Interest Management for Massively Multiplayer Games

Jean-Sébastien Boulanger

Master of Science

School of Computer Science

McGill University

Montréal,Québec

August 2006

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements of the degree of
Master of Science in Computer Science

ACKNOWLEDGEMENTS

First, I would like to thank my supervisors Jörg Kienzle and Clark Verbrugge for their support, ideas, opinions, comments, and precious time. I would also like to thank them for giving me the opportunity to get fully involved in the Mammoth project over the last year. I would also like to thank all the students who worked on the implementation of the Mammoth framework (Loc Bui, Jeremy Claude, Alexandre Denault, Michael A. Hawker, Nadeem Khan, Marc Lanctot, Alfred Leung, Pierre Marieu, Nicolas NgManSun, Alexandre Quesnel, Russell Spence) and made the implementation of the work in this thesis possible. Although a team effort, a special thank goes to Marc Lanctot for the initiative and organization of the Orbius gaming event that allowed to collect the real-player traces used in my experiments, this data was invaluable for this thesis. I would also like to thank my parents for their support all along my studies. Finally, I will thank all my family and friends for their understanding over the last year, during which I spent a lot of my time working on this thesis instead of spending time with them. Hopefully I can catch up over the next few months.

ABSTRACT

The popularity of massively multiplayer games has increased in recent years and game providers are facing scalability problems to accomodate growing populations of users. Broadcasting all state changes to every player is not a viable solution to maintain a consistent game state in a massively multiplayer game. To successfully overcome the challenge of scale, massively multiplayer games have to employ sophisticated *interest management* techniques that only send *relevant* state changes to each player.

In this thesis we develop a space partitioning technique based on triangulation that adapts to the world's obstacles. We introduce obstacle-aware interest management algorithms that use the triangular partitioning to determine the relevance of objects based on the occlusion created by obstacles. We compare the performance of both obstacle-aware and state-of-the-art interest management algorithms based on measurements obtained in a real massively multiplayer game using human and computer-generated player actions. We show that obstacle-aware interest management algorithms can reduce the number of update messages between players and that algorithms based on our triangle-based partitioning can scale to a larger number of objects. The experiments also show that measurements obtained with computer-controlled players performing random actions can approximate measurements of games played by real humans, provided that the traces of the random players are designed adequately. As the size of the world and the number of players

of massively multiplayer games increases, adaptive interest management techniques such as the ones studied in this thesis will become increasingly important.

ABRÉGÉ

La popularité des jeux massivement multijoueurs a augmenté de façon phénoménale au cours des dernières années. Les fournisseurs de jeux rencontrent de plus en plus de problèmes d'extensibilité pour supporter des populations croissantes de joueurs. La diffusion à tous les joueurs des changements réalisés dans le monde virtuel n'est pas une solution viable pour maintenir une vision cohérente du monde dans un jeu massivement multijoueurs. Pour surmonter ce défi d'extensibilité, les jeux massivement multijoueurs doivent utiliser des techniques de *gestion d'intérêt* sophistiquées qui relaient seulement l'information pertinente vers chaque joueur.

Dans cette thèse nous développons une technique de partition de l'espace qui s'adapte aux obstacles du monde virtuel en utilisant une triangulation. Nous présentons des algorithmes *sensible-aux-obstacles* de gestion d'intérêt qui emploient les partitions triangulaires pour déterminer la pertinence des objets pour chaque joueur, selon l'occlusion créée par les obstacles. Nous comparons l'efficacité des algorithmes sensible-aux-obstacles et d'autres algorithmes modernes de gestion d'intérêt à l'aide de données obtenues d'un vrai jeu massivement multijoueurs. À cet effet, nous utilisons à la fois des actions de vrais joueurs et des actions de joueurs générées par ordinateur. Nous démontrons que les algorithmes sensible-aux-obstacles de gestion d'intérêt peuvent réduirent le nombre de messages relayés entre les joueurs. Nous démontrons également que les algorithmes utilisant notre partition triangulaire peuvent s'adapter à un plus grand nombre d'objets tout en conservant de bonnes

performances. Nos expériences suggèrent également que les résultats obtenus à partir de joueurs contrôlés par ordinateur se déplaçant aléatoirement ce rapprochent des résultats obtenus avec de vrais joueurs, dans la mesure où les actions aléatoires des joueurs sont conçues adéquatement. Avec la croissance des mondes virtuels et l'augmentation du nombre de joueurs des jeux massivement multijoueurs, les techniques de gestion d'intérêt adaptatives comme celles que nous avons étudiées deviendront de plus en plus importantes.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
Introduction

Since 1997 with the creation of Ultima Online, a new genre of online game has emerged, the massively multiplayer online (role-playing) game, short MMOG or MMORPG. Compared to a traditional multiplayer game in which usually up to 16 players play a relatively short-lived game, MMOGs offer the possibility for thousands of players to play together in a persistent world [29]. The popularity of online gaming and MMOGs has only increased with years, popular games such as Everquest, Lineage, the World of Warcraft have hundreds of thousand of subscribers [39]. The industry forecasts that from 2002 to 2008 the worldwide online game usage will be growing to 35 billion hours per year. The successful MMOG, Lineage by NCSoft was the first online game to generate earnings of $100 million per year and set the tone for many other successful MMOGs [19].

The highest number of concurrent users in a unique game world was recorded in March 2006 by EVE Online to be near 26,000 [1]. Although game providers use many tricks (e.g., running multiple clones of the same game world) to support more players [29], MMOGs still face huge scalability problems in order to be able to support hundreds of thousand of players in a continuous, unique world.

In a typical game, each client sees a graphical representation of the world and controls a player – an avatar – which can perform actions. Basic building blocks of such actions are, e.g., moving the avatar, picking up objects, or communicating with

other players. The scalability problems that MMOGs face rise from the fact that 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.

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. The simplest approach is for each player to maintain a full copy of the game state and that all players broadcast updates to all other players. The problem with this approach is that it does not scale: as the number of players increases, the messages sent over the network and to be processed by each client increase exponentially.

One of the most effective strategies to address this problem is to send to a player's computer only the messages that are relevant to its avatar (e.g., only the update message of objects it can see, or that are near). The world space of MMOGs contains a lot of information and a single player needs only to know about a subset of that information. *Interest management* is the process of determining which information is relevant to each player [31].

Interest Management

Interest management has the potential to considerably reduce the amount of information that must be exchanged between players and the resources it consumes; on the other hand, filtering the information for each individual player has a computational cost. There is an important trade-off between the resources that can be saved

from filtering information and the cost to filter that information. There is constant need to develop interest managements that can more accurately filter information at a lower cost.

The information relevant to a player usually corresponds to the perception of its avatar. The perception, or expression of interest [31] of an avatar in interest management schemes is often based on proximity, modeled as a sphere around the avatar. However, the most common type of perception in MMOGs is what an avatar can see, which does not always correspond to proximity. In particular, game worlds usually contain static obstacles that occlude regions of the game space. An object that is close to an avatar, but behind a wall, is not relevant to that player.

Many interest management techniques [31] have been proposed and implemented in distributed simulations, networked virtual environments and games (see Chapter 2). However, very few experiments have been performed to evaluate and compare interest management techniques, especially in the context of MMOGs. Furthermore, most evaluation that has been done used simulations with randomly generated data. It is not clear beforehand that results from random data will hold in a real world environment.

Overview

In this thesis, we compare and evaluate eight interest management algorithms in the context of an MMOG. Three of the algorithms we evaluate are state-of-the-art proximity based interest managements which simply consider the radius around the player as the region of interest. The five other algorithms take into account obstacles in the world and attempt to leverage the fact that a player does not need to be

updated about objects that are occluded. In particular, we propose a partitioning of the world space into regions based on Delaunay triangulation that adapts to obstacles. We develop four algorithms which use the partitioning to perform more scalable obstacle-aware interest management. Our obstacle-aware algorithms have the property to take into account obstacles such that occluded objects are not necessarily considered as interesting.

The eight algorithms are evaluated within an MMOG, the Mammoth massively multiplayer online game development framework [6]. Experiments are performed using game play traces from 28 real human players that were collected in a gaming event. We also perform the same experiments using two randomly generated traces, in order to compare the results with the results from real player traces.

Contributions

More precisely the four main contributions of this thesis are:

- A comparison of multiple interest management algorithms in an MMOG setting using real-player data. Our evaluation provides insights on the filtering performance and cost of interest managements.
- The elaboration of a world partitioning technique that takes into account obstacles and preserves information about the geography of the world. The technique was designed to be suitable for interest management but could also be relevant for use in other concerns such as path finding and load balancing.
- The development of two scalable interest management algorithms, one based on visibility, and one based on reachability that could be useful in MMOGs with densely occluded environments.

- An assessment on the validity of using randomly generated traces rather than real-player data to evaluate interest management algorithms.

Road Map

The remainder of this thesis is structured as follows: the next chapter presents background on massively multiplayer games and interest management, as well as an overview of the related work in that area. Chapter 3 describes the development of our obstacle-aware partitioning technique based on triangulation. In Chapter 4 we describe the eight interest management algorithms that we evaluated. Chapter 5 explains how we abstracted and integrated interest management into the Mammoth massively multiplayer development framework, how we collected the traces, and performed our measurements. Chapter 6 discusses the results, and the last chapter draws some conclusions and discusses future work.

CHAPTER 2
Background and Related Work

Massively multiplayer online games (or MMOGs) are computer games in which a large number (thousands) of simultaneous users interact in a shared virtual world. In a typical role-playing game, each client sees a graphical representation of the world similar to Figure 2–1.

The development of MMOGs with virtual worlds that can accommodate larger and larger populations of concurrent users faces many challenges, which are discussed in the first section of this chapter. The underlying communication architecture can influence the design and implementation of interest management; we discuss communication architectures in Section 2.2 of this chapter. We also discuss background and related work on interest management, most of which is from the networked virtual environments and military simulations literature. Finally, we also discuss a few other techniques that are commonly used in combination with interest management to deal with the same technical challenges, and we introduce the Mammoth massively multiplayer development framework.

2.1 Massively Multiplayer Online Games and their Challenges

The development of MMOGs comprises many technical challenges: distributed consistency, fault-tolerance, administration of live production servers, preventing cheating, scalability, and others [9, 29]. One of the biggest challenge is to scale the game state while providing distributed consistency (i.e., the shared sense of space)

Figure 2–1: Player's avatar in the Mammoth MMORPG world.

to a *massive* number of concurrent users. 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. As the number of objects that can change and the number of players in the world increase, the amount of information that must be exchanged between players also increases.

However, the amount of information that can be exchanged between computers in an MMOG is bounded by at least two technical limitations: *network bandwidth* and *processing power* [35, 37].

Another important requirement of an MMOG is that the information about the game state must not only be exchanged between players, but it must be exchanged in

a timely manner. Game state changes must be propagated within a delay that does not affect the game play, usually below human perception. Delivery of information in a timely manner must deal with the technical limitation of *network latency* [35, 37].

Network Bandwidth. The bandwidth is the amount of data that can be transmitted over a network in a fixed amount of time. The bandwidth is limited by the underlying hardware that connects the computers. On the Internet many factors can affect the available bandwidth: the user's connection to its Internet Service Provider (ISP), the ISP's hardware and software infrastructures, traffic congestion, and others. Furthermore, each user of an MMOG is likely to have different bandwidth capabilities. Ultimately, the amount of information that can be exchanged between computers in order to keep the game state consistent in an MMOG is bounded by the network bandwidth capabilities of the users. In order to scale an MMOG it must be possible to add new users without dramatically increasing the bandwidth need of other users.

Processing Power. The processing power is the amount of calculations that can be performed by a computer in a fixed amount of time. Every computer's CPU has a fixed processing power capacity. In a game, processing power is required for multiple concerns such as physics, collision detection, graphic rendering. In networked games such as MMOGs, processing power is also required to send and receive network messages. The amount of information exchanged between computers does not only affect network bandwidth, but it also consumes CPU resources.

Network Latency. Latency is the amount of time it takes a message to travel over a network from source to destination. Although latency is ultimately bounded

by the physical limitation of the underlying hardware, congestion caused by a large amount of information sent over the network also affects latency.

Many techniques have been developed to mitigate the effect of those three limitations (see Section 2.5), but one of the most effective techniques is interest management. Interest management reduces the amount of information sent between players by sending to a player only the messages that are relevant to its avatar.

2.2 Communication Architecture

Interest management is a technique to reduce network messages and is closely related to the underlying communication protocol as well as the network software architecture. In this section we introduce the elements of the communication architecture that are relevant to interest management.

2.2.1 Data Communication

Data communication in larger multiplayer games can be performed using a variety of packet delivery methods. This includes basic unicast as the most popular current choice, but also broadcast and multicast approaches.

Unicast. Standard unicast approaches in games focus on the difference between TCP and UDP protocols. UDP is a simple best-effort protocol that offers no reliability and no packet ordering guarantee. It has very little overhead, making it appropriate for highly interactive games (e.g., first-person shooter, car racing) where speed of packet delivery is paramount. TCP guarantees ordered delivery of packets; this simplifies application programming, at a cost of noticeable overhead. TCP also has the advantage of working more transparently across firewalls, and has ended up

being the protocol of choice for many commercial MMOGs (e.g., EVE Online [1], Lineage II [4], and World of Warcraft [5]).

Broadcast. In more restrictive settings actual broadcast can be used. Local area networks (LANs) can be configured to allow a single packet to be sent to all hosts simultaneously, in a manner similar to UDP. This can make transmission of state data in MMOGs extremely efficient and simple. Unfortunately, broadcast is not typically allowed across router boundaries, and Internet-based MMOGs are not able to take practical advantage of this efficiency.

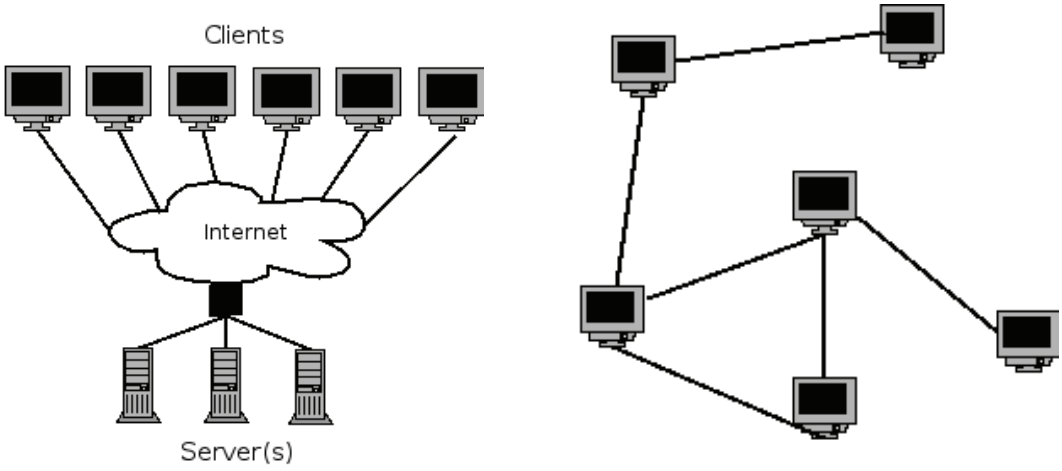
Multicast. Multicast systems provide unreliable, group-based packet delivery; a host can subscribe to one or many multicast addresses and receive all messages sent to those addresses. Transmission is not quite as efficient as basic broadcast, but is usually much more efficient than multiple unicast operations. Interest management systems in games often specify multicasting as a mean of efficiently implementing interest groups and associated network communication [43]. Unfortunately, not all ISPs provide access to the multicasting internet layers; access, firewall, and resource concerns mean multicasting is still not a general choice for MMOGs.

2.2.2 Network Architectures

Interest management can be used in the context of different network software architectures. There are two main paradigms of architecture in use for MMOGs: client-server, and peer-to-peer [36]. There are also "hybrid" architectures that are in between the two main paradigms such as grid computing and distributed computing.

Client-server. In a client-server architecture (see Figure 2–2 a), each client has a single connection with the server, which is responsible to relay the information

between clients. The server can be one or a cluster of dedicated machines that are usually maintained by the game provider. The main advantage of the client-server architecture is that the control over the game is centralized, and retained by the provider. It also facilitates implementation of the network layer, load balancing, security, and other concerns. The main drawback of the client-server architecture is that the server is an architectural bottleneck and limits the scalability of the system. The client-server approach is still the most popular among game providers.



(a) Client-server **(b) Peer-to-peer**

Figure 2–2: Network architectures

Peer-to-peer. In a peer-to-peer architecture there is no central point of control (see Figure 2–2b). The machines of players (i.e., peers) in the game are used as resources to run and manage the game in a distributed fashion. Peers may be connected with an arbitrary number of other peers and they exchange network messages directly between each other. The advantage of the peer-to-peer architecture is that it is scalable because, as opposed to the client-server architecture, there is no central

bottleneck. It also has the potential to provide better fault-tolerance as there is no central point of failure. The drawback is that most other concerns (network layer, load balancing, security, ...) are much more difficult to address in a peer-to-peer context, and there are still many challenges to address before a pure peer-to-peer architecture is widely adopted.

2.3 Interest Management

In this section we discuss the background concepts and related work on interest management.

2.3.1 Publish-Subscribe Abstraction of Interest Management

Interest management can be abstracted using a publish-subscribe model [13, 20, 32]. *Publishers* are objects that produce events, *subscribers* are objects that consume events, and an object can be both a publisher and a subscriber (e.g. a player's avatar). In this model, interest management consists of determining when a subscriber subscribes or unsubscribes to/from a publisher's updates.

Interest management can have multiple domains, the most common domain is visibility, but there can be other domains (e.g., audible range, radio contact). Each interest management domain can have different transmission and reception properties as well as different sets of publishers and subscribers. Middleware such as Quazal's Duplication Spaces allows relatively arbitrary functionality to be used for regulating multiple, co-existing information domains [32]. *e-Agora* also uses a system of multiple, independent domains; e.g., both chat and navigation data [28].

2.3.2 Space-based Interest Management

Interest management schemes can usually be separated into two broad categories: space-based and class-based, or extrinsic and intrinsic respectively [31]. *Space-based* interest management is determined based on the relative position of objects in the virtual environment, while *class-based* is determined from an object's attributes (e.g., type of object). Space-based interest management is usually the most important in MMOGs because the relevant information to a player is usually closely related to its position in the environment.

Space-based interest management is usually based on proximity, and can be understood in terms of an *aura-nimbus* information model [12]. The *aura* is the area that bounds the presence of an object in space, while the nimbus or *area-of-interest* is the space in which an object can perceive other objects. In its simplest model, both the aura and nimbus can be represented by fixed-size circles around the object (see Figure 2–3). An object X is then aware of another object Y when the nimbus of X intersects the aura of Y.

Aura-nimbus interest management. The pure aura-nimbus model has been implemented in many systems, such as MASSIVE-1 [22], Morgan et al.'s approach based on standard message-passing middleware [30], and commercial middleware such as through Quazal's *Duplication Spaces* technology [32]. In Quazal's *Duplication Spaces*, the developer is responsible to implement a game-specific interest management function.

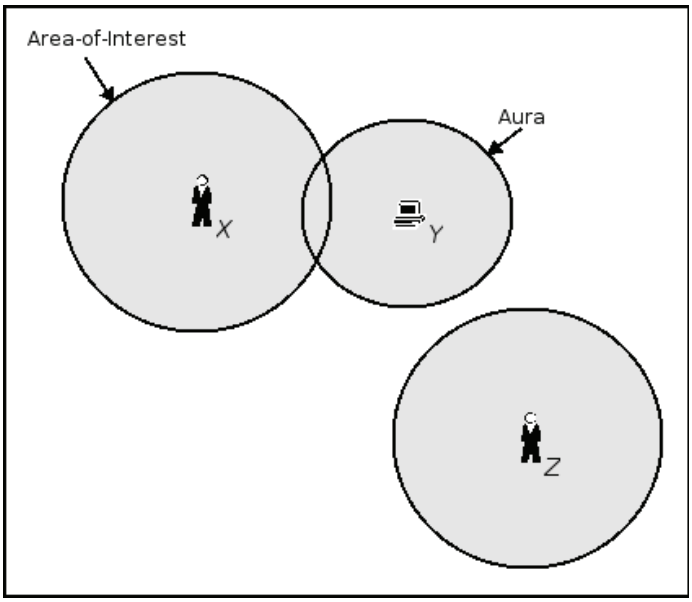


Figure 2–3: Aura-nimbus model: X is aware of Y, but Z is not aware of Y.

The advantage of a pure aura-nimbus implementation is that it allows fine-grained interest management in which only the relevant messages are sent to subscribers. It is especially suitable when there is a connection for each client with the server (e.g., TCP connection). The drawback of a pure aura-nimbus model is that it does not scale well because of the cost of computing the intersection between the area-of-interest and the auras of objects [35]. The computation of intersections between subscriber and publishers has a complexity of $O(MN)$ where M is the number of subscribers and N the number of publishers in the world. This computation can become a bottleneck in systems without broadcast or multicast capabilities.

Region-based interest management. To mitigate the limitations of a pure aura-nimbus model, region-based interest management is used by many systems as an approximation [8, 20, 21, 27]. In region-based interest management the world

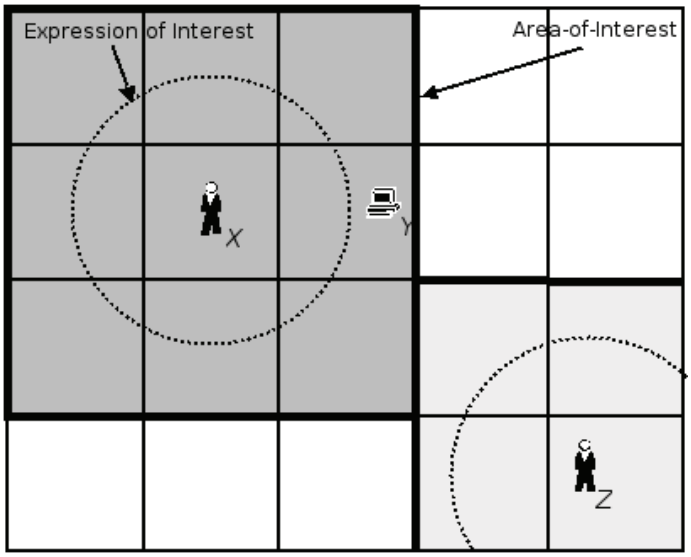


Figure 2–4: Square regions interest management: X is aware of Y because Y is within a region that spawns X's expression of interest.

space is first partitioned into static regions. The interest management determines the regions that intersect the expression-of-interest of the subscriber and forms the area-of-interest from the union of the intersected regions (see Figure 2–4). The area-of-interest represents an approximation of the true expression-of-interest; this approximation, however, is often cheaper to compute than a pure aura-nimbus model.

The quality of an interest region approximation is highly dependent on the shape and size of regions. Regular square partitionings are quite popular and straightforward to implement. Square regions also have the advantage that they allow for more dynamic partitioning schemes that work at different granularities using an octree structure [8, 15]. Another popular shape is hexagonal regions [20, 27, 42]. Studies have shown that hexagons can better approximate the aura-nimbus model [20]. If communication groups are each assigned to an hexagon that is large enough,

hexagons also have the advantage that the maximum number of subscribed groups is bounded to three instead of four for squares. Other systems such as Spline allow the designer to create regions of any shape or size [11]. It has the advantage that regions can be specially designed to adapt to the geography of the world or to address anticipated hot spots. Steed and Abou-Haidar have proposed a similar approach in which regions are created from an analysis of the players movement data in the world [38]. Using the collected usage data, the technique partition the world in a heuristically optimal way. However, there are limitations to the anticipation of the creation of hot spots, these strategies can also add an additional burden on the designers of the game.

Region-based interest management maps nicely onto multicasting. In NPSNET, for example, the space is divided into hexagons, and a multicast group is assigned to each hexagon [27]. A publisher sends events to the multicast group of the hexagon it occupies, and subscribers subscribe to multicast groups within their area-of-interest. If hexagon sizes are carefully chosen to be large enough, the number of subscribed groups of an object can be bound, limiting the number of subscription and unsubscription requests. Region-based interest management works best when objects are evenly distributed among the regions, and load balancing is important in situations where many objects gather in the same large region. Dynamic techniques such as the "three-tiered" interest management partially addresses the load balancing problem by providing a dynamic subdivision of regions using an octree structure [8].

Visibility-based interest management. Interest management approaches discussed so far mostly consider an area-of-interest for the subscriber that is independent of the geography of the environment. Visibility-based interest management considers the vision of players instead of a fixed radius. *RING,* for instance, implements visibility-based interest management by dividing the environment into rectangular regions and precomputing visibility between regions [21]. At run-time a player will receive updates about objects that are within regions that are visible from his or her current region. Hosseini et al. [24] developed another visibility-based interest management in which the visibility information is taken from each client's existing visibility culling performed in the course of graphic rendering. The advantage of this technique is that the visibility of objects is determined precisely and at no more cost since the information is already computed for rendering. Of course clients must first receive information about the position of all objects that may be visible in the world to be able to compute the visibility. Thus, if position is the main source of messages, the technique is not advantageous.

2.3.3 Other Criteria for Interest Management

Interest management need not always be a binary decision, and it has also been investigated with respect to scaling information quality. Han et al. build on the observation that some information (e.g., closer objects) is more relevant than other and make a distinction between high and low fidelity data [23]. Users interested in a common area create groups for which a representative is elected and responsible to send low fidelity data to other peers. This allows for observation of distant areas, but at reduced scale, and thus reduced bandwidth requirements.

Aarhus et al. also proposed a mechanism to grade the importance of update messages [7]. Higher priority messages are always sent before, and messages that are no longer relevant can be discarded before being sent to the client.

2.4 Evaluations of Interest Managements

As part of our study we attempt to evaluate multiple interest management schemes under multiple workloads. Others have also looked into the relative performance of different approaches, mostly using simulation or artificially-generated movement data. Han et al., for instance, compare their interest-based group approach with aura-based approach using simulation [23]. More convincing simulation results are given by Zou et al.; they evaluate two grouping techniques: cell-based and entity-based grouping for interest management with multicast. Extensive simulation study is then done to evaluate the trade-offs of group formation versus message dissemination [43]. Fiedler et al. compare the use of hexagonal versus rectangle partitioning using randomly generated player movement. Their results show that a smaller number of channels are subscribed too when using hexagons. They also show that the use of smaller regions result in a smaller percentage of the world being subscribed to and a smaller number of events received [20]. Funkhouser compares the RING visibility-based approach with full message broadcast using randomly generated player movements [21]. Morgan et al. evaluate the scalability of their middleware based interest management approach with randomly generated movements that attempt to reproduce realistic gathering of players by randomly positioning common targets within the world [30].

Using real user data from a military simulation, Rak and Van Hook [33] evaluate region-based interest management under multicast. They find that while the smaller the region size the better the interest management filtering, it is nevertheless expensive to subscribe and unsubscribe to multicast groups: optimality represents a trade-off between the region size and the number of multicast groups.

2.5 Other Techniques to Address MMOGs Limitations

Interest management is one technique to mitigate the technical limitations of MMOGs. However, interest management usually needs to be combined with other strategies in order to reach true scalability. In this section we discuss briefly some of these techniques.

2.5.1 Dead Reckoning

Dead reckoning [35] is a technique to reduce the number of network messages exchanged between computers by using predictions to estimate the current state of a remote object. Imagine a player moving from point A to point B. To render the movement of the player and perform collision detection, the continuous movement will be discretized into small steps corresponding to the frame rate of the game (e.g., 20 steps per second). Each discrete change in position of the player corresponds to a change to the game state. In order to keep the game consistent, each copy of the game state should receive an update of that change. If you have a large number of objects moving at the same time, sending a network message for each one at each frame would clearly generate a large amount of data sent over the network.

Dead reckoning reduces the number of messages sent by predicting the movement of objects at each copy of the game state. Given the current position and

velocity of an object, dead reckoning predicts the trajectory of the object and performs the movement locally. If the trajectory changes and the prediction becomes erroneous, dead reckoning performs a correction of the trajectory using a convergence mechanism. Dead reckoning allows for slightly inconsistent state, but it leverages on the trade-off between consistency and throughput.

2.5.2 Message Aggregation

Message aggregation reduces the update message transmission frequency by aggregating multiple game messages into a single network message [37]. Message aggregation can reduce the bandwidth usage and processing power usage by eliminating the need to transmit redundant information for each message (e.g., message header, Java serialization information). On the other hand, message aggregation has the potential to introduce additional network latency by delaying some messages.

2.5.3 Message Compression

Message compression reduces the bandwidth usage by compressing the data of network messages [37]. Message compression reduces bandwidth at the expense of more processing power, it can be useful if bandwidth is the bottleneck.

2.6 Mammoth: An MMOG Development Framework

The Mammoth project is an attempt to develop an MMOG in Java that can be used as a development framework to conduct academic research. It allows researchers to implement and experiment with novel algorithms that try to address the challenges of MMOGs.

Mammoth implements a game of the role-playing genre (i.e., MMORPG) [29]; it consists of a large, unique, continuous virtual world that the player sees from

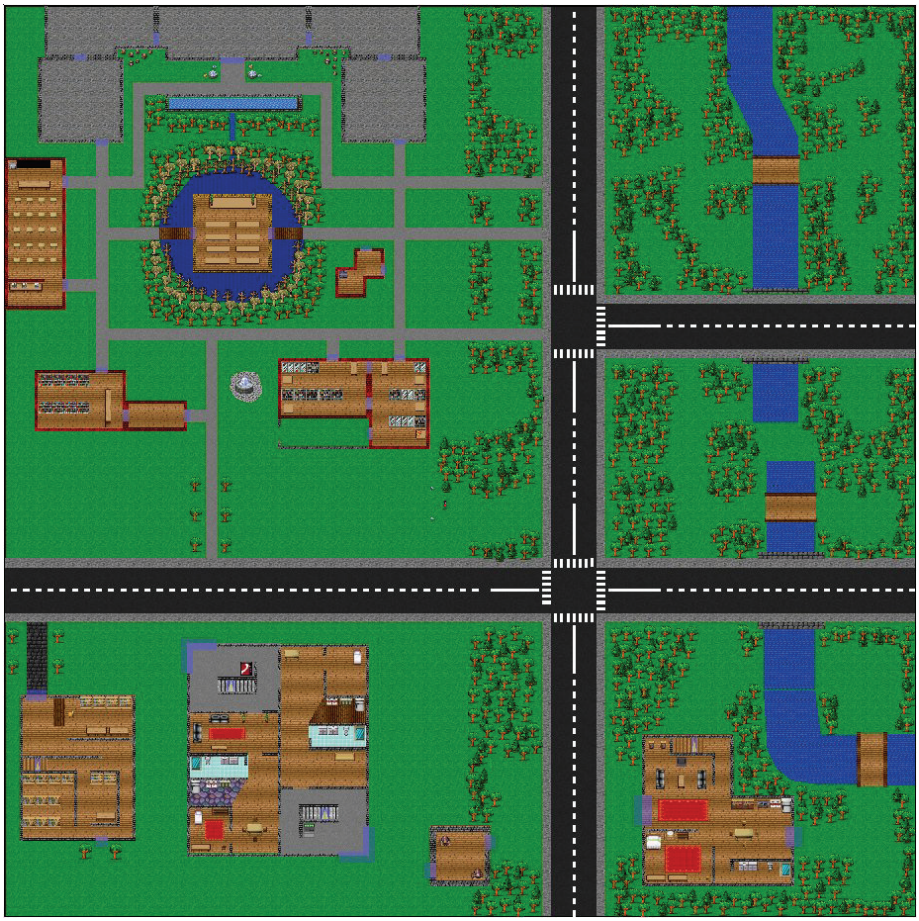


Figure 2–5: Mammoth world map

above (a god view) (see Figure 2–1 and 2–5). The view that the player has over the world is 2-dimensional, but buildings and other features of the landscape can have overlapping levels in a third dimension. All the artifacts that make up the world (obstacles, avatars, items, and others) are persistent, which means that the world continuously evolves in time and is never reset to an initial state. The "game" in the framework has no precise goal in itself; a player can simply inject himself into an avatar, explore the world, collect items and chat with other players. However, the

framework allows the creation of games with a goal (e.g., Orbius) as plug-ins to the framework. The world of Mammoth is made up from different elements, and we will describe them briefly.

Players' Avatars. Players' avatars exist permanently within the world and are not created by the players. Players may inject themselves into existing avatars to play them. When an avatar is not played by a human player, it continues to exist in the world and it can be controlled by artificial intelligence. Basic actions may be performed by players such as: moving the avatar, picking and dropping items, and chatting with other players. Each player maintains an inventory of items he/she has collected in the game.

Items. Items are objects in the world that can be collected by players. A player's avatar must be within a reaching distance of the item to pick it up. Items can be books, TVs, food, or anything else decided by the game's designer. Items are objects that have a mutable state.

Obstacles. Obstacles are objects that prevent a character from moving in a straight line and occlude the visibility of players. Obstacles have a polygonal shape; they represent walls, buildings, trees, or anything else. Obstacles are static objects and they cannot be moved.

CHAPTER 3
Triangulation-based World Space Partitioning

In this chapter we introduce a space partitioning technique that uses polygon triangulation (i.e., the decomposition of a polygon into triangles) to produce small regions that accommodate to the geography of the world. Our technique excludes the space occupied by obstacles from the partitioning to only consider space that can be occupied by players and objects. Triangles are used because it is the simplest shape that can divide any polygon and accommodate arbitrary polygonal obstacles.

Triangulation of a polygon is a well-studied problem in computational geometry, so the first step is to transform the game's world space into a polygon with holes. The contour of the polygon is formed from the limits of the world and obstacles are represented as holes. The conversion is relatively straightforward for world spaces that are planar (or can be mapped to a plane). Figure 3–1 shows the Mammoth world that has been mapped to a polygon with holes, it contains 155 rectangles that represent obstacles (i.e., walls). The polygon can be input to a state-of-the-art triangulation algorithm to produce a triangulation.

Triangulation of the world space has been used before in games for other concerns such as path finding [25]. In path finding, the representation of the planar space as a triangulation allows to efficiently find collision-free paths in the world space. However, the requirements for the triangulation in interest management and path finding are slightly different.

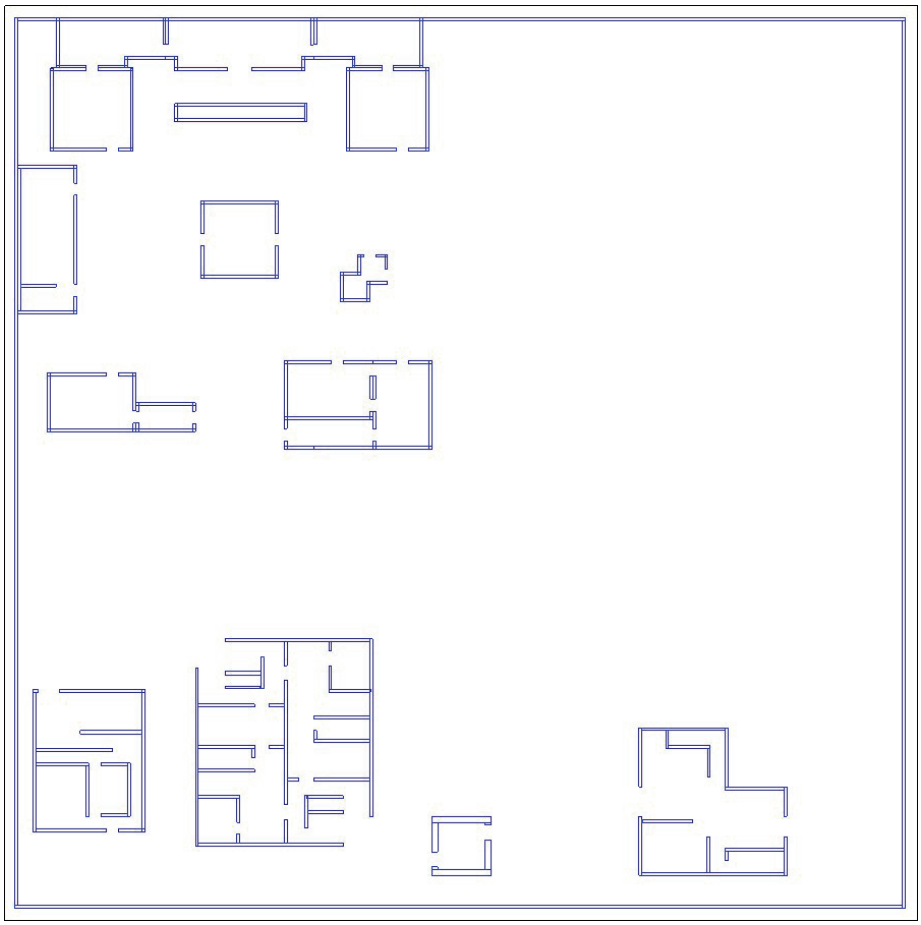


Figure 3–1: Mammoth world map as a polygon with holes.

In a triangulation being used for interest management, the existence of long, thin triangles is undesirable; the distance between two points in relatively flattened triangle can be much larger than for regular triangles, and so they are less appropriate for approximating a player's area-of-interest. For example, one endpoint of a flat triangle could be interesting to a player, but the other two points, also included as part of the area-of-interest approximation, could be very far from the player's interest. In this case the player would receive updates about objects that are neither

interesting nor soon to be discovered. Our choice of triangulation algorithm must reflect the requirement of avoiding long and thin triangles.

The partitioning of the world space for interest management is a process that is expected to be done offline or very infrequently, thus, performance issues with triangulation algorithms is not considered to be a concern.

The remainder of this chapter explains in details the technique we developed to partition the world into triangular regions that are suitable for interest management. In the remainder of this thesis, we will refer to those relatively small regions of the space that are suitable for interest management as *tiles*.

3.1 Delaunay Triangulation

There exists many types of triangulations, but Delaunay triangulations [16] have the property of maximizing the minimum angle in every triangle of the triangulation. Heuristically, this helps in avoiding thin triangles that are undesirable for interest management. We studied three variants of Delaunay triangulations before achieving an adequate partitioning: *conforming Delaunay*, *constrained Delaunay*, and *constrained conforming Delaunay*.

We performed a preliminary experiment using the Mammoth world map (see Figure 3–1) in order to assess the quality of triangles and find a triangulation that suits our requirements. Table 3–1 compiles a comparison of the different algorithms we tried. The second column (AvgMA) contains the average minimum angle (in degrees) of all the triangles in the triangulation, and the third column (MedianMA) contains the median. The fourth column (Count) contains the number of triangles

output by the triangulation. We will describe each entry in the table in the following text.

Table 3–1: Comparison of Triangulation Algorithms

Triangulation Alg.	**AvgMA**	**MedianMA**	**Count**
Conforming DT	12.064	8.130	6222
Constrained DT	16.603	13.707	1168
CCDT $a \leq 1.0$	36.062	40.481	2335
CCDT $a \leq 0.5$	40.219	43.290	3752
CCDT $a \leq 1.0$ and rm small	37.303	41.540	2225
CCDT $a \leq 1.0$ and rec to lines	38.077	41.973	1678

3.1.1 Conforming Delaunay

A conforming Delaunay triangulation is a triangulation that ensures all triangles are Delaunay, each triangle's circumcircle does not contain any other point of the polygon. The algorithm may add extra *Steiner points* in order to achieve a conforming triangulation. Steiner points are points that are not part of the triangulated polygon and are added by the algorithm to produce triangles that are Delaunay. Figure 3–2 shows a conforming Delaunay triangulation of the world map (shown in Figure 3–1). Although each triangle is Delaunay, the triangulation is clearly not suitable for interest management, it contains many long and skinny triangles. The median minimum angle of the conforming Delaunay triangulation (Conforming DT) confirms our observation, it is the smallest of Table 3–1 (i.e., 8.130). Furthermore, it outputs triangles of many different sizes. In particular, it outputs very large triangles in regions where there are no obstacles. This is not suitable for interest management since some triangles are too large to approximate the area-of-interest of a player.

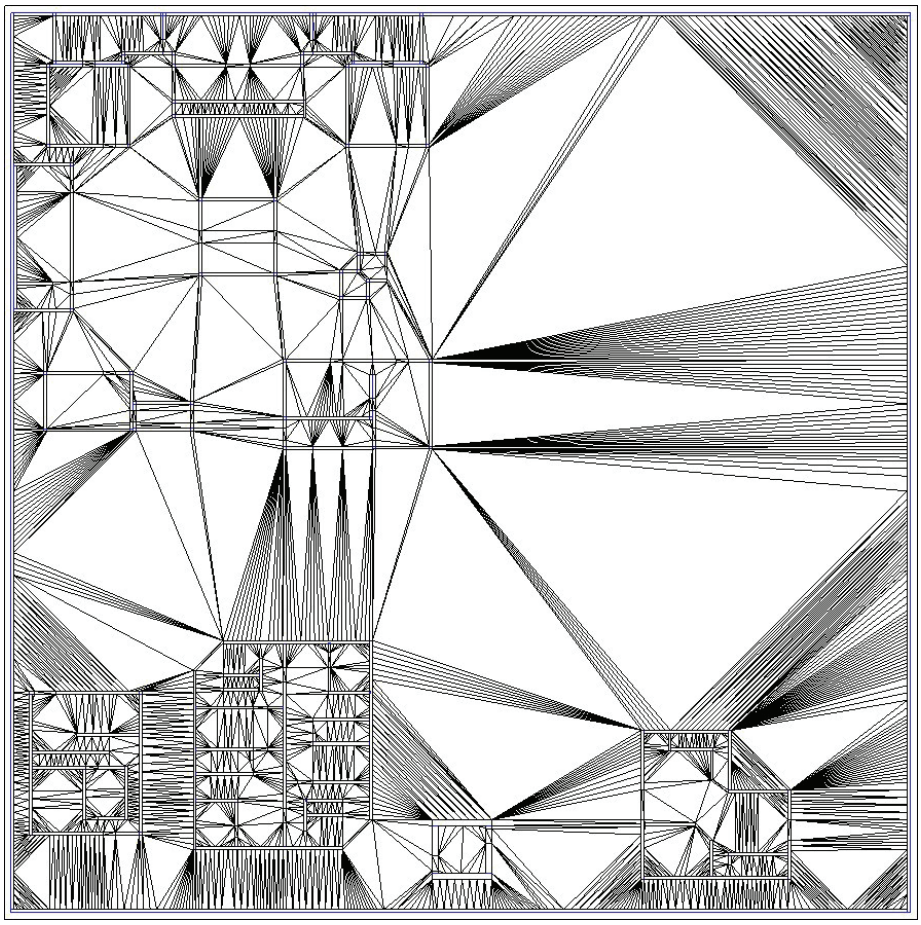


Figure 3–2: Conforming Delaunay triangulation of the map.

3.1.2 Constrained Delaunay

A constrained Delaunay is a triangulation in which each segment of the polygon is constrained to be part of the triangulation and that adds no extra Steiner points, but triangles are not guaranteed to be Delaunay. Figure 3–3 shows a constrained Delaunay triangulation of the world map.

The constrained Delaunay triangulation also has the problem that it outputs triangles of different sizes, large triangles in regions where there are no obstacles

are too large to approximate the area-of-interest of a player. The average minimum angle of constrained Delaunay triangulation (Constrained DT) in Table 3–1 shows an improvement over the conforming triangulation, but the minimum average angle is still very small.

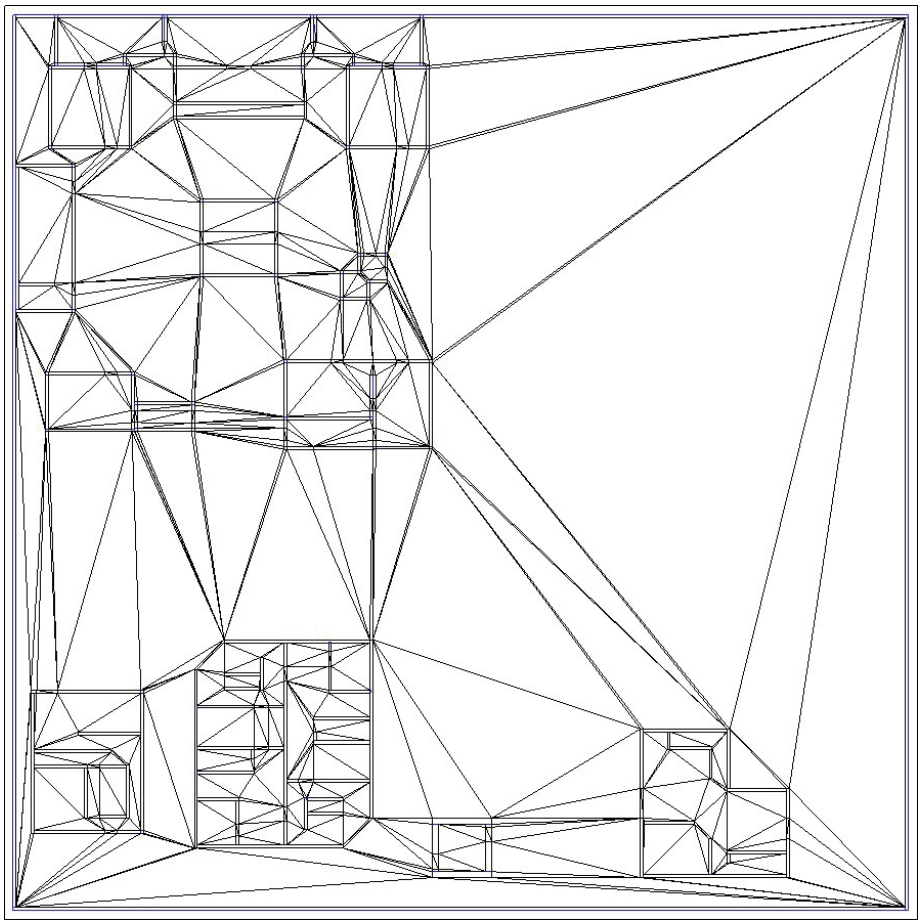


Figure 3–3: Constrained Delaunay triangulation of the map.

3.1.3 Constrained Conforming Delaunay

A constrained conforming Delaunay triangulation (CCDT) is a constrained Delaunay triangulation that uses extra Steiner points. We can also impose a constraint

on the maximum area of a triangle output by the triangulation such that we can explore fine or course-grained partitionings. Figure 3–4 shows the triangulation of the 30x30 world map with an area constraint of 1. The median minimum angle (CCDT $a \leq 1.0$) in Table 3–1 is three times better than for the constrained triangulation, a significant improvement.

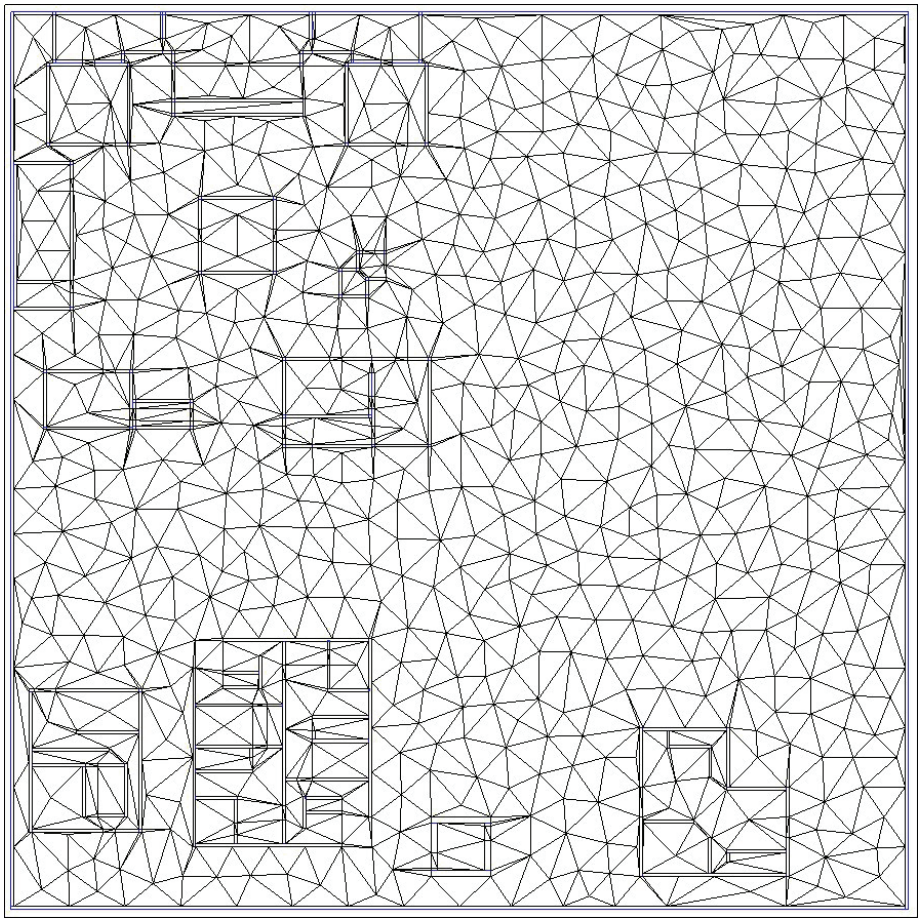


Figure 3–4: Delaunay triangulation of the world map with a maximum area constraint of 1.0.

By varying the area constraint we can produce a finer or courser-grained partitioning. Figure 3–5 shows the triangulation of the world map with an area constraint of 0.5. In Table 3–1 we can observe that by reducing the size of triangles, the average minimum angle increases. However, the number of triangles also increases and there is an overhead associated with a greater number of triangles. It is not clear a priori what would be an ideal maximum area for a triangular tile, we investigate this question in Chapter 6.

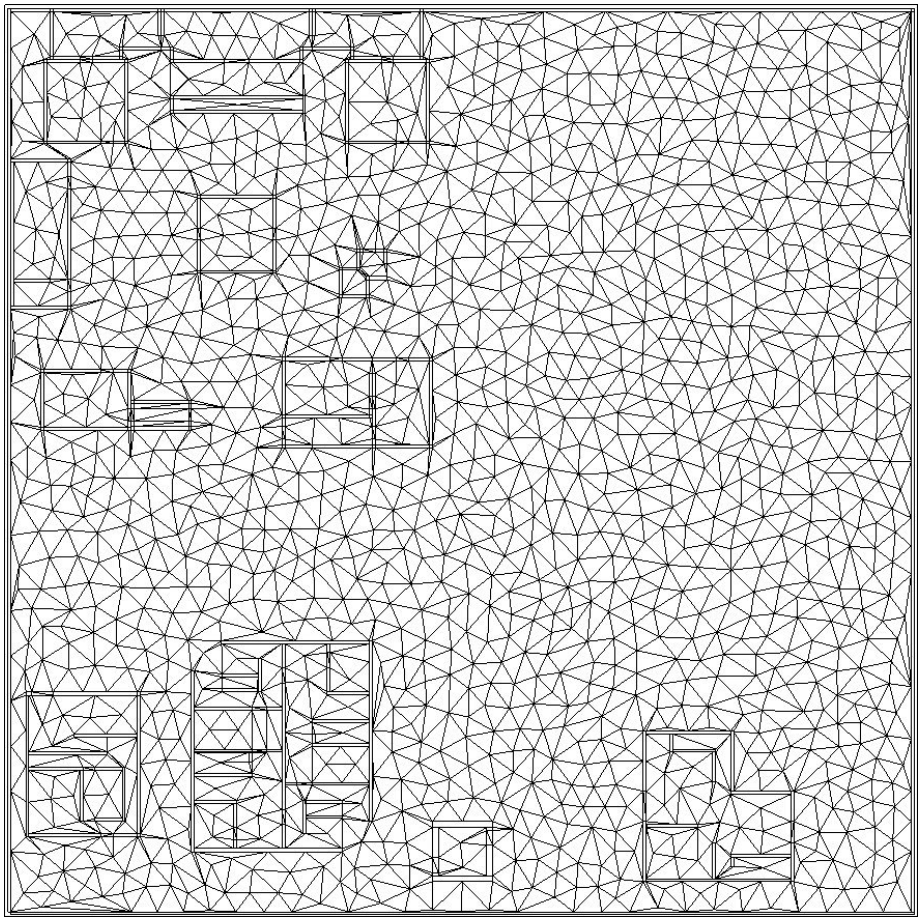


Figure 3–5: Delaunay triangulation of the world map with a maximum area constraint of 0.5.

3.2 Improving the Triangulation

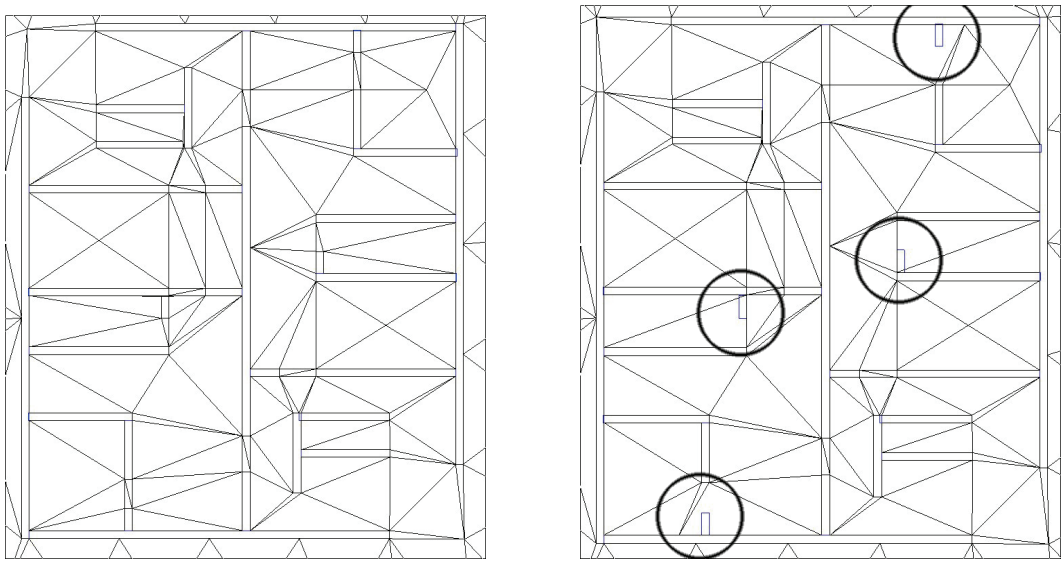
We found that constrained conforming Delaunay triangulation with an area constraint to be the most suitable to generate a triangulation useful for interest management. However, the triangulation still produces undesirable thin and long triangles. We investigated two additional preprocessing steps that could be used to eliminate a large portion of thin triangles and make the triangulation of better quality for interest management.

3.2.1 Remove Small Obstacles

The first preprocessing technique is to remove obstacles smaller than a parameterizable threshold. This is motivated by the fact that small obstacles do not occlude a significant part of the world. A telephone post for example would occlude a very small space, furthermore, as soon as an avatar moves slightly to the right or left it will see any object hidden behind the pole. Hence, from an interest management point of view, small obstacles are not really significant. Furthermore, triangulation with small obstacles results in some inconveniently small or thin triangles (since one edge has to follow the obstacle). Figure 3–6 illustrates the effect of removing obstacles with edge shorter than 0.3. We can see that it eliminates a few small triangles that had the small obstacles as an edge (circled in the figure). Table 3–1 (CCDT $a \leq 1.0$ and rm small) shows that removing small obstacles, does indeed reduce the number of thin triangles.

3.2.2 Convert Thin Rectangular Obstacles to Lines

Arbitrarily wide, but quite thin obstacles also reduce triangle quality since the algorithm generates thin triangles that have an edge on the shortest edge of a thin

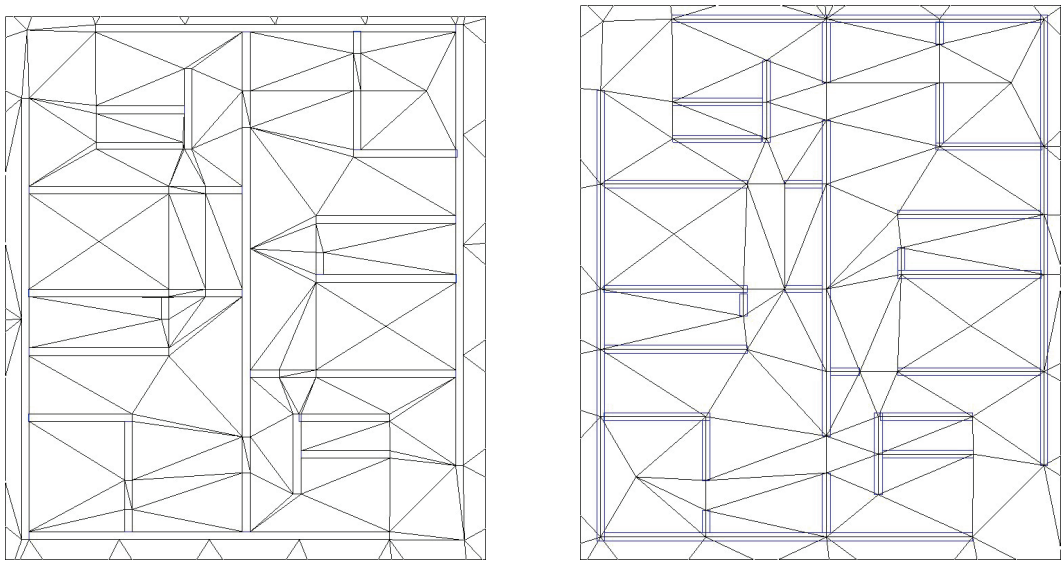


(a) With small obstacles **(b) Without small obstacles**

Figure 3–6: Removing obstacles smaller than 0.3.

rectangle. To avoid this problem, thin rectangular obstacles are converted to lines as a second preprocessing step. Figure 3–7 illustrates the advantage of converting thin rectangles to lines. Table 3–1 (CCDT $a \leq 1.0$ and rec to lines) also demonstrates a small gain in the quality of triangles. The table also shows a drop of 25% in the number of triangles. Since the surface covered by triangles is the same, it means that there is less small triangles, which reduce the overhead associated with a higher number of triangles. Some of the triangles of the resulting triangulation will overlap with the obstacles, but it is not a problem since the space occupied by an obstacle cannot be occupied by objects.

The algorithm we use to convert thin rectangles to lines is fairly straightforward (see Figure 3–8). First, we must fix a minimum edge size, if a rectangle has an edge shorter than this threshold, it is considered to be a thin rectangle and it will be

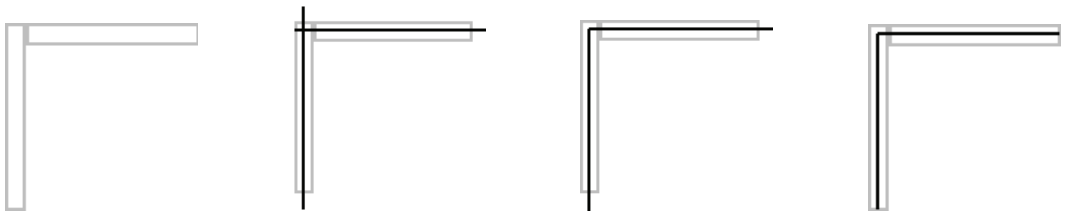


(a) With thin rectangles **(b) With lines**

Figure 3–7: Converting thin rectangles to lines.

converted into a line (Figure 3–8 a). For each thin rectangle a line is traced between the centers of the two short edges. The traced line is then temporarily extended at both of its ends in order to discover the intersections between lines (Figure 3–8 b). If two lines intersect, the algorithm replaces the previous endpoint of the line with the intersection point (Figure 3–8 c). Line extensions that do not intersect with another line are shortened back to their initial length (Figure 3–8 d).

The algorithm seems to generally work well to eliminate many thin triangles, however, there are cases in which it fails to eliminate them all. One such case is when we have two thin rectangles that are parallel and side-by-side (Figure 3–9 a). In this case, converting the two rectangles into a single line would eliminate thin triangles, but the algorithm converts them into two lines that are close to each other (Figure 3–9 b), and generates thin triangles (Figure 3–9 c). Furthermore, it may

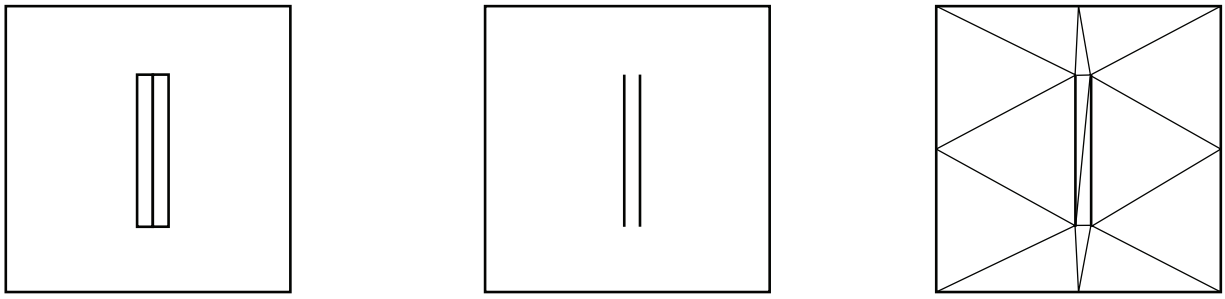


(a) **(b)** **(c)** **(d)**

Figure 3–8: Algorithm to convert thin rectangles to lines.

generate useless triangles in between the two lines that occupy a space that is non-existent in the world space.

To eliminate this preprocessing step along with its problems, it would also be possible to design the content editor of the game such that walls are drawn and encoded as lines rather than polygons.



(a) Two walls side-by-side **(b) Conversion to lines** **(c) Triangulated space**

Figure 3–9: Problem with two parallel side-by-side walls.

3.3 Storing and Using the Triangulation as a Graph

A triangulation algorithm will output a set of triangles. However, in order to be useful, we store the triangulation information as a neighbor graph. A *neighbor graph* is a graph in which each tile is a vertex and two vertices are connected by an

edge if the corresponding tiles are neighbors (i.e., have a point or an edge in common that is not on an obstacle). Using a graph to store the partitioned space has the main advantage that the problem of determining an area of interest can be done using a simple graph search. When using a breadth-first search rooted at the current tile of a player, only a localized subset of tiles that is proportional to the size of the player's area-of-interest is visited. Furthermore, the graph embeds information about occlusion of space: two regions that are close in Euclidean distance but separated by an obstacle are equally separated in the graph.

3.3.1 Building the Neighbor Graph

To build the neighbor graph, we first build the edge neighbor graph in which two tiles are connected if they have an edge in common. The pseudo-code of the construction of the graph is shown in Algorithm 1. The algorithm finds each pair of triangular tiles that have an edge in common, if the common edge does not overlap with any of the obstacles the two tiles are edge neighbors.

From the edge neighbors graph we can build a neighbor graph in which two tiles are connected if they have a point in common. The details of the algorithm are shown in Algorithm 2. The general idea is that for each tile we perform a breadth-first search in the edge neighbor graph until we found all other connected tiles that have a point in common with the tile.

The neighbors and edge neighbors of each tile are stored in two sets carried by the tile and can be retrieved efficiently. While the number of edge neighbors for a triangular tile is bound to three, tiles can have an unlimited number of point

neighbors. The neighbor relation is always symmetric, if the tile a is neighbor to tile b, tile b is neighbor to tile a.

Algorithm 1 Build Edge Neighbor Graph

```
T: triangles from the triangulation
E_O: edges of obstacles
for each t_i in T do
  for each t_j in T do
    if t_i and tj have an edge e_ij in common then
      if E_O does not contain an edge that is collinear with e_ij then
        create a neighbor edge between t_i and t_j in the graph
      end if
    end if
  end for
end for
```

3.3.2 Determining the Current Tile of an Object

To determine the current tile of an object we use an algorithm that computes the current tile of an object from the previous tile of the object. The algorithm first checks if the object has a previous tile associated with it, it might not if the object was recently added to the world. If the object does not have a previous tile, the algorithm performs a brute-force search that iterates through all the tiles of the world to find the current tile of the object. The brute-force search can be costly (i.e., $O(|T|)$ where T is the set of tiles), but it should be performed only once when the object is inserted into the world. The subsequent updates of the current tile of an object are performed using a breadth-first search from the previous tile. If updates are frequent enough, the search should usually be performed at only one level of depth, because the object would be in the same tile as the previous tile or in a neighboring tile. However, in the worst case the algorithm will search all the

Algorithm 2 Build Point Neighbor Graph

```
T: triangles from the triangulation
E_O: edges of obstacles
T_V: set of triangles that have already been visited
for each triangle t_i in T do
  initialize a queue Q
  initialize T_V
  enqueue the edge neighbors of t_i in Q
  add t_i to T_V
  while Q is not empty do
    t_q ← dequeue the first triangle from Q
    create a neighbor edge between t_q and t_i in the graph
    for each edge neighbor t_qn of t_q do
      if tile t_qn has a point in common with t_i then
        if T_V does not contain t_qn then
          enqueue t_qn into Q
          add t_qn to T_V
        end if
      end if
    end for
  end while
end for
```

tiles and take $O(|T| + |N|)$ where T is the set of tiles and N is the set of neighbor relations.

3.3.3 Searching the Neighbor Graph

Once constructed, we can determine the area-of-interest of a player using graph searches (e.g., depth-first, breadth-first, A*) rooted at the player's current tile. This technique has the advantage that only a local (to the player) fraction of the space will be visited by the algorithm. Furthermore, the notion of locality in the graph respects obstacles, two tiles separated by an obstacle will be equivalently separated in the graph.

3.3.4 Caching Information in the Neighbor Graph

Another advantage of the static partitioning is that it is possible to cache (or pre-compute) information such as the result of a search in the tile of the neighbor graph. The cached information can be re-used by other players that visit the same tile, and reduce the computation cost. We use this technique to improve the performance of some of our algorithms in the next chapter.

CHAPTER 4
Interest Management

In this chapter we describe eight interest management algorithms that we implement and compare in the later chapters. The eight algorithms can be separated into three distinctive categories: *proximity-based*, *visibility-based*, and *reachability-based*.

Proximity-based. Proximity-based interest management algorithms are algorithms that are based on the Euclidean distance between publishers and subscribers independently of the world's geography. Proximity-based algorithms do not take into account obstacles that may occlude parts of the world. We describe three algorithms that are based on proximity: Euclidean Distance (Section 4.1), Square Tiles (Section 4.2), and Hexagonal Tiles (Section 4.3). The Euclidean Distance algorithm is purely based on the Euclidean distance between objects while the other two are approximations that use a partitioning of the world into regions.

Visibility-based. Visibility-based algorithms try to leverage on the occlusion created by obstacles in the world. They restrict the area-of-interest to only the space that is visible to the player. We present two algorithms that are visibility-based: Ray Visibility (Section 4.4) and Tile Visibility (Section 4.5). Ray Visibility computes the exact visibility between each object; on the other hand, Tile Visibility does an approximation by precomputing the visibility between static regions.

Reachability-based. Reachability-based algorithms restrict the area-of-interest to the regions that are *reachably* close to the subscriber. It is similar to proximity-based algorithms, but reachability takes into account obstacles in its definition of distance between objects (i.e., the distance between a player and an object is the distance the player would travel to reach that object). Also, contrary to visibility-based algorithms, objects that are not visible (e.g., behind an obstacle) may be subscribed to if they are within a reachable distance. We describe three algorithms that are based on reachability: Tile Distance (Section 4.6), Tile Neighbor (Section 4.7), and Path Distance (Section 4.8). Tile Distance is somewhat of a hybrid between proximity-based and reachability-based; it considers the tiles that are within the radius of interest of a player, but that are also connected to the current tile of the player within the radius of interest. Tile Neighbor is purely based on the neighbor relationship between tiles, while Path Distance is based on the length of the paths between tiles.

For all algorithms we assume that a player can only see to a maximum distance from his current position, thus an expression of interest that corresponds to a radius around the player, the *interest radius*. Although our work could be extended to support changing interest radius (see future work in Chapter 7), in this work we assume that the interest radius of players is static.

4.1 Euclidean Distance Algorithm

The Euclidean Distance algorithm (see Figure 4–1) is a simple implementation of the aura-nimbus model (see Section 2.3.2). The area-of-interest is a circle around the position of the player with a radius that covers the maximum distance a player is

interested in, the aura is the position of the object. If the distance between an object and a player is smaller than the radius of the area-of-interest, the player subscribes to the object's updates.

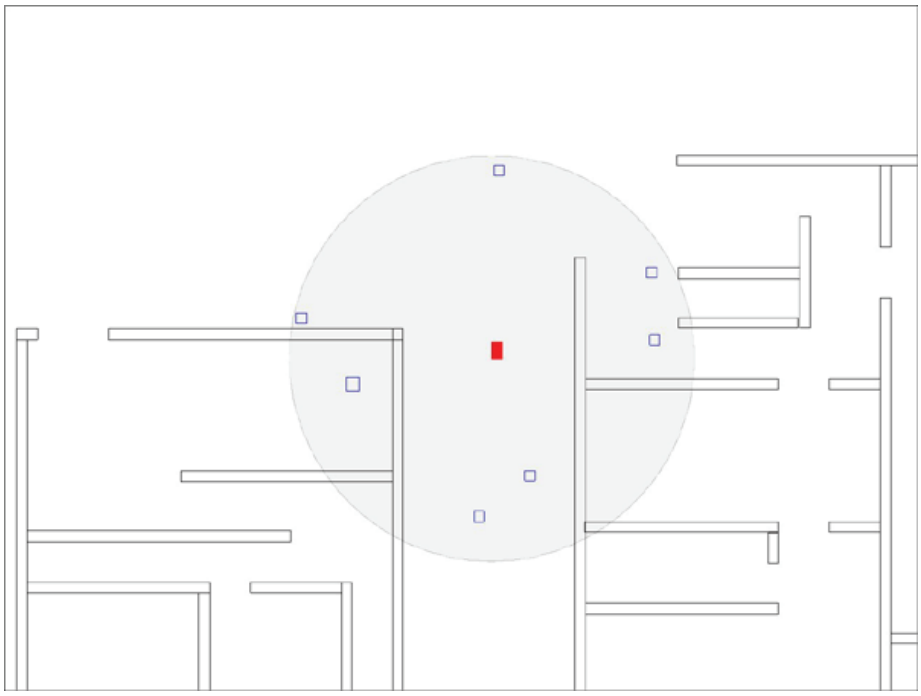


Figure 4–1: Euclidean Distance algorithm with interest radius of 2.0.

The pseudo-code of the Euclidean Distance algorithm is shown in Algorithm 3. The algorithm iterates through each subscriber/publisher pair in the interest management domain and computes the Euclidean distance between their centers. If the distance is smaller than the interest radius of the subscriber, the subscriber is subscribed to the updates of the publisher; if it is greater, the subscriber is unsubscribed from the publisher.

The main advantages of this algorithm are that it is easy to implement and computing the Euclidean distance between two points is inexpensive. The main disadvantage is that the algorithm must compute the distance between all pairs of subscribers and publishers in the space. The time complexity of this naive algorithm is $O(|S||P|)$ where S is the set of subscribers and P is the set of publishers. As the number of objects increases in the game, the algorithm does not scale well (see Chapter 6). However, the algorithm could possibly be combined with other techniques such as the one used in the following tile-based algorithms to achieve better time complexity. Another disadvantage is that objects behind obstacles are discovered by players even though they are irrelevant. For example, in Figure 4–1 there is an object inside the building on the bottom left that is discovered by the player, but it is irrelevant since the object is not visible or soon to be visible to the player.

4.2 Square Tiles Algorithm

The Square Tiles algorithm (see Figure 4–2) is a region-based interest management that divides the world into equal-sized squares. The size of squares is set according to the radius of interest of players. At any location, the subscriber is interested in at most nine tiles, the subscriber's current tile and the eight (or less) neighboring tiles. The algorithm can be implemented as Algorithm 6 (described in Section 4.7) in which the $maxDepth$ is fixed to 1. Whenever a player performs an action, the action is broadcast to all players subscribed to the square in which the action has taken place.

The Square Tiles algorithm scales well as the complexity of the computation to determine the area of interest is constant. However, it is a rather bad approximation

Algorithm 3 Euclidean Distance Algorithm

```
S: set of subscribers
P: set of publishers
L_s: set of subscriptions for subscriber s
for each subscriber s_i in S do
  for each publisher p_j in P do
    if the distance between s_i and p_j is smaller than interest radius of s_i then
      if L_{s_i} does not contain p_j then
        subscribe s_i to p_j
        add p_j to L_{s_i}
      end if
    else
      if L_{s_i} contains p_j then
        unsubscribe s_i from p_j
        remove p_j from L_{s_i}
      end if
    end if
  end for
end for
```

of the radius of interest of the player (see Figure 4–2), and it also does not take obstacles into consideration.

4.3 Hexagonal Tiles Algorithm

The Hexagonal Tiles algorithm (see Figure 4–3) divides the world into equal-sized, regular hexagons. A player subscribes to objects in the tiles that intersect its radius of interest. Algorithm 5 shows how the hexagonal tile algorithm can be implemented using a breadth-first search. For each subscriber we perform a search from its current tile to find all the tiles that are contained or that intersect with the subscriber's radius of interest. The subscriber subscribes to all publishers contained within those tiles. The set of "interesting" objects is also compared against the

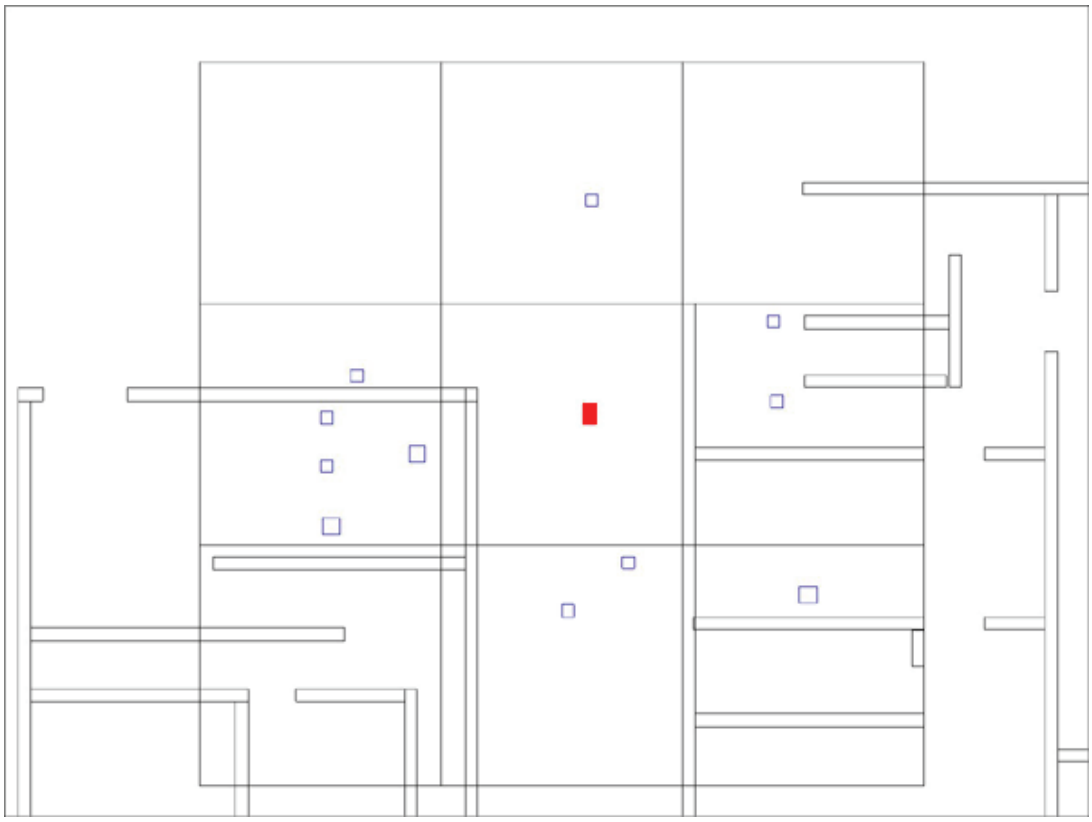


Figure 4–2: Square Tiles algorithm with tiles of size 2.0 and one neighbor.

subscription set to unsubscribe from objects that are no longer interesting. The algorithm could also be implemented using a depth-first search.

Although the worst case of the algorithm has a time complexity of $O(|T| + |N|)$ where T is the set of tiles and N the neighbor relations between the tiles, in practice the time complexity of the algorithm will be proportional to the radius of interest of the subscribers. Hexagonal tiles are known to be a good approximation of a player's circular area-of-interest [20]; it will be a good benchmark against which to compare the triangle-based algorithms.

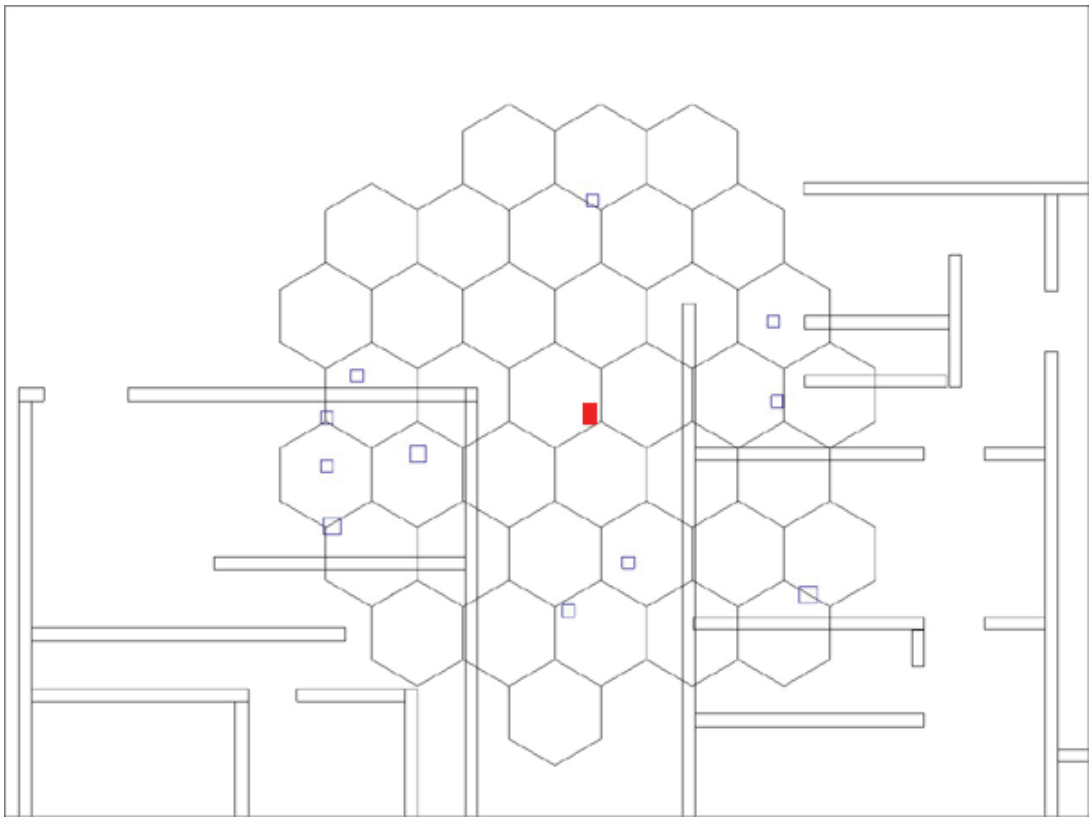


Figure 4–3: Hexagonal tiles of area size 1.0 with interest radius of 2.0.

4.4 Ray Visibility Algorithm

When using Ray Visibility, the only objects of interest are those that a player sees (see Figure 4–4). To determine if an object is visible to a player, we trace a line from the position of the player to the position of the object, up to a maximum length. If the line does not intersect with any obstacle in the world, the two objects are visible to each other. We assume that an object is visible or not by a player based on their center points only, an object cannot be partially visible. The details of the algorithm are shown in Algorithm 4.

Algorithm 4 Ray Visibility Algorithm

```
S: set of subscribers
P: set of publishers
L_s: set of subscriptions for subscriber s
O: the set of obstacles
for each subscriber s_i in S do
  for each publisher p_j in P do
    if the distance between s_i and p_j is smaller than interest radius of s_i then
      create a ray r with endpoints at s_i and p_j
      blocked ← false
      for each obstacle o_k in O do
        if r intersects with o_k then
          blocked ← true
        end if
      end for
      if not blocked then
        if L_{s_i} does not contain p_j then
          subscribe s_i to p_j
          add s_i to L_{s_i}
        end if
      else
        if L_{s_i} contains p_j then
          unsubscribe s_i from p_j
          remove s_i from L_{s_i}
        end if
      end if
    else
      if L_{s_i} contains p_j then
        unsubscribe s_i from p_j
        remove s_i from L_{s_i}
      end if
    end if
  end for
end for
```

Ray Visibility is in a sense the perfect interest management algorithm, since it accurately calculates the exact area of interest of a player at a given point in time. It therefore provides the lower bound on the number of messages that must be exchanged between players, and an indication of the cost of a relatively expensive interest calculation. Note, however, that optimal visibility is not always desired: to prevent slow game play caused by network latency, it is often recommended to *pre-fetch* information about objects that are "soon to be discovered," if perhaps not actually visible yet. The inflexibility of ray visibility in this respect means that in practice it is more subject to problems such as the missed interaction problem [30] or late discovery of objects.

The Ray Visibility algorithm has a time complexity of $O(|S||P||O|)$ where S is the set of subscribers, P is the set of publishers, and O is the set of obstacles in the world. The algorithm is computationally expensive and unless the number of objects and obstacles in the game is small, the algorithm does not scale (see Chapter 6).

4.5 Tile Visibility Algorithm

The Tile Visibility algorithm (see Figure 4–5) is based on the visibility between tiles. The algorithm involves a precomputing step during which the visibility between each pair of tiles is computed. A tile is considered visible from another tile if there exist a point in each of the two tiles that can be connected by a line segment that does not intersect an obstacle. The algorithm takes advantage of the fact that the visibility between tiles is a static property as opposed to the visibility of players that changes dynamically with their position. The algorithm approximates the visibility of a player by selecting the tiles that are visible from the player's current tile.

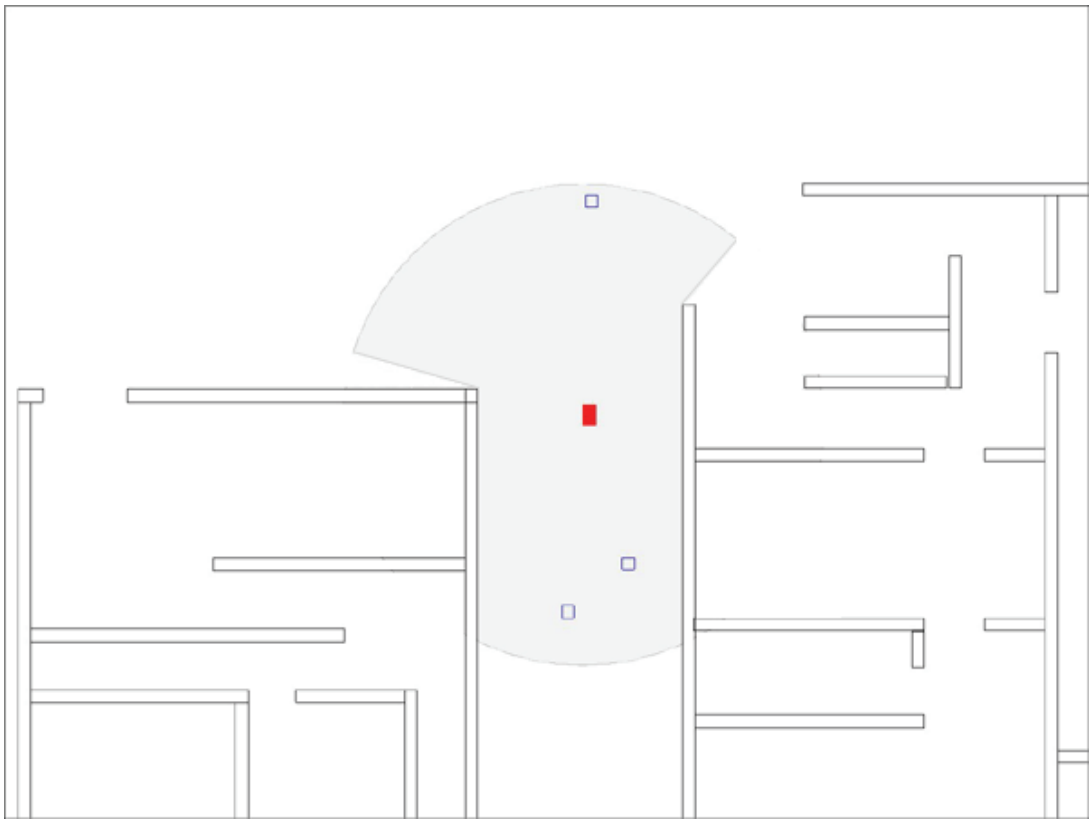


Figure 4–4: Ray Visibility algorithm with interest radius of 2.0.

Determining the set of tiles visible from a given tile is a computationally expensive operation, it is the reason why it must be precomputed. Here we outline an algorithm that computes the visibility between tiles using *weak visibility polygons*. A weak visibility polygon is the area that is visible from an edge inside of a polygon [10]. The intuition behind our algorithm is that two tiles are visible to each other if the weak visibility polygon of one of the edges of one tile touches the other tile. Our algorithm uses the algorithm by Suri and O'Rourke [40] to compute weak

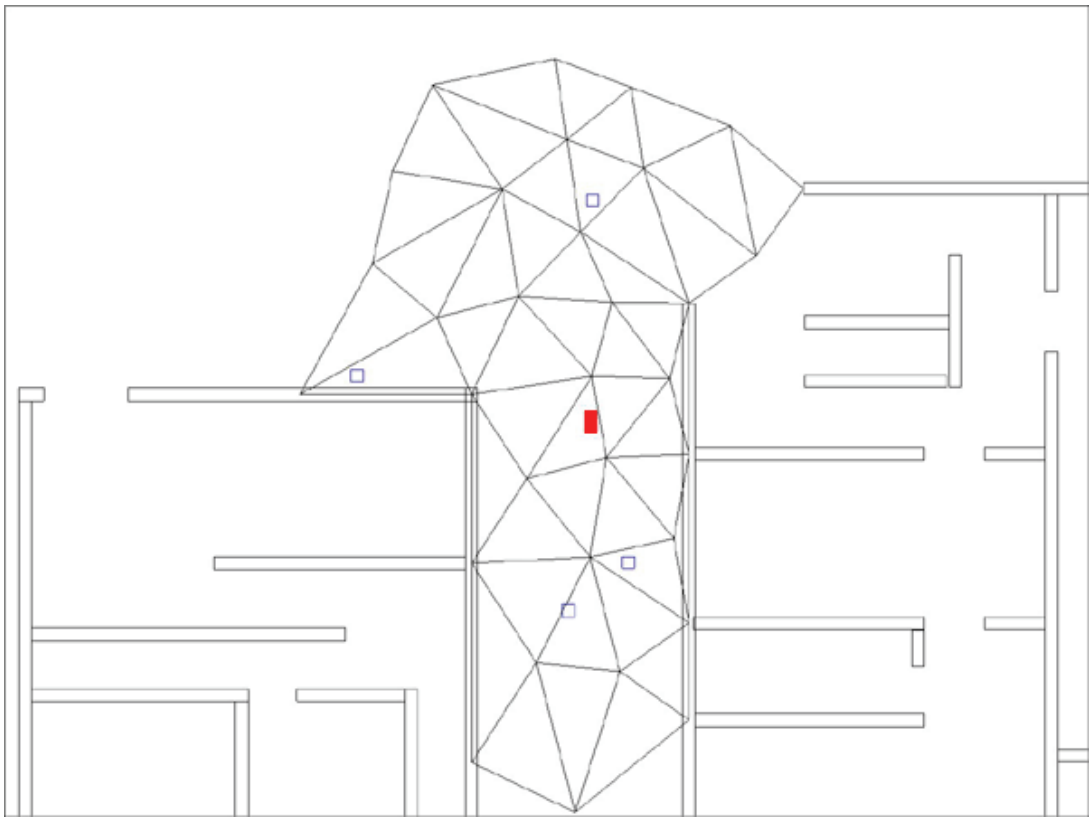
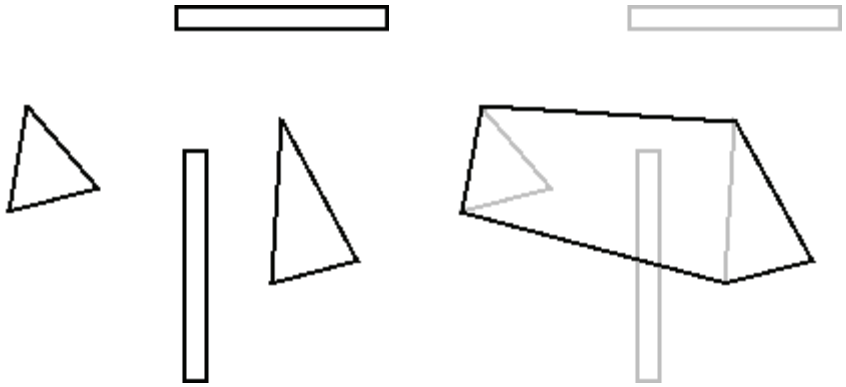


Figure 4–5: Tile Visibility algorithm with interest radius of 2.0.

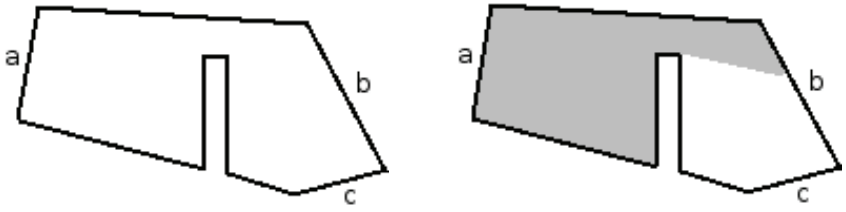
visibility polygons. Our algorithm was not designed to be optimal, but for ease of implementation.

The algorithm considers each pair of triangular tiles in the space, for example, Figure 4–6 (a) considers one such pair of triangular tiles. The first step of the algorithm aims to simplify the problem for that pair of triangles. We compute the convex hull around the two triangles as the boundary for the problem, if the two triangles see each other it will be within that convex hull (see Figure 4–6 b). Then we find all obstacles in the world that are contained or intersect the convex hull.

We eliminate the portion of the intersecting obstacles that lies outside the convex hull (see Figure 4–6 c). The resulting polygon is relatively simpler than the polygon of the whole world. In the reduced polygon, for each edge of the first tile that lies on the convex hull (edge a in the figure), we compute the weak visibility polygon using Suri and O'Rourke algorithm. If the visibility polygon touches one of the edges of the second tile (edge b or c in the figure), the tiles are visible to each other. In Figure 4–6 (d), the weak visibility polygon from edge a is shown as a shaded area, since it touches edge b the two tiles are visible to each other.



(a) Two triangular tiles with two obstacles (b) Trace the Convex Hull



(c) Simplify the Polygon (d) Visibility from edge a

Figure 4–6: Computing the visibility between two tiles.

Once the tile visibility for each tile has been precomputed and stored, a player's area-of-interest corresponds to the set of tiles visible from the tile it occupies. This information can be retrieved in constant time. The main disadvantages of the Tile Visibility algorithm is the expensive precomputing step and the fact that it is considerably more difficult to implement than other algorithms.

4.6 Tile Distance Algorithm

The Tile Distance algorithm (see Figure 4–7) is based on the Euclidean distance between a player and a triangular tile. The set of tiles of interest for a player is computed as the set of tiles connected to the current tile of the player that intersect the player's radius of interest. The algorithm (shown in Algorithm 5) implements a breadth-first search from the player's current tile. The Tile Distance algorithm is an approximation of the player's radius of interest, but also has the property that tiles that are not reachable within the player's area-of-interest are ignored. We can see this property of the algorithm in Figure 4–7: the tiles inside the building on the right are visible because they are connected to the player's current tile. On the other hand, the tiles inside the building on the left are not chosen because there is no path within the interest radius connecting them to the player. The algorithm has the same time complexity as described for the Hexagonal Tiles algorithm (see Section 4.3).

4.7 Tile Neighbor Algorithm

The Tile Neighbor algorithm (see Figure 4–8) determines tiles of interest for a player based on neighbor relationships between tiles. The algorithm performs a breadth-first search from the current tile of the player, and collects all the tiles

Algorithm 5 Tile Distance Algorithm

```
S: set of subscribers
L_s: set of subscriptions for subscriber s
Q: a queue
for each subscriber s_i in S do
    P_{s_i}: initialize empty set of publishers
    T_V: initialize empty set of tiles that have already been visited
    enqueue the current tile of s_i in Q
    add the current tile of s_i to T_V
    while Q is not empty do
        t ← dequeue the first tile from Q
        for each publisher p_t on tile t do
            if L_{s_i} does not contain p_t then
                subscribe s_i to p_t
                add p_t to L_{s_i}
            end if
            add p_t to P_{s_i}
        end for
        for each neighbor t_n of tile t do
            if T_V does not contain t_n then
                if the radius of interest of s_i intersects t_n then
                    enqueue the tile t_n in Q
                end if
                add the tile t_n to T_V
            end if
        end for
    end while
    for each publisher p_j in L_{s_i} do
        if P_{s_i} does not contain p_j then
            unsubscribe s_i from p_j
            remove p_j from L_{s_i}
        end if
    end for
end for
```

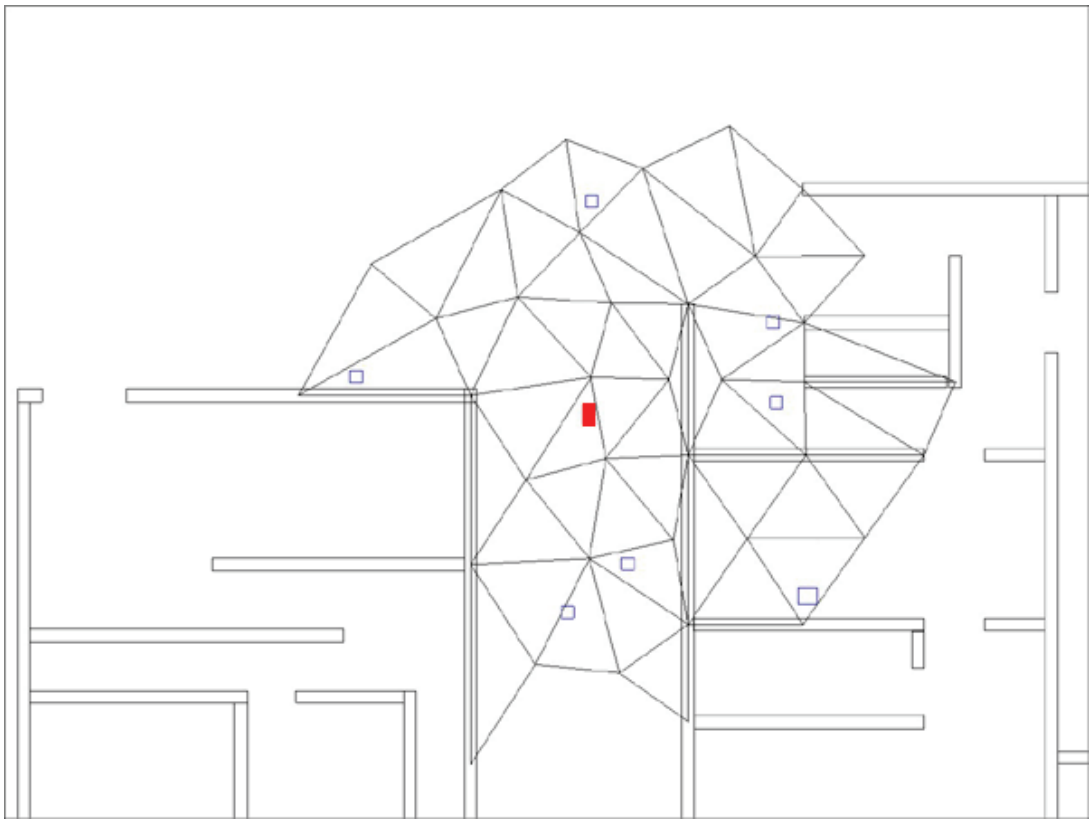


Figure 4–7: Tile Distance algorithm with interest radius of 2.0.

until it reaches a parameterized depth. The details of the algorithm are shown in Algorithm 6. For example, for a depth of one, the algorithm will only collect the immediate neighbors of the current tile. Figure 4–8 shows an example with a depth of three (the depth of each tile is indicated by a number).

The main advantage of this algorithm is that it is very simple to compute and the time complexity of the algorithm is proportional to the search depth (the radius of interest). The disadvantage is that since the shape and size of triangles is not

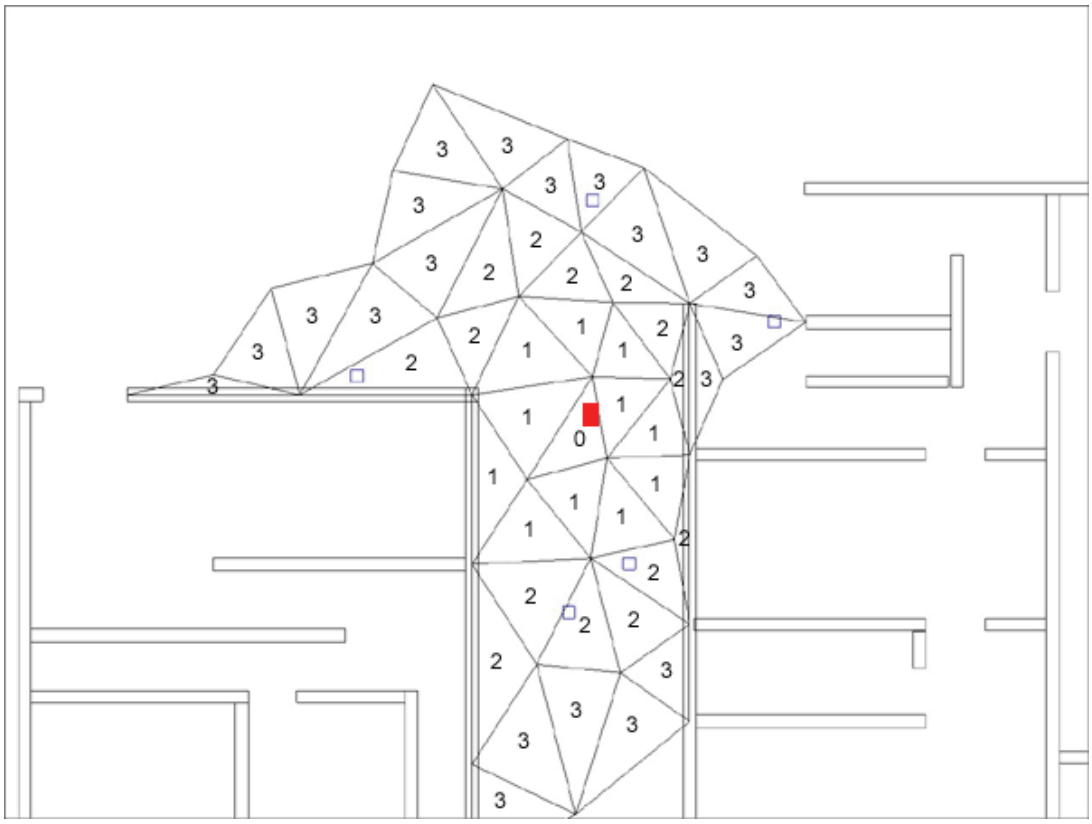


Figure 4–8: Tile Neighbor algorithm with a depth of 3 neighbors.

uniform, it is difficult to predict if a given depth will fully cover the radius of interest of a player.

4.8 Tile Path Distance Algorithm

The Tile Path Distance algorithm (see Figure 4–9) is somewhat similar to the neighbor algorithm, but instead of taking the graph depth as a limiter for its search, it takes the shortest-path distance. We define the *path distance* between two tiles as the sum of the distances between the centers of the triangles connecting the two tiles. The intuition behind the algorithm is that the subscriber is interested in the

Algorithm 6 Tile Neighbor Algorithm

```
maxDepth: the depth of neighbors to stop the search
S: set of subscribers
L_s: set of subscriptions for subscriber s
Q: a queue
for each subscriber s_i in S do
  P_{s_i}: initialize empty set of publishers
  T_V: initialize empty set of tiles that have already been visited
  enqueue depth marker 0 and the current tile of s_i in Q
  add the current tile of s_i to T_V
  while Q is not empty do
    if the first element of Q is a depth marker then
      depth ← dequeue(Q)
    end if
    t ← dequeue the first tile from Q
    for each publisher p_t on tile t do
      if L_{s_i} does not contain p_t then
        subscribe s_i to p_t
        add p_t to L_{s_i}
      end if
      add p_t to P_{s_i}
    end for
    if depth < maxDepth then
      for each neighbor t_n of tile t do
        if T_V does not contain t_n then
          enqueue a depth marker of depth + 1
          enqueue the tile t_n in Q
          add the tile t_n to T_V
        end if
      end for
    end if
  end while
  for each publisher p_j in L_{s_i} do
    if P_{s_i} does not contain p_j then
      unsubscribe s_i from p_j
      remove p_j from L_{s_i}
    end if
  end for
end for
```

tiles within a certain "reachable" distance from his current position. The algorithm is an approximation of that distance.

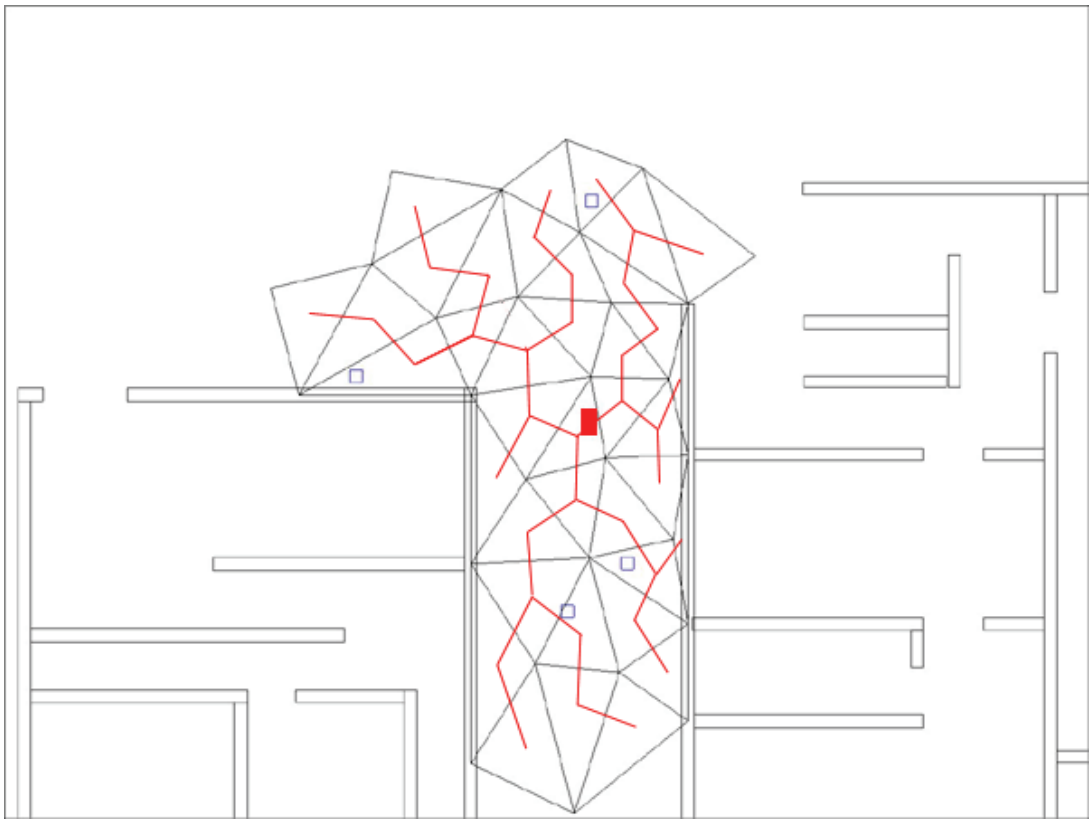


Figure 4–9: Path Distance algorithm with a distance of 3.0.

The details of the algorithm are shown in Algorithm 7. The algorithm performs an A* search in which paths are ordered by their length in a priority queue. Each path corresponds to a channel of connected tiles for which the length is computed from center to center. Figure 4–9 illustrates the shortest paths from the player's current tile to other tiles. When a path is dequeued and processed, a new path is created for each edge neighbor of the last tile on the path. The reason for taking

only edge neighbors rather than point neighbors is that we want to make sure we do not compute distances across obstacles. When a path is longer than the fixed length, the path is discarded.

The algorithm has the same problem as the Neighbor algorithm, i.e., it cannot guarantee to fully cover the radius of interest of a player. However, since it uses a criteria based on the distance, it is possible to determine qualitatively or heuristically a reasonable path distance for a given interest radius, and it tolerates cases of abnormal triangles. This property can be observed by comparing Figure 4–8 and 4–9; the path distance algorithm has a more "roundish" shape, for instance it cuts out the thin triangle on the left side along the wall.

The time complexity of the Path Distance algorithm is also proportional to the radius of interest of the subscribers. However, since the Path Distance algorithm uses static information (such as the distance between neighboring triangle centers), the results of searches can be cached (or even precomputed) for all players which allows for improved performance.

Algorithm 7 Path Distance Algorithm

```
S: set of subscribers
L_s: set of subscriptions for subscriber s
Q: a priority queue sorted by path length
for each subscriber s_i in S do
  P_{s_i}: initialize empty set of publishers
  T_V: initialize empty set of tiles that have already been visited
  add the current tile of s_i to an empty path g
  enqueue g in Q
  while Q is not empty do
    g ← dequeue the shortest path from Q
    t ← the last tile on path g
    if T_V does not contain t then
      for each publisher p_t on tile t do
        if L_{s_i} does not contain p_t then
          subscribe s_i to p_t
          add p_t to L_{s_i}
        end if
        add p_t to P_{s_i}
      end for
      add the tile t to T_V
      for each edge neighbor t_n of tile t do
        if T_V does not contain t_n then
          g_c ← clone path g
          add tile t_n to g_c
          if the length of g_c is less than maxPath then
            enqueue g_c in Q
          end if
        end if
      end for
    end if
  end while
  for each publisher p_j in L_{s_i} do
    if P_{s_i} does not contain p_j then
      unsubscribe s_i from p_j
      remove p_j from L_{s_i}
    end if
  end for
end for
```

CHAPTER 5
Experimental Setting and Implementation

In this section we describe the environment and methodology we used to perform experiments with interest managements. The goal of the experiments was threefold:

- Evaluate and compare the use of different interest management algorithms in an MMOG-like setting.
- Evaluate the feasibility of using triangulation-based tiling to perform obstacle-aware interest management.
- Compare results obtained from real-player traces with results from randomly generated traces.

The eight algorithms described in Chapter 4 were implemented within an MMOG developed at McGill University, the Mammoth framework. Each experiment consisted of a replay of trace data collected from either the movements of real-players playing a non-trivial multiplayer game (*Orbius,* described below), or by using artificial, randomly generated data.

In the first section of this chapter we introduce the Mammoth framework and its architecture. Then we describe how interest management was abstracted and integrated into the Mammoth framework. The next section explains how the real player data was collected in a gaming event consisting of 28 participants. The last two sections explain how the random traces were generated and how the measurements were conducted in our experiments respectively.

5.1 Mammoth Software Architecture

The Mammoth framework was developed by a group of students prior to the implementation of the work in this thesis; however, the framework was considerably refactored in the course of our work in order to achieve better modularity and scalability. Figure 5–1 shows an overview of the refactored Mammoth software architecture. The software architecture is divided into two dimensions: layers and services. *Layers* provide the first degree of separation of concerns; they separate the main concerns, such as the graphical interface, the application, the network and the data storage. However, these concerns are very broad for the application layer, and we need a finer-grained level of separation of concerns: services. A *service* implements a specific concern of the system that can span multiple layers. For example, game functionalities is one of the most important concern of the system. Session management is another example of a concern in the system.

The described architecture of Mammoth implements the client-server paradigm. However, the software architecture of Mammoth leaves the flexibility to extend the framework to other paradigms such as peer-to-peer.

Layers are the first degree of separation of concerns of the architecture. Although the implementation of these concerns may change, layers represent important commitments. For example, commitment to a particular graphic library, network library or data storage. Layers may only interact with their adjacent layers. Layers interact through well-defined interfaces, and a layer should not be dependent on a particular implementation of another layer. Layers contribute to providing better modifiability by localizing the main concerns and limiting the scope of their interactions with the

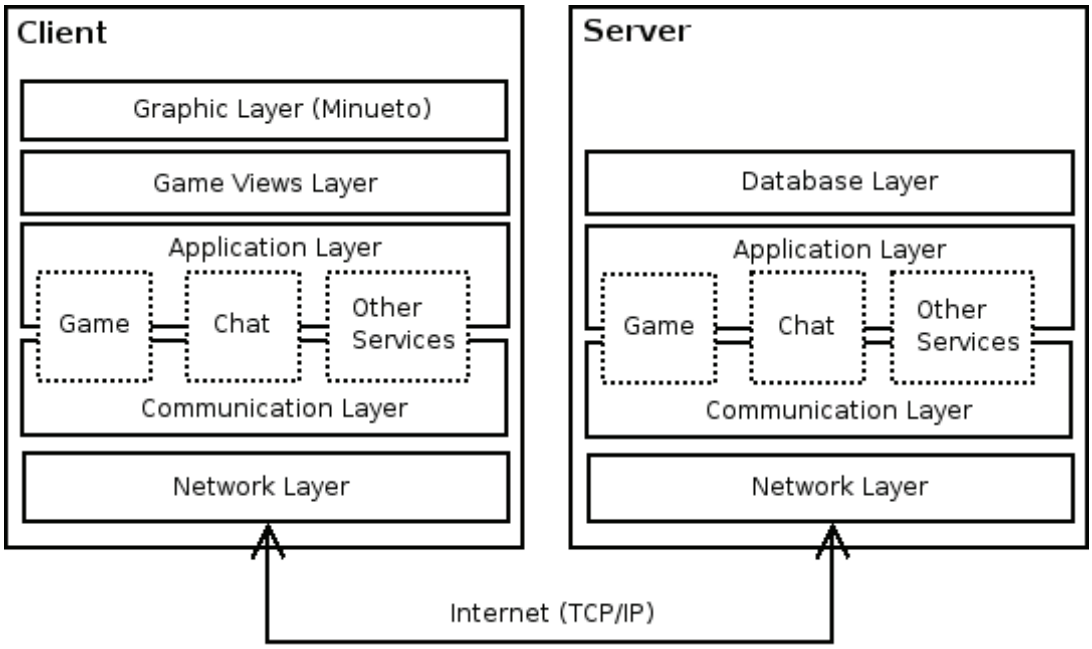


Figure 5–1: Mammoth software architecture

remainder of the system. Furthermore, they decouple the external libraries from the implementation of the application layer. We will describe each of the layers that compose the architecture.

Graphic Layer. The graphic layer is responsible for displaying the graphics to the end user. Mammoth uses the *Minueto GL* graphic library as its graphic layer. Minueto GL is a simple and intuitive 2D graphic library that has been designed for undergraduate student game development [18, 17].

Game Views Layer. The game views layer is a mediating layer between the application and the graphic layer. It uses the graphic layer to display the objects of the application layer. It also observes changes in the application layer and reflects these changes, using the graphic layer, in the view presented to the player.

Application Layer. The application layer implements all the logic related to the game. It includes the game functionalities as well as the application logic of all the services that contribute to the system.

Communication Layer. The communication layer implements the logic related to the communication of the game. It knows about the network layer and the application layer. Communication is done through messages. The communication layer is responsible for handling and translating the messages to the application layer. It is also responsible for creating messages, determining the destination and sending them by using the network layer.

Network Layer. The network layer implements lower level primitives for communication between the nodes in the system. It is not specific to the game, but implements an interface that is used by the communication layer. The network layer provides synchronous and asynchronous communication through virtual channels that are implemented over TCP using Java NIO [2].

Database Layer. The database layer provides persistence to the data of the application layer. It is used by the application layer to save and load data from persistent storage. It knows about the persistent data of the game.

5.2 Implementation of Interest Management in Mammoth

In order to investigate different interest management strategies, we abstracted and integrated interest management as a separate service in the application layer of Mammoth, the *Replication Engine*. The Replication Engine is responsible for replicating and updating the state of game objects across computers according to an interest management policy. Figure 5–2 shows an high-level overview of the

Replication Engine. The Replication Engine has one or more replication spaces, each representing a different interest management domain with a different interest management policy. A replication space controls the replication across computers and manages the propagation of update events for a set of objects in the game state. The Replication Engine assumes a network layer that provides group communication facilities such as channels. The network primitives required by the Replication Engine are abstracted by the Communication Strategy; our current implementation uses Mammoth's Network Engine (i.e., Network Layer) as the underlying network library. The game services register the mutable objects of the game state (e.g., players and items) with a replication space; when the state of a game object changes it publishes an update event to a replication space of the Replication Engine which is responsible for propagating the event. Below we explain the role of each component in more details.

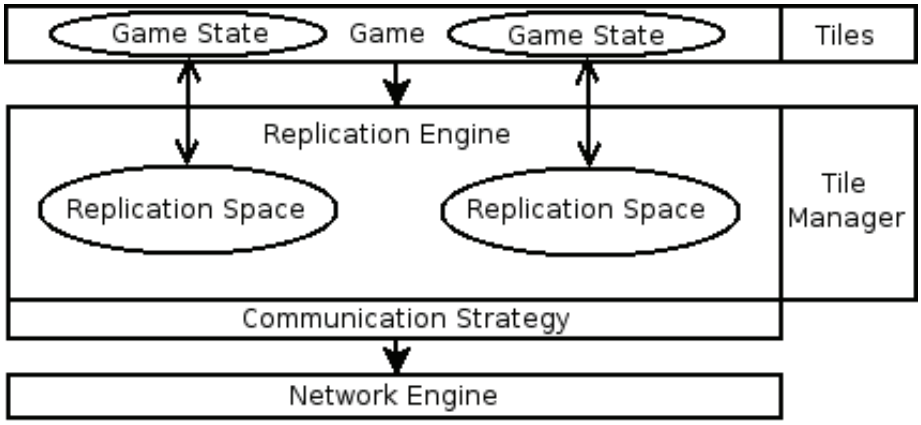


Figure 5–2: Replication Engine

5.2.1 Replication Space

The most important abstraction of the Replication Engine is the Replication Space. A *replication space* represents an interest management domain in which objects can discover and be discovered by other objects based on a programmer-defined, game-specific interest management policy.

For a defined replication space, each computer in the system has a local instance of the replication space which serves as a container for local objects' replicas. For example, in the client-server architecture the server has a copy of all objects in the replication space and the clients only have the subset of the objects relevant to their gaming experience. Figure 5–4(a) shows the replication space instances of a server and two clients. In Figure 5–3, 5–5, and 5–6 we outline our implementation of the replication space. We will discuss each element of the implementation in more details in the next paragraphs.

Subscribers and publishers. Within a replication space, objects can be assigned two roles: subscriber and/or publisher. A *subscriber* is an object that can discover other objects and subscribe to their update events (i.e., players). A *publisher* is an object that can be discovered by subscribers and that publishes update events (i.e., players and items). Each publisher is assigned a network channel on which it publishes its events and a subscriber subscribes to a channel if it is interested in the object. A publisher can be replicated in multiple replication space instances, but only one instance *owns* the publisher; the owner of the publisher is the authority over its state and is solely responsible for creating replicas of the publisher and broadcasting update events. In a simple client-server architecture such as the

one of Figure 5–4 and the one that we implemented, the server's replication space instance would be the owner of all publishers. Figure 5–3 shows the methods to add a subscriber or a publisher to the replication space (i.e., `addSubscriber` and `addPublisher` respectively).

```
class ReplicationSpace
{
   Map subscriptions = new Map   // Mapping of a subscriber to a set of publishers
   Set subscribers = new Set     // Set of subscribers.
   Set publishers = new Set      // Set of publishers.

   addSubscriber(Object subscriber) {
      subscribers.add(subscriber)
      subscriptions.add(subscriber, new Set)
   }

   addPublisher(Object publisher) {
      if(publisher.isOwned()) {
         // Assign a communication channel to the publisher
         CommunicationStrategy.initializeChannel(publisher)
      }
      publishers.add(publisher)
   }

   ...
```

Figure 5–3: Replication space implementation.

Discovery mechanism. A replication space is responsible for determining if there are publishers matching the interest of subscribers in the space. When a subscriber is interested in a publisher, the object is replicated through a *content message* that is sent over the network to the subscriber's replication space instance. The subscriber is also subscribed to the network channel of the publisher. Figure 5–5 shows our implementation of the replication space's method that performs the subscription (i.e., `subscribe`). The method first checks that the subscriber does not already have a subscription to the publisher; then, the subscriber is subscribed to

the channel of the publisher and a replica of the publisher is sent to the subscriber through a content message.

Figure 5–4 illustrates the full discovery process in a replication space. The server replication space has two subscribers (S1 and S2), each corresponding to a client's local replication space copy, and three registered publishers (P1, P2, P3). In Figure 5–4 (b), the server detects that according to its interest management policy the subscriber S1 is interested in the publisher P2. The server's replication space sends a replica of the object P2 to the replication space of S1, and subscribes S1 to the channel of P2. Then, the client has a copy of the object P2 and will receive update events published by P2 Figure 5–4 (c).

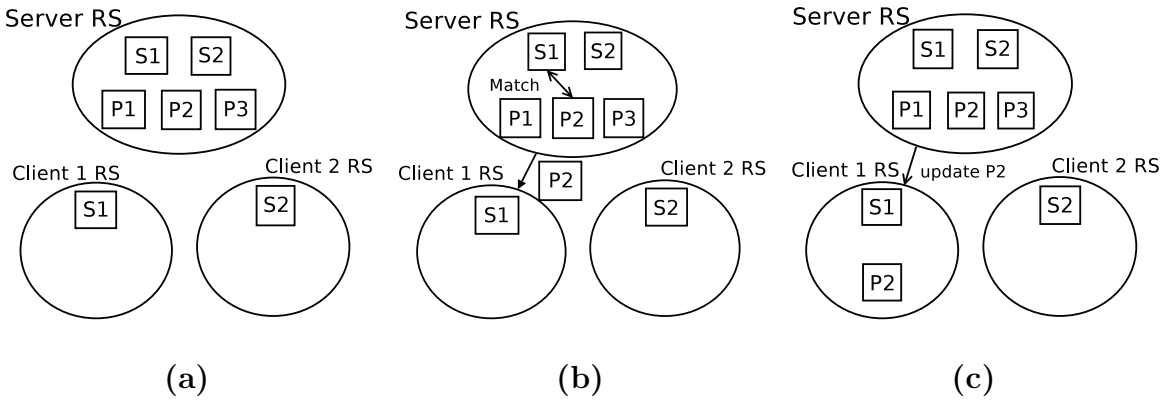


Figure 5–4: Discovery of a publisher in a Replication Space (RS).

Publishing events. When a publisher in a replication space publishes an event (e.g., an object changes position), the event is sent to the publisher's channel if the replication space is the owner of the publisher. If the current replication space instance is not the owner of the publisher, the event is forwarded to the legitimate

```
...

subscribe(Object subscriber, publisher) {
   if(not subscriptions[subscriber].contains(publisher)) {
      // Subscribe   subscriber to channel of publisher in the network layer.
      CommunicationStrategy.subscribe(subscriber, publisher)

      // Send
      CommunicationStrategy.sendContentMessage(replicaOf(publisher))

      subscriptions[subscriber].add(publisher)
   }
}

unsubscribe(Object subscriber, publisher) {
   if(subscriptions[subscriber].contains(publisher)) {
      CommunicationStrategy.unsubscribe(subsriber, publisher)
      subscriptions[subscriber].remove(publisher)
   }
}

...
```

Figure 5–5: Subscribe and unsubscribe methods of the replication space implementation.

owner. In a simple client-server implementation such as ours, if an event is published in a client's replication space the event is forwarded to the server. Figure 5–6 shows the method of the replication space that publishes an event.

Eviction mechanism. If the replication space that is the owner of a publisher detects that a subscriber is no longer interested in that publisher, it unsubscribes the subscriber from the network channel of the publisher (see Figure 5–5's `unsubscribe` method). The subscriber's replication space will stop receiving updates for that object, but it is responsible for the local garbage collection of the replica. For example in the scenario of Figure 5–4 (c), the server's replication space could detect that the subscriber S1 is no longer interested in the publisher P2 and unsubscribe S1 from P2's update events. Then, the replication space of Client 1 would be responsible for determining when P2 can be garbage collected and removed from the space. It is

```
   ...

   publish(Object publisher, Object update) {
      if(publisher.isOwned()) {
         // Send update message on the channel of the publisher
         CommunicationStrategy.send(publisher, update)
      }
      else {
         // Send update message to owner (i.e., the server)
         CommunicationStrategy.send(publisher.getOwner(), update)
      }
   }

   // The interest management strategy is left to be implemented by subclasses.
   virtual refresh()

}
```

Figure 5–6: Publish method of the replication space implementation.

important that an object is not garbage collected before it is unsubscribed by the server, otherwise the client could receive updates about an object for which it does not have a replica. A simple way to deal with this is to have a buffer area between the moment the object is unsubscribed and the time the object is garbage collected. For example the object could be unsubscribed if it is out of a radius of 5 from the subscriber, but garbage collected only if it is out of a radius of 6. This techniques works well as long as the inconsistencies between the server and the clients are within the buffer area. It would be possible if update messages are considerably delayed that the position of an object differs greatly (more than the buffer area) between the server's copy and the client's copy of the object. To deal with exceptional cases of larger inconsistencies we could add a mechanism that allows the client to request a replica of an object if it detects that it receives update from a publisher for which it does not have a replica. We did not need to implement this mechanism for our experiments.

The ReplicationSpace class defines the basic operations of a replication space (see Figure 5–3, 5–5, and 5–6), but the interest management policy is left to be implemented such that different strategies can be implemented as different subclasses. To implement the eight algorithms described in the previous chapter we made two different subclass implementations of the Replication Space: one for object-based interest managements (i.e., Euclidean Distance and Ray Visibility), and one for tile-based interest managements.

The Replication Space for object-based interest management is shown in Figure 5–7, it iterates through all subscriber/publisher pairs in the space by default, and uses a defined boolean *interest function* to determine if the subscriber is interested in the publisher. Implementing different object-based interest management strategies is just a matter of implementing different interest functions. For example, for the Euclidean algorithm we implemented an interest function that computes the distance between the subscriber and the publisher; the function returns *true* if that distance is smaller or equal to the subscriber's radius of interest (see `matchInterest` in Figure 5–7). In our implementation the `refresh` method, which triggers a new iteration over all objects, is called at a fixed time rate by a dedicated thread.

The Replication Space for tile-based interest management has a different iteration mechanism, instead of going through all possible subscriber/publisher pairs, for a given subscriber it only considers the publishers that are within the set of interesting tiles. The tiles considered as tiles of interest are determined in a *tile visiting* function that can be overridden to implement the different algorithms. For example, for the Tile Neighbor algorithm the tile visiting function performs a breadth-first

```
class ObjectBasedReplicationSpace extends ReplicationSpace {

   refresh()
   {
      foreach (Object subscriber in subscribers) {
         foreach (Object publisher in publishers) {
            if (matchInterest(subscriber, publisher)) {
               if(publisher.isOwned()) {
                  subscribe(subscriber, publisher)
               }
            }
            else {
               if(publisher.isOwned()) {
                  unsubscribe(subscriber, publisher)
               }
               else {
                  // Garbage collect the replica.
                  publishers.remove(publisher)
               }
            }
         }
      }
   }


   matchInterest(Object subscriber, Object publisher): boolean {

      Player player = (Player) subscriber;
      WorldObject object = (WorldObject) publisher;

      return player.getPosition().distance(object.getPosition()) <   player.getRadiusOfInterest();

   }
}
```

Figure 5–7: Implementation of the object-based replication space with an interest function for Euclidean distance.

search at a given depth to determine the set of tiles of interest (as shown in Algorithm 6 of Section 4.7). Only the publishers in the tiles of interest are considered for that subscriber.

5.2.2 Tile Manager

For tile-based interest managements there is an additional burden which is to determine the current tile of each object. The Tile Manager is the sub-component of the Replication Engine that keeps track of the current tile of an object. The

updates are performed in a lazy way; whenever the position of an object changes the Tile Manager marks the object as "dirty". Before the Replication Engine runs the interest management function (i.e., the `refresh` method of the replication space), the Tile Manager updates the current tile for all dirty objects using the algorithm described in Chapter 3.3.2. The current tile of an object is stored in the object itself so it can be retrieved in constant time. An object only has one current tile which corresponds to the tile that contains its current position (a point). Tiles also store all the objects that intersect their shape; more than one tile can store the same object if the shape of the object intersects those tiles.

5.2.3 Tilers

The partitioning of the world is done by a *tiler*. A tiler takes a polygon that corresponds to the boundaries of the world and a set of polygons that correspond to the obstacles in the world. We implemented three different tilers: a square tiler, a hexagon tiler, and a triangle tiler. The implementation of the square and hexagon tilers are straightforward: the tiler generates squares or hexagons of a parameterized size to cover the polygon that defines the boundaries of the world.

The triangulation tiler is implemented on top of a triangulation library, the Triangle library by Jonathan Richard Shewchuk [34]. The tiler converts the input polygons into Triangle's format, runs the program, and converts the triangulation result back into game tiles that are stored as part of the static game state. The triangle tiler performs a constraint conforming Delaunay triangulation (see Section 3.1.3) with a constraint on the triangle area. The triangulation can be configured with two parameters: minimum obstacle polygon size, and maximum area of triangles. The

minimum obstacle polygon size defines the minimum length that the longest edge of a polygon must be so that the polygon is considered in the triangulation; if all polygon's edges are smaller it is discarded before the space is triangulated. The *maximum area* is the constraint on the triangle area; it restricts the size of the output triangles.

5.2.4 Communication Strategy

The Replication Engine assumes that the underlying network library provides some kind of grouping mechanism through channels. Clients (or subscribers) can be subscribed and unsubscribed by the server from a channel, and a message sent on a channel will be sent to all clients subscribed to that channel. The Mammoth Network Engine provides virtual channels implemented on top of Java NIO as described in Section 5.1.

5.2.5 Game Use of the Replication Engine

In Mammoth we only implemented one interest management domain using a replication space, the visibility of players. Players that have an active session on the server are subscribers, and all players and items are publishers (i.e., objects that are mutable). We generally assumed that the expression of interest of players corresponds to a radius around the player; a player can see objects that are not occluded up to a certain distance. When the state of a publisher changes such that it affects its visibility (e.g., position change), the publisher publishes an update event in the replication space.

5.3 The Orbius Game Trace

Most investigations of interest management make use of data generated randomly; whether this is a reasonable strategy is unclear, so we collected a trace of real-player movements that we can compare with randomly generated data.

The real-player trace that was used for experiments was collected in a gaming event involving 28 participants playing a custom Mammoth game implementation, *Orbius* [26]. The Orbius game was designed to reflect the general characteristics of larger multiplayer games. Players had to explore the world, collaborate as teams, and interact with opposing teams in order to win the game. The game was played in the 30x30 world shown in Figure 2–5 where a single player covers an area of 0.017. The movements of the players were recorded in a game trace.

The Orbius game trace was replayed for the experiment using "bot" game clients that can read the trace and replay the actions of a player. For the experiments below, 28 bots (one per player trace) were run on seven computers (four bots per computer) connected to the server through a local area network.

Note that due to the distributed nature of the system, there are multiple non-deterministic factors that makes it impossible to guarantee an identical replay of the same game. One problem is that we cannot guarantee that update messages from the distributed clients arrive at the server in the same identical order for each replay of the game, unless we would use a complex distributed synchronization protocols that would add considerable overhead. We computed a variation by replaying the same experiment five times and found that the largest variation in number of messages between experiments was 0.4%.

5.4 Random Traces

It is much easier to generate random traces than to acquire data from real human players. We generated two random player movement traces to compare with Orbius data and see if the results are similar. Both random traces were designed to send the same number of movements over the same period of time as the Orbius trace. Each of 28 clients sends one random move of length 0.5 in a randomly-chosen direction, at a constant rate of one move every 740 milliseconds; these parameters were derived from the average number of messages sent by players divided by the experiment length. The main difference between the two random traces is the starting position of players. In the first trace players were all initialized in the space outside of buildings, similar to the way Orbius players were initialized; in the second trace 4 players were initialized outside and 24 inside of buildings—14 were initialized within the same building.

There are two main reasons for considering these variations in random traces. From our observations of random movements, players initialized outside buildings are unlikely to end up inside of buildings, and symmetrically players initialized inside a building are unlikely to exit from that building. This has the potential to make a significant difference in interest management performance—players inside a building have many opportunities for occluded sight, and thus may be able to take better advantage of visibility-based and reachability-based algorithms. Relative distribution is another important property with respect to region-based interest management; the second trace thus also allows us to examine a scenario in which half of the players are located in one place rather than being more uniformly distributed.

5.5 Measurements

To evaluate our implementations we considered two main types of messages between the server and clients: *content messages* and *update messages.* Content messages are used when an object is discovered by a player that has no copy of that object; a full copy of the state of the target object is sent to the player—this represents a replication cost. Update messages are sent to existing subscribers to inform them of a change in the state of an object that is already known. In our experiments these are primarily player position updates. Content messages are generally more expensive because they contain the full state of an object while update messages contain only a partial state. In the Orbius trace, we calculated that on average a content message was 1.5 times the size of an update message.

For each experiment we measured the number of each type of message received at each client and computed an average over the 28 clients. We also measured the number of subscriptions to network channels performed on the server side. Finally, we measured the CPU consumption of the dedicated machine running the server, a dual-core 3GHz Pentium D with 2GB of memory. CPU utilization was polled every second using JSysmon [3], and averaged over the total length of the experiment (12 minutes).

CHAPTER 6
Experimental Results

In this chapter we present results from experiments we conducted. First, based on the Orbius (real player) movement data, we discuss the trade-offs provided by the choice of tile size in terms of message filtering capability versus CPU overhead. We then compare the effectiveness at relevance filtering of the eight algorithms, followed by results on the scalability of our different interest management approaches. There are more than one way to map channels to the elements of the game world, we compare the number of subscriptions to network channel for two channel mapping schemes. Finally, we compare results obtained from real-player traces with results obtained from randomly generated traces. This is intended to give some indication of the kinds and magnitude of factors in workload data that can influence or obscure results.

6.1 Tile Size

We investigated the effect of changing the tile size (area). Our hypothesis is that smaller tiles would better approximate the player's area-of-interest and filter more irrelevant messages; however, it would also have a higher computation overhead.

Figure 6–1 and Figure 6–2 show the average number of content messages and the average number of update messages, respectively, sent from the server to a client for different tile sizes. The size of tiles is expressed in terms of tile area; in the case of triangular tiles it corresponds to the maximum area constraint that is set in the

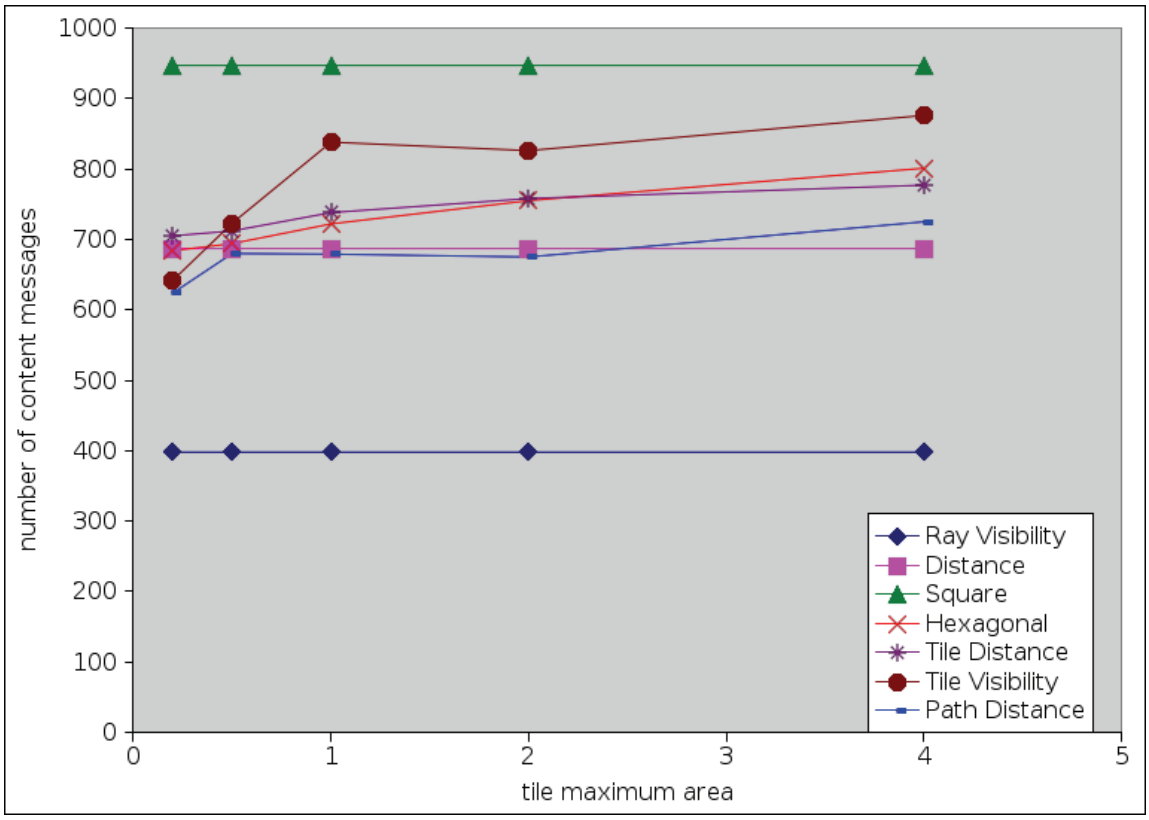


Figure 6–1: Average number of content messages received by a client for varying tile areas.

triangulation algorithm. The players' interest radius is fixed to five, a reasonable value based on player experiences in the Orbius game.

In most cases a smaller tile size results in slightly less content and update messages, with the Tile Visibility algorithm gaining the most from a smaller tile size. The effect is surprisingly subtle; even with 28 players Mammoth is not a densely populated world, and tile size does not have an overall large impact. Unsurprisingly Ray Visibility and (Euclidean) Distance do not change; these interest management

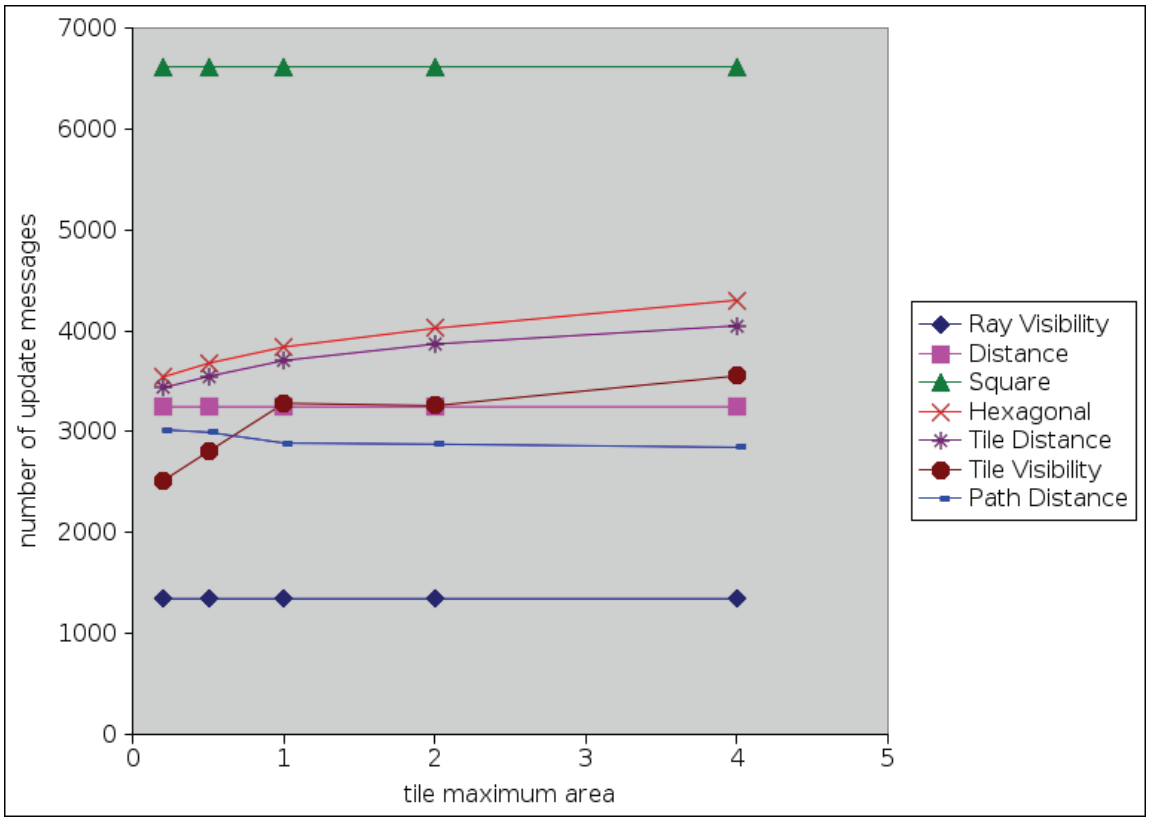


Figure 6–2: Average number of update messages received by a client for varying tile areas.

schemes are based on absolute distance, and tiling has no direct impact. Square also does not change since its tiling does not change for a fixed interest radius.

Figure 6–3 shows the average CPU consumption at the server for each tile size. Here the impact of tile size is more obvious. Algorithms such as Hexagonal, Tile Distance, Neighbor, and Path Distance make use of breadth-first search, and naturally as tile size decreases the number of tiles that must be visited increases, and so does CPU cost. Tile Visibility displays the smallest increase due to reduced tile size;

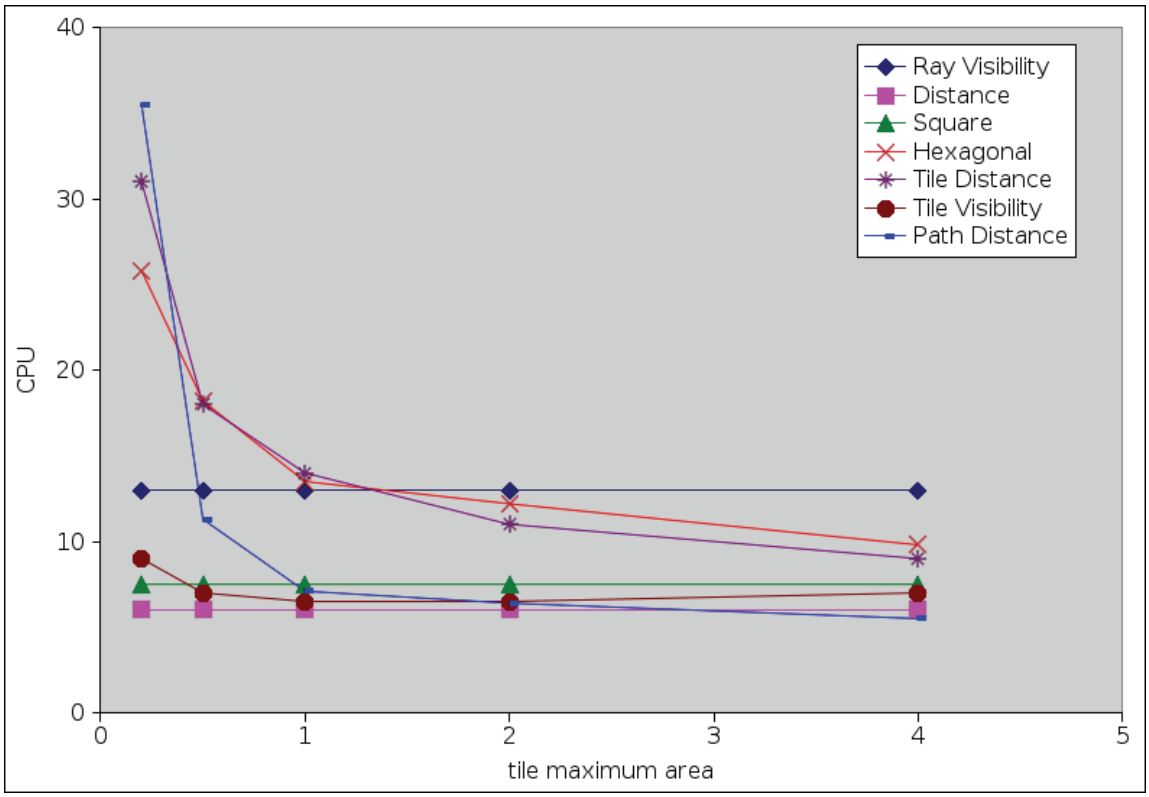


Figure 6–3: CPU consumption of the server for different tile areas.

this is likely due to using precomputed information rather than dynamic searches in order to determine tile visibility.

The results of Figure 6–1, 6–2, and 6–3 suggest two things. First, the gains from the use of smaller interest management regions are not necessarily large, and may depend on the game environment. Secondly, while there is a definite and large trade-off between quality of the approximation and the overhead cost of smaller tiles, particularly at very small tile sizes, preprocessing of the environment can greatly mitigate the costs. For our subsequent experiments we used a tile size of 1.0, as a

reasonable point where improvements in the number of content messages are mostly realized while CPU cost is still not excessive.

6.2 Message Filtering

Message filtering is the primary role of interest management; in order to determine the effectiveness of the different interest management algorithms we measured the number of update and content messages received by each client from the server. Figures 6–4 and 6–5 show the average number of messages per client for the eight interest management algorithms under different interest radii. The Ray Visibility algorithm represents a theoretically perfect filtering of messages and can be used as a reference to evaluate the efficacy of other algorithms.

The first observation to be made is that there is a considerable difference in number of messages between Ray Visibility and Distance. This suggests that there is considerable potential to reduce the number of messages by taking advantage of obstacles—interest management approaches that consider visibility should perform significantly better at filtering.

Not surprisingly, by far the worst filtering is provided by the square tiling. For the smallest radius of interest the number of messages sent (update or content) is more than twice the number of Ray Visibility. This trend only gets worse as the radius of interest increases: our square tiling algorithm guarantees a fixed region of nine tiles in all cases, and so as the tile size increases to accommodate a larger visibility radius the squares provide a worse and worse approximation of the real player interest region. Regular tilings do not take advantage of actual visibility, and

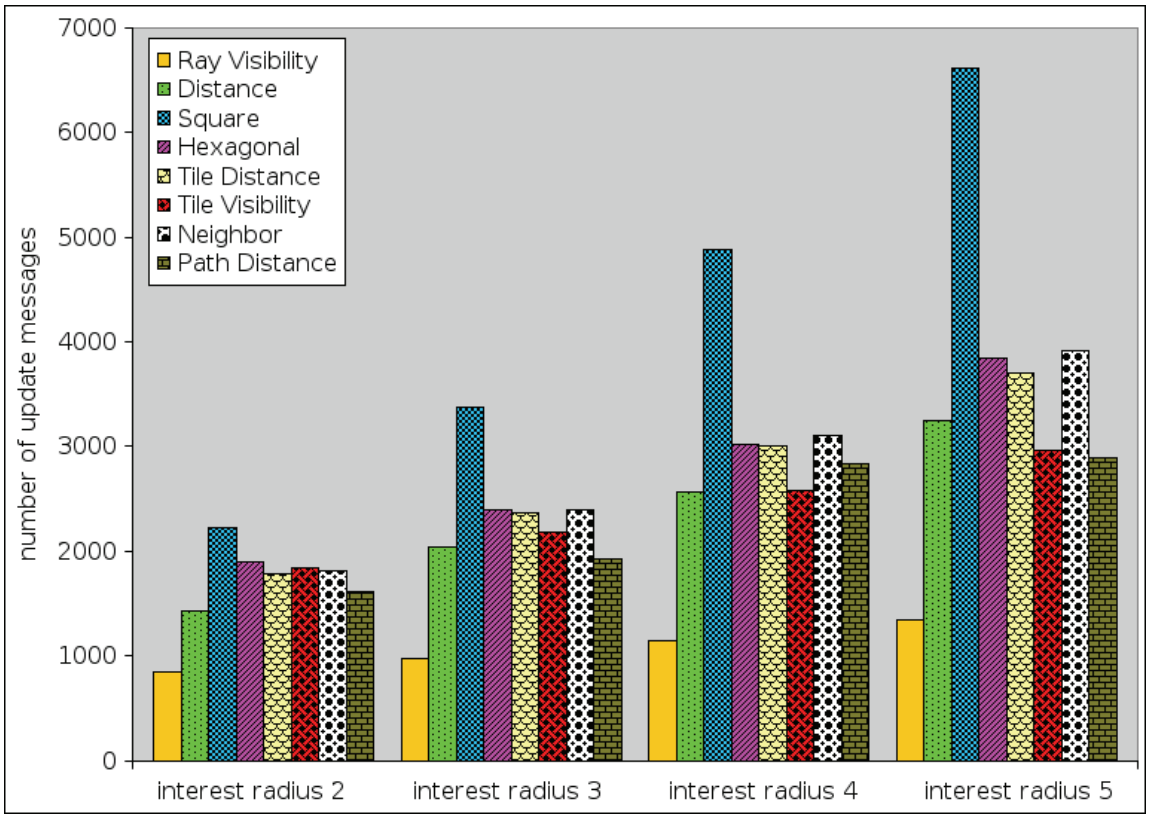


Figure 6–4: Average number of update messages per player with various interest radius sizes.

this is also exacerbated by a larger interest radius, which heuristically includes more obstacles.

The hexagonal tiles algorithm represents a significant improvement over square tiles. If we compare the number of messages for the hexagonal tiling with our pure (Euclidean) Distance in Figure 6–4 and 6–5 we find there are only 17% more update messages and 14% more content messages. This improvement over square tiles is mainly due to the accuracy of approximating the real region of interest. Neither our square nor hexagonal tile based approaches take advantage of actual visibility,

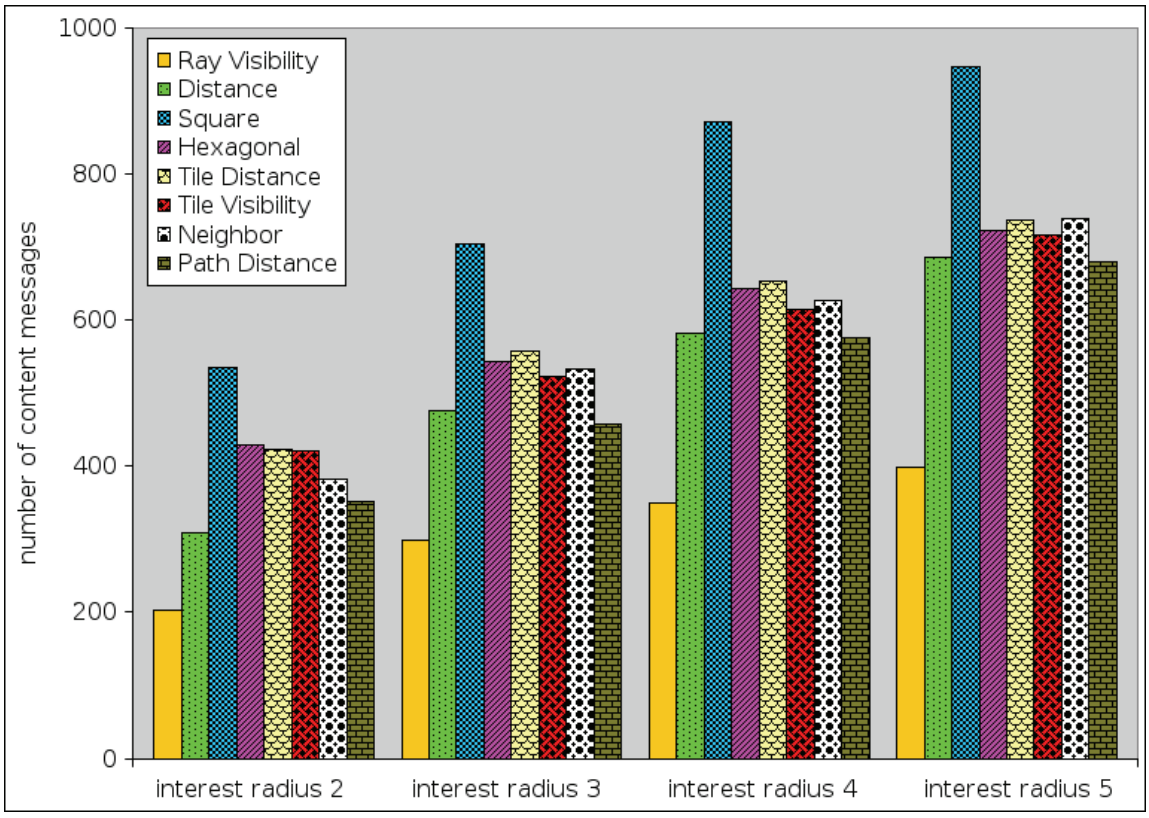


Figure 6–5: Average number of content messages per player with various interest radius sizes.

but as the interest radius increases in proportion to the tile size the small hexagonal tiles better approximate a circular region of interest. An obvious improvement to the square tiles algorithm would be to use smaller tiles that are not proportional to the interest radius, but it was already shown by others [20] that hexagonal regions can better approximate a player's interest radius than square regions.

Tile Distance is an obstacle-aware algorithm, and thus should show improved results over Hexagonal. Here, however, it performs only marginally better in terms of update messages, and marginally worse with respect to content messages. There

are many obstacles in the Mammoth world, but mainly concentrated in a few areas, and Orbius players tended to stay largely outside. Moreover, obstacles in Mammoth do not tend to result in complex, maze-like environments; tiles within a given area-of-interest tend to be connected and thus included in the tile distance. Overall, Tile Distance reduces only about 6% of update messages and 1% of content messages over an algorithm that considers all triangles intersecting the area-of-interest irrespective of connectivity. It suggests that the approximation of the interest radius made by triangles performs similarly to the approximation made by hexagons.

The Neighbor algorithm filters slightly less update messages than Hexagonal in most cases, and filters only slightly more content messages. The problem with Neighbor arises from the difficulty in determining the depth that matches a given interest radius, primarily due to the fact that tiles have irregular sizes. For our experiments we determined the depth qualitatively by choosing a value that would fully cover the area-of-interest in a surface where there are no obstacles.

Tile Visibility can filter more update messages and slightly less content messages than Hexagonal tiles. The advantage of Tile Visibility seems to grow with the increase of the interest radius. For an interest radius of two it filters 3% more update messages than Hexagonal, and for an interest radius of five, 23% more. We explain this increase in effectiveness by the fact that obstacles occlude a larger proportion of the area-of-interest when the surface is greater.

Path Distance performs better than the similar Neighbor algorithm; it filters about 16% more update messages and 10% more content messages. This can be

mostly attributed to the fact that Path Distance is easier to adjust to a given interest radius. Path distance also performs slightly better than Tile Visibility in most cases (5% less update messages and 10% less content messages), but is subject to approximation errors—it is possible that the tile area does not fully cover the interest radius of the player. Path Distance does, however, have a potential advantage in allowing for more "incremental" discovery of the world. Under Tile Visibility, large groups of tiles can be added to the area-of-interest by players moving only a small distance, peeking past a corner for instance. Path Distance will have already included many of the newly visible triangles. Overall, Path Distance is the algorithm that has the closest results to the Ray Visibility lower bound, generating about twice the number of messages.

The results suggest that both Visibility and Path Distance seem to be reasonable algorithms to perform interest management in a world with obstacles. The Path Distance algorithm, however, has a few practical advantages over the Tile Visibility algorithm. Most importantly it does not require a complex preprocessing step, and thus is much more accommodating to changes in parameters, such as the interest radius.

6.3 Scalability

To evaluate the scalability of the algorithms in densely populated areas we increased the number of objects (i.e., items) in the world and measured the number of content messages received by each player, as well as the CPU consumption of the server. The first run of the experiment had 186 objects, which was the initial number of objects put by the designer in the map; here we increased that number

to 1000, 2000, and 3000 objects. The added objects do not move, but can only be discovered by players; thus, they will only create an increase in the number of content messages sent by the server. The number of players is fixed to the 28 Orbius players throughout the experiments, and the players' interest radius is fixed to five.

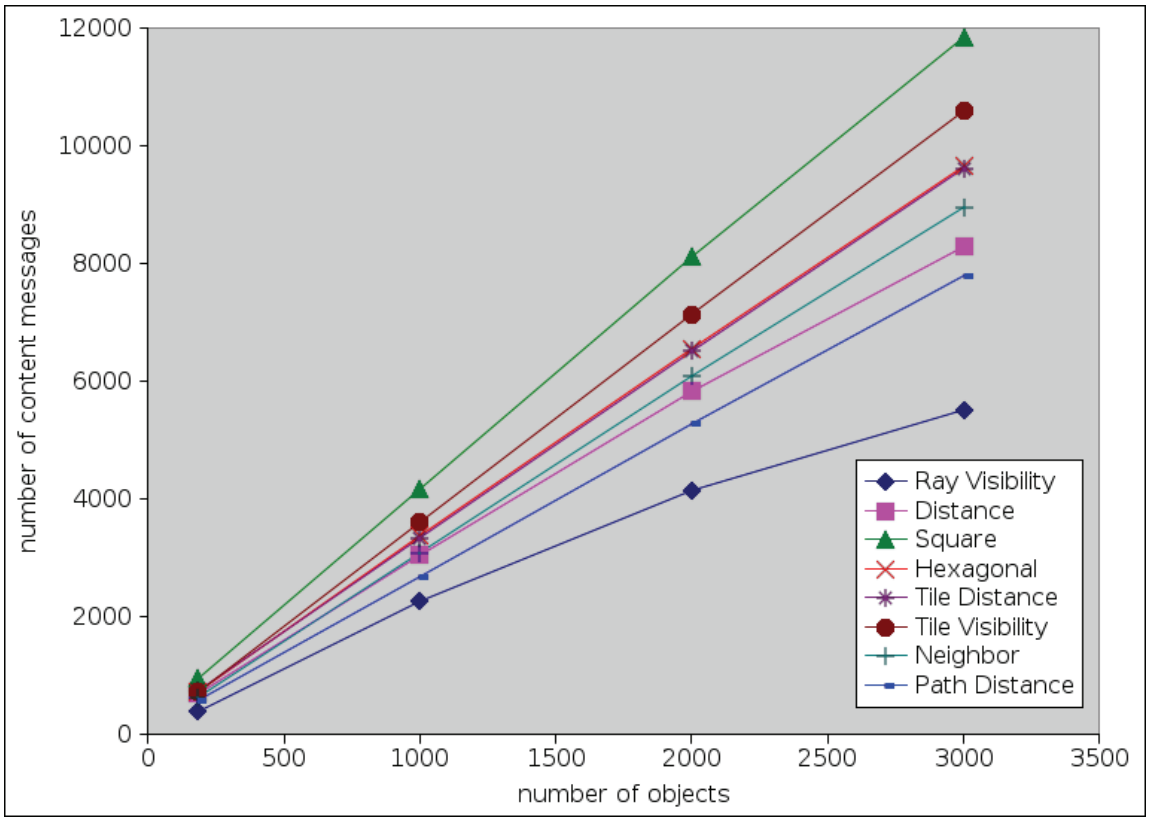


Figure 6–6: Average number of content messages per player with an increasing number of objects in the world.

Figure 6–6 shows the increase in average number of content messages received by each client, and Figure 6–7 shows the server CPU consumption corresponding with the increase in object density. The former shows a linear relation for most of our algorithms; content messages themselves are not a major source of scalability

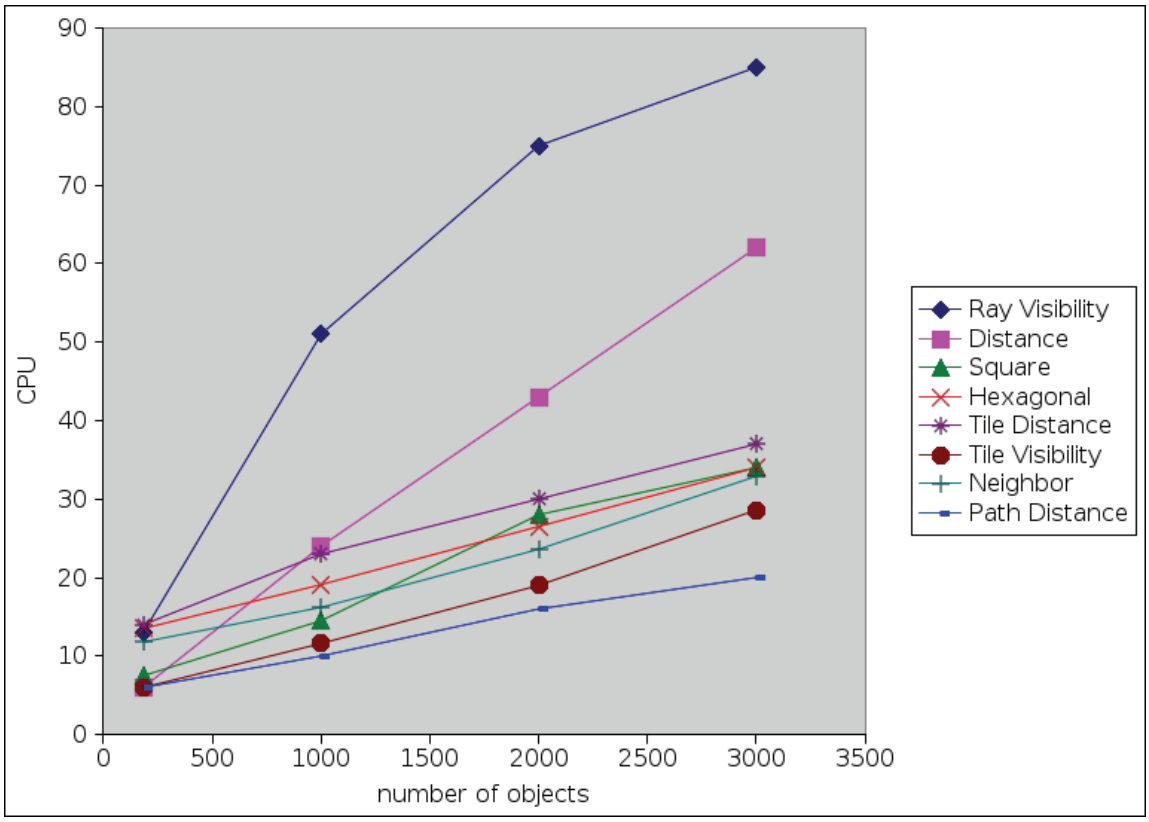


Figure 6–7: CPU consumption with an increasing number of objects in the world.

concern. CPU consumption shows much more separation. Here it is clear that algorithms based on tiles have a great scaling advantage over algorithms such as Ray Visibility and Distance. A subscriber in a tile-based situation is localized to tiles that are close to the subscriber while for the other two algorithms the interest computation is done with every publisher in the world. This makes tiling much more appealing at all but the lowest density of game objects.

6.4 Subscriptions

Depending on the underlying network group communication library that is used, the cost associated with subscription and unsubscription to network channels may vary. When virtual channels are implemented over TCP/IP (e.g., Mammoth's Network Engine), subscription and unsubscription to channels is a virtually free operation (i.e., negligible CPU cost). On the other hand, when multicast is used, there is a substantial cost associated with joining or leaving a multicast group. For this reason, in this section we compare two popular paradigms to map channels to game objects: object-channel and tile-channel mapping. *Object-channel* mapping is the paradigm that was assumed so far in which one network channel is assigned to each publisher. A subscriber subscribes to the channels of the object it is interested in and a publisher publishes events on its assigned channel. *Tile-channel* mapping is the paradigm that is usually used in region-based interest managements that use multicast (e.g., NPSNET [27]): a channel is assigned to each region or tile. Subscribers subscribe to the network channel of tiles that span their radius of interest; publishers publish events to the channel assigned to their current tile. The difference between the two paradigms was already investigated by Zou et al. [43].

Figure 6–8 shows the number of subscriptions during the experiment length for an increasing number of objects (i.e., items) in the world. The players' interest radius is fixed to five. Each algorithm in the legend has two corresponding data sets in the graph, one for each channel mapping paradigm. The tile-channel mappings are flat lines, they do not change since the number of subscriptions to tile channels will be the same independently of the number of objects in the world. On the other

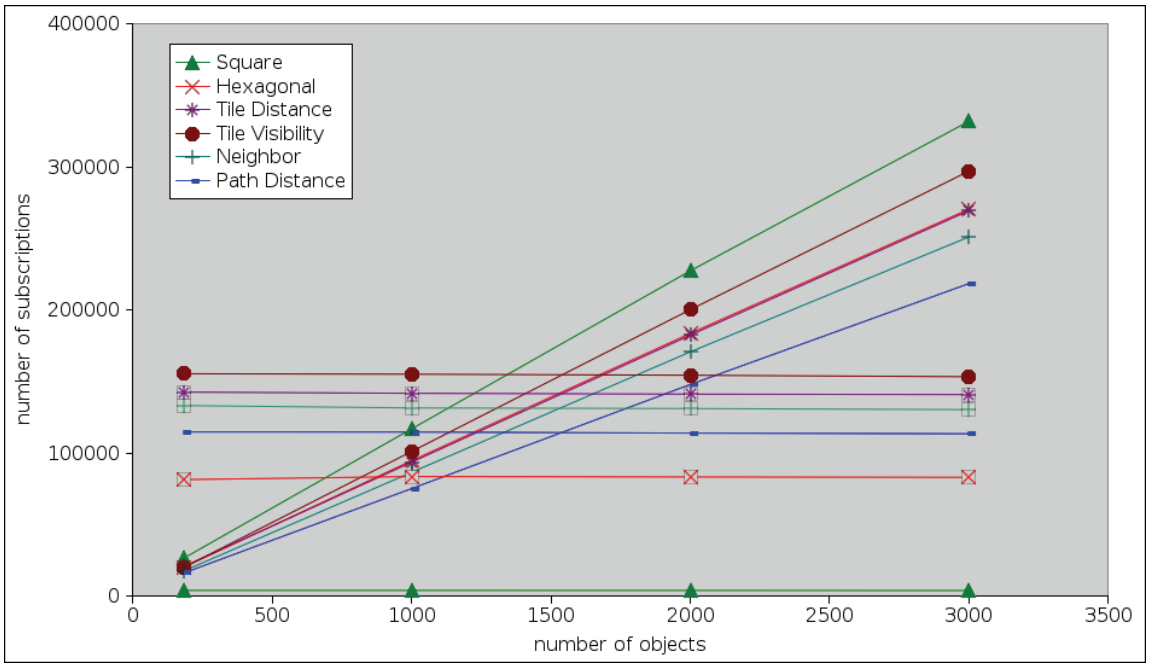


Figure 6–8: Number of subscriptions during the experiment for increasing number of objects in the world.

hand, the number of subscriptions for the object-channel mappings increases with the number of objects in the world.

The results are quite intuitive. For all algorithms the tile-based mapping becomes advantageous over object-based mapping when the number of objects in the world is greater than the number of tiles. For algorithms that use triangular tiles this number is around 1600, which corroborates with the number of tiles in the world for those experiments. In the case of square tiles the world has only 36 tiles, so tile-based mapping has less subscriptions in all cases.

6.5 Real-Player Movements versus Random Movements

Many interest management analyses make use of randomized data. Real player behaviour, however, has the potential to be quite different from any simple randomized model. To determine if experiments using real player traces and experiments using randomly generated traces would give similar results, we compared the results from our Orbius trace with the results from the two randomly generated traces (see Section 5).

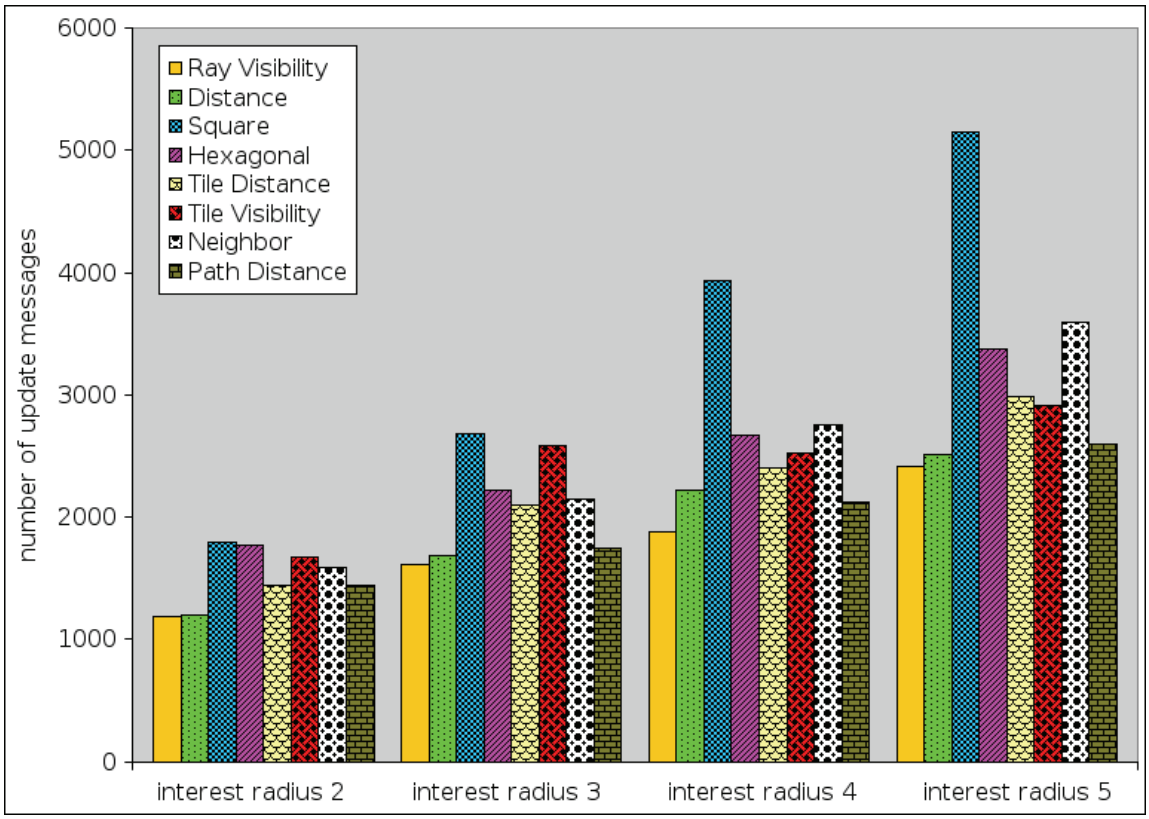


Figure 6–9: Average number of update messages per player with various interest radius sizes for random data starting outside buildings.

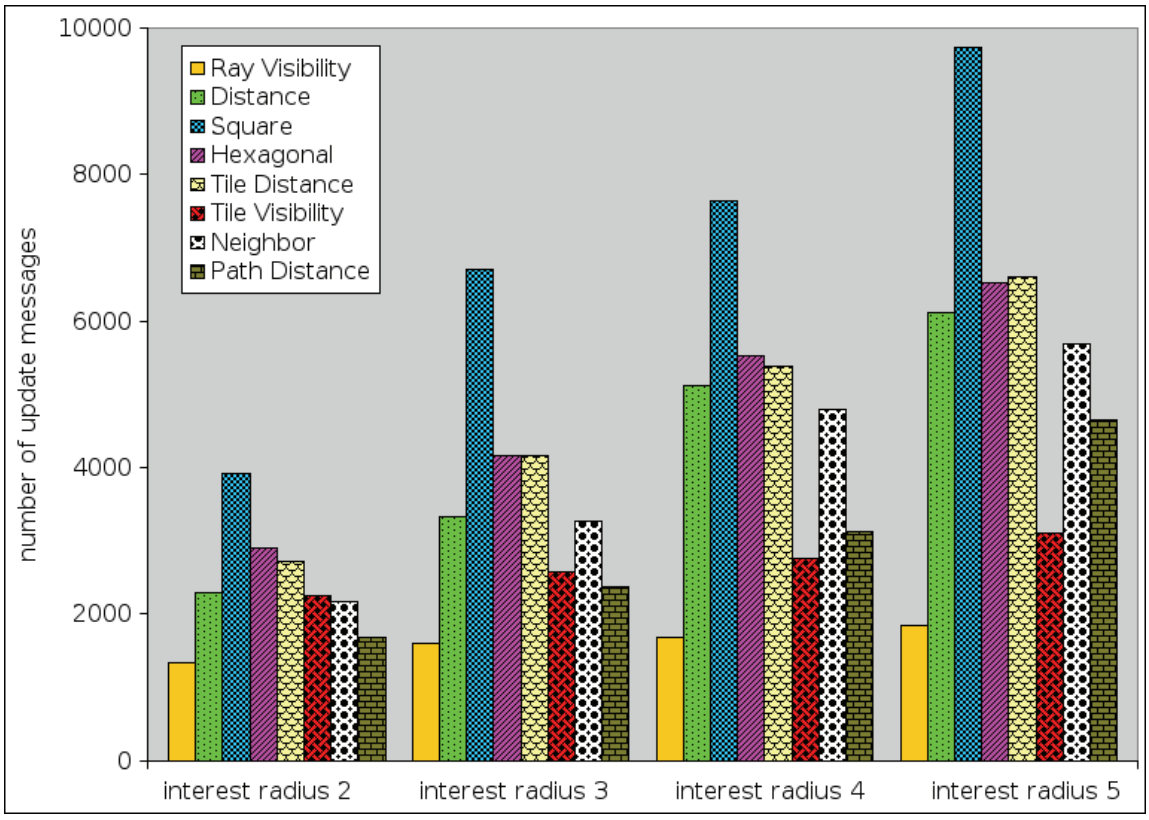


Figure 6–10: Average number of update messages per player with various interest radius sizes for random data starting mostly inside buildings.

Figure 6–4, 6–9, and 6–10 shows the average number of update messages received per player for the three sets of traces (i.e., Orbius, random outside, and random inside, respectively). The three traces give similar relative results between algorithms. For instance, in most cases Tile Visibility and Path Distance algorithms perform better than Square and Hexagonal tiles. In absolute terms, however, there is considerable difference in the number of messages received. For example, Tile Visibility filters 40% more messages over Hexagonal with the random inside trace, only 6% more with the random outside trace, and 12% with the Orbius trace. The Distance

algorithm nearly matches Ray Visibility in random outside, but is much worse in random inside.

These differences correlate with the general properties of player behaviour in these games. In random inside players largely move in an environment dense with obstacles, and thus obstacle-aware algorithms do quite well. Players in random outside are far from obstacles, and obstacle-aware algorithms do not improve the performance nearly as much. This is also evident in Table 6–1. Orbius data has more of a mixture of inside and outside movements, and Ray Visibility and Tile Visibility thus perform better in Orbius than in outside random data, and show less improvement in Orbius with respect to the use of obstacle-dense, than random inside data. Table 6–1 further shows the variance induced by the different workloads. Real game movements are particularly amenable to visibility based schemes, with Path Distance having overall good absolute and relative performance.

Algorithm	RO Avg	RO Max	RI Avg	RI Max
Ray Visibility	-37.9%	-44.3%	-33.4%	-38.8%
Distance	17.5%	22.8%	-43.2%	-49.7%
Square	20.4%	22.2%	-40.3%	-49.6%
Hexagonal	9.3%	12.1%	-41.0%	-45.4%
Tile Distance	17.4%	20.0%	-41.3%	-44.2%
Tile Visibility	0.5%	-15.4%	-11.0%	-18.6%
Neighbor	10.5%	12.0%	-27.4%	-35.3%
Path Distance	13.7%	25.1%	-17.4%	-37.8%

Table 6–1: Relative difference in number of update messages between Orbius data and the two Random data sets. Columns 2 and 3 show the change from Random Outside to Orbius, while columns 4 and 5 show the relation between Random Inside and Orbius. Negative values indicates fewer messages for Orbius, and max difference is in terms of absolute value.

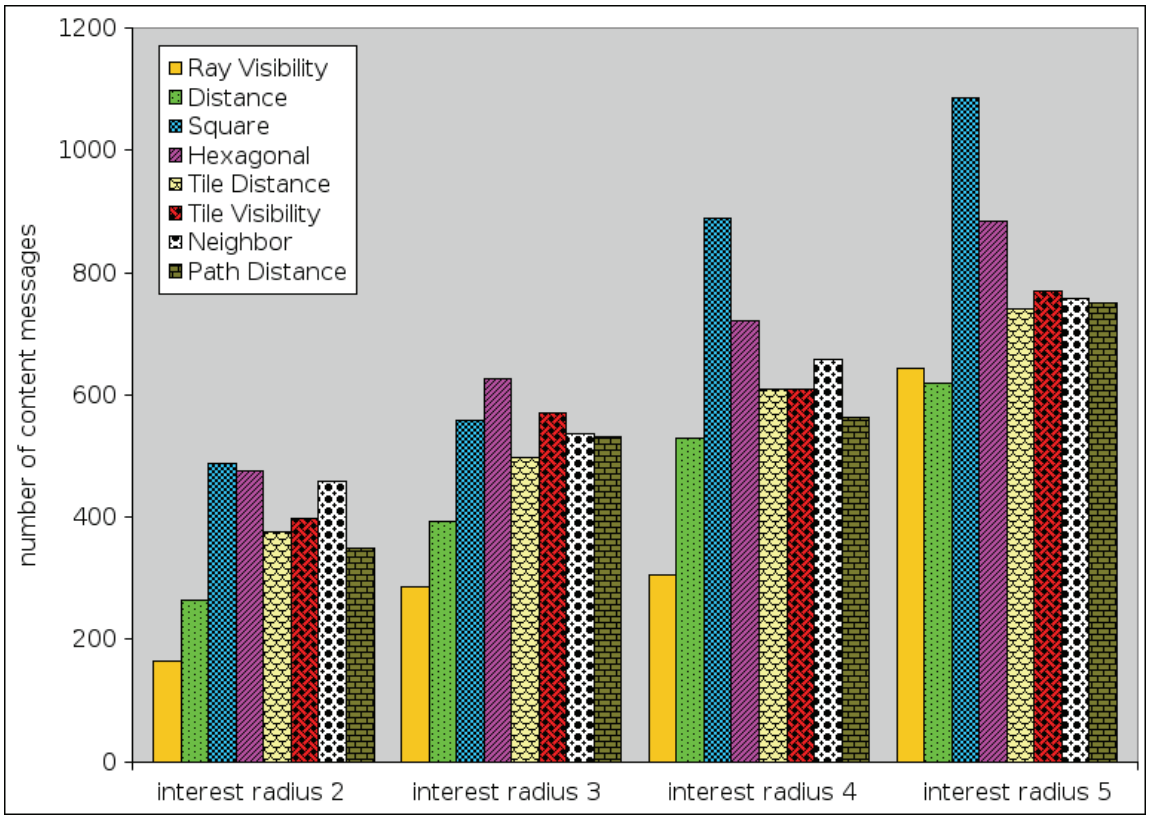


Figure 6–11: Average number of content messages per player with various interest radius sizes for random data starting outside buildings.

Figure 6–5, 6–11, and 6–12 shows the average number of content messages received per player for the three sets of traces (i.e., Orbius, random outside, and random inside, respectively). The three traces give fairly different results, especially the random inside trace. Figure 6–12 shows a few cases in which a greater interest radius results in less content messages. This phenomenon is explained by the fact that a smaller radius can cause more discovery/undiscovery of object than a greater radius, this is especially true when players are enclosed within a small space. For example, if the interest radius fully spans a building, players will discover each other

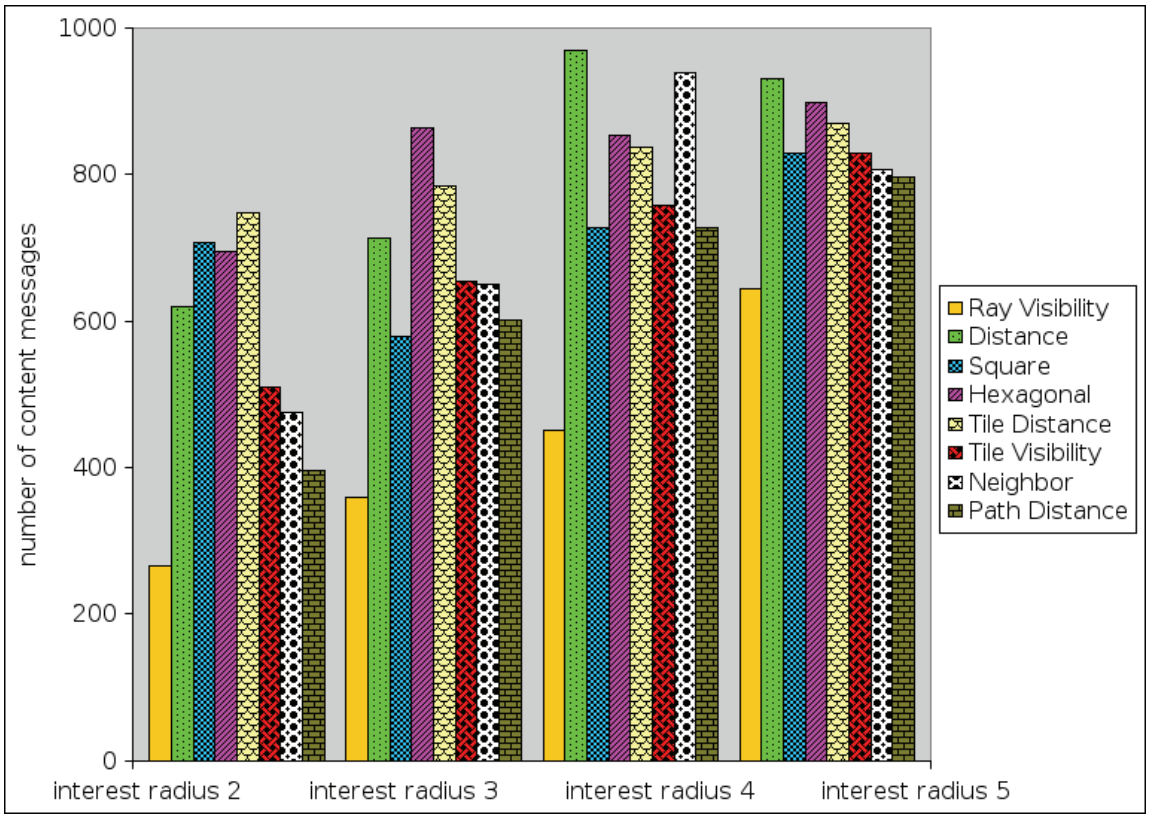


Figure 6–12: Average number of content messages per player with various interest radius sizes for random data starting mostly inside buildings.

only once; on the other hand, if the radius only spawns half a building, players will continuously discover and undiscover each other as they move. It shows that there is a trade-off between the cost of updating an object and the cost of rediscovering an object. If the probability of rediscovering an object is high, it may be cheaper to keep it updated than to garbage collect it and rediscover it.

CHAPTER 7
Conclusion and Future Work

Good interest management designs are important to good network performance in massively multiplayer games, and interest management will play a central role in scaling MMOGs to larger populations of players. In this thesis we have presented and compared a variety of interest management algorithms incorporating various levels of visibility and map conformance. We have performed experiments within the Mammoth framework that have shown that taking obstacles into account has the potential to greatly reduce the amount of information that has to be exchanged between players.

We introduced a tile partitioning technique that breaks the space according to the geography of the world. The technique uses a constrained conforming Delaunay triangulation to partition to world into triangles of a maximum area size in order to achieve a partitioning suitable for interest management. The partitioned space is stored in a neighbor graph that preserves the geographical information and can be used to search a space local to a player. We introduced four interest management algorithms that use our neighbor graph to perform visibility-based and reachability-based interest management.

We abstracted and implemented interest management within the Mammoth framework and performed experiments using a movement trace that was collected in

an event involving 28 real human players. We also performed the same experiments using two randomly generated traces.

Our experiments showed that it is possible to define regions of interest based on a partitioning of the world space into triangular tiles, and that those algorithms are more scalable than object-based algorithms, in particular when they are used in combination with caching or preprocessing.

Among the different tile-based interest management algorithms, Tile Visibility and Tile Path Distance are both algorithms that could be useful for interest management, but the Tile Path Distance algorithm seems to exhibit the most interesting properties. The number of update messages that have to be sent between players is the closest to the ideal number (given by the Ray Visibility algorithm), but the computational effort required to run the algorithm is 3 to 6 times lower. In addition, the unnecessary update messages sent to a player are the ones concerning game state that is very likely to be of interest to the player in a near future, which can increase game responsiveness in case of network lag.

Our experiments also showed the properties of the trade-off between the performance of the relevance filtering and the CPU overhead when varying the maximum tile size. We showed that past a certain point, smaller tiles have a much higher computation cost, although the improvement in reducing irrelevant messages is subtle.

Finally, we have demonstrated that measurements taken during a game using computer-controlled players performing random movements can be used to predict measurements taken during a game with real human players. However, factors such as the ratio of players starting inside buildings or other closed spaces compared to

those starting outside can have a significant impact on the results, and needs to reflect the situation in the real game.

7.1 Future Work

Our work opens the door to many avenues that could be explored in the future. In particular our obstacle-aware partitioning of the world is a useful methodology that could be used not only for interest management, but for other concerns such as path finding and load balancing. We will discuss some of the future work that could find its foundation on the work of this thesis:

Heuristic for Tile Path Distance. As we have shown and discussed before, our Tile Path Distance algorithm exhibits interesting properties. However, in our experiments we determined the mapping between an interest radius and the maximum path length qualitatively. In order to use the algorithm more easily in practice, it would be valuable to develop a heuristic that allows to determine an adequate maximum path length for a given interest radius.

Hybrid / adaptive interest management. The observations in this thesis suggest that it would be valuable to investigate the performance of hybrid / adaptive interest management algorithms in the future. For instance, it makes sense to use a fast distance-based interest management algorithm such as "Euclidean Distance" when players are mainly outside (in an area with very few obstacles), and then switch to a reachability-based interest management algorithm such as "Path Distance" when players congregate inside buildings or other areas with many obstacles.

Integrate interest management with path finding. Path finding can also be performed efficiently using a partitioning of the world into triangles. Furthermore,

our Path Distance algorithm performs a search very similar to the first step of the path finding algorithm by Kallmann in which the shortest channel between the current position and the destination of a player is found [25]. The same data structure and part of the computation could be possibly shared between the two concerns.

It would also be interesting to investigate if interest management could take advantage of knowing the destination of the player and the path that the player will use to optimize the objects that are discovered accordingly. For instance, interest management could reach further in the direction that the player is traveling or prefetch objects along the anticipated path.

Interest management for dynamic expression-of-interest. In our experiments we assumed that the expression-of-interest of players was a static radius around the player, however, in games the expression-of-interest could change dynamically. For example, a game that supports zooming would allow a player to see a smaller or larger portion of the world with finer or coarser details, respectively. It would be interesting to investigate how we can use tile-based algorithms with dynamic expression-of-interests.

Extending interest management to a distributed server. The integration of interest management into the Mammoth framework for our experiments was implemented for a single server. However, a distributed server that can run on multiple machines is now available. The Replication Space abstraction could be extended to run in the distributed scenario. Instead of having one server with a replication space containing all the subscribers and publishers in the world, each server would

have a replication space that performs interest management for a subset of the objects in the world. The subset of objects would probably correspond to the objects contained within a subspace of the whole world. Objects near the boundaries of the subspace would have to reside in the replication space of both servers. The future challenges reside in determining in which server the replication space objects reside and how that changes with the movement of objects.

Use triangular partitioning for load balancing. A distributed server introduces the possibility for dynamic load balancing. Many load balancing techniques are based on a partitioning of the world space into regions where each server manages the game objects of one or multiple regions [14, 41]. It would be interesting to investigate the use of our obstacle-aware triangular partitioning to perform load balancing. Since our algorithms can reduce the amount of information that needs to be exchanged, it could also reduce the overhead of inter-server communication.

References

[1] EVE online. http://www.eve-online.com/.

[2] Java NIO. http://java.sun.com/j2se/1.4.2/docs/guide/nio/.

[3] JSysmon. http://jsysmon.sourceforge.net.

[4] Lineage II. http://www.lineage2.com/.

[5] World of warcraft. http://www.worldofwarcraft.com.

[6] Mammoth: The massively multiplayer prototype. http://mammoth.cs.mcgill.ca, 2006.

[7] Lars Aarhus, Knut Holmqvist, and Martin Kirkengen. Generalized two-tier relevance filtering of computer game update events. In *Proceedings of the 1st workshop on Network and system support for games*, pages 10–13, 2002.

[8] Howard Abrams, Kent Watsen, and Michael Zyda. Three-tiered interest management for large-scale virtual environments. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 125–129, 1998.

[9] Thor Alexander, editor. *Massively Multiplayer Game Development 2*. Game Development Series. Charles River Media, Hingham, Massachusetts, 2005.

[10] Tetsuo Asano, Subir K. Ghosh, and Thomas C. Shermer. Visibility in the plane. In J.-R. Sack and J. Urrutia, editors, *Handbook of computational geometry*, pages 829–876. Elsevier Science B.V., Amsterdam, The Netherlands, 2000.

[11] John W. Barrus, Richard C. Waters, and David B. Anderson. Locales and beacons: efficent and precise support for large multi-user virtual environments. In *Proceedings of the IEEE Virtual Reality Annual International Symposium*, pages 204–213, 1996.

[12] Steve Benford and Lennart E. Fahlen. A spatial model of interaction in large virtual environments. In *Third European Conference on Computer Supported Cooperative Work*, pages 107–123, 1993.

[13] Sergio Caltagirone, Matthew Keys, Bryan Schlief, and Mary Jane. Architecture for a massively multiplayer online role playing game engine. Technical report, The University of Portland, 2002.

[14] Jin Chen, Baohua Wu, Margaret Delap, Björn Knutsson, Honghui Lu, and Cristiana Amza. Locality aware dynamic load management for massively multiplayer games. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 289–300, 2005.

[15] Eric Cronin, Burton Filstrup, and Anthony Kurc. A distributed multiplayer game server system. Technical report, University of Michigan, 2001.

[16] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry Algorithms and Applications*. Springer, 1997.

[17] Alexandre Denault. Minueto, an undergraduate teaching development framework. Master's thesis, McGill University, Montréal, Canada, apr 2005.

[18] Alexandre Denault and Jörg Kienzle. Minueto, a game development framework for teaching object-oriented software design techniques. In *FuturePlay 2006: The International Conference on the Future of Game Design and Technology*, Montréal, Canada, 2006.

[19] DFC Intelligence. *Challenges and Opportunities in the Online Game Market*, 2003. `http://www.dfcint.com/game_article/june03article.html`.

[20] Stefan Fiedler, Michael Wallner, and Michael Weber. A communication architecture for massive multiplayer games. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 14–22. ACM Press, 2002.

[21] Thomas A. Funkhouser. RING: a client-server system for multi-user virtual environments. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 85–92, 1995.

[22] Chris Greenhalgh. Awareness-based communication management in the MASSIVE systems. *Distributed Systems Engineering*, vol. 5, no. 3:129–137, 1998.

[23] Seunghyun Han, Mingyu Lim, and Dongman Lee. Scalable interest management using interest group based filtering for large networked virtual environments. In *VRST '00: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 103–108, New York, NY, USA, 2000. ACM Press.

[24] Mojtaba Hosseini, Steve Pettifer, and Nicolas D. Georganas. Visibility-based interest management in collaborative virtual environments. In *Proceedings of the 4th international conference on Collaborative virtual environments*, pages 143–144, 2002.

[25] Marcelo Kallmann. Path planning in triangulations. In *Proceedings of the Workshop on Reasoning, Representation, and Learning in Computer Games*, pages 49–54, 2005.

[26] Marc Lanctot, Nicolas Ng Man Sun, and Clark Verbrugge. Path-finding for large scale multiplayer computer games. Technical Report GR@M 2006-2, GR@M, McGill University, July 2006.

[27] Michael R. Macedonia, Michael J. Zyda, David R. Pratt, Donald P. Brutzman, and Paul T. Barham. Exploiting reality with multicast groups. *IEEE Comput. Graph. Appl.*, 15(5):38–45, 1995.

[28] Michal Masa and Jiří Žára. Generalized interest management in virtual environments. In *Proceedings of the 4th international conference on Collaborative virtual environments*, pages 149–150, 2002.

[29] Roger Delano Paul McFarlane. Network software architecture for real-time massively-multiplayer online games. Master's thesis, McGill University, 2005.

[30] Graham Morgan, Fengyun Lu, and Kier Storey. Interest management middleware for networked games. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 57–64, 2005.

[31] Katherine L. Morse. Interest management in large-scale distributed simulations. Technical report, Department of Information & Computer Science, University of California, Irvine, 1996.

[32] Quazal. *Duplication Spaces^{TM} Quazal Multiplayer Connectivity White Paper*, January 2002. `http://www.quazal.com`.

[33] Steven J. Rak and Daniel J. Van Hook. Evaluation of grid-based relevance filtering for multicast group assignment, 1996.

[34] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[35] Sandeep Singhal and Michael Zyda. *Networked Virtual Environments: Design and Implementation*. SIGGRAPH Series. Addison-Wesley and ACM Press, New York, 1999.

[36] Max Skibinsky. The quest for holy scale - part 1: Large-scale computing. In Thor Alexander, editor, *Massively Multiplayer Game Development 2*, Game Development Series, pages 339–354. Charles River Media, Hingham, Massachusetts, 2005.

[37] Jouni Smed, Timo Kaukoranta, and Harri Hakonen. Aspects of networking in multiplayer computer games. In *Proceedings of International Conference on Application and Development of Computer Games in the 21st Century*, pages 74–81, 2001.

[38] Anthony Steed and Roula Abou-Haidar. Partitioning crowded virtual environments. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 7–14, 2003.

[39] Bruce Sterling. *An Analysis of MMOG Subscription Growth - Version 21.0*, June 2006. `http://www.mmogchart.com`.

[40] Subhash Suri and Joseph O'Rourke. Worst-case optimal algorithms for constructing visibility polygons with holes. In *Proceedings of the second annual symposium on Computational geometry*, pages 14–23, 1986.

[41] Bart De Vleeschauwer, Bruno Van Den Bossche, Tom Verdickt, Filip De Turck, Bart Dhoedt, and Piet Demeester. Dynamic microcell assignment for massively multiplayer online gaming. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–7, 2005.

[42] Anthony (Peiqun) Yu and Son T. Vuong. MOPAR: a mobile peer-to-peer overlay rrchitecture for interest management of massively multiplayer online games. In *Proceedings of the international workshop on Network and operating systems support for digital audio and video*, pages 99–104, 2005.

[43] Li Zou, Mostafa H. Ammar, and Christophe Diot. An evaluation of grouping techniques for state dissemination in networked multi-user games. In *Proceedings of the ninth Internation Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 33–40, 2001.