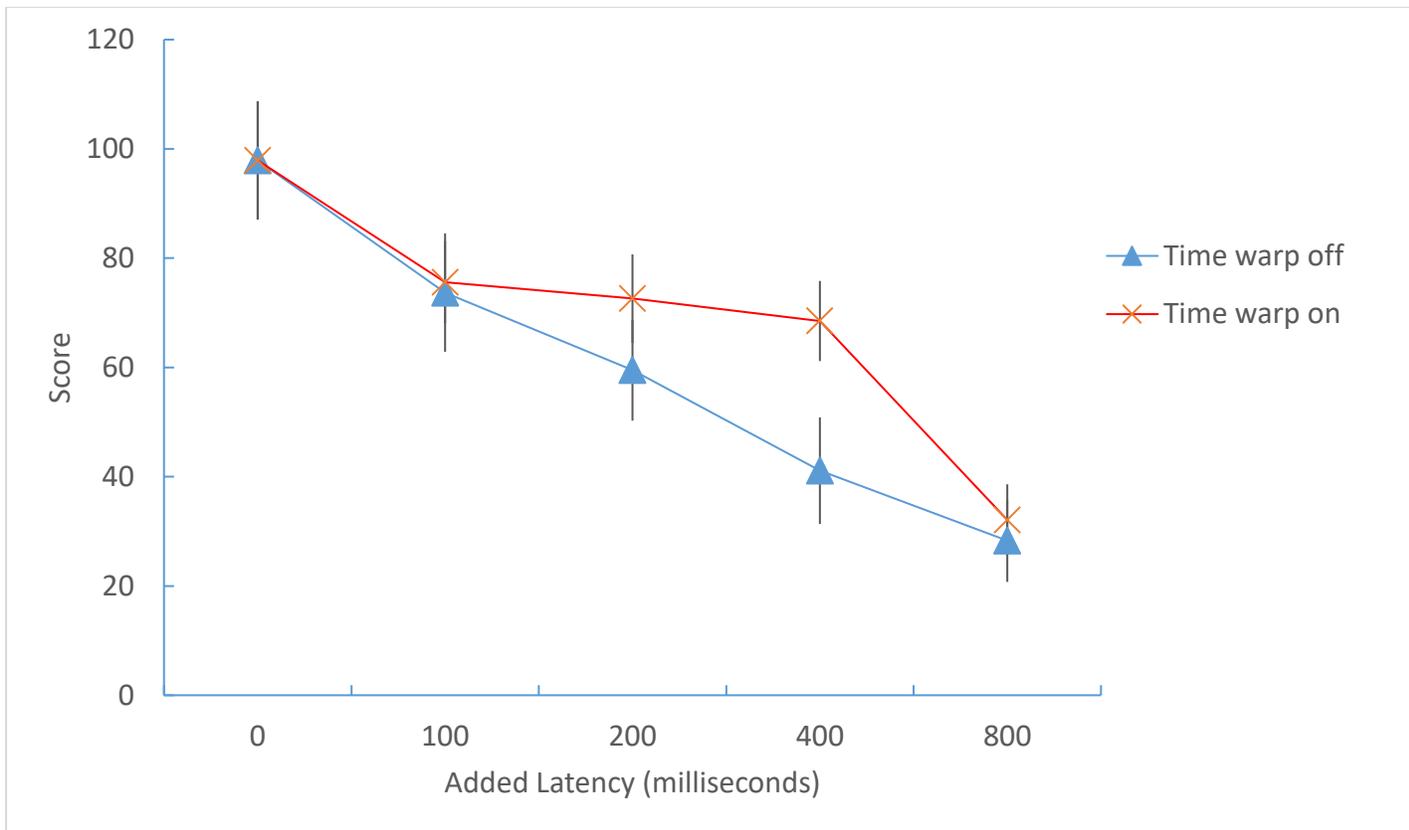


Figure 4.6. Relation of player's score with latency

Figure 4.7 shows the relation of responsiveness and consistency with latency. The x axis is the different network latencies and each point is the average responsiveness or consistency

with standard error bars. The blue trend line is with time warp algorithm off and the red trend line is with time warp algorithm on.

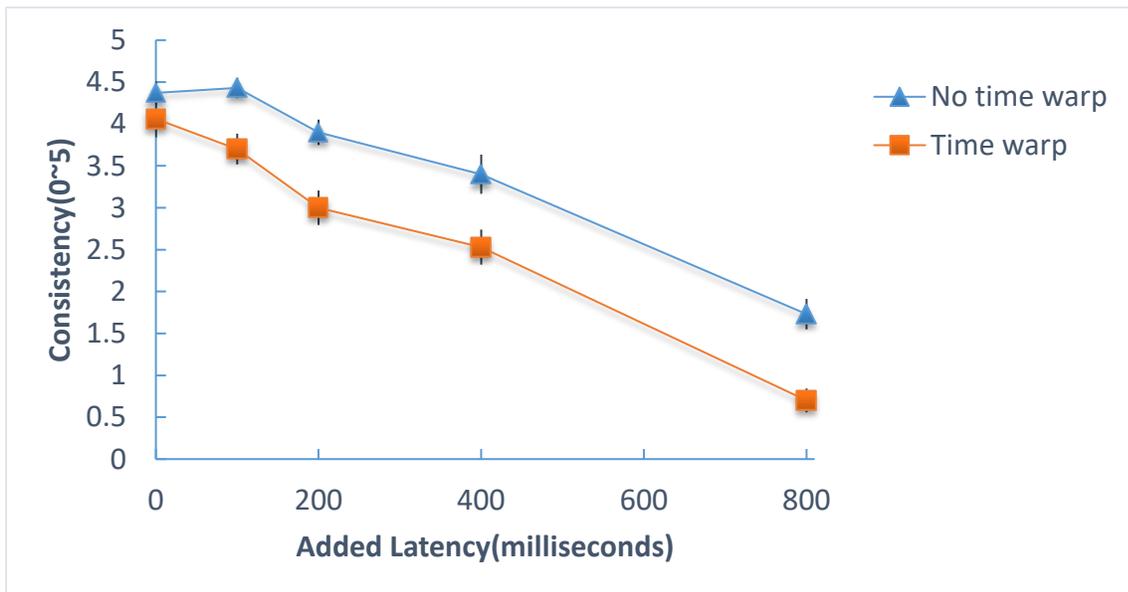
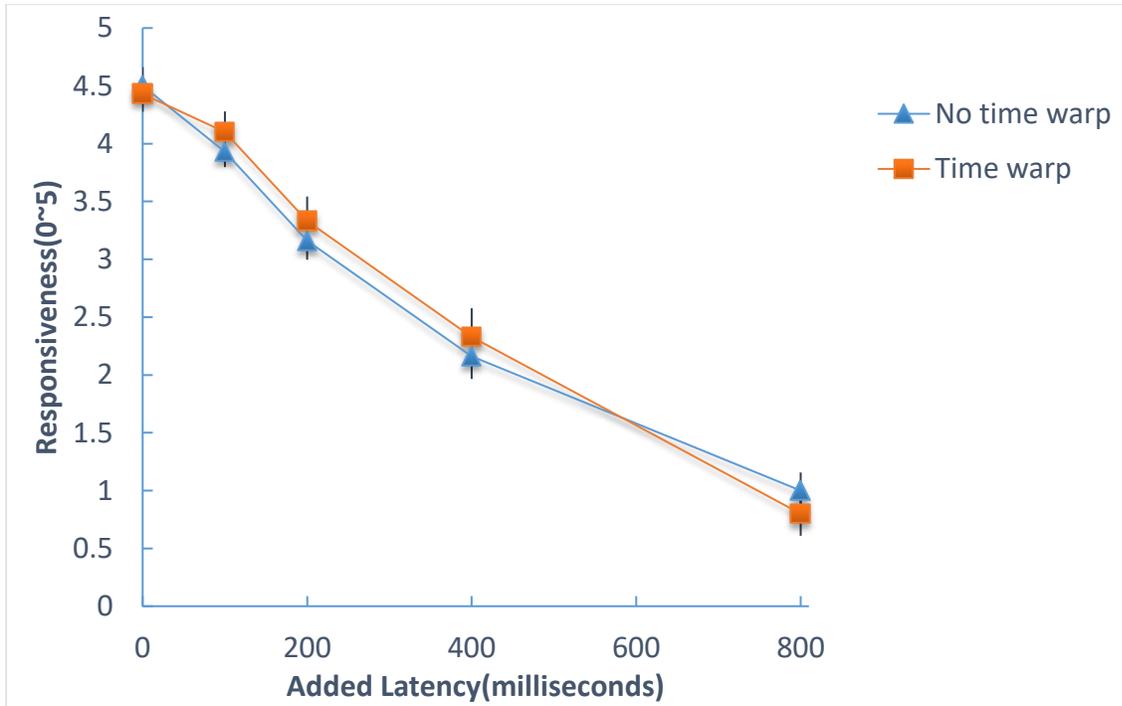


Figure 4.7. Result of survey on responsive and consistent with different latencies and time warp

As we can see from the Figures, players did not feel much difference in responsiveness with time warp on or off, while they feel significantly less consistency with time warp on. This survey also suggests setting up a threshold for turning on the time warp. For example, if the network condition is really good, latency is lower than 50ms, it is unnecessary to use time warp because it will affect the graphics consistency which may lead to player confusion. While if the network condition is bad and the latency is greater than 800ms, it is unnecessary to use time warp, either, because under high latency, time warp algorithm cannot improve performance much but will lead to more inconsistency.

4.2 Objective

This section analyzes the performance of time warp with respect to memory usage, then performance for sending drawing commands with the respect to network bandwidth.

For the time warp algorithm, the server is required to store every game world in memory. In Drizzle, I set the max size of the storing list to 300, which means if the game runs at 30 frames per second, Drizzle can store up to $300/30 = 10$ s of previous game worlds. As the number of objects become larger, it takes more time to deep copy every game world and store it in memory. For example, assuming every object has 10 attributes, every game world has 100 objects, then for the deep copy, it needs to copy least $10*100 = 1000$ attributes and allocates new space 1000 times for these attributes. This copy overhead has to happen each frame. From a memory aspect, assuming the server stores 300 game worlds, 10000 objects per game world, each object has 10 attributes, and each attribute needs to take up 4 bytes of space. Then, the server needs at least $300*10000*10*4$ bytes= 120MB of RAM for the game world alone.

For sending drawing commands to replace sending images or compressed video, I compared the network bandwidth needed per frame with the same saucer shoot game. The result is shown in Figure 4.8:

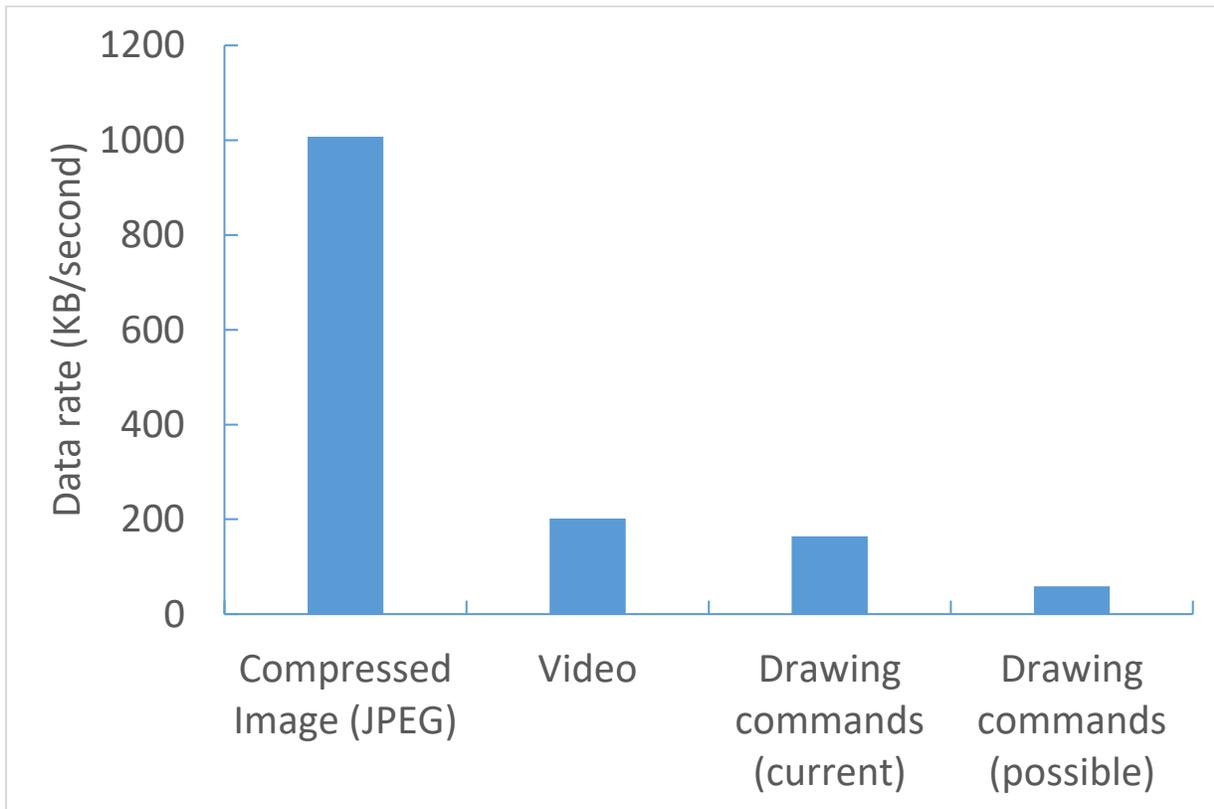


Figure 4.8. Network bandwidth comparison for three game screen transmitting methods

First, I considered transmitting images between the server and the client. I used SFML `capture()` to save the game scene as a JPEG image. Every frame would produce an image from 12KB to 59KB. I run a full-length game session and got the average size of the images as 33.58KB. So, it would transmit $33.58 \times 30 = 1007.4$ KB per second. For transmitting with compressed video, I used FFmpeg to convert one game session to a compressed video [10]. After compressing a video for a whole game session, the average 30 frames per second video is 6.7KB. Thus, the data rate of this method would be $6.7 \times 30 = 201$ KB per second. As for Drizzle's current

sending drawing commands, every draw command packet size is 28 bytes, every object has around 10 draw commands, and the average number of objects in the game session is 20. Thus, the total size for 1 second of game play is $20 * 10 * 28 * 30 \text{ bytes} = 164.1 \text{ KB}$. The transmission packet can be reduced from 28 bytes to 10 bytes, by sending only X, Y location (8 bytes), character (1 byte) and color (1 byte). In this situation, the total data rate for the same game session would be $20 * 10 * 10 * 30 \text{ bytes} = 58.6 \text{ KB per second}$. Compared with sending images, transmitting drawing commands can significantly reduce the network bandwidth.

I also evaluated the relationship between network bandwidth of different sending methods and number of objects. Figure 4.9 shows the result: The x-axis is the number of objects and the y-axis is the data rate in KB per frame. There are four trend lines in the figure: blue is the method of sending JPEG images, the red is the method of sending compressed video, grey is Drizzle's current sending drawing command method and yellow is a proposed sending drawing command method. From the figure, all methods increase linearly with the increasing number of objects. From the trend of these lines, we can see if the number of objects keep increasing, Drizzle's current sending method will require more network bandwidth than the method of sending compressed video, but it will always be much lower than the method of sending JPEG images. However, the improved sending drawing method would have a lower network bandwidth than either JPEG images or compressed video.

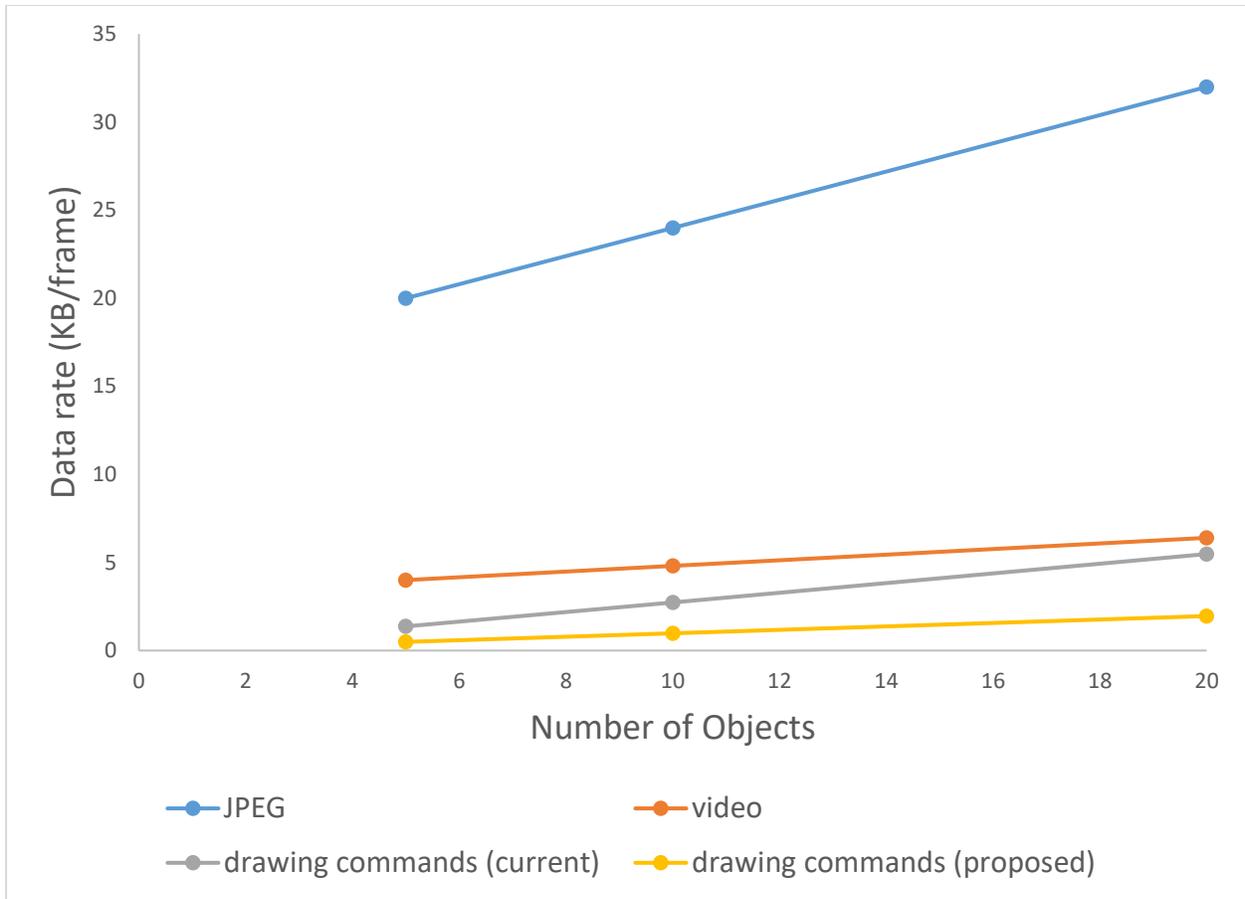


Figure 4.9. Network bandwidth for different sending methods and number of objects

I evaluated the CPU load on the server side for different game actions. The whole game loop consists mainly of 4 parts: Sending drawing commands, deep-copying the previous game world, re-applying the game world fast-forward and doing an update. As I increased the number of objects, I calculated the CPU time for each part. Figure 4.10 shows the result.

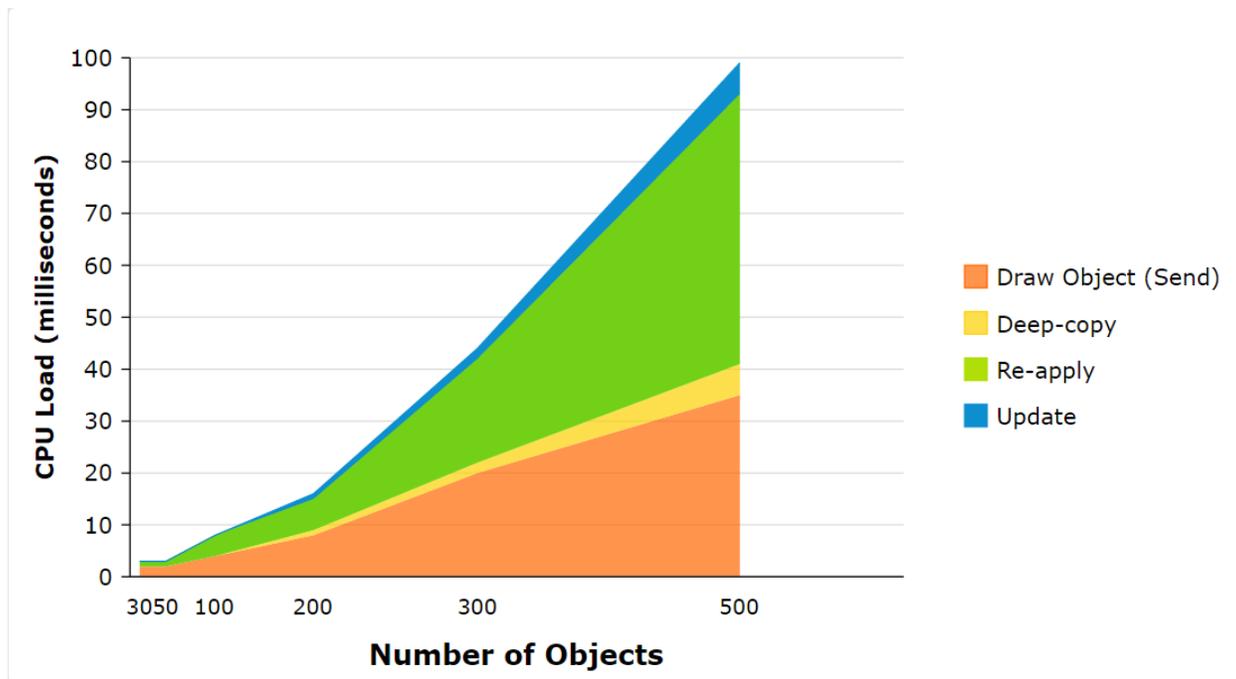


Figure 4.10. CPU loads distribution as the objects increase.

The experiment had 200ms latency. When the number of objects become over 300, drawing objects (sending all characters) and re-apply took up most of the CPU time. The reason for this is that each object has around 10 characters to draw, so the server needed to iterate and call $10 \times \text{number of objects}$ times `send()`. In the re-apply period, because there was 200ms latency, the server needed to re-apply and deep copy at least $200/30 = 6$ previous game worlds. Updating each game world needed almost the same time as an independent update phase.

5. Conclusion

With the rapid development of the Internet, cloud gaming has increasingly gained attention. While cloud gaming is promising for the distribution of games, it is still in its infancy with several performance challenges to address [1]. Conventional cloud games need high network bandwidth [7]. Network latency is also a serious factor affecting the cloud game experience [3]. Trying to address these challenges, I implemented a cloud game engine called Drizzle. Drizzle uses a new method of transmitting drawing commands rather than sending images from the server to the client, and implements a time warp algorithm for latency compensation.

To evaluate Drizzle, I designed and implemented a cloud saucer shoot game. Then I set up a survey, invited people to play the game under different latencies and time warp conditions, and asked them to fill in the survey. Besides the survey, I analyzed time warp performance with respect to memory and CPU load. In addition, I compared three transmission methods with respect to network bandwidth: sending images, sending compress video and sending drawing commands.

Based on the evaluation, sending drawing commands did reduce the network bandwidth, and reduced the workload on the server so that the server could run the game more smoothly. From the survey results, time warp could make the game feel more responsive to players.

Beyond this thesis, there are still many ways to improve the method and algorithm. Explorations may reduce the network bandwidth by optimizing the transmitting packets. For example, Drizzle could send the object drawing commands instead of character drawing

commands. To improve the time warp algorithm, I could survey the best-working range for time warp and only enable it when the latency is in this range to get a better user experience.

6. References

- [1] C. Huang, C. Hsu, Y. Chang, and K. Chen. "GamingAnywhere: An Open Cloud Gaming System," in Proceedings of the ACM Multimedia Systems Conference (MMSys'13), Oslo, Norway, February 2013.
- [2] Meng Luo. Uniquitous: Implementation and Evaluation of a Cloud Gaming System in Unity3d, Master Thesis, Worcester Polytechnic Institute, May 2014.
- [3] K.-T. Chen, P. Huang, and C.-L. Lei. How Sensitive are Online Gamers to Network Quality? *Commun.ACM*, 49(11):34–38, Nov. 2006.
- [4] Mark Claypool. Dragonfly Program a Game Engine from Scratch. 2015 <<http://dragonfly.wpi.edu/>>
- [5] "GamingAnywhere - An Open Source Cloud Gaming System." GamingAnywhere - An Open Source Cloud Gaming System. N.p., n.d. Web. 20 Nov. 2016. <<http://gaminganywhere.org/>>.
- [6] Daniel, Chris. "Get Your Game on with SHIELD TV." The Official NVIDIA Blog, 5 Jan. 2017, <blogs.nvidia.com/blog/2017/01/04/shield-tv-gaming-ces/>
- [7] Grenville Armitage, Mark Claypool, and Philip Branch. *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*, John Wiley and Sons, Ltd., June 2006. ISBN 0470018577.
- [8] Claypool, Mark, Tianhe Wang, and McIntyre Watts. "A Taxonomy for Player Actions with Latency in Network Games." Proceedings of the 25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video. Portland, OR, March 18 - 20, 2015.
- [9] De Winter, Davy, et al. "A Hybrid Thin-client Protocol for Multimedia Streaming and Interactive Gaming Applications." Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video, Newport, Rhode Island, May 22-23, 2006.
- [10] "FFmpeg." FFmpeg. N.p., n.d. Web. 20 Nov. 2016. <<https://www.ffmpeg.org/>>.
- [11] "Simple and Fast Multimedia Library." SFML. N.p., n.d. Web. 20 Nov. 2016. <<http://www.sfml-dev.org/>>.
- [12] "TCP Connection Flow." IBM Knowledge Center. N.p., n.d. Web. 29 May 2017.
- [13] "The X-Windows Disaster". Art.Net. Retrieved 10 November 2009.