B.Sc. in Computer Science and Engineering Thesis

An Improved Dead Reckoning Approach for 2D Top-Down Multiplayer Car Racing

Submitted by

Abdul Jawad
1105006

Supervised by

Dr. Md. Saidur Rahman

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology

Dhaka, Bangladesh

February 2017

CANDIDATES' DECLARATION

This is to certify that the work presented in this thesis, titled, "An Improved Dead Reckoning Approach for 2D Top-Down Multiplayer Car Racing", is the outcome of the investigation and research carried out by us under the supervision of Dr. Md. Saidur Rahman.

It is also declared that neither this thesis nor any part thereof has been submitted anywhere else for the award of any degree, diploma or other qualifications.

Abdul Jawad
1105006

CERTIFICATION

This thesis titled, **"An Improved Dead Reckoning Approach for 2D Top-Down Multiplayer Car Racing"**, submitted by the group as mentioned below has been accepted as satisfactory in partial fulfillment of the requirements for the degree B.Sc. in Computer Science and Engineering in February 2017.

Group Members:

Abdul Jawad

Supervisor:

Dr. Md. Saidur Rahman
Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

ACKNOWLEDGEMENT

First and foremost, I would like to thank my research supervisor Professor Dr. Md. Saidur Rahman. Without his assistance and dedicated involvement in every step throughout the process, this thesis would have never been accomplished. I am also thankful for his continuous support in reviewing and correcting my language and suggesting new way of thinking.

Finally, my sincere gratitude to Lecturer Shareef Ahmed for his help in reviewing my thesis.

Dhaka
February 2017

Abdul Jawad

Contents

List of Figures

List of Tables

ABSTRACT

One of the critical challenges for online real-time multiplayer games is to provide the players with a consistent view of the virtual world despite the network delays. Prediction algorithms are mainly used in order to achieve this and usually all forms of games use the popular DIS dead reckoning algorithm. This algorithm takes the position, velocity and acceleration of the last known time frame into account and predicts the new position at the required time frames. Different versions of the algorithm gathers this velocity and acceleration in different ways but they will have a significant performance overhead. We propose a newer approach in the domain of racing games where we will only need the position as the sole parameter. Alongside, we will use the predetermined slope of the static tracks in the algorithm to calculate the new positions. It reduces the necessity of passing the velocity and acceleration within the packet through the network. The velocity and the acceleration will be calculated from the previous data with the help of the predetermined slope of the track. As a result, this newer approach will avoid some typical performance overheads and also reduce the error rate of our eventual results.

Chapter 1

Introduction

The techniques of latency hiding exist for LAN (local area network) and even for WAN (wide area network) connection. But these techniques fail in the domain of mobile networks because of the latencies as it can be up to 2 magnitudes larger than a typical WAN configuration. With the advancement in processing power in mobile devices, *Multiplayer Online Games*(MOGs) becomes more popular. In the United States alone in 2016 Consumers have spent 24.3 billion US dollars on video games [24]. MOGs make up a huge portion of one of the largest entertainment industries on the planet. About 72% of the total game community play online [24].Which is about 700 millions in numbers. 62% of gamers play games with others, either in-person or online, 36% of gamers play games on their smart phones and 25% of gamers play on their wireless device [6]. In Figure 1.1 a graph have been shown which indicates the massive growing nature of game industry.

Defined by the IEEE standard 1278.1, MOGs are a kind of *distributed interactive simulation* (DIS) which is an infrastructure that links simulations of various types at multiple locations so that it can create realistic, complex, and virtual worlds for the simulation of highly interactive activities. DISs are intended to support a mixture of computer controlled behavior (computer generated forces) with virtual entities, virtual entities with live operators (human in-the-loop simulators), live entities (operational platforms and test and evaluation systems), and constructive entities (war games and other automated simulations) [10]. Protocol data units (PDUs), are exchanged on a network between simulation applications. Major issues that DIS faces are loss and delays of PDUs. Network latency or delay refers to the time it takes for packets of PDUs to travel from sender to receiver. *Delay* caused by the time it takes for a signal to propagate through a given medium, plus the time it takes to route the signal through routers. Moreover jitter is a term used as a measure of the variability over time of delay across the network [9] and *packet loss* is when one or more packets of data travelling across a computer network fail to reach their destination which is typically caused by network congestion. A lack of consistency between remote participants, various entities jittery movement and loss of accuracy in the simu-

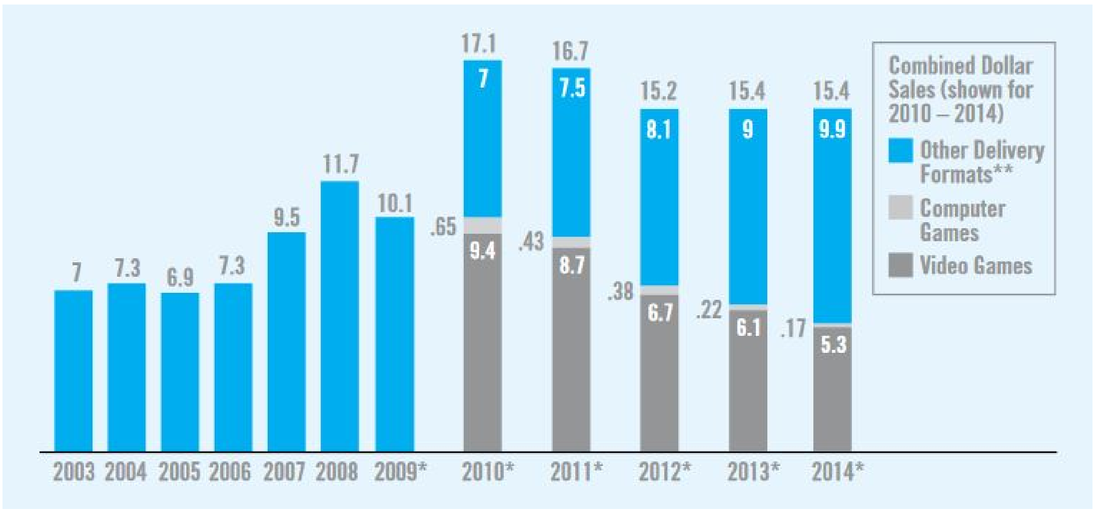

Figure 1.1: The NPD Group Retail Tracking Service - Games Market Dynamics: U.S.

lation are the prime results of delay and loss in a DIS. For its distributed nature, MOGs are way more hard to design and produce than a locally played traditional video game. In particular, players playing in geographical locations thousands of kilometers away from each other need to have their actions appear to be executed in the same virtual space.

So the main objective of the MOGs is the maximization of user experience by minimizing the odd effects of the network during play. For delay or lost packet transmission objects in a scene rendered at out-of-date locations. When objects are rendered at their latest known position, their movement is jittery and sporadic because they are being drawn at a location where they actually are not, and this looks unnatural.

In the rest of the chapter some introductory concept of real time game is provided. In Section 1.1 the challenges for developing a real time multiplayer game is given. Then in Section 1.2 application of such prediction technique is described. In Section 1.3 previous works on this area have been described. Finally Section 1.4 is the description of the organization of thesis.

1.1 Challenges

Network bandwidth and latency are important factors in achieving playability in Multiplayer Online Games. Because of the recent widespread of broadband internet access, the network bandwidth problem is not a big issue any longer. On the other hand, network latency problem is hard to solve due to the physical characteristics related to data transmission. The latency in typical Ethernet LAN is generally less than 0.3 ms and varied in WAN environment. The main cause of network latency is the distance between players, but network situation such as congestion can be a major contributing factor. According to Cheshires experiments in 1995, it took 84.5 ms for a round-trip from the East Coast to the West Coast of the United States.

Table 1.1: Round Trip Time *(RTT)* of some cities ($milisecond$)

City List			
City Name	Paris	Washington	Barcelona
Los Angeles	166.172	73.843	155.858
New York	78.085	33.793	91.274
Stockholm	33.353	155.536	56.04
Berlin	20.845	100.772	52.818

Usually, *round-trip time* (RTT) for international transmission is longer. According to the preliminary experiments performed for this research, it took about 300 ms for a round-trip from Dhaka, Bangladesh to Seoul, Korea [3].

1.2 Application

Path prediction like dead reckoning is used in various works. From modern warfare to simple gaming path prediction algorithms are used verily. Moreover, delay or lag compensating technique like dead reckoning are required not only for MOGs but also for all other DIS application like space exploration, military and medical organizations. So maximization of user experience ultimately entails improving the quality of such applications. But in here the main focus would be on gaming.

According to the report by Newzoo gamers worldwide will generate a total of $99.6 billion in revenues in 2016, up 8.5% compared to 2015. For the first time, mobile gaming will take a larger share than PC with $36.9 billion, up 21.3% globally. Asia Pacific (APAC) continues to dominate worldwide, accounting for 47% of the market. China alone accounts for one quarter of all global game revenues. This growth represents a 10.7% year over year increase. North America is the second largest region with estimated revenues of $25.4 billion in 2016, a year over year growth rate of 4.1%.

1.3 Previous Works

Qualitative studies were performed to determine the odd effect of network states on the players. Participants commented on the quality of play at different levels of lag and jitter. This was measured by Mean Opinion Score (MOS) vs delay (Lag) and jitter [28]. Figure 1 and 2 of that paper show the relation between them. Their result proves that higher quantities of lag and jitter correlate with poor player experience.

In a typical first person shooting game tournament *Unreal Game 2003* players performance was measured through total kill and death [26]. Through 20 different scenarios of lag , different players experiencing different lag it was shown that higher lag tends to lower players performance [5, 21, 26, 28]. When measuring the odd network effect on the players experience, score of a player is a trivial metric. Fewer successful shots and kill (Player score) are the result of higher degree of delay jitter and loss. For example Aggarwal et al ran test in a fast-paced tank shooting game called BZFlag, Wattimena et al. [28] gathered data from the popular shooting game *Quake IV* and Ishibashi et al. [17] developed a distributed version of *Quake III* to test in. Performance of a player in these type of first person shooting game is highly based on reflexes and instant player input. So as a result even slight delay or loss in the network pays off very badly. But these metric should only be considered for a selected genre or type of game. For example in [17], popular *Real Time Strategy* game (RTS) *Warcraft III*, latency up to 3 seconds has only marginal effect on performance. This is because in *Warcraft III* it is more important to execute strategy than good performance. So [8] adverse effect of network will yield a perception of poor game quality.

Another lag hiding technique is introducing local lag [21, 23] in the game. *Local lag* is the time gap between when an input is given and when its execution takes place. As local lag give some extra time to the PDUs to reach the server subsequently to the other clients it improves simulation accuracy by hiding lag. This method is effective as it ensures time synchronization of distributed network and it also allows all the connected clients to receive an event before it executes. But there can arise some problems for local lag because it creates lag between when a player issues a command and when it is executed. So for shooting game like Counter Strike or racing game like Need For Speed a player is more likely to notice delay regarding player movement or mouse movement input than delay regarding firing or shooting. For this reason the common method in most popular action games is to set a delay while firing or shooting, but to have no delay in regard to player movement or mouse movement input.

Liang and Boustead in [21] propose a method to further reduce the negative effects of lag. Data packets can arrive at an odd time even it can arrive out of order. So event needs to be sorted in a way for maintaining temporal accuracy. Such a technique is *Time Warp*. In this technique simulations are rewind to execute event at the appropriate time. Which in turn retain consistency of the events between different parties geographically. In another [11, 23] version of a time warp E. Cronin, B. Filstrup and A. R. Kurc proposed that features a trailing *state synchronization* method wherein instead of rewinding to time stamps when detecting a change, whole game states are simulated that are slightly behind the current time as to allow more time for late information to arrive. When an inconsistency is detected, the leading state need only rollback to a previous state.

For jittery movements of the game objects and player in the game scene it is convenient to draw objects in the past which is outlined by the popular game development community *Valve*. This allows the receiver to smoothly transit between the past location to the newly received positional data update [1,2]. The method works because time is rewound for that object, allowing current or past information about an object to represent the future information of that object. Then to draw the object, a position is interpolated from information ahead of and behind the new current position (which is actually in the past),as shown in Figure2. If interpolation time is greater than the lag then the interpolation is possible. *Interpolation* time is a constant or equal to the lag of the server. In Half Life II [1,2] a constant 100ms interpolation time is used. Which seems sufficient for that game to cover large percentage of lag or loss in the network.

The most used lag compensating technique is *dead reckoning* for mitigating all the odd and adverse effect of the network. Dead reckoning a fundamental feature of the DIS standard was developed to reduce the amount of communication between the participating entities. Previous research works on dead reckoning have been largely focused on the evaluation of the *extrapolation equation* [16]. In general to control error dead reckoning use a fixed threshold regardless of the relationship between entities. It restricts the PDUs to be sent while it maintains a threshold. Data packets are only sent when the simulation crossed the threshold value. Dead reckoning is defined as any process to deduce the approximate current position of an object based on past information. The position of the participating entities are usually calculated based on the last known update. An IEEE standard dead reckoning formula [10] is given by

$$P = P_0 + V_0 \Delta t + \frac{1}{2} A_0 \Delta t^2$$

Here,

$$P = \text{newly predicted position}$$

$$\Delta t = \text{elapsed time}$$

$$P_0 = \text{previous position}$$

$$V_0 = \text{previous velocity}$$

$$A_0 = \text{previous acceleration}$$

This equation works well for real time game by assuming that object does not change direction during this elapsed time.

In [5] S. Aggarwal and H. Banavar draw the difference between a *time stamped* and a *non time stamped* dead reckoning. Time synchronization and time stamped dead reckoning means that receiver will receive packets in proper order and execute in exact same condition as they were generated [5, 15, 18, 21, 22]. Time synchronization while adding network traffic greatly improves simulation accuracy and reduce export error. [5, 19, 20]

In [4] Jacob Agar proposed an *EKB scheme* that can be utilized for distributed intractive simulation system with a team scenario such as military training simulations. Their algorithm is better in terms of *average export error* over *interest scheme* [19] and traditional dead reckoning [10]. In terms of packets transmitted across the network, his hybrid improved EKB [4] out performs interest scheme in all different network latency situation. And the hybrid improved EKB sends almost as few packets as traditional dead reckoning, while providing more realistic movement and more accurate predictions than traditional dead reckoning.

Traditional prediction algorithm works by taking both the velocity and acceleration constant. linear prediction fails to maintain an accurate result because player movement is rarely linear in nature. Different types of video games [25] needs different types of prediction system which was discussed in Wolf and Pantels work. They discussed 5 different prediction schemes which are constant velocity , constant acceleration, constant input position , constant input velocity and constant input acceleration. The experiments show that prediction schemes can be useful for networked games. They cannot reduce the latency directly, but they can be used to reduce its impact.

1.4 Thesis Organization

The rest of the thesis is organised as follows. In Chapter 2 we have given some important definitions for our experiments. In that chapter we have also introduced necessary assumption that is needed for our experiments. Chapter 3 is where we have proposed our improved technique. In that chapter some variations of DR is also discussed. In Chapter 4 we presented our experiment result and have given a relative comparison with DR. There we also discussed about the experiment procedure and data collection procedure of our experiment. And finally, Chapter 5 is a conclusion.

Chapter 2

Preliminaries

In this chapter we define some basic terminologies of games and interpolation methods. Definitions that are not included in this chapter will be introduced as they are needed. At first Section 2.1 some basic definitions have given. In Section 2.2 definition and classification of racing game is given. Finally in Section 2.3 we define some basic and preliminary assumption of this thesis.

2.1 Basic Terminology

In this section we give some definitions of the games that are used throughout the reminder of this thesis.

2.1.1 What is Game?

A *game* is structured form of play, usually undertaken for enjoyment and sometimes used as an educational tool. Games are distinct from work, which is usually carried out for remuneration, and from art, which is more often an expression of aesthetic or ideological elements. However, the distinction is not clear-cut, and many games are also considered to be work (such as professional players of spectator sports or games) or art (such as jigsaw puzzles or games involving an artistic layout such as Mahjong, solitaire, or some video games). Key components of games are goals, rules, challenge, and interaction. Games generally involve mental or physical stimulation, and often both. Many games help develop practical skills, serve as a form of exercise, or otherwise perform an educational, simulational, or psychological role. A game is a system in which players engage in an artificial conflict, defined by rules, that results in a quantifiable outcome [27].

2.1.2 What is Video Game?

Video games are computer or microprocessor-controlled games. Computers can create virtual spaces for a wide variety of game types. Some video games simulate conventional game objects like cards or dice, while others can simulate environs either grounded in reality or fantastical in design, each with its own set of rules or goals. A computer or video game uses one or more input devices, typically a button/joystick combination (on arcade games); a keyboard, mouse or trackball (computer games); or a controller or a motion sensitive tool (console games). More esoteric devices such as paddle controllers have also been used for input. the first commercial video game, Pong, was a simple simulation of table tennis. As processing power increased, new genres such as adventure and action games were developed that involved a player guiding a character from a third person perspective through a series of obstacles. This *real-time* element cannot be easily reproduced by a board game, which is generally limited to *turn-based strategy*; this advantage allows video games to simulate situations such as combat more realistically. Additionally, the playing of a video game does not require the same physical skill, strength or danger as a real-world representation of the game, and can provide either very realistic, exaggerated or impossible physics, allowing for elements of a fantastical nature, games involving physical violence, or simulations of sports. Lastly, a computer can, with varying degrees of success, simulate one or more human opponents in traditional table games such as chess, leading to simulations of such games that can be played by a single player [14].

2.1.3 What is Multiplayer Online Game (MOG)?

Multiplayer online games are computer/video games that can be played by multiple players at the same time. Players can be teamed up to achieve the same goals or compete against each other. The popularity of multiplayer online games is significant and continues to rise, mainly due to the increased strategic complexity of game play against other human being, as opposed to computer artificial intelligence (AI) [14].

2.1.4 Interpolation

In the mathematical field of numerical analysis, *interpolation* is a method of constructing new data points within the range of a discrete set of known data points. In engineering and science, one often has a number of data points, obtained by sampling or experimentation, which represent the values of a function for a limited number of values of the independent variable. It is often required to interpolate (i.e. estimate) the value of that function for an intermediate value of the independent variable. One of the simplest methods is linear interpolation (lerping). Consider the above example of estimating $f(2.5)$. Since 2.5 is midway between 2 and 3, it is reasonable to take $f(2.5)$ midway between $f(2) = 0.9093$ and $f(3) = 0.1411$, which yields 0.5252. Generally, linear interpolation takes two data points, say (x_0, y_0) and (x_1, y_1), and the interpolation is given by:

$$\frac{y - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0}$$

for a value x in the interval (x_0, x_1)

Polynomial interpolation is a generalization of linear interpolation. Note that the linear interpolant is a linear function. We now replace this interpolant with a polynomial of higher degree.

Spline interpolation uses low-degree polynomials in each of the intervals, and chooses the polynomial pieces such that they fit smoothly together. The resulting function is called a spline. The classical approach is to use polynomials of degree 3 the case of *cubic splines*. Cubic splines or *Cubic Hermite splines* are typically used for interpolation of numeric data specified at given argument values x_1, x_2, ... ,x_n to obtain a smooth continuous function. The data should consist of the desired function value and derivative at each x_k. (If only the values are provided, the derivatives must be estimated from them.) The Hermite formula is applied to each interval (x_k, x_{k+1}) separately. The resulting spline will be continuous and will have continuous first derivative.

2.1.5 Trend line

The yellow line in the middle of the track in Figure 2.3 is the *trend line* which is calculated from the data values of 50 consecutive games played by 4 different players.

The data is collected with 0.2 sec time interval. The intermediate points are calculated using cubic spline interpolation. This yellow line or Trend Line represent the typical gamer behavior in 2D top down game. This line will define the direction of the velocity and acceleration of a player.

Figure 2.1 represents the changes in x coordinate values with respect to time and Figure 2.2 is the changes in y coordinate values with respect to time. These graphs are drawn from the data of 50 consecutive games played by 4 different players.

2.1.6 Export error

Export error is the difference between the predicted position and the actual position of that time. As data comes to server a little after the actual event took place and takes some more time to propagate to the other players there will always be some deviation from the actual position. This deviation from the actual position cause the export error. More precise position prediction

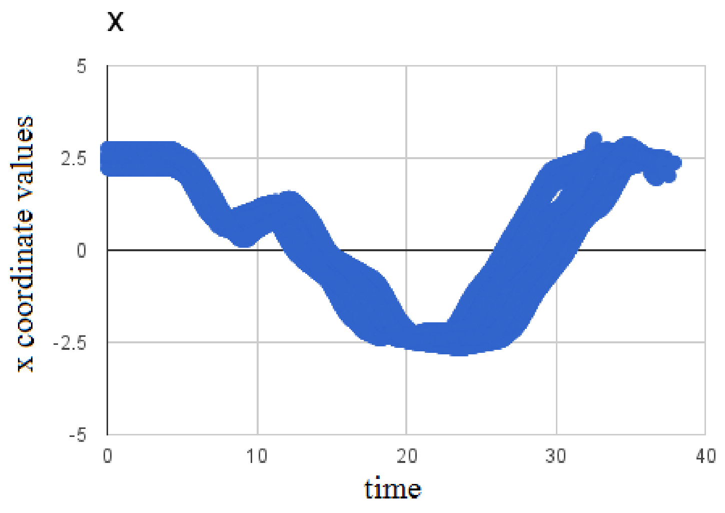

Figure 2.1: Changes in x coordinate values with time.

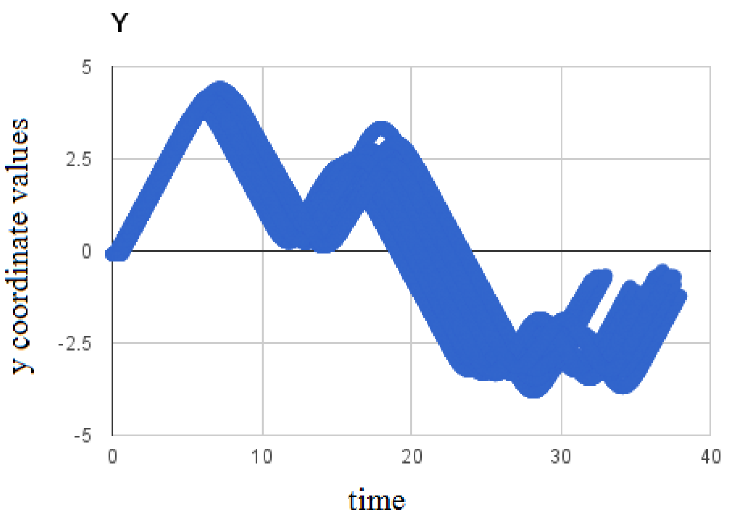

Figure 2.2: Changes in y coordinate values with time.

results in less export error.

Performance of the various path prediction technique can be measured from the export error. Export error also increases with delays in network. If P_t and E_t are the predicted and actual position at time t then *export error*, *total export error* and the *average export error* are represented by the following equation:

$$\text{Export Error} = P_t - E_t$$

$$\text{Total Export Error} = \sum P_t - E_t$$

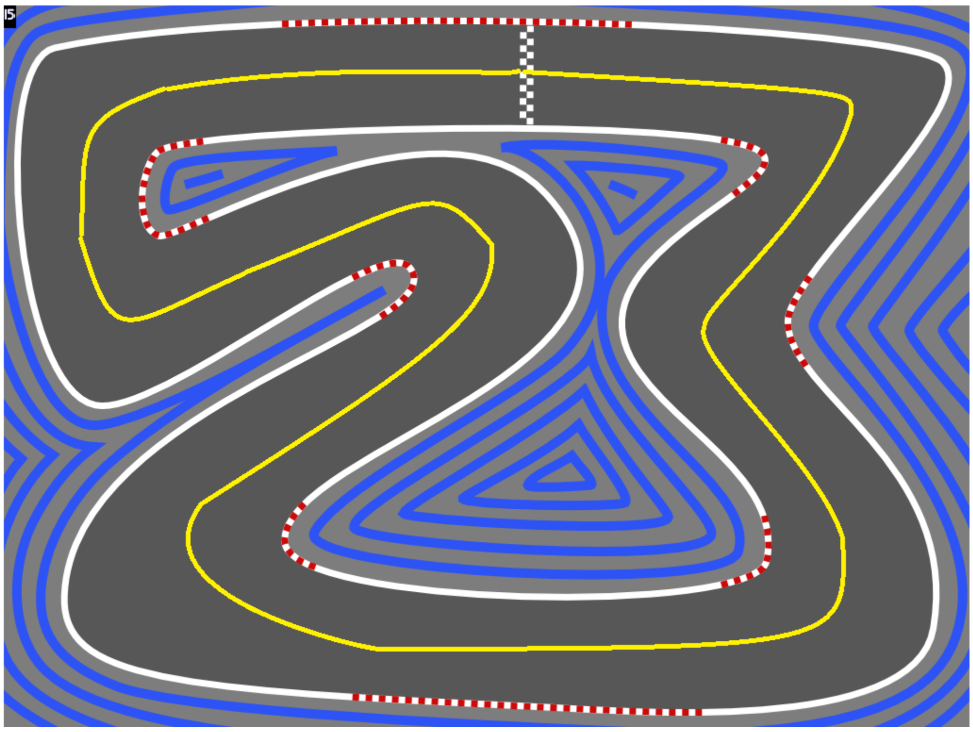

Figure 2.3: Trend line.

$$AverageExportError = \frac{\sum_{i=1}^{i=n} P_t - E_t}{n}$$

2.2 Multiplayer Racing Game

The *racing video game* genre is the genre of video games, either in the first-person or third-person perspective, in which the player partakes in a racing competition with any type of land, air or sea vehicles. They may be based on anything from real-world racing leagues to entirely fantastical settings. In general, they can be distributed along a spectrum anywhere between hardcore simulation and simpler arcade racing games. Racing games may also fall under the category of sports games. The most important assumption is that the main goal of each participant in a typical race is to reach the finish in the shortest time possible. Although this will not be true for every player but it does model an average player for typical race games.

2.3 Behavioral Assumptions

Some assumptions about the behavioral pattern of the player can be made in order to increase the performance of our prediction yet not compromising reliability. We will model the typical player in a typical racing game based on these assumptions. In most general racing games, players run the race in order to win which means covering the environment (or track) in the shortest possible time. But in a 2D top down racing, the track remains constant. Under these assumptions, we can neglect any unusual behavior from the player and focus only to those that are necessary for the win.

Although some players could opt for upgrades or some atypical behavior like skidding through the track (subject to particular games), the major goal remains to finish the touch line in the shortest time possible. To make this behavior appear convincing, a good prediction should be based on the idea that each simulated vehicle will attempt taking turns following the most ideal trajectory in order to achieve the best completion time. The assumptions should get adjusted accordingly time to time if sometimes the prediction becomes too optimistic. Though it poses another difficult challenge about how to adjust. Predicting the position of other vehicles will be smoother if these adjustments (or corrections) are done in a regular manner which often requires a trade-off between correctness and reliability. For this reason, every new corrective prediction should be done gradually in order to avoid any abrupt change [12].

There is no point in trying to simulate fast-paced changes (e.g. collisions), because to the participating drivers, these changes are some approximated realities [12]. Two player might not sense the collision at the same exact point, as local realities may differ. So, an arbitrating server may be used to decide whether collision happened or not. But still, todays mobile networks will not send this information to the player before 2-3 seconds, long after any sensible visual feedback could be achieved by the action.

As traditional dead reckoning acts after the information of a large error arrive at other clients, it fail to properly predict curves on the track. An improvement would be predicting positions onto the center line of the track. This results in a greater visual effect globally as the vehicle keeps on track while it approaches a curve. But it might lead to a monotonous and unnatural simulation. This limitation could greatly be avoided if we project the predicted position not onto the center line but onto a varied approximation of the ideal driving line.

Chapter 3

Algorithmic Design Description

In this chapter we explain a new method of prediction algorithm. At first in Section 3.1 we explain the dead reckoning technique and in Section 3.2 we will give the proposed improvement of the dead reckoning algorithm for 2D top down multiplayer car race game.

3.1 Traditional Dead Reckoning Algorithms

Traditional dead reckoning algorithm sends a positional update to clients when an object deviates from its predicted path by some threshold. That is why a dead reckoning algorithm that successfully improves path prediction not only minimizes the odd effect of delay but also minimizes the network traffic by sending only the necessary updates when needed. Duncan and Gracanin [13] propose a method the *pre-reckoning scheme*, that sends an update when it anticipate that an object will cross some predefined threshold. For calculating the threshold change, the angle of the current movement and the last movement is analyzed. If this angle is larger then the threshold then it assumes that the threshold will be crossed very soon and a PDU is sent. Pre-Reckoning yields better results when variability in player movement is low.

Cai et al. [7] present an auto adaptive dead reckoning algorithm. The method outlined in [7] assumes a *perspective* view on the world such that farther away objects are smaller and less visible. However, in a 2D video game where an *orthographic* view is used, all the objects in view of the player is of normal size and therefore almost all of the objects are of interest to the user. The results suggest a considerable reduction in update packets without sacrificing accuracy in *extrapolation*. Dynamic threshold for predicting objects results in less data needed to be sent over the network. So it reduces the packets which will be sent through the network. But it does not eliminate the requirement for increasingly accurate prediction schemes. A dynamic threshold allows farther away objects to not require a high degree of accuracy. But regardless, closer objects still need to be predicted accurately. Furthermore, while all the prediction methods referred above are capable of predicting players movement relatively accurate. More elaborate methods should be considered to handle high amounts of lag. Once there is a high amount of network delay, traditional methods of dead reckoning become too inaccurate and the export error becomes too large. Ultimately players start to notice a loss of consistency.

First order dead reckoning is less accurate compared to the second order dead reckoning. Following are the equations of the first and second order dead reckoning.

$$\text{First order } P_t = P_{t-1} + V_t \Delta t$$

$$\text{Second order } P_t = P_{t-1} + V_t \Delta t + \frac{1}{2} A_t \Delta t^2$$

Methodologically, our proposed solutions will be compared with the IEEE standard dead reckoning algorithm [10].

3.2 Improved Dead Reckoning

As players velocity and acceleration is not constant in a racing game in our proposed algorithm we will calculate the velocity and acceleration using previous data values. Direction of this velocity and acceleration vector will come from the *trend line*. The equation for calculating V_t and A_t is given below:

$$V_t = \frac{2(P_t - P_{t-1})}{\Delta t} - V_{t-1}$$

$$A_t = \frac{V_t - V_{t-1}}{\Delta t}$$

This values will be the magnitude of the resultant velocity and acceleration vector. The direction of the vector will be calculated from the trend line.

If the current position is (x_a, y_a) and (x_p, y_p) is the point on the curve such that line between (x_a, y_a) and (x_p, y_p) is *perpendicular* to the trend line then the direction would be calculated with the following equation:

$$\vec{d} = \frac{(x_m - x_n)\vec{i} + (y_m - y_n)\vec{j}}{\sqrt{(x_m - x_n)^2 + (y_m - y_n)^2}}$$

where (x_m, y_m) and (x_n, y_n) are the two nearest point from (x_p, y_p) on the trend line. In Figure 3.1 the direction vector has shown visually. The black line in the Figure 3.1 is the trend line.

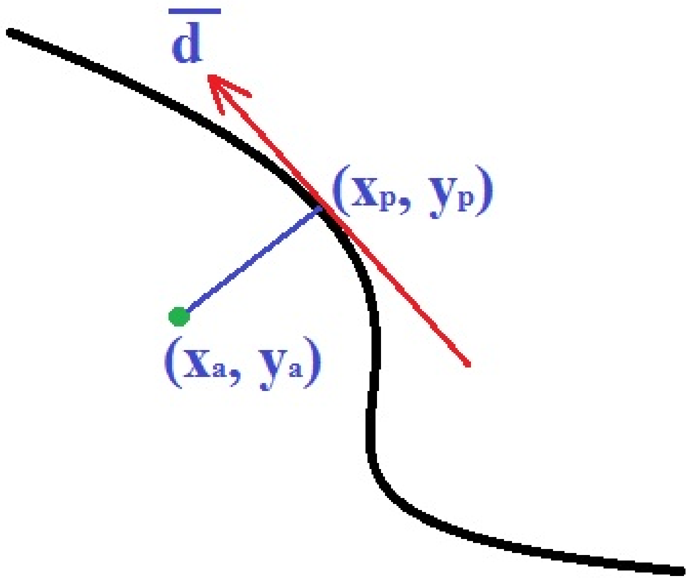

Figure 3.1: Direction Vector

So,

$$\vec{V_t} = V_t\vec{d}$$

$$\vec{A_t} = A_t\vec{d}$$

This V_t and A_t is then used in the dead reckoning equation:

$$\vec{P} = \vec{P}_{t-1} + \vec{V_t}\Delta t + \frac{1}{2}\vec{A_t}\Delta t^2$$

Chapter 4

Exploratory Strategy, Outcomes and Relative Evaluation

In this chapter we have discussed the experimental strategy and also the comparative evaluation. In Section 4.1 we have explained the experimental procedures. Then in Section 4.2 the game-play of the car race have been discussed. Section 4.3 is the data collection procedure. Finally in Section 4.4 and Section 4.5 the testing methodology and the results have been described.

4.1 Experimental Procedures

In this section we will explain the experimental procedures that have been performed to complete this thesis. In Section 4.1.1, the total procedure have been shown step by step.

4.1.1 The test domain

The following methods are conducted in order to complete this experiment. The main objective of the followings is to develop the technique and compare it with the existing dead reckoning algorithm.

- Collect data from 50 test play sessions of 4 players
- Analysis of the collected data
- Extract typical player's play pattern from the collected data
- Analyze various play pattern and construct the Trend Line.
- Develop a new technique for 2D top down car racing

- Implement the technique and compare it with the basic Dead Reckoning algorithm
- Compare the result and draw conclusion

For collecting player's input, to analyze the collected data (for pattern extraction) and conduct empirical and comparative evaluations we implemented an interactive distributed test environment which results in a multiplayer online game named *Car Race*. In Section 4.2, we introduce the major functionality of this test environment.

4.2 The Game-play

The game-play of the Car Race is similar to the traditional top down racing game. In here a player will host the game and act as a server for the game. The other player can join with the server and after a certain interval the game starts. In the game both the player will have a separate car entity of their own which can move freely in all the directions in the 2D plane. Player who touches the finish line will be the winner. The track has collider around it so that players can not take any kind of short cuts and have to be in the track all the time during the play. A screen shot of the game is given in Figure 4.1. When a player touches the finish line the game simulation stops. The game is developed using *Unity Game Engine v5.0*. In this game engine any constant lag can be used for simulation. Simulation can also include packet loss and jitter for simulating actual test environment.

4.3 Data Collection Procedure

Data are collected using various metrics. Positions of the player are tracked as a 2D vector and collected against time and distance covered shown in Figure 2.1 and Figure 2.2. These data are then placed in an excel sheet for further analysis. Different graphs are drawn from the data. From each sessions player spawning times are also recorded. So that it can be used for making the trend line. Player input record consist of a positional 2D vector, total distance covered and total play time.

4.4 The Testing Methodology

4.4.1 Play testing

For obtaining the data , 4 play test sessions had conducted, each ran for between 5-8 minutes and with a total of 4 players showed in Table 4.1. The first play test ran for 6 minutes and had

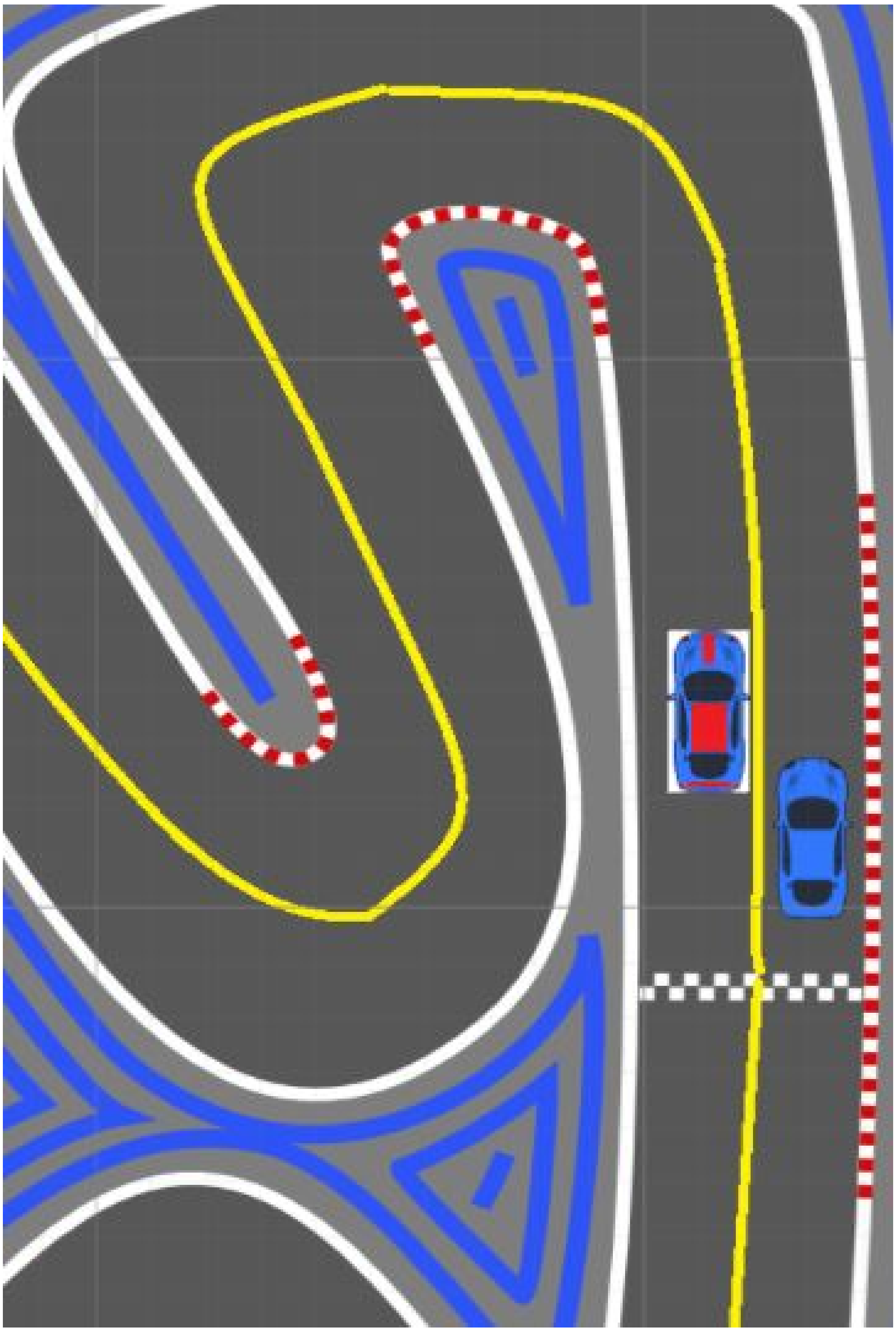

Figure 4.1: Screen shot of the Car Race game

Table 4.1: Play Test Outline

	Duration	# of players	Usage
Session 1	6	2	evaluation
Session 2	5	2	algorithm development
Session 3	8	2	evaluation
Session 4	8	2	evaluation

2 participating players. The second play test ran for 5 minutes and had 2 participants. The third play ran for 8 minutes and had 2 participants. And the final session ran for 8 minutes and have 2 players All participants have played video game before and have experience in 2D top down car race. All participants were male, between the ages of 22-25 and were undergraduate students. Half of the collected data were the testing set and the rest for evaluation of two algorithms. More specifically, we used the second play test as the testing set, and the remaining session 1, 3 and 4 for our evaluations. All the players played in 2 independent computers connected by a LAN.

4.4.2 Performance testing and matrices

Both the algorithm is analyzed with obtained data set. Unity game engine can simulate any amount of delay in the simulation. At the time of a prediction, different metrics can be measured from the data. Both the algorithm is measured by the metric average export error. AEE is the average distance from the predicted position and the actual position of the player for all predictions made. The calculation of AEE is shown in equation below.

$$AEE = \frac{\sum_{i=1}^{i=n} P_t - E_t}{n}$$

E_t and P_t is the predicted position of the player and the actual position of the player respectively at time t and n is the total number of predictions made throughout the game session. The AEE is the best metric in determining the accuracy of any given prediction method [4].

4.5 Results and Analysis

The result of this experiment shows that proposed improved dead reckoning technique gives better results. But for more delays it gives less accurate result in our designed car race game. For 50 ms delay its average export error is 173.45 where the dead reckoning gives average export value 193.33. The AEE values of both of the technique is given in Table 4.2. In Figure 4.2 and Figure 4.2 a comparative graph is also given.

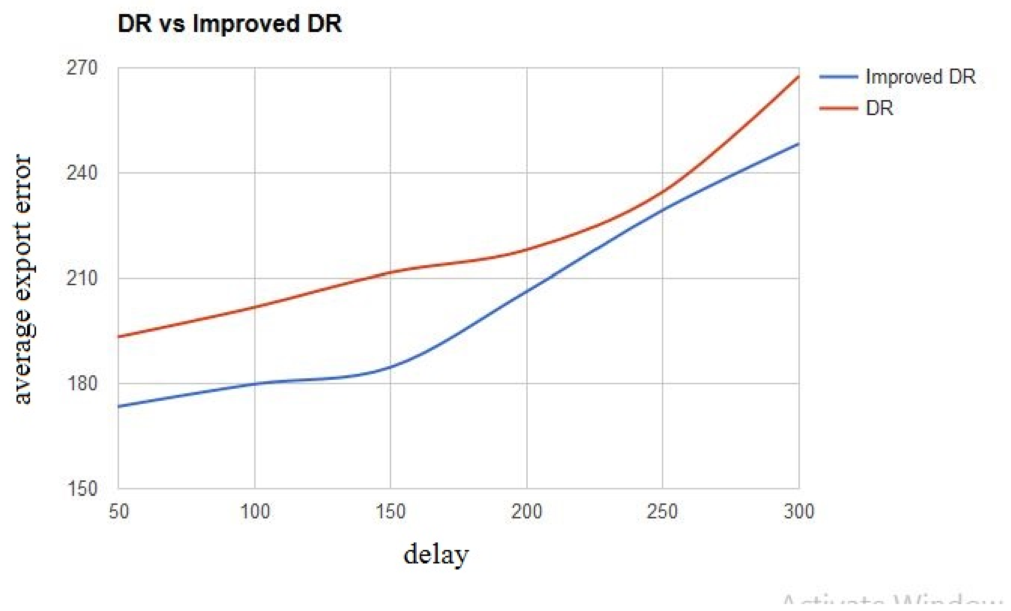

Figure 4.2: AEE of DR and improved DR

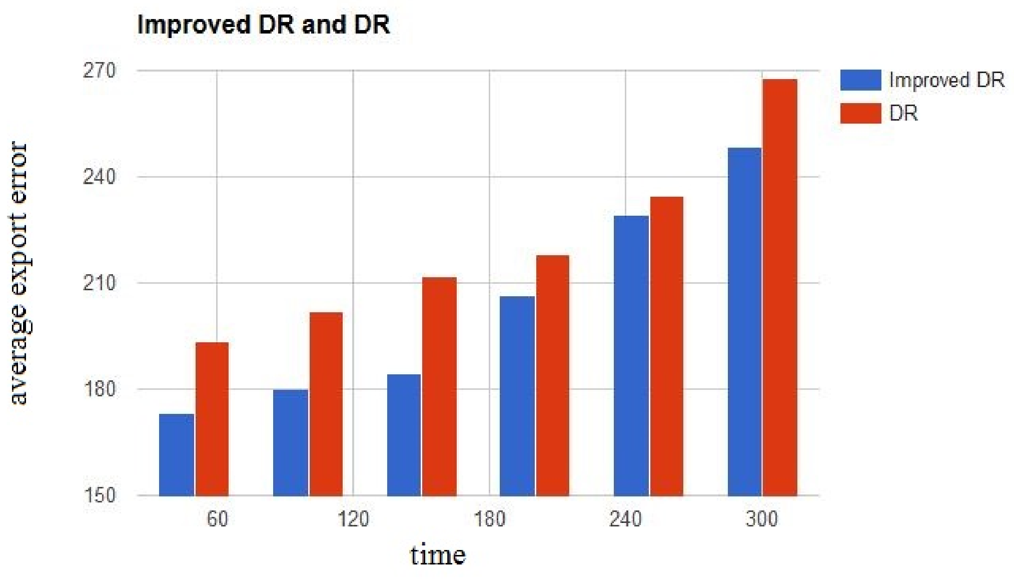

Figure 4.3: AEE of DR and improved DR

Table 4.2: AEE of Dead Reckoning and Improved Dead Reckoning at high latency

	Improved Dead Reckoning	Dead Reckoning
50	173.45	193.33
100	179.83	201.74
150	184.66	211.59
200	206.257	218.06
250	229.43	234.642
300	248.321	267.675

Chapter 5

Conclusion

In the experiment above we have tried to give an effective technique for designing a multiplayer 2D top down game. In Chapter 1 we have given the necessary motivation about the importance of such technique also have shown previous studies for designing such technique. In Chapter 2 we have given some necessary important definitions for our experiments. In that chapter we have also introduced necessary assumption that was needed for our experiments. Chapter 3 is where we have proposed our improved technique. In that chapter some variations of DR is also discussed. In Chapter 4 we presented our experiment result and have given a relative comparison with DR. There we also discussed about the experiment procedure and data collection procedure of our experiment.

Experiment results shows that prediction procedure of the improved DR is good for 2D top down game. It gives less AEE value compared to the DR scheme. In this procedure the velocity and acceleration values remain same as the DR. Only the change in direction gives a better path approximation.

References

[1] Latency compensating methods in client/server in-game protocol design and optimization. https://developer.valvesoftware.com/wiki/LatencyCompensatingMethodsinClient/ServerIn-gameProtocolDesignandOptimization. Accessed : 2010-09-30.

[2] Multiplayer networking. https://developer.valvesoftware.com/wiki/SourceMultiplayerNetworking. Accessed : 2012-03-3.

[3] Ping online from multiple dierent locations worldwide. http://www.startping.com. Accessed 07.02.2015.

[4] AGAR, J., CORRIVEAU, J.-P., AND SHI, W. Play patterns for path prediction in multiplayer online games. In *Communications and Networking in China (CHINACOM), 2012 7th International ICST Conference on* (2012), IEEE, pp. 12–17.

[5] AGGARWAL, S., BANAVAR, H., KHANDELWAL, A., MUKHERJEE, S., AND RANGARAJAN, S. Accuracy in dead-reckoning based distributed multi-player games. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games* (2004), ACM, pp. 161–165.

[6] ASSOCIATION, E. S., ET AL. Essential facts about the computer and video game industry, 2010.

[7] CAI, W., LEE, F. B., AND CHEN, L. An auto-adaptive dead reckoning algorithm for distributed interactive simulation. In *Parallel and Distributed Simulation, 1999. Proceedings. Thirteenth Workshop on* (1999), IEEE, pp. 82–89.

[8] CLAYPOOL, M. The effect of latency on user performance in real-time strategy games. *Computer Networks 49*, 1 (2005), 52–70.

[9] COMER, D. E. *Computer networks and internets*. Prentice Hall Press, 2008.

[10] COMMITTEE, D. S., ET AL. Ieee standard for distributed interactive simulation-application protocols. *IEEE Standard 1278* (1998).

[11] CRONIN, E., FILSTRUP, B., KURC, A. R., AND JAMIN, S. An efficient synchronization mechanism for mirrored game architectures. In *Proceedings of the 1st workshop on Network and system support for games* (2002), ACM, pp. 67–73.

[12] DECARPENTIER, G. J., AND BIDARRA, R. Behavioral assumption-based prediction for high-latency hiding in mobile games. *Proceedings CGAMES* (2007).

[13] DUNCAN, T. P., AND GRAČANIN, D. Algorithms and analyses: pre-reckoning algorithm for distributed virtual environments. In *Proceedings of the 35th conference on Winter simulation: driving innovation* (2003), Winter Simulation Conference, pp. 1086–1093.

[14] EDERY, D., AND MOLLICK, E. *Changing the game: how video games are transforming the future of business*. Ft Press, 2008.

[15] FERRETTI, S. Interactivity maintenance for event synchronization in massive multiplayer online games. *Doktorat. Sveučilište u Bologni, Odjel za kompjutorsku znanost, ožujak* (2005).

[16] FOSTER, L., AND MAASSEL, P. The characterization of entity state error and update rate for dis. In *Proceedings of 11th DIS workshop on Standards for the Interoperability of Distributed Simulation* (1994).

[17] ISHIBASHI, Y., HASHIMOTO, Y., IKEDO, T., AND SUGAWARA, S. Adaptive δ-causality control with adaptive dead-reckoning in networked games. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games* (2007), ACM, pp. 75–80.

[18] LI, F. W., LI, L. W., AND LAU, R. W. Supporting continuous consistency in multiplayer online games. In *Proceedings of the 12th annual ACM international conference on Multimedia* (2004), ACM, pp. 388–391.

[19] LI, S., AND CHEN, C. Interest scheme: A new method for path prediction. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games* (2006), ACM, p. 41.

[20] LI, S., CHEN, C., AND LI, L. A new method for path prediction in network games. *Computers in Entertainment (CIE) 5*, 4 (2008), 8.

[21] LIANG, D., AND BOUSTEAD, P. Using local lag and timewarp to improve performance for real life multi-player online games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games* (2006), ACM, p. 37.

[22] LIN, Y.-J., GUO, K., AND PAUL, S. Sync-ms: Synchronized messaging service for real-time multi-player distributed games. In *Network Protocols, 2002. Proceedings. 10th IEEE International Conference on* (2002), IEEE, pp. 155–164.

[23] MAUVE, M., VOGEL, J., HILT, V., AND EFFELSBERG, W. Local-lag and timewarp: providing consistency for replicated continuous applications. *IEEE transactions on Multimedia 6*, 1 (2004), 47–57.

[24] MORSE, K. Video games, 2017. Online Report.

[25] PANTEL, L., AND WOLF, L. C. On the suitability of dead reckoning schemes for games. In *Proceedings of the 1st workshop on Network and system support for games* (2002), ACM, pp. 79–84.

[26] QUAX, P., MONSIEURS, P., LAMOTTE, W., DE VLEESCHAUWER, D., AND DEGRANDE, N. Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games* (2004), ACM, pp. 152–156.

[27] SALEN, K., AND ZIMMERMAN, E. *Rules of play: Game design fundamentals*. MIT press, 2004.

[28] WATTIMENA, A., KOOIJ, R. E., VAN VUGT, J., AND AHMED, O. Predicting the perceived quality of a first person shooter: the quake iv g-model. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games* (2006), ACM, p. 42.

Index

Chapter 6

Codes

6.1 Car Behavior Code

We use this code to calculate the car position.

```
 using UnityEngine;
using System.Collections;
using System.IO;
using UnityEngine.Networking;

public class carBehavior : MonoBehaviour {

        public GameObject car;
        private GameObject thiscar;
        private GameObject front;
        private string xfilename = "X";
        private string yfilename = "Y";
        private float oldY;
        private bool NowPrint = false;
        StreamWriter x ;
        StreamWriter y ;

        [Range( 0, 10)]
        public float speed;

        [Range( 0, 1)]
        public float positionspeed;

        [Range( 0, 10)]
        public float rotationspeed = 0.1f;

        float distance = 0.0f;
        Vector3 OldPos, NewPos;

        private float PreTime;
        private float r = 0.0f;
        // Use this for initialization
        void Start () {
                OldPos = car.transform.position;
                PreTime = Time.time;

                x = File.CreateText(xfilename);
                y = File.CreateText(yfilename);
                x.WriteLine("D␣--------␣X");
                y.WriteLine("D␣--------␣Y");

        }

        // Update is called once per frame
        void Update () {

                if (!isLocalPlayer) {
                print("i␣am␣in");
                return;
                }
                //printing in every 0.2s
                if (Time.time - PreTime > 0.20f)
                {
                        Debug.Log("P␣"+car.transform.position.ToString("F
                        //Debug.Log("R "+car.transform.rotation.ToString(
                        print(car.transform.position);
                        PreTime = Time.time;
                }
                print(Time.time - PreTime);
                PreTime = Time.time;
                if ((Time.time - PreTime > 0.016f) && NowPrint){
                        distance += Vector3.Distance(OldPos, car.transfor
                        Debug.Log("P␣" + car.transform.position.ToString(
                        x.Write(distance.ToString("F4") + "\t\t");
                        x.WriteLine(car.transform.position.x.ToString("F4
                        y.Write(distance.ToString("F4") + "\t\t");
                        y.Write("e");
                        y.Write(car.transform.position.x.ToString("F4")+"
                        y.WriteLine(car.transform.position.y.ToString("F4

                        PreTime = Time.time;
                }
                if (Input.GetKeyDown(KeyCode.A)) {
                        distance = 0.0f;
                        NowPrint = true;
                }
                if (Input.GetKeyDown(KeyCode.Z)) {
                        x.Close();
                        y.Close();
                }
                //for velocity
                if (Input.GetKey (KeyCode.UpArrow))
                {
                        car.transform.position = Vector3.MoveTowards (car
                        car.gameObject.GetComponent<Rigidbody2D>().AddFor
                }
                //for break
                if (Input.GetKey (KeyCode.DownArrow))
                {
                        car.transform.position = Vector3.MoveTowards (car
                }

                if (Input.GetKey (KeyCode.LeftArrow))
                {
                        Vector3 upDir = front.transform.up;
                        float x1 = Mathf.Acos (upDir.x) + rotationspeed;
                        float x2 = Mathf.Cos (x1);

                        //float y1 = Mathf.Acos (upDir.x) + rotationspeed
                        float y2 = Mathf.Sin(x1);

```

```
                        Vector3 NewVec = new Vector3 (x2, y2, 0.0f);

                        Vector3 NewDir = Vector3.Slerp (upDir, NewVec, 0.

                        car.transform.rotation = Quaternion.FromToRotatio
                        r += rotationspeed;
                        print ("Rotation␣" + car.transform.eulerAngles);
                        print ("eular␣␣" + car.transform.rotation.eulerAn

                        car.transform.eulerAngles (car.transform.rotation
                        car.transform.eulerAngles = new Vector3(car.trans

                        Vector3 val = Vector3.Slerp (front.transform.up,
                }

                //codes for car rotation
                if (Input.GetKey (KeyCode.RightArrow))
                {
                        r -= rotationspeed;
                        car.transform.eulerAngles = new Vector3(car.trans
                        //car.transform.Rotate (car.transform.rotation.x,
                        //car.transform.position = new Vector2 (car.trans
                }

        }
        //for collision detection
        void OnCollisionEnter(Collision col) {

                if (col.gameObject.tag == "Finish") {
                        Destroy(car);
                }
        }

        [Command]
        void CmdSpawn()
        {
                var go = (GameObject)Instantiate(car,
                   transform.position + new Vector3(0, 1, 0),
                   Quaternion.identity);

                NetworkServer.SpawnWithClientAuthority(go, connectionToCl
        }

        //for tracking local player
        public override void OnStartLocalPlayer() {

                thiscar = (GameObject)Instantiate(car, new Vector3(2.76f,
                car = thiscar;
                //Destroy(thiscar);
                car.transform.GetChild(0).gameObject.SetActive(false);
                car.transform.GetChild(3).gameObject.SetActive(true);
                NetworkServer.SpawnWithClientAuthority(car, connectionToC
                thiscar.SetActive(true);
                front = car.transform.GetChild(1).gameObject;
                PreTime = Time.time;
                oldY = car.transform.position.y;
        }
}

//Codes for Dead reconing
public class DeadReckoning : MonoBehaviour
{

        // Use this for initialization

        public List vector3 posUpdate;
        public List vector3 velocity;
        public List vector3 accelaration;
        void Start()
        {
                if (!init())
                {

                        Application.exit(0);
                }
        }

        // Update is called once per frame
        void Update()
        {


        }
}

private vector3 CalculateVelocity(float t)
{

        int size = posUpdate.Len();
        vector3 prepos = posUpdate(size - 1);
        vector3 preprepos = posUpdate(size - 2);
        vector3 vel = prepos - preprepos;
        vel = vel / t;
        velocity.add(vel);
        return vel;
}

private vector3 CalcAcce(vector3 vel1, vector3 vel2, float t)
{
        vector3 acce = vel2 - vel1;
        acce = acce / t;
        accelaration.add(acce);
        return acce;
}

public vector3 newpos(vector3 acce, vector3 vel, float t)
{
        int size = posUpdate.Len();
        vector3 u = velocity(size - 1);
        u = u * t;
        vector3 p = 0.5 * acce * t * t;
        return u + p;
}

public virtual bool OnServerConnect(NetworkConnection conn)
{
        if (conn != null)
```

```
                return true;
}

public virtual void OnServerReady(NetworkConnection conn)
{
        NetworkServer.SetClientReady(conn);
}

// called when disconnected from a server

public virtual void OnClientDisconnect(NetworkConnection conn)
{
        StopClient();
}

// called when a match is joined

public bool OnMatchJoined(JoinMatchResponse matchInfo)
{
        return !matchInfo.isnull;
}

public override bool OnSerialize(NetworkWriter writer, bool forceAll)
{
        if (forceAll)
        {
                // the first time an object is sent to a client, send all
                writer.WritePackedUInt32((uint)this.int1);
                writer.WritePackedUInt32((uint)this.int2);
                writer.Write(this.MyString);
                return true;
        }
        bool wroteSyncVar = false;
        if ((base.get_syncVarDirtyBits() & 1u) != 0u)
        {
                if (!wroteSyncVar)
                {
                        // write dirty bits if this is the first SyncVar
                        writer.WritePackedUInt32(base.get_syncVarDirtyBit
                        wroteSyncVar = true;
                }
                writer.WritePackedUInt32((uint)this.int1);
        }
        if ((base.get_syncVarDirtyBits() & 2u) != 0u)
        {
                if (!wroteSyncVar)
                {
                        // write dirty bits if this is the first SyncVar
                        writer.WritePackedUInt32(base.get_syncVarDirtyBit
                        wroteSyncVar = true;
                }
                writer.WritePackedUInt32((uint)this.int2);
        }
        if ((base.get_syncVarDirtyBits() & 4u) != 0u)
        {
                if (!wroteSyncVar)
                {
                        // write dirty bits if this is the first SyncVar
                        writer.WritePackedUInt32(base.get_syncVarDirtyBit
                        wroteSyncVar = true;
                }
                writer.Write(this.MyString);
        }

        if (!wroteSyncVar)
        {
                // write zero dirty bits if no SyncVars were written
                writer.WritePackedUInt32(0);
        }
        return wroteSyncVar;
}

public class MyNetworkManager : MonoBehaviour
{
        public GameObject alienPrefab;

        NetworkClient myClient;

        // Create a client and connect to the server port
        public void SetupClient()
        {
                ClientScene.RegisterPrefab(CarPrefab);

                myClient = new NetworkClient();

                myClient.RegisterHandler(MsgType.Connect, OnConnected);
                myClient.Connect("127.0.0.1", 4444);
        }
}


//codes for improved dead reckoning

public class IDeadReckoning : MonoBehaviour
{

        // Use this for initialization
        public List vector3 datapoints;
        public List vector3 posUpdate;
        public List vector3 velocity;
        public List vector3 accelaration;
        void Start()
        {
                if (!init()) {
                        Application.exit(0);
                }
        }

        // Update is called once per frame
        void Update()
        {

        }
}

private vector3 direction3D()
{
        int size = posUpdate.len();
        vector3 prepos = posUpdate.get(size - 1);
        for (int i = 0; i < datapoints.len(); i++)
        {
                if (prepos.dot(datapoints.get(i)) == 0)
                {
                        vector3 d = datapoints.get(i + 1) - datapoints.ge
                        d.normalize();
                        return d;
                }
                else {
                        //print("Something weird :");
                }
        }
}

private vector3 CalculateVelocity(float t)
{
        int size = posUpdate.Len();
        vector3 prepos = posUpdate(size - 1);
        vector3 preprepos = posUpdate(size - 2);
        vector3 vel = prepos - preprepos;
        vel = vel / t;
        velocity.add(vel);
        return vel;
}

private vector3 CalcAcce(vector3 vel1, vector3 vel2, float t)
{
        vector3 acce = vel2 - vel1;
        acce = acce / t;
        accelaration.add(acce);
        return acce;
}

public vector3 newpos(vector3 acce, vector3 vel, float t)
{
        int size = posUpdate.Len();
        vector3 u = velocity(size - 1);
        u = u * t;
        vector3 p = 0.5 * acce * t * t;
        return u + p;
}

public virtual bool OnServerConnect(NetworkConnection conn)
{
        if (conn != null)
                return true;
}

public virtual void OnServerReady(NetworkConnection conn)
{
        NetworkServer.SetClientReady(conn);
}

// called when disconnected from a server
public virtual void OnClientDisconnect(NetworkConnection conn)
{
        StopClient();
}

// called when a match is joined
public bool OnMatchJoined(JoinMatchResponse matchInfo)
{
        return !matchInfo.isnull;
}


public override void OnDeserialize(NetworkReader reader, bool initialStat
{
        if (initialState)
        {
                this.int1 = (int)reader.ReadPackedUInt32();
                this.int2 = (int)reader.ReadPackedUInt32();
                this.MyString = reader.ReadString();
                return;
        }
        int num = (int)reader.ReadPackedUInt32();
        if ((num & 1) != 0)
        {
                this.int1 = (int)reader.ReadPackedUInt32();
        }
        if ((num & 2) != 0)
        {
                this.int2 = (int)reader.ReadPackedUInt32();
        }
        if ((num & 4) != 0)
        {
                this.MyString = reader.ReadString();
        }
}
```

Generated using Undegraduate Thesis LaTeX Template, Version 1.2. Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh.

This thesis was generated on Friday 17th February, 2017 at 1:57pm.