Mammoth
A Massively Multiplayer Game Research Framework

Jörg Kienzle, Clark Verbrugge, Bettina Kemme, Alexandre Denault, Michael Hawker
School of Computer Science, McGill University, Montreal, QC, Canada
{Joerg.Kienzle,Clark.Verbrugge,Bettina.Kemme}@mcgill.ca,
{Alexandre.Denault,Michael.Hawker}@mail.mcgill.ca

ABSTRACT

This paper presents Mammoth, a massively multiplayer game research framework designed for experimentation in an academic setting. Mammoth provides a modular architecture where different components, such as the network engine, the replication engine, or interest management, can easily be replaced. Subgames allow a researcher to define different game goals, for instance, in order to evaluate the effects of different team-play tactics on the game performance. Mammoth also offers a modular and flexible infrastructure for the definition of non-player characters with behavior controlled by complex artificial intelligence algorithms. This paper focuses on the Mammoth architecture, demonstrating how good design practices can be used to create a modular framework where researchers from different research domains can conduct their experiments. The effectiveness of the architecture is demonstrated by several successful research projects accomplished using the Mammoth framework.

1. INTRODUCTION

In the last decade, the video game industry has shown unparalleled growth, both in revenue and in development complexity. With the advent of the Internet, multiplayer and massively multiplayer games have become more and more popular. Compared to a traditional multiplayer game in which usually up to 16 players play a relatively short-lived game, *massively multiplayer games* (MMOGs) offer the possibility for thousands of players to play together in a persistent world. MMOG implementations face huge scalability problems since they have to handle a massive amount of connected players, presenting them with a consistent view of the world, and still providing good performance and hence, an enjoying experience.

Many academic researchers in the fields of distributed systems, distributed simulations, databases, and fault tolerance over the last 40 years have addressed scalability and consistency issues in small and large-scale distributed systems. However, serious research aiming specifically at the development of (massively) multiplayer games has only recently started. Whereas 100% data consistency and fault tolerance is required in the database world and most other domains, the situation is slightly different with computer games. The most important element in computer games is that the game experience is enjoyable, i.e. the game implementation has enough performance to allow for a smooth game play, and the game states perceived by the players are similar enough to not give an unfair advantage to any of the players.

Producing experimental results for multiplayer and especially for massively multiplayer games can be a daunting task. First, considerable infrastructure is needed to run experiments. Not only does each player usually play on a different machine, most distributed game architectures require machines with considerable processing power and high-end network connections to act as game servers. Second, in order to get realistic and statistically significant measurements, a large number of human players have to actually play the game for a long period of time. Finally, and most importantly, since the experiments are supposed to take place in a computer game setting, considerable development effort has to be spent in implementing an actual game. There are many important aspects to consider while designing a computer game, such as graphics, animation, sound, interactivity, realism, and storytelling. If any one element is neglected too much, the game's "fun" factor can be diminished.

To avoid this problem, some universities have formed industrial alliances with strategic partners, allowing them to experiment with commercial game software. Although this strategy can represent a great saving in development time and effort, initial design decisions of the game and implementation remain in the hands of the commercial partners. Also, it is not easy for the researchers to vary different parameters of the game, or even swap out entire engines, in order to compare alternative solutions. Another option for academics is to reuse a commercial game software which was publicly released as a basis for experimentation. However, most commercial game software released to the public is poorly documented. It is coded for efficiency, not good design. As a result, the implementation of different concerns is usually tightly tangled and hence hard to modify. Because of this, many researchers decide to not run realistic experiments, but instead simulate the aspects of MMOGs relevant to their specific research. The hope is, of course, that good algorithms and techniques validated using simulation also perform well in a real computer game. However, this is not necessarily true.

Our solution to this problem is Mammoth [1], a massively

ICFDG 2009, April 26 – 30, 2009, Orlando, FL, USA.

multiplayer game research framework designed for realistic experimentation in the context of multiplayer and massively multiplayer games. One of the key features of Mammoth is that its architecture is composed of loosely coupled components that can be individually replaced as necessary to suit the needs of the researcher.

The remainder of the paper is structured as follows. Section 2 presents related work in this area. Section 3 presents the Mammoth architecture and the components it is comprised of. Section 4 gives an overview of the design and programming techniques that have been used in the implementation of Mammoth. Section 5 points to research results that have already been obtained using Mammoth and the last section draws some conclusions.

2. RELATED WORK

Although not numerous, several frameworks for MMOG development have been proposed in research contexts. Primarily, these focus on demonstrating or evaluating a specific technique such as an optimized communication network. *DoIT, Lucid,* and *ATLAS*, for instance, provide relatively complete environments for game development based on improved network models [14, 17, 16]. Such designs typically aim at supplying a singular overall abstraction to the game developer, such as a basic client/server architecture design. A similar approach can be applied to many game development paradigms: the *RTF* middleware game framework emphasizes publish/subscribe for interest management, replication and migration systems [12], while *Colyseus* incorporates techniques that improve performance in FPS games by using weak(er) game consistency models [4]. Complete game frameworks for examining Artificial Intelligence in games have also been presented. Both *Stratagus* and *ORTS* provide full game facilities that accommodate various AI designs in a non-trivial game context [19, 6]. These approaches, however, are heavily focused on a specific genre (real-time strategy), and the challenges of modelling AI in that context. Other game aspects are not as easily changeable for research in different domains.

To help with game research a basic modularity of components is critical; the ease and flexibility with which various aspects can be replaced, modified, and measured is essential for research investigations from different perspectives. Fletcher et al., for example, describe *plug-replaceable* concurrency and consistency control, showing a flexible means to explore different game consistency models [10]. *NGS* allows for prototyping a variety of region-based network architectures, including P2P and client/server designs [24]. These approaches allow for rapid evaluation of different parameters and designs applied to a subset of game components. For consideration of other in-game aspects, such as realistic player movement or the impact of object visibility, a full game implementation is of course still necessary.

Of course for instruction purposes in game development and software design, a number of game frameworks have been proposed. *SAGE, Gedi,* and *DXFramework,* for instance, all supply a basic C/C++ code-base for game development, and have been used successfully for teaching game design on the Windows platform [18, 7, 23]. Many pedagogical uses of games are possible; *Alice*, developed at CMU, acts as a gentle introduction to programming for younger students by emphasizing a simple scripting interface with good graphical feedback [8]. Our approach here is less aimed at instruction and more at academic research, where multiple, major design changes and associated evaluations must be efficiently supported. In addition, it should be noted that Mammoth is not a game development engine by itself, and should not be compared to development tools such as Ogre [2].

3. MAMMOTH

Mammoth is a massively multiplayer game research framework. It was created as a collaborative project between a group of McGill professors and students in early 2005, and has evolved considerably during the last 3 years. Its goal is to provide an implementation platform for academic research related to multiplayer and massively multiplayer games in the fields of distributed systems, fault tolerance, databases, networking, concurrency. During the last 3 years, several other side projects have started using Mammoth to conduct experiments in the fields of artificial intelligence, modeling and simulation, and content generation. It should be noted that Mammoth features relatively primitive graphics and is not an appropriate platform for graphics related research.

3.1 The Mammoth Game

Like most multiplayer and massively multiplayer games, in Mammoth players take control of a game character, also called an *avatar*. A game session consists of moving around in a virtual world and interacting with the environment by executing *actions*. Basic building blocks of such actions are, e.g., moving the avatar, picking up or dropping *items*, or communicating with other players. Unlike other games, there is a fixed number of avatars in the world of Mammoth who always exist, even when players are not playing the game. A player logging into Mammoth takes control of one of the avatars in the world during the gaming session.

Items are game objects that players can manipulate. Similar to object-oriented programming, *item types* define the classes of items which exist within the Mammoth world, together with their attributes and actions that can be applied to them. Just as classes, item types can form hierarchies if the item's attributes and/or actions are related. When creating a world map, items are created by instantiating the corresponding item type and placing the instance on the map at a chosen position.

Currently there are no goals to be achieved by players in the Mammoth world. An avatar just has an inventory with a maximum carrying capacity, items have a weight and a value. We are planning to eventually evolve the Mammoth world into a "Sims"-like environment, where players have to perform certain basic actions, e.g. eating, to "survive" in the game. Additionally, subgames can be used to add purpose to the Mammoth world, as described in Section 3.2. A screenshot of the 3D Mammoth client is shown in Fig. 1.

3.2 Subgames

Actions which players perform through their avatar in the virtual world are usually motivated by the goals of the game. In highly competitive games each player performs actions strictly to his own advantage: all other players are considered enemies. Cooperative game play on the other hand can be observed in games that allow (temporary) teams of players to be created.

Since game goals considerably influence the interaction

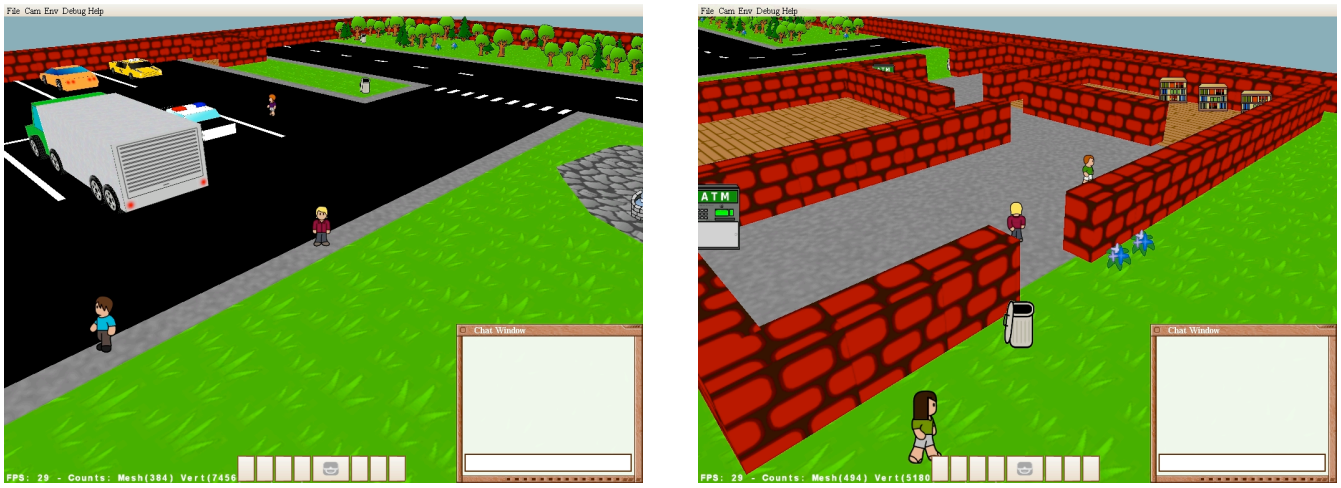

Figure 1: The Mammoth 3D Client

patterns that can be observed between players and their environment in multiplayer games, it is essential that the Mammoth framework allows a researcher to define game goals which motivate the players to behave in a way that is relevant for the research at hand. In Mammoth, game goals can be created by defining subgames.

A subgame is simply a game within a game[1]. A subgame can alter the game rules of the game in which it takes place in one of the following ways:

- Creation or deletion of new items
- Creation of new item types
- Changing the effects of existing actions
- Defining new actions
- Define a scoring system
- Define team rules

In order to allow players to play the game, a subgame definition also has to include instructions to customize the user interface, for instance to trigger game-specific actions. For more information on the subgame capabilities of Mammoth and their implementation, the interested reader is referred to [13].

The following paragraphs describe two examples of subgames implemented in Mammoth.

Find the Trophy. In *Find the Trophy*, a unique "trophy" is placed somewhere in the game world, and it is up to the players of the game to find the trophy. Whoever finds it gets a point and becomes "it." They then have a set amount of time in which to hide the trophy again somewhere else in the world. To prevent other players from following them to see where the trophy is hidden, they have an increased walking speed. Once the trophy is hidden again, the process repeats, with other players seeking out the trophy etc... In cases where the trophy is not found, whoever hid it gains another point and the trophy is randomly moved to a new location for everyone to find again.

In order to define the *Find the Trophy* subgame, a new item type had to be defined for trophies, and a single instance had to be placed in the world. The subgame also had to redefine the *pickup* and *drop* actions in order to detect when players find or hide the trophy, and increment the *speed* property of the player carrying the trophy.

Orbius. In *Orbius* players are divided into at least two separate teams and assigned a team color. Their objective would be to find five specifically sized orbs (ranging from small to large) scattered across the world and return them to an agreed upon container, which would become their "base". After assembling a complete "base" with five of their team-colored orbs, a "golden" orb would appear somewhere in the game world. Team members would then need to find that golden orb and place it in an enemy base.

An experiment with over 30 human players revealed that Orbius players engage in interesting team behavior. Some of the team members spread out to find the orbs, while others would stay close to the chosen container to protect the base. *Orbius* introduces a new *tickle* action that allows a player to force another player to drop an item he is carrying. Therefore, players would also often be chasing after other players. Besides adding the tickle action, the Orbius subgame also had to redefine pickup and drop, define the orb item types, and instantiate and distribute many orbs of different colors and weights throughout the game world. In addition, Orbius modified the maximum carrying capacity of the players in order to force them to carry large orbs one at a time.

3.3 Distributed Game Architecture

In multiplayer and massively multiplayer games, in order to provide a shared sense of space among players, each player must maintain a copy of the (relevant) game state on his computer. When one player performs an action that affects the world, the game state of all other players affected by that action must be updated.

Different strategies for the distribution of the game state can have a profound impact on the scalability, consistency and performance of the game. Since Mammoth is specifically designed for experimentation with different distribution approaches, Mammoth defines an intuitive object-based interface between the game layer and the framework components that handle distribution.

[1]Although theoretically hierarchies of games can be created, most often a subgame is just a game that takes place in the Mammoth world.

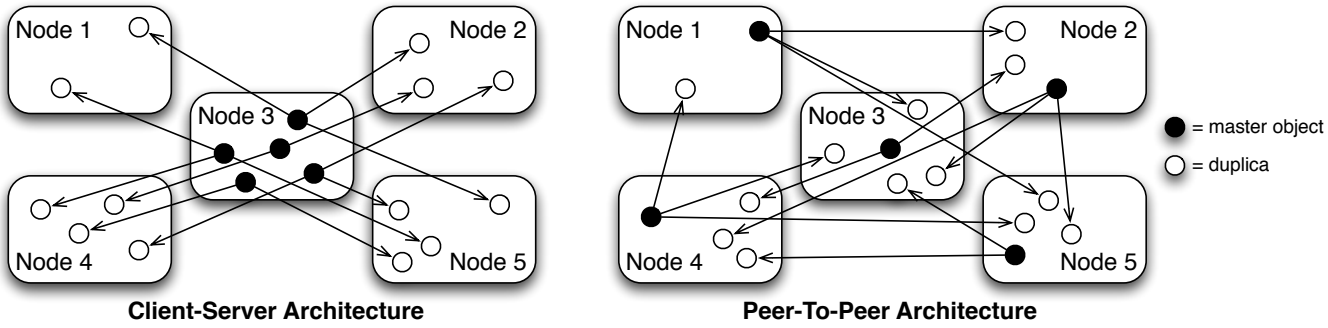

Figure 2: Creating Different Network Topologies by Migrating Master Objects

In our approach, game objects are mapped to *duplicated objects*, which encapsulate the state of the game objects that has to be distributed to players. Every node that needs access to the game state encapsulated by a duplicated object creates a new local instance of the object, a *duplica*. Whenever the game executes a *read* operation on a game object, the state of the local duplica is read. *Modifying* operations, however, cannot be executed locally for consistency reasons. If local execution was allowed, it would be possible for concurrent modifications to take place, which could result in serious inconsistencies visible to the players.

Consistency in our design is guaranteed by designating one of the copies of the duplicated objects as being the *duplication master*. Modifying operations are always executed sequentially on the node that holds the duplication master. After the operation has finished executing, update messages are broadcast to all duplicas.

The remote execution of modifying operations is completely transparent to the game layer. The game simply invokes the operation on the game object: our duplicated objects redirect the call to the duplication master node, if necessary. This transparency is not only convenient for the programmer. It also makes it easy to migrate the duplication master from one node to another node for load balancing or fault tolerance reasons.

By assigning the duplication master objects to nodes, different network topologies can be built. The left hand side of Fig. 2 illustrates that, for instance, a typical client-server architecture can be built by assigning all master objects to a single machine. At the other extreme, a peer-to-peer topology, shown on the right hand side of Fig. 2, can be created by uniformly distributing master objects over all machines. The left hand side and the right hand side actually show an identical game state distribution, but the network connections used to send state updates are different. Of course, any other intermediate network topology, for instance server clusters, can be built by migrating master objects from one node to the other, or even load-balancing dynamic topologies that evolve according to the current game situation using run-time master object migration.

3.4 Mammoth Framework Components

In order to allow researchers to easily conduct experiments, the Mammoth framework has been designed as a collection of collaborating components that each provide a distinct set of services. The components interact with each other through two types of well-defined interfaces, engines and managers. The general architecture is depicted in Fig. 3.

At the highest level, the Mammoth architecture follows the Model-View-Controller paradigm. The main components in the model are the *WorldEngine*, the *SubgamesManager*, the *PhysicsEngine* and the *PathFindingManager*. The main components in the view are the *PersistenceManager*, the *WebMonitor & Logging* component, the *XMLTools* and various Mammoth clients, which also act as controllers. Currently Mammoth has a 3D client, a 2D client, a wireless client (which remotely connects to a PDA), and a NPC client (which is a client without graphical user interface that runs AI algorithms for controlling the movements of a player). The *Model* is connected to the *Views* and *Controllers* though the *ReplicationEngine*, which implements the run-time support for duplicated objects. It contains the *InterestManager* and interfaces with the *NetworkEngine* for low-level communication.

The services offered by the individual components are briefly explained below.

Engines. Engines are core components that can be completely replaced to experiment with alternative implementations. A classic example of this would be the multiple network engines available in Mammoth. The engines can be interchanged transparently, as long as they provide the required features determined by their interface.

- **World Engine:** The *World Engine* stores all the components contained in the game world and provides an easy interface to retrieve these components. At first glance, the *World Engine* doesn't seem like a component that needs to be replaceable, since it basically is just a big data structure storing game objects. However, careful profiling has revealed that a rather large percentage of CPU time (more than 20%) is spent in the *World Engine* searching for various game objects. Thus, optimizing the different data structures used to store the game components in the *World Engine* is an interesting research problem. Our first implementation of the *World Engine* used one large hashtable to store all the different game components. The current implementation has demonstrated an important increase in efficiency by storing the different game objects in smaller separate hashtables.
- **Graphics Engine:** The graphics engine must be capable of displaying the world to the player, and allow the player to visualize his inventory and trigger actions such as moving, manipulating objects or chatting to

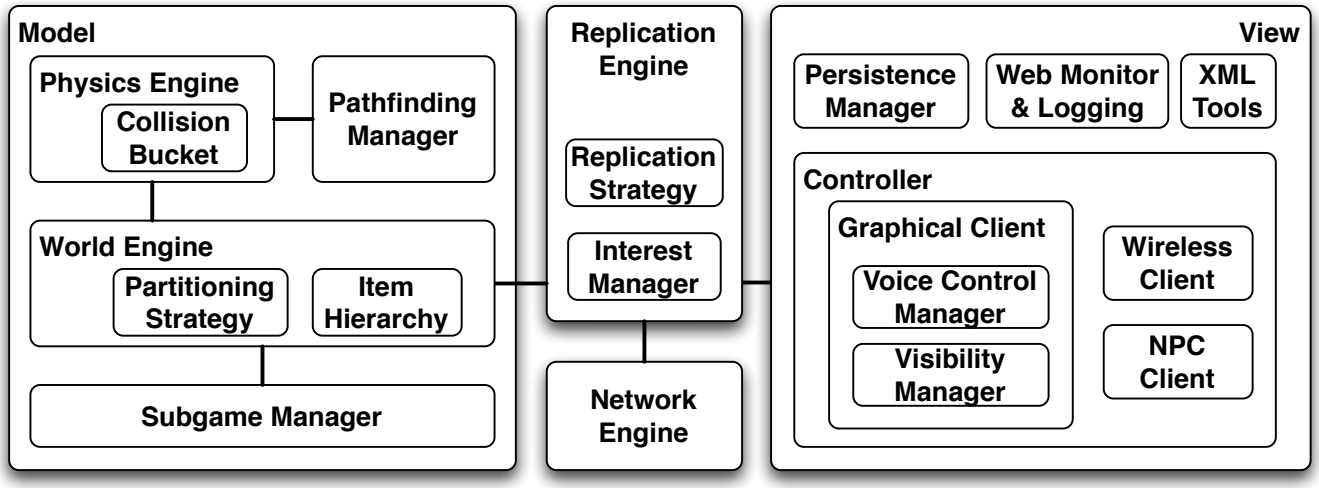

Figure 3: Components of the Mammoth Framework

other players. Over the last years, different graphical Mammoth engines have been implemented. The first graphics engine was implemented on top of *Jogl* [22], Java's interface to OpenGL. Our latest graphics engine is based on *JMonkey* [20], a high performance Open Source Java-based 3D graphics library. The windowing system used to display the player's inventory and action buttons is a Swing-like library running on top of OpenGL called *Feng GUI* [21].

- **Physics Engine:** The physics engine implements interactions between game objects that are based on the laws of physics. Currently, our physics engine is very simple. It only implements basic collision detection. However, we are planning to implement a physics engine for Mammoth that calculates object dynamics, i.e., assigns a mass to every object and calculates updates for the position of all objects based on their current momentum and external forces that might be applied to them.
- **Replication Engine:** The replication engine is the primary component responsible for distributing the state of the world across multiple clients, i.e. creating master duplication objects, assigning them to nodes, and distributing the duplicas. The replication engine uses the *Interest Manager* (see description below) to determine to which node duplicas have to be sent to. Finally, the replication engine allows a game designer to define *duplication spaces* [3]. Duplication spaces are dimensions in which objects can discover other objects, and be discovered by other objects. For instance, *3D Geometry* is an example of a duplication space commonly used in MMOGs. An object that occupies physical space in the virtual world is a *publisher* in the *3D Geometry* duplication space, an object that can see objects by observing the virtual world is a *subscriber* in the *3D Geometry* duplication space. Objects can simultaneously be publishers and subscribers not only in one, but also in multiple duplication spaces. For example, an avatar carrying a radio would be a publisher and subscriber in the *3D Geometry* duplication space, and a subscriber in the *Radio Frequency* space.

  Currently, the Mammoth framework has two interchangeable replication engines, a *networked* version and a *local* version. The local version is used to run unit tests, while the networked version is used to run a standard multiplayer game.
- **Network Engine**: The network engine component provides basic communication means to the framework. In order to support the communication needs of duplicated objects, the network engine provides *direct asynchronous messaging*, *direct synchronous messaging* (in the form of *remote method calls*), and *publish / subscribe-based broadcast* capabilities. Currently, the Mammoth framework includes three interchangeable network engines: *Stern* (communication is routed through a central hub), *Toile* (fully connected network), and *Postina* (a self-organizing peer-to-peer network engine using tree-based broadcast). In addition, a *Fake* network engine is provided, which uses shared memory and emulated serialization to route messages across components. *Fake* is mainly used when executing unit tests on components that depend on a network engine.

Managers. Managers are components designed to manage multiple implementations of a given algorithm or strategy. As opposed to engines, which allow a single implementation of a particular component, managers allow multiple implementations of a given functionality to be registered with the system. A classic example of this would be the *PathFinding* manager, which provides several different path finding algorithms. Different algorithms can then be assigned to different players, allowing for experimentation in a live setting.

- **Pathfinding Manager:** The *Pathfinding Manager* is responsible for managing the different registered path finding algorithms. It also acts as a central hub for receiving pathfinding requests, allocating the necessary resources (such as threads from the thread pool in case the pathfinding algorithm is run in the background) and dispatching them to the proper algorithm.
- **NPC Manager:** The *NPC Manager* is the central administration unit for the behavioral AI components

in Mammoth. Once an AI component is registered with this manager, an AI controller can be spawned and assigned to a player. This AI component can then control the actions of that player. The manager can also be used to monitor the behavior of a player, which is particularly useful for AI algorithms that learn by observing human-controlled movement.

- **Persistence Manager:** The *Persistence Manager* administers the different strategies used by Mammoth to save the state of the game to stable storage. Both push (automated recording of events) and pull (explicit retrieving of the state of game objects) strategies are supported. In addition, the *Persistence Manager* provides the necessary *Data Access Objects* (DAO) required to access stable storage, which is most often a relational database. Given that the persistence strategy implementations are kept separately from the DAOs, it is trivial to experiment with different strategy / storage medium combinations.
- **Interest Manager:** The object replication scheme used by the replication engine uses an elaborate publish/subscribe system to propagate state updates throughout the system. The interest manager is responsible for matching publisher duplicated objects to subscriber duplicated objects. This can be done using different criteria, for instance, based on distance, on zones, on visibility, on reachability, etc. We currently provide 9 different interest management strategies.
- **Subgame Manager:** The *Subgame Manager* is a component designed to coordinate the loading, joining, leaving, starting, and intra-coordination of subgame entities. Every machine in the Mammoth network has it's own *Subgame Manager,* which loads the same set of subgames to be available to players. Once loaded, player requests to join subgames are routed to the *Subgame Manager*, which either connects the player to an existing game in progress, or creates a new subgame instance for the player, if needed.

3.5 Logging and Monitoring Capabilities

Given that Mammoth is a research framework that is intended for experimentation, extensive work has been done in developing tools to monitor and record the state of the game world while a game is executing. Basic logging facilities are provided by *Log4j*, a popular logging package for Java. However, Mammoth offers some more elaborate logging and monitoring features as explained below:

- **Logging Player Movement:** One of the key feature of using Mammoth for experimentation is the ability to record the activities of players. Given the high amount of movement actions a player can generate, a special custom movement logging solution was implemented that uses a combination of rotating memory buffers to cache writes. That way, writing to disk is done in a controlled fashion. The movement logging component also includes tools to replay previously recorded movements of players using a special client designed for that purpose.
- **Web Monitor:** Debugging a distributed application is a tricky task, given that the game state is distributed over several machines. However, this task is greatly simplified if a developer can easily inspect the state of a client without altering its execution, as debuggers will often do. In the Mammoth framework, all participants (both servers and clients) are equipped with an embedded webserver. By instructing a standard web browser to connect to this webserver, developers can display the properties of objects found in the replication and world engine at run-time without altering, disrupting, or pausing gameplay.

4. MAMMOTH IMPLEMENTATION

The implementation of Mammoth is done almost exclusively using the Java programming language. This was a practical decision. Many researchers at the School of Computer Science of McGill University use Java for their experiments, and many tools have been developed for research and performance analysis in Java. Furthermore, the cross-platform nature of Java facilitates access to Mammoth for the students, and makes maintenance easier. We are of course aware that an industrial implementation of our framework using a non-interpreted language such as C++ would provide even better performance. However, our experiments are still valid, since they provide insight into the complexity of our algorithms and techniques as the number of players, game objects and nodes increases. We are currently working with Quazal, an industrial partner, to integrate the best ideas into their commercial product Net-Z.

In order to achieve flexibility and extensibility, many advanced programming techniques have been used in the development of Mammoth. Given the modularity requirements, many of the design patterns proposed in [11] are put to good use. This section outlines the importance of interfaces and listeners in the design of this modular architecture, and also describes how XML is used to successfully deal with changing data structures.

4.1 Interfaces

One of the key elements in the Mammoth architecture is the flexibility with which engines can be replaced, and new algorithms registered with the managers. In order to make this possible, engines and manager strategies define their own *interfaces*. All interaction between components in Mammoth is done by calling interfaces, i.e. at the abstract level: no concrete implementation is ever directly referred to. This is very similar to the *bridge* design pattern [11], where abstractions are decoupled from their implementations. As a result, the implementation of a component can be changed without the need to modify any of the depending components. However, the definition of the abstraction is fixed: changes in the interfaces themselves are more complex and would require significant changes, and should therefore be avoided. Fortunately, after 3 years of development, the interfaces of the major components are fairly stable.

Object factories, and their respective configuration files, are used to control the instantiation of the different implementations. A researcher can simply specify the component to be used in the Mammoth configuration file before starting the game. The factories read the researcher's choice from the configuration file and instantiate the appropriate engine or instruct the managers to use a specific strategy. In this manner, the researchers do not need not worry about the initialization details of a component. Some factories even use the Java reflection API to automatically recognize new available implementations of components. This allows the addition of new implementations without the need to modify existing factories.

4.2 Listeners

Modularity and separation of concerns is essential when developing a complex research framework. Strong dependencies between components greatly reduce the flexibility and maintainability of the source code. As a result, team development is complicated, since changes required to implement a specific feature within one module can have a major impact on other parts of the framework.

Within Mammoth, for example, the persistence engine requires knowledge on how and when a world object is modified. The world engine could directly inform the persistence manager about the state update, but that would create a dependency. Changing the implementation of the persistence strategy might then again require the modification of the world object. Such modifications could be risky, since the world engine is a central component of the Mammoth framework.

The Mammoth framework addresses this problem through the extensive use of *listeners*. As described by the *observer* design pattern [11], core objects containing the game state are considered *subjects*. Components requiring information about a subject can register themselves as observers with the subject by implementing the appropriate listener interface. Whenever the state of a subject changes, all registered observers are notified of the change.

The most notable example of the use of listeners within Mammoth is the graphical game client, which is designed as a view object registered with every world object. However, listeners are also used in various other components, such as the network engine, the replication engine and the authentication engine for various purposes. For instance, the NPC manager, which requires information about player logins and logoffs, registers itself as a listener to the authentication engine and the network engine.

4.3 XML

One of the biggest challenges when working on a research framework is dealing with data structure changes. The perfect example of this is the Mammoth world map, whose format has changed countless times since the beginning of the project. Creating a map is a time consuming task, and many researchers have created custom maps for a specific research purpose. It is therefore impractical to recreate or manually convert all available maps each time the data structures change. Only newly created maps that use the new features introduced into the data structure need to be created using the new data format, provided that it is still possible to load maps saved in the old format into the game.

In Mammoth, data structure changes are made possible through the use of XML. For instance, world maps and item type definitions are stored in XML. XML tags are assigned to every object and every attribute. Elaborate attributes, such as “shape” which describe the shape of an object, have their own sub tag. Whenever the format of a data structure changes, a new XML reader / interpreter is written for the new format, and the readers of the older formats are updated to simply convert the old format to the new one. That way, any data structure that is read from disk is automatically converted. Of course, if the data structure is saved back to disk, the most recent save format is used.

5. PAST EXPERIMENTS

Mammoth has been successfully applied to several different research problems representing different aspects of game design. The modular design, for example, enabled relatively simple investigation of the problem of interest management. Different approaches can be implemented and compared in the context of a real complex game environment. Our study in this area used player data gathered from actual gameplay (Orbius experiments); by replaying player movements we were able to examine the network performance of a variety of space partitionings designed for interest management, as well as the potential impact on consistency [5].

[9] uses Mammoth to determine the impact of different network topologies on the performance of a massively multiplayer game as the number of players and objects in the world increases. A client/server, a peer-to-peer, and a hybrid topology where nodes are organized in a tree-like hierarchy are evaluated. The experiments for this study used an average of 180 clients in a small enclosed space, to simulate a heavy-load situation on a game server. Although this figure might seem relatively small when compared to the number of concurrent users support by modern commercial MMOG, these games feature much larger worlds distributed and duplicated accross several node. As such, these MMOG will typically reduce load by limiting the number of players that can multually interact with each other, most often through game design decision.

Other experimental work has focused on more in-game algorithmic concerns. Path-finding is an optimization issue that has seen significant investigation in the literature, but which remains an interesting problem in the context of a realistic, complex environment. Our work on this problem made use of Mammoth to gather and measure real player data, in this case to investigate the performance of several path-finding implementations, showing the relative impact of algorithm design and workload variation [15].

A further research project developed different techniques for providing persistent game state. Persistence is important to recover from server failure. While critical events must be written synchronously to the persistent storage, a set of approximation strategies have been proposed and analyzed that are suitable for events with low consistency requirements, such as player movements. An evaluation showed that a distance-based solution offers the scalability and efficiency required for large-scale games as well as low error bounds [25].

6. CONCLUSION AND FUTURE WORK

Conducting experiments in the context of multiplayer and especially massively multiplayer games can be a daunting task. A considerable development effort is needed to implement an actual game. There are many important aspects to consider while designing a computer game, such as graphics, animation, sound, interactivity, realism, and storytelling. If any one element is neglected too much, the game's "fun" factor can be diminished, if the experiments are supposed to take place in a real game setting, i.e., with human players.

This paper presented Mammoth, a massively multiplayer game research framework designed specifically for experimentation in an academic setting. Mammoth provides an interactive virtual game world without any specific game goals. Players take control of an avatar, manipulate items, and communicate with other players. Subgames allow researchers to define game goals for specific research purposes.

The Mammoth architecture has been designed to be ex-

tremely flexible and extensible. The Mammoth components, engines and managers, encapsulate different implementation strategies or algorithms, and can be easily replaced. Existing implementations can be used by researchers simply by activating the desired component in the configuration file. This flexibility has been achieved through the use of advanced design techniques such as interfaces, listeners, factories, other design patterns, and flexible data storage formats based on XML.

The Mammoth project has been very successful from an academic point of view. Over the last 3 years, a total of 2 Ph.D. students, 8 master students (6 completed theses so far), and 15 undergraduates (9 honors or semester projects) have worked on Mammoth. The project has been demonstrated at various conferences in Canada and has gained support and collaboration from game companies such as Electronic Arts and Quazal. Current research is focussing on investigating the fault tolerance properties of peer-to-peer game architectures, integrating cheat detection and fault tolerance, consistency between multiple subgame instances, content generation, NPC behavior generation based on statechart models, and automated AI generation based on observation of human behavior.

7. ACKNOWLEDGMENTS

This research has been partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Canadian Foundation for Innovation (CFI). Thanks also to Mark Lanctot for inventing and implementing the Orbius subgame.

8. REFERENCES

[1] Mammoth: A Massively Multiplayer Game Research Framework. http://mammoth.cs.mcgill.ca/.

[2] Ogre: Open source 3d graphics engine. http://www.ogre3d.org/, 2001.

[3] Quazal Technologies Inc., duplication spaces patent # 6,907,471, DDL patent # 7,096,453. http://www.quazal.com, 2008.

[4] A. Bharambe, J. Pang, and S. Seshan. Colyseus: a distributed architecture for online multiplayer games. In *NSDI'06: Proceedings of the 3rd Symposium on Networked Systems Design & Implementation*, pages 155–168, Berkeley, CA, USA, 2006. USENIX Association.

[5] J.-S. Boulanger, J. Kienzle, and C. Verbrugge. Comparing Interest Management Algorithms for Massively Multiplayer Games. In *Proceedings of Netgames 2006: 5th Workshop on Network and System Support for Games*, pages 1 – 12, October 2006.

[6] M. Buro. ORTS: a hack-free RTS game environment. In *International Computers and Games Conference (CG'02)*, Edmonton, Canada, 2002.

[7] R. Coleman, S. Roebke, and L. Grayson. Gedi: a game engine for teaching videogame design and programming. *J. Comput. Small Coll.*, 21(2):72–82, 2005.

[8] M. Conway, S. Audia, T. Burnette, D. Cosgrove, and K. Christiansen. Alice: lessons learned from building a 3D system for novices. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 486–493, New York, NY, USA, 2000. ACM.

[9] A. Denault, J. Kienzle, C. Dionne, and C. Verbrugge. Object-Oriented Network Middleware for Massively Multiplayer Online Games. Technical report, McGill University, Montreal, Canada.

[10] R. D. S. Fletcher, T. C. N. Graham, and C. Wolfe. Plug-replaceable consistency maintenance for multiplayer games. In *NetGames '06*, page 34, New York, NY, USA, 2006. ACM.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, USA, 1995.

[12] F. Glinka, A. Ploß, J. Müller-lden, and S. Gorlatch. RTF: a real-time framework for developing scalable multiplayer online games. In *NetGames '07*, pages 81–86, New York, NY, USA, 2007. ACM.

[13] M. A. Hawker. Subgames in Massively Multiplayer Online Games. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, June 2008.

[14] T.-Y. Hsiao and S.-M. Yuan. Practical middleware for massively multiplayer online games. *IEEE Internet Computing*, 9(5):47–54, 2005.

[15] M. Lanctot, N. N. M. Sun, and C. Verbrugge. Path-finding for large scale multiplayer computer games. In *Proceedings of the 2nd Annual North American Game-On Conference (GameOn'NA 2006)*, pages 26–33, Monterey, California, sept 2006. Eurosis.

[16] D. Lee, M. Lim, S. Han, and K. Lee. ATLAS: a scalable network framework for distributed virtual environments. *Presence: Teleoper. Virtual Environ.*, 16(2):125–156, 2007.

[17] D. Liang and P. Boustead. Using local lag and timewarp to improve performance for real life multi-player online games. In *NetGames '06*, page 37, New York, NY, USA, 2006. ACM.

[18] I. Parberry, J. R. Nunn, J. Scheinberg, E. Carson, and J. Cole. SAGE: a simple academic game engine. In *Proceedings of the Second Annual Microsoft Academic Days on Game Development in Computer Science Education*, pages 90–94, 2007. http://larc.csci.unt.edu/sage/.

[19] M. J. Ponsen, S. Lee-Urban, H. Muñoz-Avila, D. W. Aha, and M. Molineaux. Stratagus: An open-source game engine for research in real-time strategy games. Technical Report AIC-05-12, Navy Center for Applied Research in Artificial Intelligence, 2005.

[20] M. Powell. JMonkey Engine. http://www.jmonkeyengine.com/.

[21] J. Schaback. Feng GUI: Java GUIs with OpenGL. http://www.fenggui.org/.

[22] Sun Microsystems. Java Binding to OpenGL. http://jogl.dev.java.net/.

[23] J. Voigt. DXFramework: A pedagogical computer game engine library. http://dxframework.org/, 2006.

[24] S. D. Webb, W. Lau, and S. Soh. NGS: an application layer network game simulator. In *IE '06: Proceedings of the 3rd Australasian conference on Interactive entertainment*, pages 15–22, 2006.

[25] K. Zhang, B. Kemme, and A. Denault. Persistence in massively multiplayer online games. In *NetGames '08*, pages 1 – 10, New York, NY, USA, 2008. ACM.