# A Time Rewind System for Multiplayer Games

Hesam Rahimi, Saurabh Ratti, Ali Asghar Nazari Shirehjini, and Shervin Shirmohammadi

***Abstract*—In video gaming, time rewind is an engaging feature that gives players the chance to recover from their missteps. Manipulating the logical flow of events adds a completely new dimension and unpredictability to gameplay. Yet time rewind is rarely seen beyond single player games, due to technical challenges and logical dilemmas that require complex designs. In this paper, we propose a game engine architecture that achieves the concept of time rewind for networked multiplayer games, explaining our design choices and showing with a proof-of-concept game that our approach works over various network latencies.**

***Index Terms*—Computer games, multiplayer games, time rewind.**

## I. Introduction

Manipulating the temporal dimension for a single player has been done in games such as Prince of Persia: The Sands of Time (time rewind and slowdown) [1], Max Payne (time slowdown) [2], and Braid (time rewind, warp, and relative time progression) [3]. However, combining time rewind with multiplayer gaming is a new and underexplored concept in game engineering. Certain logistic and engineering challenges arise when considering time rewind in a multiplayer setting, as multiple players have the ability to initiate time rewind simultaneously. In order to execute comprehensible time manipulation, an effect delineation mechanism is required to determine where players' individual time rewind influence occurs. Due to the complexity of this, never have we seen support for environment-wide time rewind in multiplayer gaming, because of many challenges: what happens to entities once their histories are rewound? How do entities affected by time rewind interact with each other? What happens when a time-rewound entity physically collides with a non-rewound entity and they both occupy the same physical space? How do we mitigate network latency and its effect on time rewind synchronization among players? This article explains how our system addresses these challenges and how we translate the concept of time rewind and playability in that setting into design modules to build a game allowing networked players to experience time rewind.

Manuscript received May 16, 2013; revised July 22, 2013.

H. Rahimi was with the Distributed and Collaborative Virtual Environment Research (DISCOVER) Lab, University of Ottawa, Canada. He is now with the University of Toronto, Canada (e-mail: hrahimi@discover.uottawa.ca).

S. Ratti was with the DISCOVER Lab, University of Ottawa, Canada. He is now with the Government of Canada (e-mail: sratti@discover.uottawa.ca).

A. A. Nazari Shirehjini and S. Shirmohammadi are with the DISCOVER Lab, University of Ottawa, Canada (e-mail: anazari@discover.uottawa.ca; shervin@discover.uottawa.ca).

## II. Related Work

Many studies have been performed to determine the effect of network latency on multiplayer games offering traditional gameplay. [4] found that, in a collaborative virtual environment, jitter has a greater effect on player performance than latency. [5] has looked at latency in relation to the player's perspective: avatar games with first person perspective show a decline in player performance with latency greater than 100 milliseconds (msec), whereas third-person perspective games can potentially tolerate 500 msec lag. Omnipresent perspective games such as real-time strategy (RTS) games are more resilient to latency and jitter and work up to 1 second latency, as focus is on long term planning rather than reaction time [5], [6]. Player performance and gameplay is also affected by frame rate, and performance can benefit from frame rates of up to 60 frames per second (fps) in a First-Person Shooter (FPS) game [7]. While the above research works present very valuable information, they all study "normal" game-play. No work has studied the effects of latency on games with time rewind.

Combining the networked multiplayer aspect of a game with time travel capabilities is a new and underexplored field of research and development. Braid [3] is an innovative 2D platform and puzzle game with various types of time manipulations, such as rewind, warp and pause. These manipulations form the basis of puzzles within the game, in addition to being used to solve the puzzles. Braid's gameplay however, is strictly single player with no multiplayer. TimeShift [8] is a 3D FPS game that includes time manipulation abilities in a networked multiplayer mode. Rather than accomplishing environment-wide time manipulation by multiple players in parallel however, TimeShift makes use of "time bubbles" as a mechanism to limit the area of effect that time manipulation is carried out in. Achron [9] is an RTS game that features time as a traversable dimension, similar to a map's limited span of x/y/z directions. The player is able to go back and forward within a limited time window and carry out actions in the past. Effects from past actions ripple forward to the future, possibly creating time paradoxes. But emphasis is placed on managing multiple timelines per unit and resolving the associated paradoxes in their convergence, and Achron is still subject to the established thresholds of the genre. In summary, we are unaware of any work that investigates this concept beyond our initial work in [10].

## III. Proposed Architecture

Our proposed architecture and its subcomponents are shown in Fig. 1. The Game class updates, coordinates, and dispatches system state information to various

DOI: 10.7763/LNSE.2013.V1.81

subcomponents. It also manages and tracks all active in-game entities, such as characters and enemies. The primary components are:

1) Physics Engine (PE)

A custom frame-based, fixed-step, 2D physics engine tasked with processing entities for collisions, between both static and dynamic entities. We have used simple 2D graphics since our main challenge is the distributed multiplayer time-travel aspect, which can of course later be applied to a more visually pleasing and realistic 3D graphics engine as well.

2) Time Manipulation Module (TMM)

A logical module consisting of the Time Recorder and state related classes, highlighted in Fig. 1, for time recording and rewinding. The TMM is invoked by the PE to save and restore entities' states in order to rewind and replay their timelines independently. An entity's "state" is the minimum necessary information to replicate its location, behaviour and appearance at a given time. While TMM has been previously described in [10], a relevant summary is provided to make this article self-contained.

3) Networked Multiplayer Support

Distributed network play allows multiple remote players to play together, both in synchronized and unsynchronized modes.

Other components of our system include a video renderer, an animation processor for custom XML animations, and stubs for entity AI and game levels.

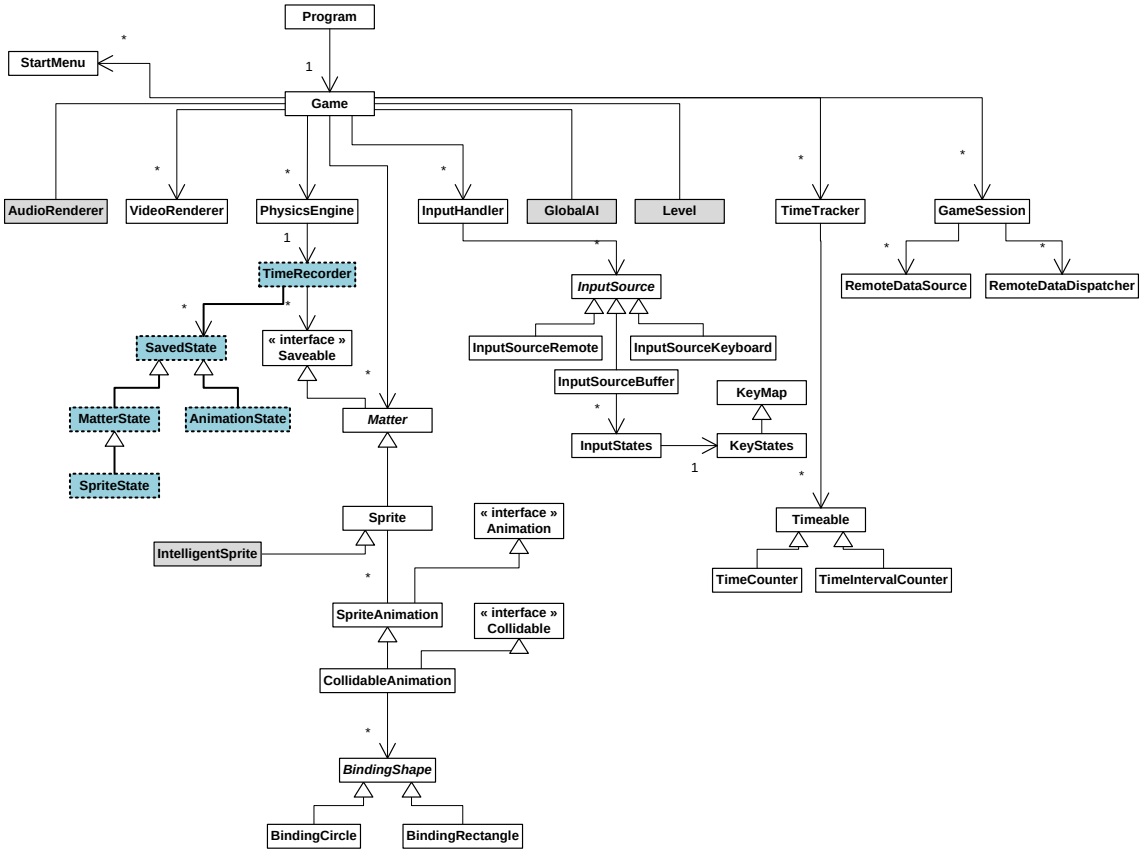



Fig. 1. UML diagram of class design, dependencies, and inheritance. Classes associated with TMM are dotted and tinted blue. Greyed classes represent stubs for components not implemented in the current incarnation.

### A. Physics Engine

Given the action based style of game-play, PE has a substantial role within the game engine. It has the responsibility of applying a set of mechanical or traditional physics laws to game entities. This includes gravity, friction, and elastic collisions with conservation of energy and momentum. The engine's updating procedure is based on a stepping system in which the Game class simply instructs the engine to advance the physical simulation state of entities by a specified amount of time, the latter used in all displacement calculations. Often, the game advances the physics state by the number of elapsed milliseconds between general game updates and, as a result, closely follows the system clock. The physics engine has two major facets: collision detection and the collision handling systems it employs.

Collision detection (CD) and its handling plays an important role when manipulating time and potentially displacing objects, hence it is important to understand how our system addresses CD. For the purposes of implementing our game, we have simplified collision detection by approximating game entities with simple geometrical shapes as bounding boxes. Mathematical formulas with the radius and side lengths are then used to calculate intersections of these shapes for each different case at every update cycle. This allows us to focus on time manipulation, but more complex CD algorithms can be used too.

The displacement formula for entities used throughout the engine is the product of an entity's velocity and the elapsed time since the last update. As such, the displacement experienced by an entity between subsequent game updates has the potential to be very large if the entity has a very high

velocity or the elapsed time between updates is lengthy.

Large elapsed time values can occur during system slowdowns, where game updates become less frequent and the time between them increases. High velocities are more frequent due to the fast-paced, real-time nature of the game. Many applied forces are large and can be repeated in short periods of time to move entities at high speeds, such as multiple punches on an enemy. If two entities on a collision course have overly large displacement values, the entities simply "warp" past each other from one update to the next, as they would collide in the time "between" updates. Since the entities never intersect during an update, the collision goes undetected and the entities essentially pass through each other.

To maintain the simplicity of the system, "line of sight" traversal algorithms are not used to solve the displacement issue; instead, a two-faceted solution is used. First, the physics system is kept on a fixed step interval, requiring the game to call the physic engine's update function every 16 msecs (approximately 60 fps). When a game update occurs after this timeout, the physics engine is invoked multiple times to match the real-time clock as closely as possible. When game updates are less than this amount of time, the physics advance is delayed and the elapsed time is carried over to the game update iteration. These specific time and frame rate values are chosen as they are well suited to the in-game velocities, in the majority of cases preventing overly large displacements. Second, a fixed speed limit is imposed for all objects as a fail-safe mechanism. In theory this contradicts the concept of elastic collisions as it results in a net loss of energy in the system, but this is reasoned as air friction imposing a maximum velocity, i.e. terminal velocity. After collisions between entities are detected, this is followed by the application of energy and momentum conservation to the object, with modifications to their direction vectors.

Collision handling also has the responsibility of dealing with illegal states. An illegal state is one where, in the context of a 2D action game, two object's physical bodies are overlapping. The distributed multiplayer module with parallel time manipulation creates a high potential for illegal physics state occurrence. The ability of objects to enter and leave the physical environment, required by the time manipulator, can create illegal states whenever these objects suddenly occlude with other entities upon re-entry. Our initial exploration determined that a preventative algorithm, which verifies the physics state for each object and can assure that the system never ventures into an illegal state, requires a large number of heuristics and highly taxed computational resources. Alternatively, a best-effort corrective algorithm is implemented successfully to handle collisions, permitting the system to enter states considered illegal for short periods of time. This technique allows entities to momentarily overlap, correcting their positions once they do so. But, because the application of elastic collisions has the effect of always directing objects towards non-overlapping positions, the number of heuristics and special cases to be treated by other interlocking modules are minimized and result in a more general and reusable system. When such momentary illegal states do happen, they are not usually perceived by the players. If they do happen to be perceived, we can justify them within the game's storyline as some sort of science fiction "time-space continuum paradox", which is not an uncommon theme in time travel/manipulation scenarios in games and movies.

### B. *Time Manipulation Module*

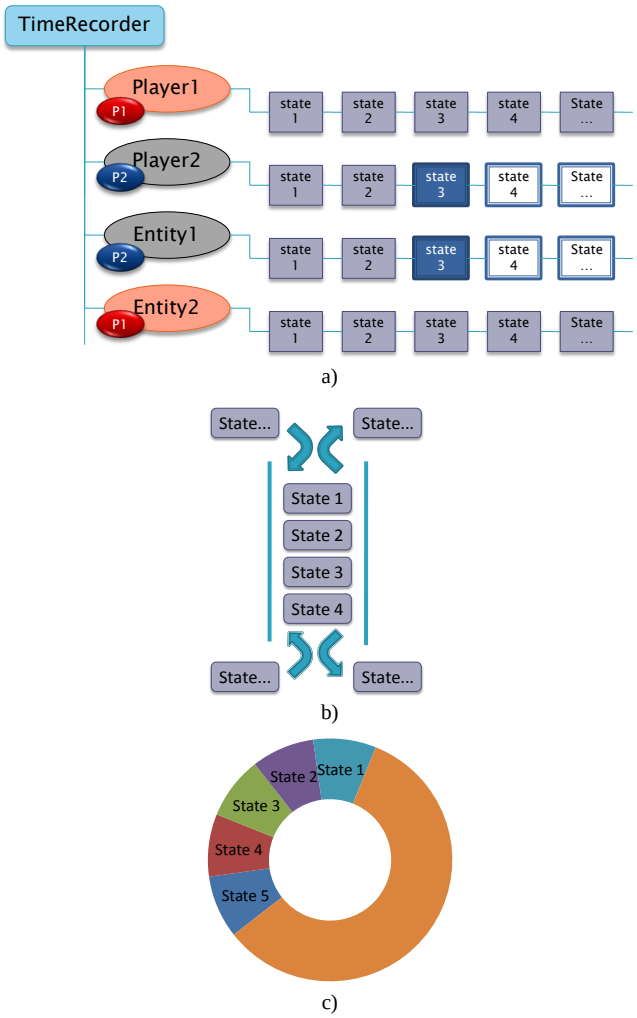



Fig. 2 a) Time Recorder structure with rewind initiated by Player 2 (P2). b) The double-ended queue (deque) data structure used by Time Recorder. c) The deque implementation internally uses a circular array.

TMM is at the heart of time rewind, and is integrated as a subcomponent into the PE. The TMM's primary function is to store all entities' past states and allow traversal of this history. States are captured at each game update cycle, another reason for the PE's fixed time stepping design. The history buffer is kept at a fixed size that limits the amount of time that can be rewound. Within the engine, it is the Time Recorder class that holds all the game entities' history buffer, as illustrated in Fig. 2 a). During gameplay, the TMM records an entity's current states as a new state which in enqueued, while old states at the end of the history buffer are dequeued. When rewinding an entity through its past actions, the Time Recorder iterates through states in reverse order like a stack. To hold the entities' states and meet the insertion/removal requirements of recording and rewinding history, each entity's history buffer in the Time Recorder is a double-ended queue (deque) data structure, as shown in Fig. 2 b). Our deque data structure is a custom implementation because it does not exist natively in the XNA framework, and it

internally uses a circular array as seen in Fig. 2 c). The circular array implementation is chosen due to its constant access time, a desirable quality since history buffers are traversed sequentially. The static length of the array is determined at creation time according to the rewind time limit imposed by the TMM. If this list becomes full, the oldest state is dequeued to maintain the preset maximum rewind time.

Entities in the game can be in one of three time modes: Normal, Rewinding or Replaying. In the Normal mode, an entity's actions are stored in its history buffer. When a player initiates time rewind, the mode of the entity is set to Rewinding, and its past states are reinstated in reverse. This mode is maintained until the command is halted or the TMM arrives at the beginning of its history buffer. The entity then goes into the Replaying mode, and the history buffer is now traversed forward so that the previously "rewound" actions are executed as they were during previous gameplay. This mode is maintained until the end of the history buffer is reached, or a new event occurs on the entity. Fig. 3 depicts the state diagram of an entity's progression through its time modes. Every update cycle, the PE invokes the TMM to determine how each game entity is analyzed for collision detection. The interactions between entities are determined by the time mode collision matrix, with the matrix used in our proof-of-concept game given in Table I. The entries within the collision matrix identifies when an entity in a given mode (a row lookup) should be compared against other entities in their specified mode (a column lookup for the current row).

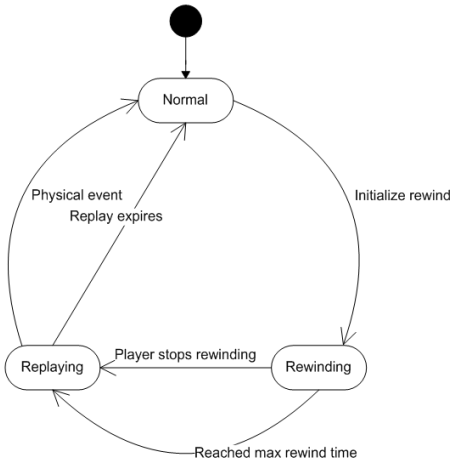



Fig. 3. Entity time mode state diagram.

TABLE I: TIME MODE COLLISION MATRIX

| ***Time Mode of Current Entity*** | ***Time Mode of Comparison Entity*** | | |
|---|---|---|---|
| | Normal | Rewinding | Replaying |
| Normal | × | | × |
| Rewinding | | | |
| Replaying | × | | |

In our collision matrix, Rewinding game entities are completely omitted from collision detection in order to allow their rewinding animations to be drawn. Replaying objects are compared exclusively with the Normal objects, to determine if collisions have occurred. If so, the Replaying objects return to a Normal state, effectively having their timelines broken from the previous history, and both game entities are processed normally by the PE. Finally, Normal entities are processed for collisions between themselves. Once collisions have been detected and then handled, the TMM records the entities' states into the history buffer, as all game entities are in valid and legal states.

### C. Networked Multiplayer Support

Multiplayer support over the network is a huge topic of its own and beyond the scope of this article. We refer the reader to [11] for a comprehensive explanation and analysis of the networking aspect of current games. In our system, networked multiplayer support is provided in either synchronized or unsynchronized modes. Synchronized gameplay is achieved by time stamping player input and ensuring that game clocks remain within temporal proximity of each other. Unsynchronized gameplay is possible because game instances are independently calculated simulations; i.e. game state is derived by applying local and remote input data to the simulation. Obviously inconsistencies are introduced when input data from a remote player is delayed due to latency, and the resulting simulation calculations differ on the screen of each player. However, our game design is such that players do not know about these inconsistencies and gameplay continues in spite of latency being present [12].

## IV. IMPLEMENTATION

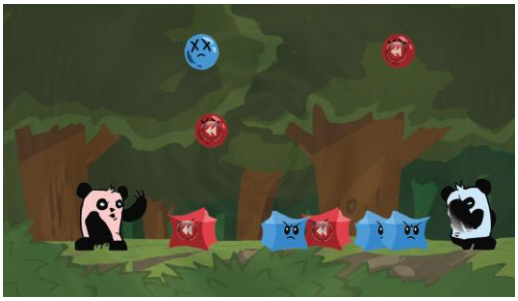


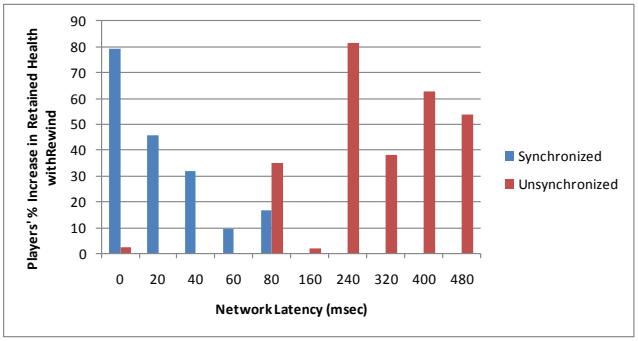
Fig. 4. Game screenshot.

We implemented our proposed engine and a specific game, collectively named FizzX [10], on top of Microsoft's XNA framework using the C# programming language for Microsoft's X-Box 360 and Windows-based PCs. As shown in Fig. 4, players represented by panda avatars cooperatively beat up enemy "slimes" and attempt to achieve a high score of the number of slimes killed. In such a setting, time rewind allows players to recover from mistakes that result in loss of health, allowing them to fight longer. As seen in the Fig, the reddish panda player on the left side has activated time rewind, indicated visually by the circular ripples in the background and arrowed sprites. To encourage players to use time rewind cooperatively and aid each other, we designed the game to have environment-wide time rewind, instead of restricting the ability to localized "bubbles". To accomplish this, we use the concept of "entity owners" as our effect-delineation mechanism, where the player to last hit an entity owns it. Owning an entity is visually representing by changing the entity's colour to match the player's colour, hence the game's association with red/blue slimes and reddish/bluish pandas, respectively. When a player initiates time rewind, only those entities owned by the player rewind through their previous states. Players work together because

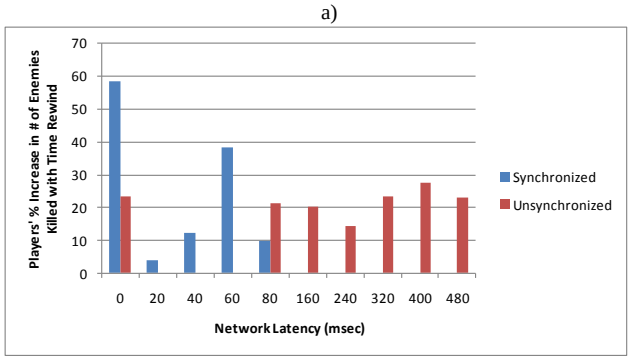
both players can kill any entity whether they own it or not. They can also rewind entities they own to help both themselves and their partner, because the rewind is environment-wide.

## V. Experiments and Results

We found that with network latencies greater than 80 milliseconds, game playability decreased sharply in synchronized mode (see Fig. 5), but we were able to achieve acceptable gameplay in unsynchronized mode with latencies as high as 480 milliseconds. The results are shown in Fig. 5.




a)




b)

Fig. 5. Effect of rewind on players' percentage increase in a) retained health b) enemies killed.

For both synchronized and unsynchronized networked gameplay, we wanted to verify that the properties of time rewind are retained; that players are able to retain a greater amount of health and kill a greater number of enemies when the time rewind ability is used. We carried out testing over a range of latencies and captured players' health and enemies killed in a cooperative fashion with and without rewind. Our test results show that in both synchronized and unsynchronized gameplay, the use of time rewind always has a positive effect on players' performance. Fig. 5 a) shows the players' percentage increase in retained health when using rewind, while Fig. 5 b) shows the players' percentage increase in the number of enemies killed when using rewind. Due to the cooperative nature of the game, players' retained health was calculated as the average retained health of both players combined, and the number of enemies killed was the sum of enemies killed by both players. In order to confirm that the observed results are not due to chance, statistical analysis was performed on the captured data to confirm the statistical significance of the differences. Paired t-tests confirmed that the difference in health retained with and without time rewind usage is statistically significant, with a p-value of 0.008 when the game is synchronized and 0.04 for unsynchronized mode. Paired t-tests also confirmed the same for enemies killed, with a p-value of 0.03 for synchronized game and 0.00023 for unsynchronized game instances.

## VI. Conclusions

The FizzX game engine architecture shows that a networked game with distributed multiplayer time manipulation over the network can be accomplished even in the presence of high network latency. This requires a physics engine that integrates with a time manipulation module responsible for entity state recording and rewind capabilities of the recorded states. The described proof-of-concept game was used to drive certain gameplay decisions and engine behaviours, but these can be changed and adapted to suit other styles of gameplay. First, the 2D style of our game could easily be changed to 3D and employ more sophisticated rendering and physics algorithms. Also as stated, the cooperative nature of the game was tied to the use of entity owners as a time rewind effect delineation mechanism. In a competitive game where players are pitted against each other, the effect of time rewind could be controlled in different ways. For example, a time rewind weapons hit on another player could force that player to go back in time, or time rewind would be limited to the physical area around a player, allowing them to only help themselves. Additionally, the time mode collision matrix can be changed to provide an alternate interaction between entities in the different time modes. Lastly, networked multiplayer synchronicity can be enhanced to provide Synchronized gameplay at greater latencies.

## References

[1] Prince of Persia: The Sands of Time, Ubisoft. [Online]. Available: http://www.ubi.com/US/Games/Info.aspx?pId=657

[2] Max Payne. Remedy Entertainment. [Online]. Available: http://remedygames.com/games/#max-payne

[3] Braid. Microsoft Game Studios. [Online]. Available: http://braid-game.com/

[4] K. S. Park and R. V. Kenyon, "Effects of network characteristics on human performance in a collaborative virtual environment," in *Proc. IEEE Virtual Reality*, Houston, Texas, 1999, pp. 104-111.

[5] M. Claypool and K. Claypool, "Latency and player actions in online games," *Communications of the ACM*, vol. 49, no. 11, pp. 40–45, November 2006.

[6] M. Claypool, "The effect of latency on user performance in real-time strategy games," *Computer Networks*, vol. 49, no. 1, pp. 52-70, September 2005.

[7] K. Claypool and M. Claypool, "On frame rate and player performance in first person shooter games," *Multimedia Systems Journal*, vol. 13, no. 1, pp. 3-17, September 2007.

[8] TimeShift. Sierra Entertainment. [Online]. Available: http://www.gamerankings.com/xbox360/929531-timeshift/index.html

[9] Achron. Hazardous Software. [Online]. Available: http://www.achrongame.com/

[10] S. Ratti, C. Towle, P. Proulx, and S. Shirmohammadi, "FizzX: multiplayer time manipulation in networked games," presented at International Workshop on Network and Systems Support for Games, Paris, France, 2009.

[11] S. Ratti, B. Hariri, and S. Shirmohammadi, "A survey of first-person shooter gaming traffic on the internet," *IEEE Internet Computing*, vol. 14, no. 5, pp. 60-69, September/October 2010.

[12] H. Rahimi, S. Ratti, A. A. Nazari, and S. Shirmohammadi, "Unsynchronized networked games: feasibility with time rewind," presented at International Workshop on Network and Systems Supports for Games, Taipei, Taiwan, 2010.

**Hesam Rahimi** was a researcher with the University of Ottawa's Distributed and Collaborative Virtual Environment Research (DISCOVER) Laboratory until 2012, where he performed research in networked games, mobile games, and adaptive game streaming. He has a Master's of Applied Science degree in electrical and computer engineering from the School of Electrical Engineering and Computer Science at the University of Ottawa, Canada, and is currently working at the University of Toronto, Canada, in the Smart Applications over Virtual Infrastructure (SAVI) research project.

**Saurabh Ratti** was a researcher with the University of Ottawa's Distributed and Collaborative Virtual Environment Research (DISCOVER) Laboratory until 2010, where he performed research in massively multiplayer online games (MMOGs), overlay networking, networked games, and distributed systems. He has a Master's of Applied Science degree in electrical and computer engineering from the School of Electrical Engineering and Computer Science at the University of Ottawa, Canada, and is currently working at the Government of Canada.

**Ali Asghar Nazari Shirehjini** received his Ph.D. degree in computer science from the Technische Universität Darmstadt, Germany in 2008, where he was with the Fraunhofer Institute for Computer Graphics from 2002 to 2008. He is currently a Visiting Research Associate at the DISCOVER Lab, University of Ottawa, Canada. Between April 2011 and October 2012 he was a PostDoc at TU Berlin and Karlsruhe Institute of Technology in Germany, and from December 2008 to April 2011 he was one of the four Vision 2010 Postdoctoral Fellows at the University of Ottawa. His research interests include ambient intelligence, context awareness, interaction with smart spaces, mobile 3-D user interfaces, rapid prototyping, context-aware systems, context-aware game streaming, and massively multiplayer online gaming.

**Shervin Shirmohammadi** received his Ph.D. degree in electrical engineering from the University of Ottawa, Canada in 2000, where he is currently a Full Professor at the School of Electrical Engineering and Computer Science. He is Co-Director of both the Distributed and Collaborative Virtual Environment Research Laboratory (DISCOVER Lab), and Multimedia Communications Research Laboratory (MCRLab), conducting research in multimedia systems and networking, specifically in gaming systems and virtual environments, video systems, and multimedia-assisted biomedical engineering. The results of his research have led to more than 200 publications, over a dozen patents and technology transfers to the private sector, and a number of awards and prizes. He is Associate Editor-in-Chief of IEEE Transactions on Instrumentation and Measurement, Associate Editor of ACM Transactions on Multimedia Computing, Communications, and Applications, and was Associate Editor of Springer's Journal of Multimedia Tools and Applications, and chairs or serves on the program committee of a number of conferences in multimedia, virtual environments, and games. Dr. Shirmohammadi is a University of Ottawa Gold Medalist, a licensed Professional Engineer in Ontario, a Senior Member of the IEEE, and a Professional Member of the ACM.