Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2014, Article ID 138596, 18 pages
http://dx.doi.org/10.1155/2014/138596

Research Article

Dead Reckoning Using Play Patterns in a Simple 2D Multiplayer Online Game

Wei Shi,[1] Jean-Pierre Corriveau,[2] and Jacob Agar[2]

[1] *Faculty of Business and I.T., University of Ontario Institute of Technology, Oshawa, ON, Canada L1H 7K4*
[2] *School of Computer Science, Carleton University, Ottawa, ON, Canada K1S 5B6*

Correspondence should be addressed to Wei Shi; wei.shi@uoit.ca

Received 5 January 2014; Revised 27 March 2014; Accepted 1 April 2014; Published 12 May 2014

Academic Editor: Abdennour El Rhalibi

In today's gaming world, a player expects the same play experience whether playing on a local network or online with many geographically distant players on congested networks. Because of delay and loss, there may be discrepancies in the simulated environment from player to player, likely resulting in incorrect perception of events. It is desirable to develop methods that minimize this problem. Dead reckoning is one such method. Traditional dead reckoning schemes typically predict a player's position linearly by assuming players move with constant force or velocity. In this paper, we consider team-based 2D online action games. In such games, player movement is rarely linear. Consequently, we implemented such a game to act as a test harness we used to collect a large amount of data from playing sessions involving a large number of experienced players. From analyzing this data, we identified play patterns, which we used to create three dead reckoning algorithms. We then used an extensive set of simulations to compare our algorithms with the IEEE standard dead reckoning algorithm and with the recent "Interest Scheme" algorithm. Our results are promising especially with respect to the average export error and the number of hits.

1. Introduction

Consumers have spent 20.77 billion US dollars on video games in the United States alone in 2012 [1]. 36% of gamers play games on their smart phones and 25% of gamers play on their wireless device. 62% of gamers play games with others, either in-person or online [2]. Multiplayer online games (MOGs) make up a huge portion of one of the largest entertainment industries on the planet. Consequently, maximizing a player's play experience while playing a MOG is key for the success of such games.

MOGs are a kind of distributed interactive simulation (DIS), which is defined by the IEEE standard 1278.1 as an infrastructure that links simulations of various types at multiple locations to create realistic, complex, and virtual worlds for the simulation of highly interactive activities. DISs are intended to support a mixture of virtual entities with computer controlled behaviour (computer generated forces), 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) [3]. Data messages, known as protocol data units (PDUs), are exchanged on a network between simulation applications. Delay and loss of PDUs are the two major issues facing DISs. *Delay* (or equivalently, network latency) refers to the time it takes for packets of PDUs to travel from sender to receiver. This delay is usually taken to be 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. *Jitter* is a term used as a measure of the variability over time of delay across the network [4]. *Loss* (often higher when delay is higher) refers to lost network packets as a result of signal degradation over a network medium, as well as rejected packets and congestion at a given network node. Delay and loss cause a DIS to suffer from a lack of consistency between remote participants, jittery movement of various entities, and a general loss of accuracy in the simulation. Consequently, MOGs are inherently more difficult to design and produce than a traditional locally played video game: the distributed nature of the former

entails finding solutions to many architectural problems irrelevant for the latter. 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.

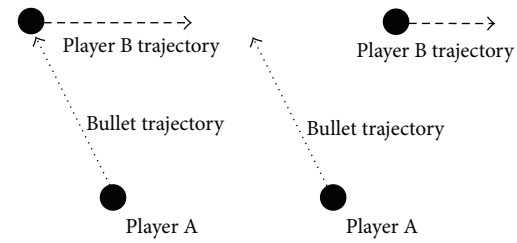
Thus, the main objective when designing the architecture of a networked video game is to maximize the user's playing experience by minimizing the appearance of the adverse effects of the network during play. When a network message (packet) is sent, there is a time delay called *lag* between the sending of the packet and the reception of the packet. Late or lost packet transmission has the effect of objects in a scene being rendered at out-of-date or incorrect locations. If objects are simply rendered at their latest known position, their movement is, as a result, jittery and sporadic. This is because they are being drawn at a location where they actually are not, and this looks unnatural.

Dead reckoning algorithms predict where an object should be based on past information. They can be used to estimate a rendering position more accurate to the true path of the object. This ensures that once the player receives the true position of the object, the positional jump to the correct location is either nonexistent or much smaller, creating the illusion that this object is behaving normally.

Lag compensation techniques are not restricted to MOGs but in fact apply to any distributed interactive simulation (DIS) application. DISs are used by military, space exploration, and medical organizations amongst others. In such contexts, improving the "user experience" ultimately entails improving the quality of such applications.

The key idea behind dead reckoning is that predicting the position of an object makes it unnecessary to receive an update for that object's motion every time it moves. Such updates are required only when there is a change in the motion. This allows for a greater degree of network lag and loss and lowers the number of update messages that are required to be sent over the network.

Traditional prediction schemes predict player position by assuming that each player moves with a constant force or velocity. Because player movement is rarely linear in nature, using linear prediction cannot maintain an accurate result. However, few of the dead reckoning methods that have been proposed focus on improving prediction accuracy by introducing new methods of predicting the path of a player. The "Interest Scheme" presented in [5] is one such innovative approach. It specifically focuses on improving prediction accuracy in a 2D tank game. The key contribution of the "Interest Scheme" is that it does so by assuming that a player's surrounding objects will have some anticipative effect on the player's path. An important restriction however is that a tank cannot "strafe" to the left and right of the forward vector but has to rotate to change direction. In this paper, we instead consider traditional team-based 2D action games (e.g., first-person, third-person, or top-down shooters) wherein players can move freely in all directions, making a player's movement highly unpredictable, and thus highly prone to inaccuracies. We propose a prediction scheme that takes *user play patterns* into account. In order to determine such patterns, we first implemented a 2D top-down multiplayer online game titled



(a) Time N at client (b) Time N + latency at server

Figure 1: Inconsistency without time warping.

"Ethereal," which serves as our test environment. Ethereal is a 2D multiplayer competitive game of 2 opposing teams in which players have freedom to move in all directions without gravity. A key facet of Ethereal is that it records not only all keyboard and mouse input of all players, but also all game world variables (such as game object and item positioning, world geometry information, and game events). We then conducted multiple play testing sessions, each involving numerous experienced players. From observing these players playing Ethereal and from subsequently analyzing half of the collected large dataset, we identified a set of typical player behaviours (i.e., play patterns). We used these patterns to create a new dead reckoning algorithm (and its associated parameters) called *EKB* (for *Experience knows best*). Another key facet of Ethereal is its ability to play back the recorded input data while simulating different network conditions. This allowed us to use the other half of our dataset to compare, under different network conditions, different versions of our path prediction algorithm with two well-known dead reckoning algorithms.

In the rest of this paper, we first discuss existing work on path prediction in the next section. Then, we introduce in Section 3 our initial EKB algorithm. In Section 4, we discuss two enhancements to this algorithm. Then, in Section 5, we present our experimental framework and compare our three versions of the EKB algorithm. Our experiments comparing the different versions of EKB with two well-known dead reckoning algorithms are summarized in Section 6. Finally, the generalization of our results, as well as other future work, is briefly discussed in the last section of the paper.

2. Related Work

2.1. Effects of Delay, Jitter, and Loss on Users. In [6], qualitative studies were conducted to determine the effects of adverse network states on the player. Participants were asked to comment on the quality of play at different levels of lag and jitter. Figures 1 and 2 of that paper show the mean opinion score (MOS) versus the amount of lag (ping) and jitter, respectively. Their findings clearly show that higher quantities of lag and jitter are correlated with a lower player experience.

In [7], the mean scores of players (the players' performance based on kills made and deaths suffered) were studied in *Unreal Tournament 2003* (a typical first person shooter video game). Through a series of 20 different scenarios of lag,

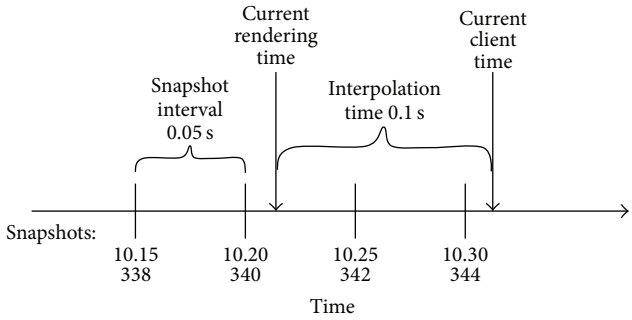


Figure 2: Object interpolation [17, 18].

with different players experiencing different amounts of lag, it was shown that high lag has a substantial negative effect on player performance. These findings are outlined in Figure 1 of that paper: it shows the scores of players unimpaired by bad network conditions versus those players experiencing bad network conditions.

A player's score in a shooting based game is a common metric used when measuring the effects of unfavourable network conditions on the player experience. Findings have been consistent that a higher degree of network loss, delay, and/or jitter results in fewer successful shots or kills made and a lower player score [6, 7, 9, 10]. For example, Aggarwal et al. [9] ran tests in a fast-paced tank shooting game called *BZFlag*, Wattimena et al. [6] gathered data from the popular shooting game *Quake IV*, and Ishibashi et al. [11] developed a distributed version of *Quake III* to test in. The performance of a player in these types of video games is based highly on reflexes and instant user input, and as a result, even fraction of second delays in the network can affect player performance. But these metrics should only be considered for a select genre or type of game. For example, in [12], running tests in the popular real-time strategy (RTS) PC Game *Warcraft III*, it was found that latency of up to 3 seconds has only marginal effects on the performances of players. This is a result of the strategic nature of the RTS genre, wherein strategic planning (as opposed to split second decision making) is more important to good performance. However, under high network lag or loss scenarios, a player's *perception* of the quality of the game can be hindered. As shown in [6], the adverse effects of the network will yield a perception of poor gaming quality. This is a result of either sporadic or jumpy positioning of game objects, or of a delay between the issuing a command for an action and the execution of that action.

2.2. Consistency. Action games use a client-server architecture to keep client-side computation to a minimum and allow for a scalable amount of players within the game world. This architecture also minimizes cheating by maintaining the game server as the single authority on all game events. In a client-server model action game, the server is the authority on game events and sends the clients game events, including the actions of other players. Client machines take input from the user and send this information to the server.

A MOG must achieve the illusion that players are playing in the same game world, when in reality they are geographically in different locations. In fact, in a MOG, players do not technically play in the same game space and are all only receiving a best-guess approximation of the game world. Given that the server is the authority on all game events, the clients must perform some form of time synchronization with the server. There are many ways to do this, each method with their own strengths and weaknesses. A discussion of these methods lies outside the focus of this paper. In summary, these distributed time synchronization techniques involve time-stamps and/or estimating lag times between machines. The time synchronization method used in our simulations is similar to the method described by Simpson in [13]: to ensure time is still synchronized, a client periodically sends a packet to the server time-stamped with the current time of the client. The server, on reception of this, immediately time-stamps the packet with its own current time and sends it back to the client. The client, from this, can determine how long it took the packet to get to the server and back again (round trip time or RTT) because of the time stamp. The client can also determine the exact time of the server, assuming it takes the same amount of time to get to and from the server (RTT/2). From here, the client will adjust its time delta between simulation ticks until its time matches the server. The client adjusts its time over several updates of the simulation because a large time jump all at once would cause objects in the scene to jump as well. If the time discrepancy is smoothed out over several frames, then there is no time-jump and the movements of the player are perceived as normal. Furthermore, because out-of-date packets and events are not important and we only need to know about the latest information regarding any given object, it is possible to employ proactive queue management techniques to drop old obsolete events in the face of arriving fresher event packets [14].

Local lag as proposed in [10, 15] refers to a network lag hiding technique wherein a delay is introduced between when an input is given and when its execution takes place. This hides lag and improves simulation accuracy because it effectively allows some time for the input packet to reach the server and subsequently to reach the other clients. Local lag allows all parties involved to receive a given event before its execution. Without local lag, in order to stay perfectly time-synchronized, a client or server would have to simulate the object or event forward to the current time, as it will have been received after its execution was supposed to take place. While this method is very effective at ensuring the time-synchronization of distributed events, it introduces a delay between when a player issues a command and when it is executed. This can be a problem, depending on what type of input the player gives. A player is more likely to notice delay regarding player movement or mouse movement input than delay regarding firing or shooting. The common method in most popular action games, for this reason, is to introduce a delay when dealing with firing or shooting, but to have no delay in regard to player movement or mouse movement input. For this reason, in our research simulation, we decided to employ no local lag for player movement but to introduce

a small amount of delay for weapons firing. This allows us to benefit from local lag, without causing the annoyance of having player movement delay.

In [10], Liang and Boustead go on to propose a method to further reduce the negative effects of lag. Since a packet, due to jitter and different lag between players, can arrive at odd intervals, and even out of order, events need to be sorted in such a way to maintain temporal accuracy. To account for this, the common method used in the video game industry is a technique called time warp. Time warp refers to the "rewinding" of the simulation to execute events at the appropriate time. This ensures that events happen the way they are supposed to, as well ensuring consistency between different parties geographically. In [16], a variation of a time-warp system is 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 roll back to a previous state.

As illustrated in Figure 1, assume player A has a network delay of 150 ms to the server and player A shoots a bullet at player B. According to player A's view (left side of Figure 1, Figure 1(a)), it looks like the bullet hit player B. But since the network message took 150 ms to reach the server, according to the lag of player B, the hit is not registered because player B has moved out of the way in the time it took for the message to arrive at the server (right side of Figure 1, Figure 1(b)). Time warp ensures this does not occur, by rewinding player A to its position 150 ms ago when the bullet was fired to check for the collision (that is, to make the server side looks like the left side of this figure Figure 1(a)). The simulation testbed used for our research includes this time warping technique, to help secure against the ill effects of lag.

2.3. Object Interpolation. A common solution to reduce the jittery movement of rendered objects, as outlined by the popular video game development community *Valve*, is to draw game objects in the past, allowing the receiver to smoothly interpolate positional data between two recently received packets [17, 18]. 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 Figure 2. Without information ahead of when rendering occurs, we can at best draw it at its current known position, which, as mentioned before, yields jittery rendering. As long as the rewind time (or interpolation time) is greater than lag, it should be possible to interpolate the position of the object. Interpolation times are traditionally a constant value or equal to the amount of lag to the server at any given time. The method used in our work is a constant interpolation value of 100 ms, as is used in the popular action game *Half-Life* 2 [17, 18]. It is a value sufficient enough to cover a large percentage of the lag or loss that will occur.

2.4. Dead Reckoning. As previously mentioned, the main goal of network compensation methods is to minimize the perceived influence of adverse network conditions on the player. Dead reckoning is a widely used method to achieve this goal. Dead reckoning is defined as any process to deduce the approximate current position of an object based on past information. In a MOG, this is done so that during high lag and loss conditions in the network, the client can approximate an object's position more accurately. That is, when data is lost or lost beyond what interpolation can solve, the current position of an object needs to be predicted. Without prediction, an object would only be rendered at the latest known position, causing discrepancies in simulation state and great jitter in object motion. When relying on dead reckoning, an assumption is made about the nature of game objects, such as adherence to certain forces or likelihoods. With small amounts of lag and loss, dead reckoning does a great job concealing the fact that there was missing information about an object.

The most common and traditional method of dead reckoning involves doing a linear projection of information received from the server about this object. An IEEE standard dead reckoning formula [3] is given by

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

where P, Δt, P_0, V_0, and A_0 represent the newly predicted position, elapsed time, original position, original velocity, and original acceleration, respectively. This equation works well to accurately predict an object, assuming that the object does not change direction. This method of prediction can become inaccurate after a time, especially for a player object, whose movement is very unpredictable.

When the receiver finally receives the actual current position from the server, after having predicted its position up until now, there will be some deviation between the predicted position and the actual position. This error difference is known as the *export error* [5, 8, 9]. Minimizing the *export error* has the effect of lowering the appearance of lag in the simulation.

In [9], the difference between a time-stamped and a nontime-stamped dead reckoning packet is explored. Without time-stamping a dead-reckoning packet, the receiver cannot be entirely sure when the packet was generated, and as a result, discrepancies between the sender and receiver's view of the world will exist. Time synchronization between players and time-stamping dead reckoning packets means that the receiving user can execute a received packet in proper order and in the exact same conditions as they were when generated. It is widely acknowledged that time synchronization, while adding network traffic, greatly improves simulation accuracy and reduces the export error [9, 10, 19–22].

As previously mentioned, traditional prediction schemes forecast a player's position by assuming each player moves using constant force or velocity. However, because player movement is rarely linear in nature, using linear prediction fails to maintain an accurate result. Furthermore, Wolf and Pantel explore and discuss the suitability of different prediction methods within the context of different types

of video games [23]. They conclude that some prediction schemes are better suited to some types of games than to others. More specifically, they look at five traditional prediction schemes: constant velocity, constant acceleration, constant input position, constant input velocity, and constant input acceleration. Each prediction scheme is compared to the others in the context of a sports game, a racing game, and an action game. As a result of the evaluation of these different prediction methods in each game, these authors demonstrate that different prediction schemes are better suited to different types of games. For example, it is shown that predicting with a constant input velocity is best suited to sports games; a constant input acceleration is best for action games; predicting with constant acceleration is best suited to racing games, and for action games, constant velocity and constant input position predictions also offer a relatively low prediction error.

Among existing dead reckoning methods, few focus on improving prediction accuracy via genuinely new (i.e., nontraditional) methods for predicting the path of a player. We discuss below some of these innovative approaches.

Traditionally, dead reckoning algorithms dictate that the server should send a positional update to clients when an object strays from its predicted path by some threshold. Thus, a dead reckoning algorithm that successfully improves path prediction does not only minimize the appearance of lag but also minimizes network traffic as well. Duncan and Gracanin [24] propose a method, called the Pre-Reckoning scheme, that sends an update just before it is anticipated that an object will exceed some threshold. To anticipate a threshold change, the angle between the current movement and the last movement is analyzed. If this angle is large enough, it is assumed that the threshold will be crossed very soon, and a dead reckoning packet is sent. The Pre-Reckoning algorithm yields better results when variability in player movement is low.

Cai et al. [25] present an autoadaptive dead reckoning algorithm that uses a dynamic threshold to control the extrapolation errors in order to reduce the number of update packets. The results suggest a considerable reduction in (the number of) update packets without sacrificing accuracy in extrapolation. While having a dynamic threshold for predicting objects does result in less data needing to be sent over the network, it does not eliminate the requirement for increasingly accurate prediction schemes. A dynamic threshold allows farther away objects to not require a high a degree of accuracy, but regardless, closer objects still need to be predicted accurately. Furthermore, the method outlined in [25] assumes a perspective view on the world, such that farther away objects are smaller and less visible. However, in a 2D video game, in which an orthographic view is utilized, all objects in view are of normal size, and therefore almost all of the objects are of interest to the user.

Work has also been done in using neural networks to enhance the accuracy of dead reckoning [26, 27]. In [26], McCoy et al. propose an approach that requires each controlling host to rely on a bank of neural network predictors trained to predict future changes in an object's velocity. Conversely, the approach proposed by Hakiri et al. in [27] is based on a fuzzy inference system trained by a learning algorithm derived from neural networks. This method does reduce network loads. While these methods have been shown to improve performance of dead reckoning, they impose extra computation on each host prior to the launching of a game and, more importantly, ultimately depend on extensive training. That is, the statistical nature of such predictors entails they must learn from very large datasets. Our proposed solution rests on the notion of *play patterns*. Machine learning could have been used to learn such play patterns from the large datasets we have gathered, but we have relied on our ability to initially recognize such patterns manually. Thus, we will not discuss further, in the context of this paper, techniques that require statistical learning.

Delaney et al. [28] describe a hybrid predictive technique that chooses either the deterministic dead reckoning model or a statistically based model. The claim of these authors is that their approach results in a more accurate representation of the movement of an entity and a consequent reduction in the number of packets that must be communicated to track that movement remotely. The statistical model rests on repeatedly observing players race to a same goal location in order to determine the most common path used. In turn, this path is used to predict the path of a player towards the same goal location. The difficulty with such an approach is that it rests on the notion of shared goal locations, which is not readily applicable to most genres of games. However, the idea of a hybrid approach to path prediction is an interesting one to which we will return later.

Finally, Li et al. propose a method called the "Interest Scheme" [5, 8] for predicting the location of a player-controlled object. That approach shows an increased accuracy of path prediction beyond traditional dead reckoning models specifically in a 2D tank game, with levels of lag up to 3000 ms. The strength of the Interest Scheme lies in the way it uses the surrounding entities of a given player-controlled entity to better predict what actions the user will take. The method works on the assumption that a player's directional input is affected by its surroundings, such as items and enemy players. Due to the introduction of an extra computational burden, especially when network conditions are adequate for play, a hybrid method is introduced into the "Interest Scheme." This method involves using traditional dead reckoning algorithms up until a fixed threshold of prediction time. However, "Interest Scheme" is designed for one very specific type of game. Thus, as previously mentioned, the success of the "Interest Scheme" is not reproducible in a traditional team-based action game.

While all the prediction methods referred to above are capable of predicting player movement relatively accurate, more elaborate methods should be considered to handle high amounts of lag. This is required by the nondeterministic manner in which players typically move in any given video game. Once there is a high amount of network delay, traditional methods of dead reckoning become too inaccurate, and the *export error* starts to become too large. Ultimately players start to notice a loss of consistency [6, 7, 18, 23, 29]. Some work, in particular the Interest Scheme outlined by Li et al. [5, 8],

has been done from this standpoint. The algorithms that we will now introduce are in the same vein. Methodologically, our proposed solutions will be compared with the IEEE standard dead reckoning algorithm [3] (hereafter referred to as TDM for "traditional dead reckoning method") and the "Interest Scheme" (hereafter IS) algorithm [5, 8].

3. The EKB Algorithm

In this section, we introduce our proposed method of prediction algorithm: *experience knows best* (EKB). We start by describing our movement prediction algorithm, which rests on the combined use of different velocities. Next, each of these velocities is discussed. We then describe some enhancements to our algorithm, followed by a discussion of its parameter space. In order to clarify the algorithm description, hereafter we use *TPlayer* to refer to the target player for prediction. The *last known position* is the latest position data that was received over the network. The *last known position time (LKPT)* refers to the time stamp associated with the *last known position*. The *last known velocity* is the velocity associated with the LKPT.

3.1. Combination of Velocities. Our approach involves predicting a player's position by predicting the potential behaviours that a player may adopt. To do so, using half of the data collected during the play sessions of Ethereal, we identified behaviours that are assumed to affect the player's next movement. These behaviours each take the form of a velocity that is exerted on the players, affecting where they will be located next. These behaviour velocities are applied at different strength levels depending on what is occurring in the game from the point of view of the player at hand. These velocities are based on the positions and states of other objects in the scene. Velocities are applied as either an attraction or repulsion towards or away from a given position in space. The magnitude of these velocities depends on several factors such as the distance to the object and the strength or weakness of the player.

The following velocities are employed in our work: the follow velocity, the bravery velocity, and the alignment velocity. They will be explained at length in the next subsection. Here we first describe how they are combined to act on a player.

Each velocity takes into account other players in the game world in order to determine direction and magnitude. They do so only if a given player is within a specified static distance threshold. In our current experiments, we set this distance to the size of the screen. Any player outside of such a region of interest is not considered in the computing of a behaviour velocity.

In order to simplify how the velocities interact with each other, we separate player behaviour into two categories: *in battle* behaviours and *out of battle* behaviours. When the player is *in battle*, the player's position is calculated by combining the follow and the align velocities. When *out of battle*, the player's position is calculated by combining the follow and the align velocities. Let us elaborate.

Whether the *TPlayer* is *in battle* or not is chosen as a simple distance check to the closest enemy, as outlined in (4). If there exists an enemy within the battle threshold W, then the player is said to be *in battle*. Equation (2) calculates the distance from the current player position ($\vec{C}$) to a given enemy player ($\vec{P}_e$). For our results, we used a threshold of $W = 800$ as this seemed to accurately represent when a player was engaged in combat or not in the context of our game. Consider

(1) **if** the player is in battle **then**
(2) $\vec{V}_r = \left(\vec{V}_{\text{follow}} \times q\right) + \left(\vec{V}_{\text{bravery}} \times (1 - q)\right)$
(3) **else**
(4) $\vec{V}_r = \left(\vec{V}_{\text{follow}} \times r\right) + \left(\vec{V}_{\text{align}} \times (1 - r)\right)$
(5) **end if**

Algorithm 1: Apply velocities.

$$D_{e_i} = \left|\vec{P}_{e_i} - \vec{C}\right| \tag{2}$$

$$closestEnemyDist = \min\left\{D_{e_1}, D_{e_2}, \ldots, D_{e_n}\right\} \tag{3}$$

$$InBattle = \begin{cases} \text{true} & \text{if } closestEnemyDist \leq W \\ \text{false} & \text{otherwise.} \end{cases} \tag{4}$$

Algorithm 1 shows how the velocities are handled. $\vec{V}_r$ is the final resultant velocity that is used to predict a player's position. Coefficients q and r are static values that are less than 1 and greater than 0 (the values of q and r will be explained shortly). They dictate how much of each velocity is used.

In summary, we first separate a player's behaviour into two states: *in battle* and *out of battle*. We then exert different velocities based on a player's current state. Finally, these velocities are combined into a resultant velocity $\vec{V}_r$ as outlined in Algorithm 1 to calculate the player's predicted position $\vec{C}_{\text{pred}}$ at the next simulation tick from this player's current position $\vec{C}$ (as shown in (5)). Consider

$$\vec{C}_{\text{pred}} = \vec{C} + \vec{V}_r. \tag{5}$$

We use $q = 0.5$ and $r = 0.6$. This is the result of trial and error tests to see what works best.

Finally, for convenience, we now give in Table 1 an explanation of each parameter that is used in the descriptions of our algorithms. The Nomenclature section lists lists these parameters, as well as all variables used in our algorithms.

3.2. Main Theoretical Component: Velocities. We now elaborate on each of the proposed velocities.

3.2.1. Follow. The follow velocity arises from our observation that the player moves towards friendly players and the player groups with these friendly players (e.g., other teammates). It is computed by taking the average position of all friendly players within a specified radius and having the player move towards that location. Furthermore, the speed of differentiation of this velocity does not depend on the distance of other

Table 1: Parameter space.

Parameter	Value	Description
W	800	Static distance threshold to differentiate between a player *in battle* or *out of battle*
k	0.4	Coefficient used in (14) to determine how much smaller we scale the strength of the friendly team
u	200	Maximum distance a player will aim to run towards or away from the friend epicenter depending on how strong each team is
l	0.1	Coefficient used to modify m (introduced below and defined in Nomenclature) so that it is in the correct range
R	60	Upper bound on m that ensures that there is always some transition that occurs from the old velocity $\vec{V}_0$ to the new velocity
q	0.5	Sets how much of each follow and bravery velocity is used to create the resultant movement velocity if in battle
r	0.6	Sets how much of each follow and alignment velocity is used to create the resultant movement velocity if out of battle
h	40	Threshold angle used to determine the angle above which the change in velocity needs to be before a change in direction is registered
x	350	Minimum threshold of lag below which the EKB method is always used

players, but it is instead always set to the maximum speed of the *TPlayer*. From our observations, the follow velocity is the most important behaviour velocity to exert on a player. This is because, in multiplayer online games, a player's actions generally proceed from a team-based strategy:

$$\vec{E}_f = \frac{\sum_{i=1}^{i=n_f} \vec{P}_{f,i}}{n_f} \tag{6}$$

$$\vec{V}_{\text{follow}} = \frac{\vec{E}_f - \vec{C}}{\left|\vec{E}_f - \vec{C}\right|} \times S. \tag{7}$$

Equation (6) calculates the averaged position of all friendly (f) players within a given threshold, where $\vec{P}_f$ is the position of a friend and n_f is the number of players of that type within the predefined region of interest (ROI). Equation (7) represents the velocity from the *TPlayer* current position $\vec{C}$ to the average position of all friendly players. S is the maximum speed of the *TPlayer*. The follow velocity is illustrated in Figure 3.

To gather friendly and enemy player data, we impose a maximum distance that a player can be before it is not considered in any calculations. The game window is 1280 pixels wide and 960 pixels tall, and so the maximum distance for considering players is 1280 pixels in the x-axis and 960 in the y-axis.

3.2.2. Align. The alignment velocity arises from a player's tendency to travel in a group with friendly players. The

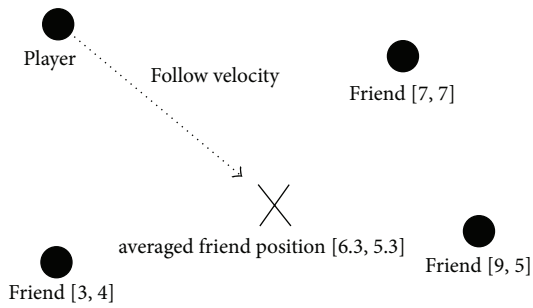


Figure 3: Follow velocity.

alignment velocity takes into account all friendly players' velocities within a specified radius. The closer a friendly player is, the more weight it carries in affecting the direction of the alignment velocity. The alignment velocity's magnitude is a function of how many friendly players are within the specified radius and how close they are. Consider

$$D_f = \begin{cases} D_{f\text{MAX}} & \text{if } D_f \geq D_{f\text{MAX}} \\ D_{f\text{MIN}} & \text{if } D_f \leq D_{f\text{MIN}} \end{cases} \tag{8}$$

$$\vec{D}_{\text{align}} = \sum_{i=1}^{i=n_f} \left(\vec{V}_{f,i} \times \left(1 - \frac{D_{f,i} - D_{f,i_{\text{MIN}}}}{D_{f,i_{\text{MAX}}} - D_{f,i_{\text{MIN}}}} \right) \right) \tag{9}$$

$$\vec{V}_{\text{align}} = \begin{cases} \vec{D}_{\text{align}} & \text{if } \left|\vec{D}_{\text{align}}\right| \leq S \\ \dfrac{\vec{D}_{\text{align}}}{\left|\vec{D}_{\text{align}}\right|} \times S & \text{otherwise.} \end{cases} \tag{10}$$

Equations (9) and (10) outline the alignment velocity. $D_f, \vec{V}_f, D_{f\text{MIN}}$ and $D_{f\text{MAX}}$ represent the distance to the friendly player, the velocity of the friendly player, the minimum distance to consider for friendly players, and the maximum distance to consider for friendly players, respectively. If the result of (9) is greater than the player's speed S, then $\vec{V}_{\text{align}}$ is set to S. $D_{f\text{MIN}}$ and $D_{f\text{MAX}}$ are each a predefined threshold. $D_{f\text{MIN}} = 60$ pixels because a player's diameter is approximately this amount, and a player within this distance is likely not be distinguished by the *TPlayer*. $D_{f\text{MAX}} = 1000$ pixels because the friendly player in question is certainly visible at this distance. Though a player may be visible up to 1600 pixels (the diagonal distance of the screen space), an object outside of 1000 pixels is unlikely to affect the alignment of the *TPlayer*.

3.2.3. Bravery. The bravery velocity arises from the observed behaviour of a player's tendency to fall back when outnumbered by the enemy and the tendency to advance on the enemy while winning. To obtain the bravery velocity, the total strength of all nearby friends and the total strength of all nearby enemies are calculated. The strength of each team is calculated by adding up the health and ammo values of each player. The relative strength of the friendly army versus the enemy army determines the direction of the resulting velocity. If the friendly army is stronger, the bravery velocity is

positive, namely, towards the enemy. Otherwise, it is negative, consequently, away from the enemy forces. The higher the magnitude of the velocity is, the farther the *TPlayer* will move away or towards the enemy:

$$\vec{E}_e = \frac{\sum_{i=1}^{i=n_e} \vec{P}_{e,i}}{n_e} \qquad \vec{E}_f = \frac{\sum_{i=1}^{i=n_f} \vec{P}_{f,i}}{n_f} \tag{11}$$

$$I_p = \frac{H_p}{\mathrm{MH}_p} + \frac{A_p}{\mathrm{MA}_p} \tag{12}$$

$$Z_f = \sum_{i=1}^{i=n_f} I_{f,i} \qquad Z_e = \sum_{i=1}^{i=n_e} I_{e,i} \tag{13}$$

$$\vec{D}_{\text{bravery}} = \left(\vec{E}_f + \left(\frac{\left(\vec{E}_e - \vec{E}_f\right)}{\left|\left(\vec{E}_e - \vec{E}_f\right)\right|}\right) \times \left(u \times \frac{\left(kZ_f + I_c\right) - Z_e}{\max\left(\left(kZ_f + I_c\right), Z_e\right)}\right)\right) - \vec{C} \tag{14}$$

$$\vec{V}_{\text{bravery}} = \frac{\vec{D}_{\text{bravery}}}{\left|\vec{D}_{\text{bravery}}\right|} \times S. \tag{15}$$

In (12), I_p is the influence of given player (whether a friendly player, enemy player, or the current player) in terms of its strength. H, MH, A, and MA are the health, maximum health, ammo value, and maximum ammo of the player, respectively. This influence value is then combined into either the enemy or friendly team influence value, depending on which team the *TPlayer* is, represented by Z_f or Z_e in (13). Z_f and Z_e are each made up of all the players on the given team that are within a predefined threshold. $\vec{D}_{\text{bravery}}$ in (14) is the direction vector used for the bravery velocity. u is a coefficient, that is, the maximum distance that a player will run away or towards the enemy and k is a coefficient that modifies the strength of the friendly influence. This is to model the fact that a player will consider his/her own strength over his/her allies' strength in a combat situation. The *TPlayer* is either moving towards or away from the enemy, in relation to the averaged friend position. This is illustrated in Figure 4. Equation (15) is the actual velocity used for bravery.

k is a coefficient used in (14) to determine how much smaller we scale the strength of the friendly team. This is done for two reasons. First, a player will tend to consider his own strength when in combat and will not adjust her behaviour if their friends are strong or weak. Second, since enemy players are more often far away than friendly players, they often fall outside the maximum distance for considering players. Thus, many enemy players that the player may be aware of are not considered in the strength calculations because they are simply too far away. This is easily adjusted by using k. In our simulations, we set $k = 0.4$. We found through trial and error that this yielded the best results and behaviour that best reflected reality. u is the maximum distance a player will aim to run towards or away from the friend epicenter depending on how strong each team is. We set u to 200 pixels, as, through trial and error, this is the value that resulted in the best prediction accuracy.

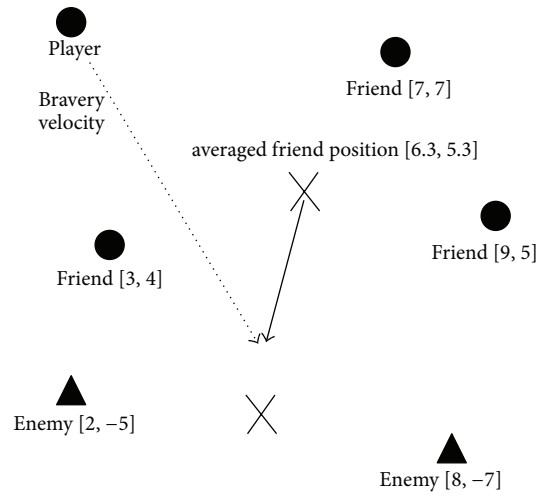


Figure 4: Bravery velocity when friendly team is stronger.

4. Algorithm Enhancements

4.1. Smooth Transition. In Algorithm 1, $\vec{V}_r$ is the final resultant velocity that is used to predict the player's position. After extensive experiments, we notice the following: if the player's velocity is immediately set to $\vec{V}_r$, the result is most often an inaccurate account of the player's movement. This is due to ignoring the player's *last known velocity*. The *last known velocity* is the last velocity information received in the last received position packet. (We use "position packet" synonymously with "dead reckoning packet," which refers to the information received from the server describing player information.) To alleviate this, we perform a smooth transition of the player from the *last known velocity* to the one determined by the combined velocities (see (16)). $\vec{V}_j$ is the velocity that should be used as the velocity of the player and j is the number of updates that have occurred since the last known velocity $\vec{V}_0$. The more updates that have passed since the last known velocity was received (i.e., the larger the value of j), the larger the value of $\vec{V}_r$ is and the smaller the value $\vec{V}_0$ is. Once j reaches the size of m (discussed below), then we exclusively use $\vec{V}_r$ to determine $\vec{V}_j$ as follows:

$$\vec{V}_j = \begin{cases} \frac{m-j}{m}\vec{V}_0 + \frac{j}{m}\left(\vec{V}_r\right) & \text{if } j \le m \\ \vec{V}_r & \text{otherwise} \end{cases} \tag{16}$$

$$m = \begin{cases} l \min\{closestEnemyDist, \\ \qquad closestFriendDist\} & \text{if } m \le R \\ R & \text{otherwise.} \end{cases} \tag{17}$$

The calculations for m are shown in (17). m is proportional to the distance between the player and the closer of the closest friendly or enemy player. This is due to the following

(1) **if** the desired A* end destination location has changed since the last update **then**
(2) recalculate an A* path to the desired location for $\vec{V}_{\text{follow}}$ and $\vec{V}_{\text{bravery}}$.
(3) **end if**
(4) **if** the next A* node in the path is reached **then**
(5) increment the desired node location to the next node in the A* path.
(6) **end if**
(7) Use the next desired node location to calculate the vectors for $\vec{V}_{\text{follow}}$ and $\vec{V}_{\text{bravery}}$.
(8) **if** the player is in battle **then**
(9) $\vec{V}_r = \left(\vec{V}_{\text{follow}} \times q\right) + \left(\vec{V}_{\text{bravery}} \times (1 - q)\right)$
(10) **else**
(11) $\vec{V}_r = \left(\vec{V}_{\text{follow}} \times r\right) + \left(\vec{V}_{\text{align}} \times (1 - r)\right)$
(12) **end if**

Algorithm 2: A*.

(1) **if** the amount of time since the last position packet was received (Q) is less than or equal to x ms **then**
(2) predict the player's position using the EKB
(3) **else if** the player has not changed direction since $LKPT - Q$ time **then**
(4) predict the player's position using the TDM
(5) **else**
(6) predict the player's position using the EKB
(7) **end if**

Algorithm 3: Hybrid approach.

observation: a player is more likely to react to a player that is close to it and is less likely to continue at the current velocity. l is a coefficient used to modify m so that it is in the right scale (we found that $l = 0.1$ works best). R is the upper bounds on m.

We used $l = 0.1$ because it allows for the best transition from the old velocity $\vec{V}_0$ to the new velocity calculated by EKB. We used $R = 60$. This value represents a maximum allowable time to still consider the old velocity $\vec{V}_0$ in the calculations. An R of 60 corresponds to 1 second.

4.2. Incorporation of A.* We employ the A* path finding algorithm [30] to further improve the accuracy of our initial algorithm. The use of the A* algorithm proceeds from observing a player's tendency to avoid walls and find an efficient path through space to a desired location. This ensures that the *TPlayer's* predicted path avoids wall objects and looks more realistic. The implementation of the A* path finding algorithm in our scheme involves modification to the follow and bravery velocities, whereas the alignment velocity remains the same. $\vec{V}_{\text{follow}}$ and $\vec{V}_{\text{bravery}}$ now point towards a desired position that is along the A* path, rather than pointing towards only the final destination. For $\vec{V}_{\text{follow}}$, this desired location is the average position of all nearby friendly players. For $\vec{V}_{\text{bravery}}$, this desired location is $\vec{D}_{\text{bravery}} + \vec{C}$. A shortest path to this desired location avoiding all obstacles is then calculated. Algorithm 2 outlines how A* is incorporated into our prediction scheme.

4.3. Hybrid Approach. In order to further improve the prediction accuracy and reduce the number of packets transmitted across the network, we develop a hybrid scheme as follows.

(i) Below x ms of lag, EKB is always used. This is because according to our experiment results EKB performs best under this lag range.

(ii) If a player has been moving in the same direction for less than or equal to the same amount of time as the network delay, then we assume the player will continue to move in this direction and thus we use TDM for the prediction.

(iii) Otherwise, EKB is used.

The hybrid method is detailed in Algorithm 3. We use $x = 350$, because below this threshold the EKB method performs significantly better than TDM and IS (as will be discussed later). The amount of time between the *current time* and the *LKPT* is how long we have not received a position packet from the server, that is, how long we have not known the true position of the *TPlayer*. We call this Q time. Equation (18) describes its calculation:

$$Q = currentTime - LKPT. \tag{18}$$

LKPT is the time stamp of the last received positional information received with regard to the *TPlayer.* We use Q as the amount of time before the *LKPT* to check to see if the player has changed direction. If the *TPlayer* has not changed direction since $LKPT - Q$, then it is assumed that the player will continue in this direction, and the TDM is used.

```
(1) initialize count to one
(2) initialize directionChange to false
(3) while count is less than Q do
(4)     if the angle between the velocity at t = last received and
            the velocity at t = last received minus count's is above
            a threshold angle h. then
(5)         set directionChange to true.
(6)     end if
(7)         increment count by dt.
(8) end while
```

Algorithm 4: Check player direction change.

To determine whether a player has changed direction, we use the method outlined in Algorithm 4.

h is a threshold angle used to determine the angle above which the change in velocity needs to be before a change in direction is registered. We use $h = 40$ degrees, so that if a player changes direction by 45 degrees, it is detected as a change in direction. x is the minimum threshold of lag below which the EKB method is always used.

5. Experimental Parameters and Results

5.1. Overhead Introduced by Our EKB Algorithm. Our algorithm computes a player's desired position based on several calculations. The constituents of this algorithm that add complexity include the A* algorithm, as well as the calculation of (a) the follow vector, (b) the align vector, and (c) the bravery vector. Computing each of these vectors requires looping through nearby players, and therefore the time complexity is $O(n)$, where n is the number of players. The time complexity of the A* calculation depends on heuristics. It is essentially a guided breadth-first search on the tiles (or grid) of the game world. In our implementation, the complexity is $O(D^2)$, where D is the depth of the solution path. Each of these calculations is computed at most once per prediction, and there are at most n players that require prediction. So the total time complexity of algorithm EKB is $O(nD^2 + n^2)$.

5.2. Experiment Conditions. In order to collect players' input, replay the collected data (for data analysis and pattern extraction), and conduct empirical and comparative evaluations, we implemented an interactive distributed test environment. This test environment takes the form of a multiplayer online game named Ethereal. We designed the test environment so that player activities would be similar to those that would be seen in any traditional action game. Players, each on a separate computer, can make a connection to the server machine to join the distributed interactive system, wherein players can interact with each other in real-time. Once a connection is made to the server, each player chooses a team to join and can then start playing the game. In Ethereal, players assume the role of a single entity that can move in all directions on a 2D plane freely. Also, they can use the mouse to aim and shoot. Gameplay is such that there are two teams, both pitted against each other in competition. Points are awarded to a team when a player from that team kills a player from the opposing team. A team wins when it reaches a certain amount of points before the other team does. A screenshot from the game can be seen in Figure 5.

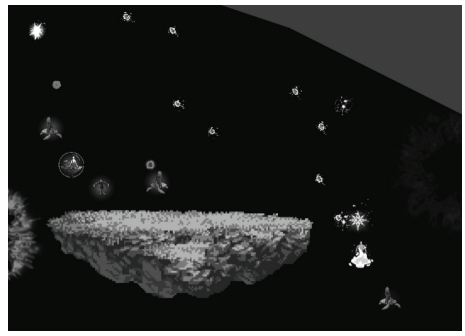


Figure 5: A snapshot from Ethereal.

To ensure adequate observations and depth of analysis, we implemented a replay system to record all events and inputs from play sessions and to conduct analysis based on this replay data. This allows us to do multiple predictions per update (i.e., tick) of the simulation on all players in the game without worrying about the analysis and collection of our results slowing down the simulation for the players during a play test. The replay system is a server-side implementation that is designed to accurately playback all game actions and events exactly as they were recorded. The replay system records all relevant information from a play session as it happens. It records player input, births time (i.e., player spawn time), death time, and state snapshots. Player input records consist of a time stamp, directional information, and mouse input at every update of the simulation. A "player snapshot" is taken every 5 seconds to ensure nothing becomes desynchronized. A player snapshot consists of all state information that is important to gameplay, namely: player position, velocity, health, and ammo. An example of such "replay raw data" can be found after the References section. The complete dataset is available at http://www.jakeagar.com/sessionInputsAndMapsAndConfigs.zip.

After recording all necessary data, the system can then play it back. This is done by spawning a given player at its

Table 2: Session parameters for algorithm evaluation.

	Number of players	Run time
Session 1	10	15 minutes
Session 2	14	35 minutes
Session 3	15	28 minutes

recorded birth time. Then, as the simulation progresses, the time stamp of the input record that is next to execute is checked. If it is time to execute this input, it gets executed. The same is done for the snapshot records. Once the death time of the player is reached, the player is killed. In this way, all recorded data is played back, such that an entire game play session can be observed after it has been recorded. Our test environment also has the capability to run and evaluate dead reckoning schemes and to measure and record metrics associated with each such scheme. At each update of the playback simulation, the replay system can check any number of criteria in order to decide if a simulated dead reckoning prediction should be made. To do so, the replay system must know the positional history of all objects. It achieves this by recording player positional data at each update of the simulation. When a dead reckoning prediction is called upon, the replay system "looks back in time" to the relevant simulated last known position. From the latter, it can predict a position at the current time, as well as measure how accurate the prediction is (by comparing it to the current position). That is, since the replay system makes only simulated dead reckoning predictions, we can easily compare such predictions against the actual player data. In particular, we can compute how far away the predicted position is from the actual position and measure the number of packets that would be sent by the current dead reckoning scheme (as will be explained shortly).

For the training and evaluation of our algorithm, we ran in total 3 play sessions of our 2D networked multiplayer game Ethereal. Table 2 outlines the number of players in and duration (i.e., "run time") of each session used for our algorithm evaluation. All participants were avid for professional video game players, male, between the ages of 17 and 30, and they were either undergraduate students, graduate students, or graduates. We used sessions 1 and 3 (25 participants) for our evaluations. We chose this many players as they constitute a representative set of set of the actual players that would be seen in an online game playing community. Furthermore, this size of sample follows suit with the sample sizes of similar studies. Finally, teams within a game were organized with respect to the configuration of the room so that players sitting close to each other were on the same team, allowing them to discuss strategy.

The play sessions were conducted in a local-area network (LAN) scenario, and thus while playing, players experienced near perfect network conditions. That is, lag was in the order of 5–50 ms at any given time. We then simulated lag on the players' input data from the replay files.

5.3. Performance Testing and Metrics. We experimented with different prediction methods, analyzing them with our replay system. We can setup any amount of delay into the simulation and test the predicted position against the actual position of any player. At the time of making a prediction, we can then measure different metrics. We measured the average export error (AEE), the number of hits, and the number of packets sent. We use these metrics to evaluate our EKB method, as well as the two other dead reckoning schemes we selected: the TDM [3] and the IS [5, 8]. We believe that these metrics accurately test and contrast the accuracy of these dead reckoning schemes. Let us elaborate.

AEE is the average distance from the predicted position and the actual position of the player for all predictions made. To calculate it, we take the median of all export errors at fixed intervals of time (e.g., 300 ms, 600 ms, etc.) to determine the general accuracy of an algorithm. The calculation of AEE is shown in (19). $\vec{P}_t$ and $\vec{E}_t$ are, respectively, the actual position of the player and the predicted position of the player at time t. Here, n is the total number of predictions made throughout the lifetime of all replay data. The AEE is a measurement of how similar the estimated behaviour of a player is related to the true and actual movement of a player. The AEE is the best metric in determining the accuracy of any given prediction method:

$$\text{AEE} = \frac{\sum_{i=0}^{i=n} \left| \vec{P}_{t,i} - \vec{E}_{t,i} \right|}{n}. \tag{19}$$

The next metric we take is hits. A *hit* is defined as when the predicted location is within a specific threshold (measured in pixels) of the actual position. It is taken at specific points in time. This metric measures how many times the prediction scheme has predicted a position correctly. Whenever the position of the player is accurately predicted as a hit, this means that the play experience is improved for the player because it means that the estimated player position will not have to be corrected to the actual player position.

We also measure the number of packets that need to be transmitted over the network during each session of play. This is done by assuming that a packet only needs to be sent when the predicted position of the player is more than a certain static threshold g distance away from the actual position of the player. We use a threshold of $g = 45$ because it is less than the width of a player. Measuring the number of packets sent is done in order to evaluate network traffic (which, ideally, should be as low as possible). Network bandwidth is often a performance bottleneck for distributed interactive simulations. When there are less packets that need to be sent per object, the game can replicate more objects over the network. We present packets sent as a single integer, representing the total number of packets that have been sent throughout all play sessions that were conducted.

To ensure we test our prediction scheme against other prediction schemes in identical situations of play, we make a prediction and measure its accuracy as often as possible. Instead of simulating realistic lag onto the players at the moment of play, we test our prediction scheme at every update of the simulation during replay playback (of the recorded replay data). Furthermore, at each tick of the simulation, we test lag at varying degrees of network delay.

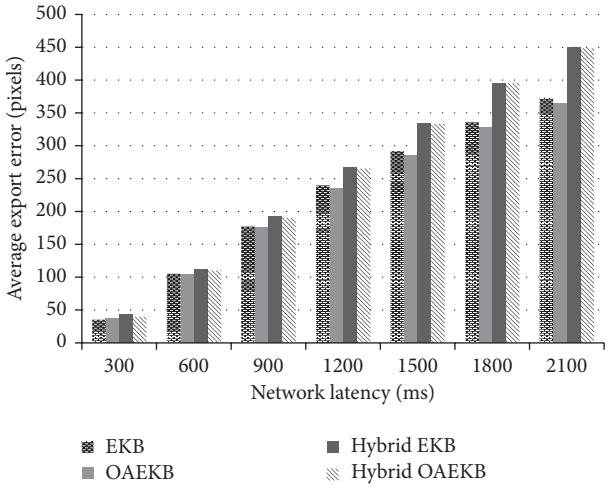


Figure 6: AEE of EKB and its variations.

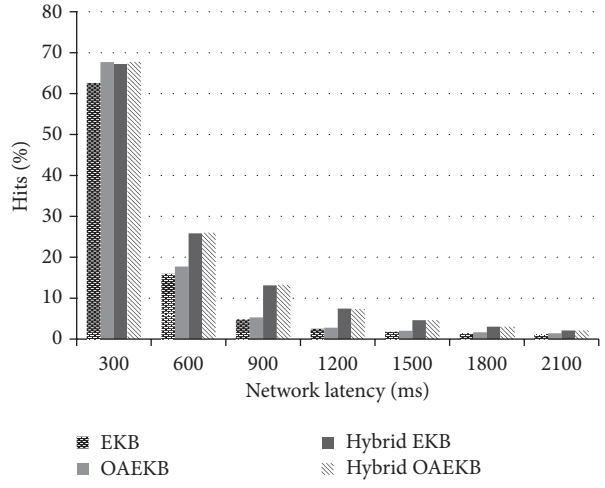


Figure 7: Hit percentages, EKB at threshold $g = 45$.

Table 3: AEE of EKB and its variations.

	EKB	OAEKB	Hybrid EKB	Hybrid OAEKB
300 ms	37.8	36.9	44.1	42.3
600 ms	106.9	104.8	112.6	112.5
900 ms	178.9	175.2	192.2	192.2
1200 ms	241.6	235.8	267.1	266.9
1500 ms	293.6	285.7	334.7	334.0
1800 ms	337.3	327.8	395.2	394.1
2100 ms	374.9	364.5	449.4	448.5

Table 4: Hit percentages and EKB at threshold $g = 45$.

	EKB	OAEKB	Hybrid EKB	Hybrid OAEKB
300 ms	62.6%	67.7%	67.2%	67.7%
600 ms	15.9%	17.7%	25.8%	26.0%
900 ms	4.8%	5.3%	13.1%	13.3%
1200 ms	2.5%	2.8%	7.5%	7.4%
1500 ms	1.7%	2.0%	4.6%	4.5%
1800 ms	1.4%	1.7%	3.0%	3.0%
2100 ms	1.2%	1.4%	2.1%	2.1%

At each tick, we simulate lag at 7 different levels, 300 ms apart. We test prediction at 300 ms of delay, 600 ms of delay, and so on up to 2100 ms of delay. We do 7 predictions per update of the simulation for each and every player in the game. We use nonrandom, deterministic lag intervals to ensure that analysis and comparison are done absolutely fair, such that nothing is left to chance. We test against lag up to approximately 2 seconds because lag scenarios above 2 seconds of network lag are not likely in today's online video games. Combining testing prediction at every update of the simulation with an all-encompassing approach to lag simulation allows the testing of every possible game scenario at every level of lag with each dead reckoning scheme. Finally, we remark that jitter is not addressed in this work.

5.4. Experimental Results and Their Analysis. In this subsection, the results of our experiments with our different versions of EKB are described. The discussion is organized with respect to each metric we consider: AEE, then number of hits, and finally packets sent.

5.4.1. Average Export Error. The average export error (AEE) is the discrepancy between the actual location of the player and the predicted location of the player (in pixels). Figure 6 and Table 3 compare the EKB method against the obstacle avoiding EKB (OAEKB) method that uses A*, the hybrid EKB, and the hybrid OAEKB method. These figures show the effect of introducing the A* algorithm on the AEE. While the introduction of the path finding algorithm A* does not improve prediction accuracy drastically, it does result in more realistic predicted motion of the player, as a player controlled by a human user will tend to avoid obstacles to achieve their objectives. AEE increases as prediction time increases. This is because as the prediction time increases, so does the time since the last player's true position was known. This suggests that the EKB performs better under higher latency conditions.

From Figure 6, we can compare the OAEKB and hybrid OAEKB Recall that movement in the hybrid method can use TDM or EKB, depending on how linear a player's past behaviour is. We introduced the hybrid method to increase the number of hits and decrease the number of packets sent. While it does this effectively, it also has the effect of lowering the accuracy of the AEE.

5.4.2. Number of Hits. Figure 7 shows the number of hits at different levels of network latency for EKB, OAEKB, hybrid EKB, and hybrid OAEKB. Table 4 shows the same data. The number of hits without using the hybrid method was much lower than with, and this is why we introduced the hybrid method. The fact that the hybrid method has the effect of improving the number of total hits while at the same time lowering AEE accuracy demonstrates that although the position of the player is often accurately predicted, it does

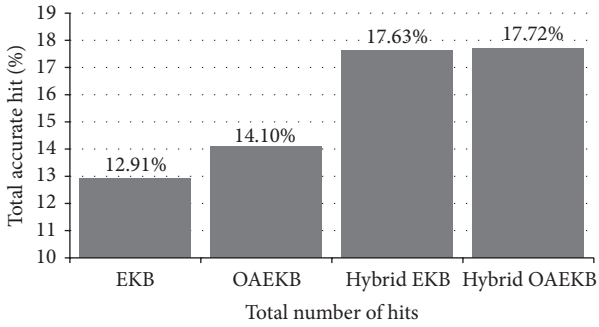


Figure 8: Total hit percentages, EKB at threshold $g = 45$.

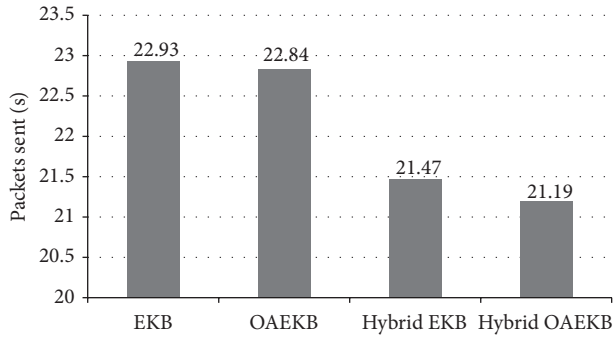


Figure 9: Packets per second, EKB at threshold $g = 45$.

not mean that the position of the player is overall better approximated. The hybrid method improves the number of hits because it allows for frequent predictions of the player moving in a straight line, during which time it is exactly accurate in predicting the player (as long as the player is moving linearly). Without the hybrid scheme, the EKB does approximate the player relatively accurate but does not produce as many exact predictions of player position.

As can be seen in Figure 8, the hybrid OAEKB method improves on the number of hits made by EKB. It can also be seen that while introducing A* into EKB did not provide relevant improvements to AEE, it increased the total number of accurate predictions made (as shown in Figure 8). This further provides evidence that A* allows our algorithm to more accurately predict the movement of the player.

5.4.3. Number of Packets Sent. We also measure the number of dead reckoning packets (or position packets) that need to be sent through the network. The "packets sent" metric is produced not by using fixed intervals of lag like for the AEE and hit metrics. Instead, as explained earlier, we work under the assumption that a packet is sent only when the predicted position is a certain threshold h distance away from the actual position of the player. To calculate packets sent per second, we add up all the packets that would be sent in this way for each dead reckoning algorithm and then divide it by total time (of the play session) such that we obtain a measure of packets sent per second.

Figure 9 shows the number of packets required to be sent by the different versions of EKB. We observe that the hybrid method reduces the number of packets that need to be sent.

In summary, our experiments show that the OAEKB is best suited to produce the lowest AEE, and the hybrid OAEKB method is best suited for increasing the number of hits and reducing network traffic. In the next section, we will compare the different versions of EKB with TDM [3] and IS [5, 8].

6. Comparative Evaluation

6.1. Average Export Error. Figure 10 and Table 5 show the AEE introduced by the TDM, IS, EKB, OAEKB, hybrid EKB, and hybrid OAEKB algorithms. From this figure we can see that the OAEKB considerably lowers the overall prediction

Table 5: AEE comparison with high latency.

	TDM [3]	IS [5, 8]	EKB
300 ms	35.6	38.2	37.8
600 ms	109.1	110.0	106.9
900 ms	198.7	191.7	178.9
1200 ms	293.3	271.8	241.6
1500 ms	388.8	345.7	293.6
1800 ms	484.2	413.8	337.3
2100 ms	579.5	477.7	374.9
	OAEKB	Hybrid EKB	Hybrid OAEKB
300 ms	36.9	44.1	42.3
600 ms	104.8	112.6	112.5
900 ms	175.2	192.2	192.2
1200 ms	235.8	267.1	266.9
1500 ms	285.7	334.7	334.0
1800 ms	327.8	395.2	394.1
2100 ms	364.5	449.4	448.5

error when predicting at large amounts of network delay. This is a consequence of the OAEKB's strong ability to approximate the position of the player. It considers various factors that would affect the player in the context of the game and uses these to predict the path of the player. The OAEKB performance at high lag is demonstrated by the slope of OAEKB's AEE-prediction time relationship decreasing as high levels of prediction time are reached, while the TDM and IS seem to take a relatively linear increase in AEE as prediction time is increased.

From Figure 10, it can be seen that while the hybrid OAEKB yields significantly worse AEE overall compared to the OAEKB results, it still outperforms the TDM and IS.

6.2. Hits Percentages. We then measure the number of times each algorithm makes an accurate *hit*. Figure 11 lays out the number of hits that were recorded at each given time interval. Table 6 shows the same data as Figure 11. Hybrid EKB and Hybrid OAEKB performed relatively well, especially at very low amounts of lag (300 ms). The TDM is a close second in terms of number of hits to the hybrid OAEKB. The strength of the TDM is its ability to predict an object moving in a linear direction. So while hybrid OAEKB can better predict

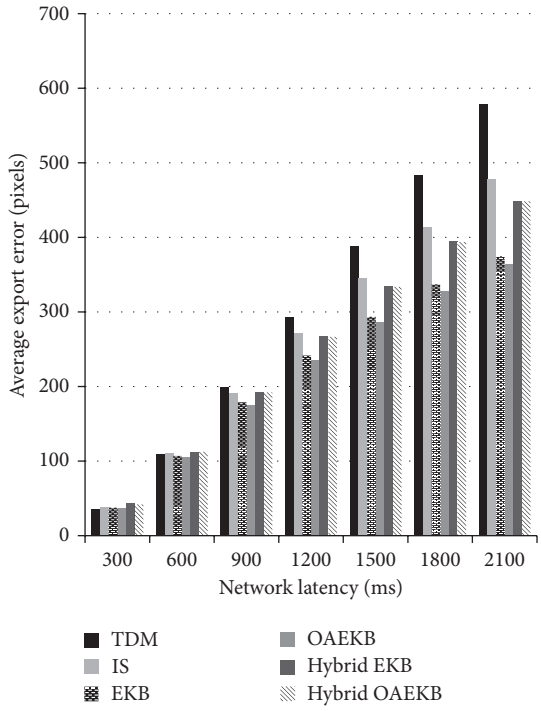


Figure 10: Average export error comparison.

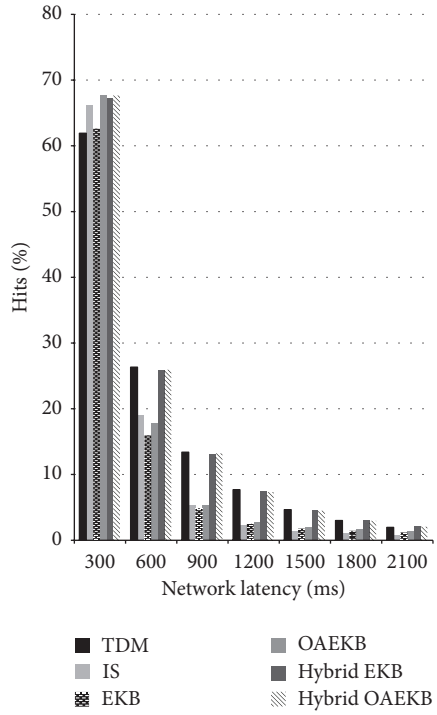


Figure 11: Hit percentages at threshold $g = 45$.

Table 6: Hit percentages at threshold $g = 45$.

	TDM [3]	IS [5, 8]	EKB
300 ms	61.9%	66.1%	62.6%
600 ms	26.3%	19.0%	15.9%
900 ms	13.4%	5.3%	4.8%
1200 ms	7.6%	2.3%	2.5%
1500 ms	4.6%	1.4%	1.7%
1800 ms	2.9%	1.1%	1.4%
2100 ms	1.9%	0.8%	1.2%
	OAEKB	Hybrid EKB	Hybrid OAEKB
300 ms	67.7%	67.2%	67.7%
600 ms	17.7%	25.8%	26.0%
900 ms	5.3%	13.1%	13.3%
1200 ms	2.8%	7.5%	7.4%
1500 ms	2.0%	4.6%	4.5%
1800 ms	1.7%	3.0%	3.0%
2100 ms	1.4%	2.1%	2.1%

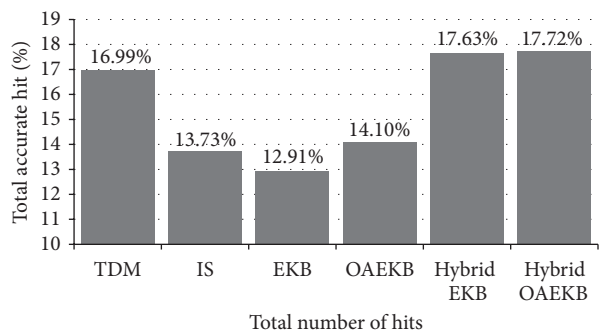


Figure 12: Total number of accurate hits at threshold $g = 45$.

a player's overall behaviour, the TDM can better predict a player's behaviour when moving in the same direction (which players will often do).

The total number of hits as shown in Figure 12 is calculated by taking a mean calculation of all the previous hit percentage results. Hybrid OAEKB and Hybrid EKB performed best, followed very closely by the TDM. This is because the TDM predicts very accurately when a player moves in a single direction, which is often the case. The IS performed poorly because it failed to account for the case when a player would move in a straight line for an extended period of time. The IS assumes that the player will maintain the original velocity $\vec{V}_0$ for only a relatively short amount of time.

6.3. Number of Packets Sent. We also measure the number of packets that need to be transmitted over the network during each time interval we monitor. Figure 13 shows hybrid OAEKB improves prediction accuracy over TDM and IS while sending as few packets as TDM. The difference in packets sent between TDM and hybrid OAEKB is negligible, whereas IS needs to send a significantly higher number of packets.

It is worth noting that, compared to TDM, EKB significantly reduces AEE but yields relatively poor improvements/results with respect to the number of hits and the number of packets sent. This is because the AEE is a measure of overall accuracy, while hits and packets sent are concerned

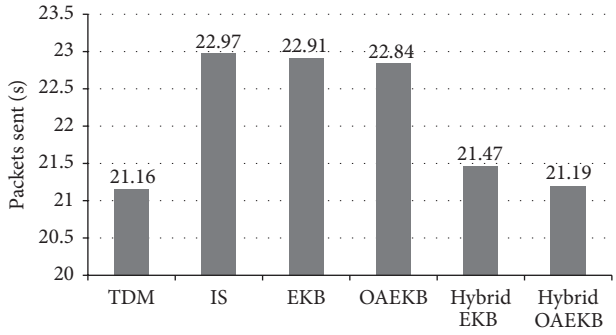


Figure 13: Comparison of packets sent per second.

with making a binary observation of whether the prediction was within a small threshold h or not. The strength of the TDM is in its ability to perfectly predict the player, as long as this player is moving in a constant direction. The strength of hybrid OAEKB is that it selectively chooses to predict the player moving in a constant direction or to a defined behaviour. In this way, AEE can be improved without adversely affecting the number of hits made or the number of packets sent.

Finally, contrary to IS, hybrid OAEKB improves the prediction accuracy without increasing the network traffic. And while the number of packets sent is not improved over TDM, hybrid OAEKB does result in more realistic player movement on account of the significant improvements made to the AEE.

7. Conclusion

In light of the limitations observed in existing work on dead reckoning, we have proposed here a new prediction scheme that relies on user play patterns. Our research takes place in the context of a 2D top-down multiplayer online game we have developed. In such typical team-based action game, a player's movement is highly unpredictable and is therefore highly prone to prediction inaccuracies, thus emphasizing the need for a better prediction method. We have evaluated our three proposed algorithms against the IEEE standard dead reckoning algorithm [3] and the recent "Interest Scheme" algorithm [5, 8]. In particular, we have shown that while our hybrid OAEKB is best suited to improve hits and packets sent, its use also reduces the improvement to AEE we get from EKB and OAEKB, while still outperforming the other two dead reckoning algorithms we considered.

We emphasize that both the proposed play patterns and algorithms proceed from the data collected for a specific 2D multiplayer competitive game opposing two teams and in which players have freedom to move in all directions without gravity. We observe that traditional 3D first-person shooter games, such as *Unreal Tournament*, *Quake III*, and *Counter-Strike*, employ mostly 2-dimensional player motion (aside from jumping). Thus, our algorithms could be well-suited for these kinds of games as well. However, our algorithms specifically assume there are teams of players. Thus, the generalization of our results to other types of games still constitutes future work. Furthermore, we remark that we still adopt a simple approach to prediction. More complex schemes that (in the spirit of swarm artificial intelligence) would enable a player to simultaneously use multiple strategies (taking into account many possible scenarios) are beyond the scope of our current research.

Additionally, we would like to explore the potential for our method to take into account past decisions and play styles of the player to increase prediction accuracy. Our method assumes all players are interested at the same rate in following teammates and reacting to enemy players. In reality, many players possess differing play styles and skill levels and will react differently in any given situation. Utilizing machine learning or data analytics with this in mind could provide greatly improved results. Just by playing, a user could see automatically personalized improvements in prediction.

We would also like to experiment with taking into account the play styles associated with different weapons or character classes in order to increase prediction accuracy. Many games have different ways to interact in the world such as different weapons and characters to use that can greatly change how a player interacts with teammates and enemies. Building a framework that would allow knowledge of such factors could increase prediction accuracy even further.

We would like to improve upon how the desired position is calculated for the player. While A* made improvements to the method, these improvements were quite minimal. This may be caused by the fact that the position that is currently being used to predict where the player would like to be is fundamentally incorrect. More work should be done in finding exactly where the player would like to be at any given time.

Another difficult question we hope to eventually address pertains to the possibility that users could possibly adapt their input based on the network latency they are currently experiencing.

Finally, work needs to be done in order to reduce the number of packets that are sent over the network. Though each of the 3 versions of the EKB results in improved AEE and hits over the two other dead reckoning methods we considered, it does so with no improvement to network traffic.

Appendix

Box 1 is an example of the raw gameplay data recorded from a play session. Map, Us, U, UI, N, P, Pos, Sy, Alignment, Si, Is, I, T and B, X, and Y represent the current map, the list of all users, information for a single user, the user's identification number, the user's name, starting information of a user's player, starting position, character type, team, spawn time of the player, list of input data, individual input entry, time of input, character representing directional and mouse input, the X position of the mouse cursor, and the Y position of the mouse cursor respectively, respectively.

```
<Map>ProvingGrounds.xml</Map>
<Us>
<U>
<Ui>0</Ui>
<N>Unnamed</N>
<P>
<Pos>
<X> −2560</X>
<Y>1024</Y>
</Pos>
<N>Unnamed</N>
<Sy>4</Sy>
<Alignment>0</Alignment>
<Si>3621</Si>
</P>
<Is>
<I>
<T>3723</T>
<B>0</B>
</I>
<I>
<T>3740</T>
<B>0</B>
</I>
<I>
<T>3757</T>
<B>0</B>
</I>
<I>
<T>3774</T>
<B>8</B>
<X> −1827.08</X>
<Y>1147.52</Y>
</I>
<I>
<T>3791</T>
<B>8</B>
</I>
<I>
<T>3808</T>
<B>8</B>
</I>
```

Box 1

Nomenclature

W : Static distance threshold to differentiate between a player *in battle* or *out of battle*

k: Coefficient used in (14) to determine how much smaller we scale the strength of the friendly team

u: Maximum distance a player will aim to run towards or away from the friend epicenter depending on how strong each team is

l: Coefficient used to modify m (explained below) so that it is in the correct range

R: Upper bound on m that ensures that there is always some transition that occurs from the old velocity $\vec{V}_0$ to the new velocity

q: Sets how much of each velocity (follow and bravery) is used to create the resultant movement velocity if *in battle*

r: Sets how much of each velocity (follow and align) is used to create the resultant movement velocity if out of battle

h: Threshold angle used to determine the angle above which the change in velocity needs to be before a change in direction is registered

x: Minimum threshold of lag below which the EKB method is always used

TPlayer: The player whose position is being predicted (target player)

$\vec{C}$: The current position of the *TPlayer*

S: The top movement speed of the *TPlayer*

$\vec{E}_f, \vec{E}_e$: Vector representing the epicenter of either friend or enemy players

$\vec{V}_{\text{follow}}$: Movement vector used to predict movement towards an epicenter of players

$\vec{D}_f, \vec{D}_e$: Distance from the *TPlayer* to a given friendly (or enemy) player

$\vec{P}$: Position of a given player

$\vec{D}_{\text{align}}$: Initial movement vector used to predict align behaviour before setting the magnitude limit S

$\vec{V}_{\text{align}}$: Movement vector used to predict movement aligning to friendly players' movement

H, A, MH, MA: Current health, current ammo, max health, and max ammo of a given player

$\vec{I}_p$: Influence of a given player

$\vec{Z}_f, \vec{Z}_e$: Sum of player influences (either friend or enemy)

$\vec{D}_{\text{bravery}}$: Initial movement vector used to predict bravery behaviour before setting magnitude to S

$\vec{V}_{\text{bravery}}$: Movement vector used to predict bravery behaviour

closestEnemyDist: Distance to closest enemy player

InBattle: True or false value based on in the player is said to be *in battle*

$\vec{V}_r$: Combination of other movement vectors used to predict the new position of the *TPlayer*

$\vec{V}_j$: Prediction movement vector used to smooth the *last known velocity* into $\vec{V}_r$

j: Number of updates since the *last known velocity*

m: Number of updates since the *last known velocity* to fully transition from $\vec{V}_j$ to $\vec{V}_r$

Q: Amount of time that has passed the *last known velocity*.

Conflict of Interests

The authors declare that there is no conflict of interest regarding the publication of this paper.

Acknowledgment

The authors gratefully acknowledge financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC) under Grant no. 371977-2009 RGPIN.

References

[1] Entertainment Software Association, *Essential Facts about the Computer and Video Game Industry*, 2013.

[2] Entertainment Software Association, *Essential Facts about the Computer and Video Game Industry*, 2011.

[3] "IEEE standard for distributed interactive simulation application protocols," IEEE Standards Board, 1995.

[4] D. E. Comer, *Computer Networks and Internets*, Prentice Hall, 2008.

[5] S. Li, C. Chen, and L. Li, "A new method for path prediction in network games," *Computers in Entertainment*, vol. 5, no. 4, article 8, 12 pages, 2008.

[6] A. F. Wattimena, R. E. Kooij, J. M. van Vugt, and O. K. Ahmed, "Predicting the perceived quality of a first person shooter: the Quake IV G-model," in *Proceedings of the 5th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames '06)*, Singapore, October 2006.

[7] P. Quax, P. Monsieurs, W. Lamotte, D. de Vleeschauwer, and N. Degrande, "Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game," in *Proceedings of the 3rd ACM SIGCOMM Workshop on Network and System Support for Games (NetGames '04)*, pp. 152–156, Portland, Ore, USA, September 2004.

[8] C. Chen and S. Li, "Interest scheme: a new method for path prediction," in *Proceedings of the 5th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames '06)*, Singapore, October 2006.

[9] S. Aggarwal, H. Banavar, A. Khandelwal, S. Mukherjee, and S. Rangarajan, "Accuracy in dead-reckoning based distributed multi-player games," in *Proceedings of the 3rd ACM SIGCOMM Workshop on Network and System Support for Games (NetGames '04)*, pp. 161–165, Portland, Ore, USA, September 2004.

[10] D. Liang and P. Boustead, "Using local lag and timewarp to improve performance for real life multi-player online games," in *Proceedings of the 5th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames '06)*, Singapore, October 2006.

[11] Y. Ishibashi, Y. Hashimoto, T. Ikedo, and S. Sugawara, "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 (NetGames '07)*, pp. 75–80, Melbourne, Australia, September 2007.

[12] M. Claypool, "The effect of latency on user performance in real-time strategy games," in *Proceedings of the 2rd Workshop on Network and System Support for Games (NetGames '03)*, pp. 3–14, Redwood City, Calif, USA, May 2003.

[13] Z. B. Simpson, "A stream-based time synchronization technique for networked computer games," 2010, http://www.mine-cont-rl.com/zack/timesync/timesync.html.

[14] C. E. Palazzi, S. Ferretti, S. Cacciaguerra, and M. Roccetti, "Interactivity-loss avoidance in event delivery synchronization for mirrored game architectures," *IEEE Transactions on Multimedia*, vol. 8, no. 4, pp. 874–879, 2006.

[15] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg, "Local-lag and timewarp: providing consistency for replicated continuous applications," *IEEE Transactions on Multimedia*, vol. 6, no. 1, pp. 47–57, 2004.

[16] E. Cronin, B. Filstrup, A. R. Kurc, and S. Jamin, "An efficient synchronization mechanism for mirrored game architectures," in *Proceedings of the 1st Workshop on Network and System Support for Games (NetGames '02)*, pp. 67–73, 2002.

[17] Valve, "Source Multiplayer Networking.," Valve Developer Community, 2011, https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking.

[18] Y. W. Bernier, "Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization," Valve Developer Community, 2009, https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization#Footnotes.

[19] Y.-J. Lin, K. Guo, and S. Paul, "Sync-MS: synchronized messaging service for real-time multi-player," in *Proceedings of the 10th IEEE International Conference on Network Protocol (ICNP '02)*, pp. 1092–1648, 2002.

[20] Y. Zhang, L. Chen, and G. Chen, "Globally synchronized dead-reckoning with local lag for continuous distributed multiplayer games," in *Proceedings of the 5th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames '06)*, Singapore, October 2006.

[21] F. W. B. Li, L. W. F. Li, and R. W. H. Lau, "Supporting continuous consistency in multiplayer online games," in *Proceedings of the 12th Annual ACM International Conference on Multimedia (MULTIMEDIA '04)*, pp. 388–391, New York, NY, USA, October 2004.

[22] S. Ferretti, "Interactivity maintenance for event synchronization in massive multiplayer online games," Technical Report UBLCS, Bologna, Italy, 2005.

[23] L. C. Wolf and L. Pantel, "On the suitability of dead reckoning schemes for games," in *Proceedings of the 1st Workshop on Network and System Support for Games (NetGames '02)*, pp. 79–84, 2002.

[24] T. P. Duncan and D. Gracanin, "Pre-reckoning algorithm for distributed virtual environments," in *Proceedings of the Winter Simulation Conference*, vol. 2, pp. 1086–1093, December 2003.

[25] W. Cai, F. B. S. Lee, and L. Chen, "An auto-adaptive dead reckoning algorithm for distributed interactive simulation," in *Proceedings of the 13th Workshop on Parallel and Distriuted Simulation (PADS '99)*, pp. 82–89, May 1999.

[26] A. McCoy, T. Ward, S. McLoone, and D. Delaney, "Multistep-ahead neural-network predictors for network traffic reduction in distributed interactive applications," *ACM Transactions on Modeling and Computer Simulation*, vol. 17, no. 4, article 16, Article ID 1276929, 2007.

[27] A. Hakiri, P. Berthou, and T. Gayraud, "QoS-enabled ANFIS Dead Reckoning algorithm for distributed interactive simulation," in *Proceedings of the 14th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications (DS-RT '10)*, pp. 33–42, Fairfax, Va, USA, October 2010.

[28] D. Delaney, T. Ward, and S. McLoone, "On reducing entity state update packets in distributed interactive simulations using a hybrid model," in *Proceedings of the 21st IASTED International*

Multi-Conference on Applied Informatics, pp. 833–838, February 2003.

[29] W. Palant, C. Griwodz, and P. Halvorsen, "Evaluating dead reckoning variations with a multi-player game simulator," in *Proceedings of the 16th Annual International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '06)*, pp. 4:1–4:6, Newport, RI, USA, May 2006.

[30] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 1995.