Visibility–based microcells for dynamic load balancing in MMO games

by

Alexei Sumila

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
September 2011

Abstract

Massively multiplayer games allow hundreds of players to play and interact with each other simultaneously. Due to the increasing need to provide a greater degree of interaction to more players, load balancing is critical on the servers that host the game. A common approach is to divide the world into microcells (small regions of the game terrain) and to allocate the microcells dynamically across multiple servers.

We describe a visibility–based technique that guides the creation of microcells and their dynamic allocation. This technique is designed to reduce the amount of cross–server communication, in the hope of providing better load balancing than other load–balancing strategies.

We hypothesize that reduction in expensive cross-server traffic will reduce the overall load on the system. We employ horizon counts map to create visibility based microcells, in order to emphasize primary occluders in the terrain. In our testing we consider traffic over a given quality of service threshold as the primary metric for minimization.

As result of our testing we find that dynamic load balancing produces significant improvement in the frequency of quality of service failures. We find that our visibility-based micro cells do not outperform basic rectangular microcells discussed in earlier research. We also find that cross-server traffic makes up a much smaller portion of

overall message load than we had anticipated, reducing the potential overall benefit from cross server message optimisation.

Acknowledgments

I would like to thank Professor James Stewart for continuing guidance and support. I would also like to thank the School of Computing for keeping me a student for all these many years.

Glossary

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Massively Multiplayer Games

Massively Multiplayer Online (MMO) games began with text-based games in the 1980s and became immensely popular and increasingly complex. A Massively Multiplayer Game distinguishes itself from other multiplayer games in two ways: quantity of users and state persistence. The number of players online simultaneously is typically expected to be at least several hundred, with regular multiplayer games typically supporting 64 players or fewer. An MMO game also maintains a persistent state for all players: the world in the game changes and progresses even if no users are interacting with it. This happens because the game world is hosted on the dedicated persistent server which maintains and progresses the world. The gameplay of MMOs differs wildly but most revolve around control and maintenance of a single in-game avatar. Modern MMO games often feature large, continuous 3D worlds and allow users near-complete freedom of movement within the world. One of the most popular MMO games, World of Warcraft, had over 11 million active subscribers as of January 2010 [4].

Due to limitations of network hardware, not all players are usually allowed on the same game map at the same time. Games employ "shards" to allow large number of players to play simultaneously. Each shard is a separate game world. Players on different shards have limited or no ability to interact with one another. Therefore new ways of increasing the maximum number of players on a single shard are continuously being developed [6] [16] [2]. The advances are generally made in two categories: reducing the load on the client's connection, and reducing the load on the server maintaining the game.

1.2 Load balancing

The load on a client's internet connection is reduced using prediction techniques [1] and with careful maintenance of the region of interest of the player [8]. Prediction techniques concentrate on minimization of player state updates. The updates are sent only when the client's action is not predicted by the model of behavior composed of the client's previous actions. A basic example of such behavior is movement. When a player reports to the server that he is moving on a certain trajectory, no further notification is necessary until that trajectory changes. Similarly, region of interest (RoI) techniques on the server send information to the player only when that information can be of benefit in the current update cycle: messages from players that are incapable of affecting the player immediately are not propagated to that player.

From the perspective of the server, the two most important issues are network bandwidth and CPU load. As total network bandwidth of the server is proportional to the average bandwidth use of each individual player, the previously mentioned predictive and RoI techniques for each client comprise a large portion of network

load reduction. CPU load on the server has to be balanced with the cost of RoI and prediction techniques, as complicated schema can simply shift the bottleneck from the network connection to processing costs, providing no overall benefit to the maximum number of players.

1.3 Server Architecture

The standard architecture of a modern MMO game is a variant of a client-server paradigm. All messages originating from the clients must be received and verified by the server before being propagated to other clients. Due to the pervasiveness of cheating in online game worlds, verification of all traffic for signs of malfeasence on the server is a necessary step. To reduce this server-side bottleneck, a modern server is an amalgamation of a number of machines all maintaining a single game state but each maintaining only a subset of total server responsibilities [8]. This eliminates a single point of failure and simultaneously increases the maximum number of players that can be serviced. Responsibilities are typically split in one or more ways, most significant of which are *area of game world* and *subsection of gameplay* splits. Subsection of gameplay subdivision will assign servers to maintain different aspects of the player state, such as player movement, player inventory and monster inventory. Game world area subdivision will assign a single physical machine to maintain a certain physical area of the virtual game world. Any player venturing into that geographical area will subscribe itself to the server maintaining the region and will receive updates through only that server.

1.4 Problem

A successful static subdivision for the "area of game world" method is dependent on player behavior and player concentration on the map to be relatively constant. Typically this assumption fails at least periodically as user-initiated or developer-initiated events suddenly and drastically alter user behavior. This is especially true of game world area subdivisions. Introduction of new content into the world or simple evolution of the player base leads to changes in player concentrations on the game map. These changes in concentrations lead to load being unevenly distributed among the various machines comprising the server. To resolve this problem, dynamic subdivision algorithms were proposed, many using a concept of microcells. Each microcell is a subset of what would normally comprise a single region of responsibility for a single machine. This microcell is precomputed and designed to be offloaded to an emptier server to re-balance the load.

1.5 Hypothesis

This thesis proposes that a microcell subdivision based on visibility within a 2.5D terrain can outperform a subdivision into even rectangular regions proposed in other work. Our measure of effectiveness is the frequency of quality of service failures in a player congregation scenario. There is an additional overhead necessary for propagating messages between users who are located on different servers but still are capable of seeing each other. Our microcells, and thus server regions, are separated in large part by visual occluders so this overhead should be minimized. We expect the use of our visibility-based microcells to reduce the frequency of quality of service failures, when compared to existing methods.

1.6 Contributions

We propose a novel approach of subdividing terrains into visibility-based micro cells. The 2.5D height map of the terrain is processed through a horizon counting algorithm to convert the height map into a 2.5D horizon occlusion map. That map is then smoothed and further subdivided into regions using a watershed technique to produce simple enclosed regions, which are then simply mapped to the original height map to produce the microcells. This approach is compared with simple square subdivision investigated in other literature [16]. Our experimental results show that this approach does *not* outperform previously proposed square subdivision in all but very specific cases. If cost effectiveness of each incremental rebalancing is a factor, our algorithm proves itself to be more resiliant. Load balancing algorithms can trap themselves by being unable to shed any more regions in a cost effective trade, a trade that would not increase the overall message load on the system as a whole. If cost effectiveness is ignored the algorithm will perform similarly to rectangular subdivision.

Chapter 2

Background

2.1 Server Architecture

2.1.1 Communication Model

The two most common networking models in use in games today are client-server and peer-to-peer (here to reffered to as P2P). P2P was used by multiplayer games since the inception of networked multiplayer gaming [12]. It provides a simple link between two or more players, making all users responsible for the integrity of the data. Recently, the P2P approach was also adapted and tested on massively multiplayer games in a research environment [9], but large commercial projects have yet to adopt this approach. Developers have traditionally decided against P2P due to the significant complexity involved in maintaining consistent game state, anti-cheat implementations and game update distribution.

The most commonly used architecture for multiplayer games, including MMOs, is client-server [12] [9]. The server is responsible for all synchronization and communication between clients. A client-server architecture consists of a single server which is responsible for maintaining a consistent game state and propagating it to all clients.

The ability to easily maintain a coherent and verified game state remains the reason for such wide spread adoption of this architecture. The separation of tasks between clients and server differs with each implementation.

In older implementations, the clients notified the server of their actions and the server checked the action against its internal game state to verify what the result of that action was. Communications between clients and server were sent across a network which added latency and negatively affected the server's ability to maintain the exact game state on all clients, degrading the users' experience. A simple two client shooting game can be used to illustrate this issue. The first client spots the second client, aims and fires. On the first client's screen he sees that the shot was aligned perfectly. The message about the trajectory of the bullet is sent with some delay to the server. Meanwhile, the second client has actually started moving in a different direction and has sent the update to the server notifying it of the action. Depending on the order in which events arrive on the server, different outcomes may be produced. If the first client's message arrives first, the second client is notified that it was hit. If second client's message arrives first, then the first client's trajectory would be invalidated and result in a miss. If the latter happened, the first client would experience a situation where his "perfect" shot produces no effect. This degrades the experience of the first player.

To alleviate this problem, modern clients often perform more elaborate computations. Clients maintain their own game state and notify the server of the game events as they see them unfold in their version [13]. The server receiving these updates validates and synchronizes them between all clients. In this architecture, the first client's "perfect shot" would have been sent to the server as a message indicating

that a direct hit has occurred. The server would receive the direct hit message and the second client's movement message in some order and would have to decide which sequence of events to declare valid and confirm for both players. In this situation, creators of the server can provide heuristics to choose outcomes based on quality of user experience. With the right implementation, the effects of network lag can be masked significantly, increasing user satisfaction.

2.1.2 Large Scale Server Design

The server in a classical client-server architecture is expected to be a single physical computer. This solution does not scale well and it introduces a single point of failure into the system. If the server fails, the game becomes unplayable. To alleviate this problem, MMO servers are often represented by large clusters of machines forming a single server from the point of view of the client. Internally, there are many ways to subdivide the tasks between the individual machines. In the Terazona [11] system, the server is split into multiple layers. The first layer is the dispatcher server that acts as an authentication and gateway server. All users are required to log on to this server to be reassigned to one of a number of game servers. These game servers are responsible for authentication of messages for the users subscribed to them and for propagation of necessary messages to all other users. When users request game state information, such as their inventory, the game servers communicate to a "sphere server" which redirects the query to an appropriate back end server. All players on the system are part of the same seamless game and they are completely unaware of the server separation, as all their communications are with the game server that they were assigned to.

A number of other commercial and research solutions follow some variant of this design. One common component of these systems is a world map subdivision into sub-regions with players physically located in these sub-regions being serviced by the servers assigned to the sub-region. Newer systems allow seamless region separation, where the user may not necessarily be aware that he is crossing a server boundary.

2.2 Load Reduction

One of the most important goals of large scale server design is to increase the maximum number of players capable of playing in the same world simultaneously. Terazona claims that their solution is capable of supporting 32,000 players simultaneously, where other solutions allow only up to 6,000 clients [11]. With such a large number of incoming connections, the number of updates propagated to users and the size of those updates play very important roles in the overall technical feasibility of the game.

2.2.1 Message Reduction

Reduction of message frequency is achieved chiefly by the use of predictive algorithms, such as Dead Reckoning [1]. Update messages notify the server of the current trajectory of the client and only send further updates when the trajectory changes. Periodically, the client also sends a general update with its entire state, so that the server can verify that information is internally consistent and that no cheating has occurred. This can result in negative artifacts on the client in some cases. In commercial MMO games, such as Anarchy Online and Guild Wars, a failure of the prediction algorithm is known as “rubber banding”. Movement and collision detection is done

on the client with trajectory updates sent to server for confirmation. If the server decides that a move performed on the client was not valid (for example due to a minor descrepency between trajectory on client and server) the user then experiences a "rubber band" where he is suddenly snapped back to the position at which the discrepancy began. The frequency of such artifacts increases as the quality of service provided by the server decreases, further highlighting the need for efficient service maintainance schema.

2.2.2 Interest Management

Once a client and server have synchronized the sequence of events between themselves, these events need to be propagated to other clients. In order to minimize the amount of forwarding necessary, the server maintains an RoI for each user. There are many different implementations of RoI [5]. Among the oldest and simplest is the Nimbus implementation [5] which subdivides the region of interest of each player into two or more circular areas centered on the client itself. The inner circle represents the area where the client will receive the maximum number of updates from other actors. Successive outward circles reduce the number of updates to reflect the fact that events happening farther away will produce a smaller effect on the client's screen. A client walking several steps away from the player is likely to cross the entire screen and will bear complete scrutiny of the player. The same event happening a long distance down the road from the client's avatar will only affect a few pixels on the player's screen; thus a crude approximation of the event is likely to look the same as a very detailed one. Some research indicates that Nimbus is among the best performing RoI schema despite its simplicity [5].

2.3 Server Subdivision

2.3.1 Static Division

One feature common to many large scale server cluster architectures is a subdivision of the game world into geographical regions. Each geographical region is maintained by one of the servers comprising the server cluster [11] [8]. Each server assumes responsibility for all players located on the part of the world assigned to it. There are two prevalent paradigms of connecting the regions belonging to different servers: *explicit gateway control* and *transparent*. If explicit gateway control is employed, the user is required to cross a specific boundary or gateway which does not allow interaction with anything on the other side of the gateway. In Anarchy Online and Guild Wars, such gateways are represented by a shimmering wall which becomes apparent when the user gets close enough to it. Beyond a shimmering wall the user can see static terrain features but he is unable to detect any interactive actors. Upon crossing the shimmering wall the user is typically greeted with a loading screen. When loading is complete the user finds himself in the next zone. The user requesting a transfer to a different region inadvertently passes through an admission control mechanism which may deny him entry if the server is overly loaded or down. This approach allows the global game system to maintain acceptable maximum loads and to easily handle server failures. If one of the many servers maintaining the world fails or is overloaded, only the people interested in entering that part of the map are affected. One example of similar work in academia is that of Assiotis and Tzanov [2], which describes splitting the terrain into segments and explores all the details of player synchronization within a segment and across segment boundaries.

In transparent region subdivision, no explicit gateway control exists. The user

perceives the map as one single region that does not require any loading screens. The differences between regions in this case are typically thematic and only affect the user in minor ways. For example, in World of Warcarft [17], the only consequence of crossing from region to region is the change in chat filters to the local region. In this method the user is able to interact with actors across server boundaries. The server that the user is subscribed to is responsible for propagating updates to the other servers if one of the users arrives close to the border region. The region in which a user's status needs to be propagated across server boundaries is typically confined to a relatively narrow band around the perimeter of the region. The width of the perimeter is dependent on the maximum extent of the RoI visibility. The exact details of message propagation depend on the specific implementation of the system and can differ significantly. Such a system typically is unable to gracefully fail a single server as all servers have to communicate information to the neighboring servers. Without an explicit gateway control the client would have little idea that he is venturing into another server and when that server is down there is no thematic way to cover up the failure. With an explicit gate on the way the player can always simply be told he was temporarily denied entry without much immersion breaking.

The additional overhead of propagating messages to the neighboring servers makes player congregation on server boundaries undesirable. To address this issue, game designers typically design server borders to be either as narrow as possible for regions with high traffic or sufficiently uninteresting to discourage players from lingering in the area. For example, primary entrances to two most populated towns in World of Warcraft [17], Ironforge and Ogrimmar, both consist of narrow winding corridors which do not allow users inside and outside of town to see each other.

2.3.2 Dynamic Subdivision

Both methods of static subdivision are susceptible to the changes of player concentration. Throughout the life cycle of an MMO, population concentrations are expected to change due to the population maturing, as well as through community events. A typical MMO design has different regions to provide different degrees of difficulty. As a player becomes better and stronger he is expected to move to more difficult regions. This leads to different distributions of players through the life cycle of a game as the number of expert players takes time to grow, and later can subside due to players leaving to play other games. This forces the designers of an MMO server split to create subdivisions which can tolerate future changes in the population density. This is challenging as such numbers are difficult to predict, because the popularity and design of each individual game can significantly alter the outcomes. Populations often experience strong temporary spikes in player density due to player-initiated events, such as community gatherings and clan or faction wars.

Several dynamic subdivision algorithms were proposed to address these issues [6] [16]. The general premise of such algorithms is a dynamic adjustment of areas of responsibility of each game server. These techniques work on the principle that the total user population remains manageable and thus adjusting the size of the regions for each server can balance the load and allow all servers to maintain the necessary Quality of Service (QoS).

One important component of dynamic subdivision strategies is the cost of performing region adjustment which occurs when population densities change. Depending on the schema involved, the cost of offloading the client from one server to the other during the region adjustment steps can prove to be a significant part of the overall

cost of the procedure. For example, in De Vleeschauwer et al. [16], the cost of migrating a player from server to server is 15, on a scale where a single message update is counted as 1. Propagating the user's message to the other server once costs 0.3 on the same scale. Other papers, however, choose to disregard this value as it is heavily dependent on the architecture of the servers. For example, in the Terazone [11] style architecture, most of the relevant player state is stored on the content servers behind the sphere server, so the migrating cost would be significantly less.

2.4 Noncommercial Research

When analyzing features available in existing game and advertised in commercial solutions such as the Terazona [11], one can see that there exists a great degree of sophistication in solutions already deployed to end users. Unfortunately, details of the techniques used in such solutions remain proprietary. In academia there is a growing literature addressing various problems. For example, Boulanger et al. [5] discuss various RoI techniques in great detail, with testing done in a real networking environment using an MMO framework. This work concludes that a concentric circles RoI performs no worse than much more complicated region of interest algorithms, a conclusion we use in the selection of our RoI implementation. In work by Assiotis and Tzanov [3], a solution to the problems arising from region–based seamless world subdivision is discussed. Details cover event propagation, locking mechanisms and conflict resolution.

Two solutions representing micro cell offloading that were the direct predecessors of this thesis are works by Chen et al. [6] and De Vleeschauwer et al. [16]. Chen applies a microcell implementation to a peer-to-peer MMO. That paper uses heuristics

of server locality and map adjacency to optimally balance the network. The paper shows the necessity of dynamic load balancing. More directly applicable to this thesis, Vleeschauwer's paper covers Client-Server architecture specifically and deals with splitting the world into starting cells, which in the paper are represented as rectangles, and then further subdividing the world into rectangular microcells. These microcells are traded between servers when quality of service on an overloaded server cannot be maintained. We use the approach of this paper as the basis of our testing.

Recently Kazem et al. [10] have proposed a visibility driven load balancing based on layered hexagonal division of the world map. This approach addresses the same issues as this thesis but does not directly take visibility into account when creating the microcells.

Chapter 3

Methodology

In order to improve on previous work, we chose to segment the terrrain based on actual visibility, as opposed to assumed RoI visibility based on a distance around each microcell. In order to simplify the task of subdivision, we chose to assume that terrains are represented by a height map. This 2.5D representation of terrains is common, as it simplifies map development itself, and allows the use of many optimised rendering technologies. To convert the height map into the representation of visibility inside of it, we chose to utilize a horizon computation algorithm [15]. The horizon algorithm computes the skyline as it is seen from every sample point of the terrain. The resulting horizon map does not guarantee that all points inside the horizon are visible from the test point, but it is reasonable to assume that the points on the horizon are major occuluders. We postulate that map subdivision based on the most prominent occluders, such as those revealed by horizon computations, would provide a good basis for visibility based subdivision.

Our segmentation algorithm is based on two assumptions: segmenting terrain based on visibility will reduce cross server boundary communication cost; and horizon maps produce good information about inter-region visibility.

This chapter describes our approach for computing microcells from a height map. We begin with an outline of the algorithm and its individual steps in Section 3.1. We proceed by providing an outline of the dynamic load balancing algorithm in Section 3.2.

3.1 Microcell Computation Overview

The microcell computation performs the following steps:

1. Data is loaded in the form of a height map, where each pixel represents one of 256 possible heights for the terrain. 2.5D terrains represent a common approximation of game terrains. Typically non 2.5D elements such as caves and buildings are accessed through portals, which excludes them from the terrain map.

2. The height map is converted into a horizons count map in which each pixel stores a count of the number of times that pixel appears on some horizon (Section 3.1.1).

3. All local minima are found on the horizons count map, and each local minimum is assigned a unique integer identifier (Section 3.1.2).

4. A waterfall algorithm is applied to grow the minima into larger regions (Section 3.1.3).

5. A minimal spanning tree connecting all of the regions is constructed (Section 3.1.4).

6. The number of regions was made to match the number of regions in a block test by merging enough of the smaller cells with their neighbours using the minimal spanning tree (Section 3.1.5).

3.1.1 Horizon Counts

For each point a horizon is computed using Stewart's algorithm [15]. The horizon at a point of the terrain consists of a set of other points that separate the terrain from the sky, as seen from that point. After computing the horizons, we count, for each terrain point, how many times it appears on a horizon. Due to the discreet nature of the data, the horizon count map (Figure 3.1) is very noisy, so the map is smoothed by using a Gaussian filter with a 7 by 7 pixel matrix and a standard deviation of one. This is done in Matlab using the following code:

H=fspecial('gaussian',7,1); I=imfilter(I,H); I=imfilter(I,H);

The filter is run twice on the image in order to connect adjacent disjointed regions, producing a less noisy map. bf New: The same result could be achieved by running a large filter one

3.1.2 Minima Finding

The local minima are identified as those points that are minimal in their 8-neighbourhood: For each pixel, P, we examine its 8 neighbors to determine if they are of the same height or higher. If the neighbor pixel is of the same height it is added to a neighborhood queue. If the pixel is higher it is left alone. If any of the pixels are lower then P is disqualified. When all neighbors are examined the algorithm repeats on each pixel in the queue, until the queue is exhausted or the whole region is disqualified.

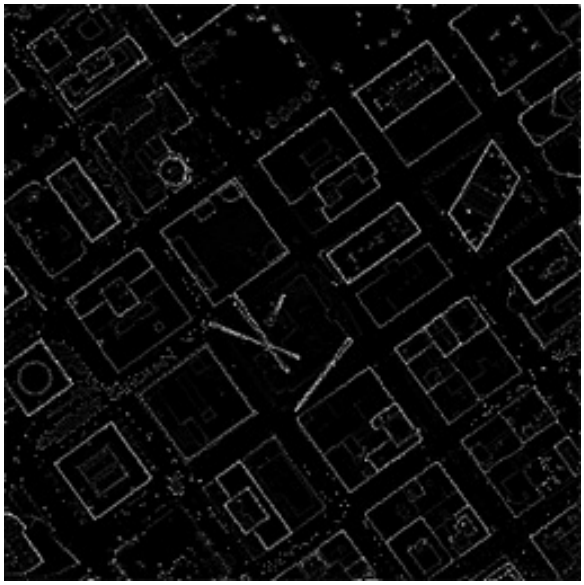

Figure 3.1: Input horizons map. This map is generated from a LIDAR scan of downtown Houston. The pixels are monochrome and represent values 0-256, where 0 is black and 256 is white. The value of each pixel is proportional to the number of times the pixel appears in a horizon count of another point.

If the queue is exhausted we designate the pixel, P, that started the queue as a local minimum. All pixels in the queue were at the same height, so any one of those pixels is a suitable candidate to be a local minimum. Any examined pixels, except for those that disqualified a region, are marked as used and are not considered in future tests.

3.1.3 Waterfall

Conceptually, the waterfall algorithm simulates submerging the terrain into a body of water. Each local minimum will create a new independent lake. The water level in the lakes will rise (Figure 3.2). Eventually, the lakes will meet and the locations where they meet will represent the watershed lines. To simulate this effect we first mark

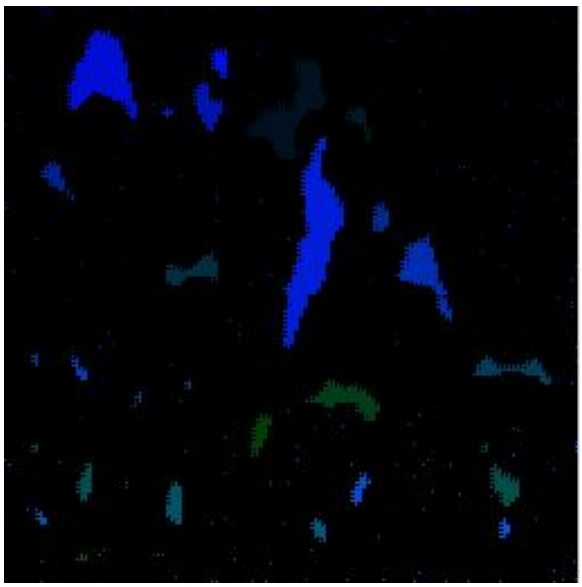

Figure 3.2: Early steps of region seeding. The region is being "flooded" from the lowest level up. At this stage only the deepest features of this Martian Canyon have been processed. Black pixels represent unprocessed levels

each minimum with a different seed colours. Then, beginning with lowest possible pixel level and proceeding to the maximum the algorithm fills all unmarked pixels at that level with special neutral color.

All of these neutral pixels that have adjacent colored pixels and all their neutral colored neighbors take up the seed color of the colored pixel. This way, pixels at the current level increase the "pool" around each seed. Neutral pixels with multiple adjacent seed colors, will pick the first color arbitrarily. These pixels occur on the watersheds. All the combinations of colors surrounding that watershed pixel and its height will be stored in an array for later use by the Minimal Spanning Tree.

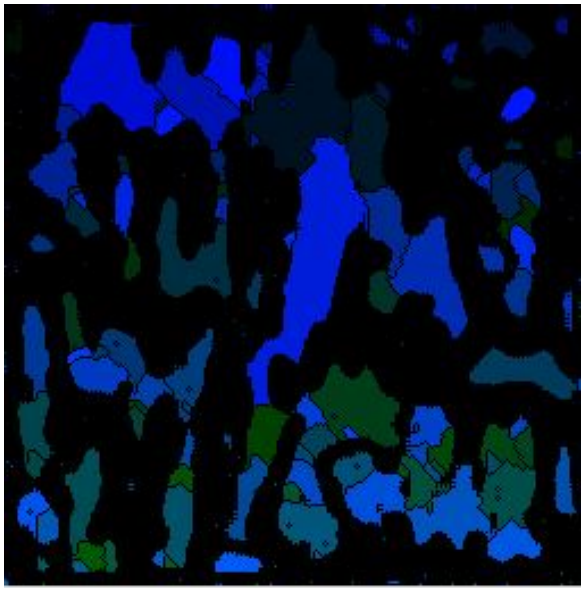

Figure 3.3: A more advanced stage of the process started in the previous figure

3.1.4 Spanning Tree

The minimal spanning tree represents the sequence of "lakes" of the waterfall joining together, where the bigger lakes would overtake smaller lakes as the flood goes over the watershed line. A leaf node of the tree represents a minimum. An interior node is made for each pair of lakes that join at a watershed. The internal node stores the watershed pixels. The spanning tree allows us to have a good subdivision of the map which we can use as guidance for joining the regions when we have to reduce their number later.

3.1.5 Cleanup

The size of each region is the number of pixels that it contains. Total number of regions was made to match the baseline geometric test. Smaller regions were merged into bigger regions until the total number of regions matched in both test. The

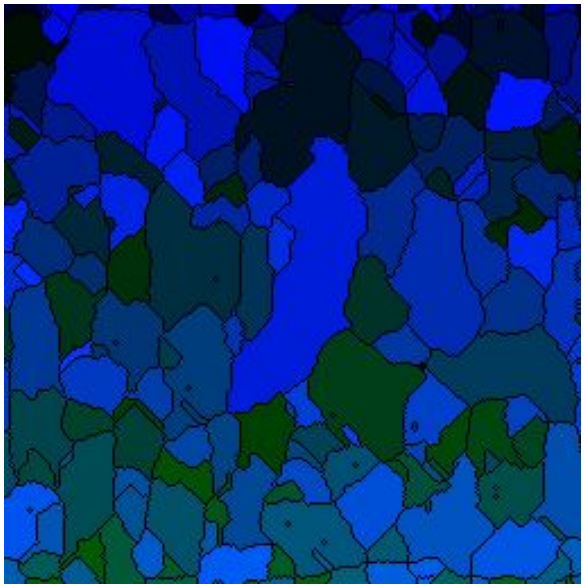

Figure 3.4: End result of the flooding. These are all available clusters before the cleanup. Each is distinguished by a specific color which becomes a unique identifier in the system

merge was performed by merging the removed regions into the parent region on the waterfall's minimal spanning tree.

3.2 Dynamic Load Balancing Overview

The Dynamic Load Balancing algorithm performs the following steps:

1. Load the microcell map, in which each pixel stores the integer identifier of its microcell.

2. The entire map of the game terrain is divided evenly into rectangular regions, among all available servers. If a server boundary divides a microcell, the server that holds the minimum pixel that began the microcell will expand its region to include the entire area of the microcell (Figure 3.5).

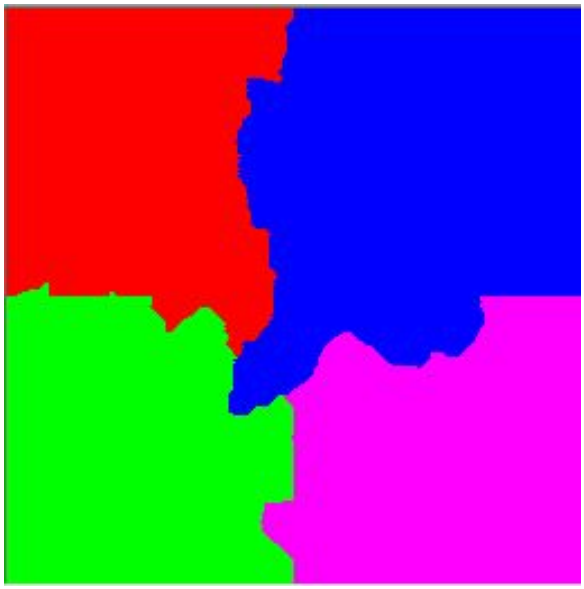

Figure 3.5: This is an initial region subdivision of the same Martian Canyon presented in previous figures. The map is divided between four regions and then lakes that span multiple regions are handed over to the region that holds the initial minimum that created the lake.

3. Users populate the server and begin to follow their normal routines. For the purposes of this paper the normal routine is moving towards a specific point relative to a given congregation point. The movement uses A* algorithm. It has been shown that A* algorithm produces movement patterns similar to that of real people. The non congregation behaviour of the clients is not modeled, because for the purposes of testing overload, random non congregation behaviour would merely provide base level load on all servers which can otherwise be modeled by modifying maximum server load threshold.

4. When a server determines that the quality of service it provides for users is too low, the server initializes a load balancing event. Quality of service can be measured in a number of ways, the most direct of which is measuring the

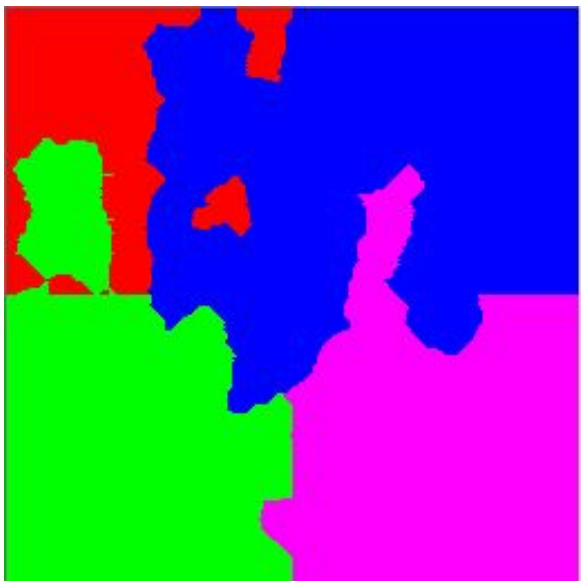

Figure 3.6: With the algorithm progressing, the microcells begin to be exchanged between servers to balance the trafic. Where as in the starting step the area of terrain handled by the four servers were roughly equal, now the red region has lost most of its area of responsibility and it was taken over by the green and blue regions. As the blue region began to overload, the purple region took over some of the microcells from the blue.

average response time between a user request and server's reply. This can be approximated by counting total number of queries processed by the server in a fixed time interval or by the number of players that the server is servicing.

5. When a load balancing event occurs, servers adjacent to the overloaded server report their load status. The server with the lowest load is chosen as the recipient of the extra load. Only the border microcells are capable of being shed to attempt to keep the regions contiguous, but that does not guarantee that they will remain as such, as can be seen in Figure 3.6 where a single red microcell was encircled by the blue region.

6. Users are associated with the piece of terrain that they reside on. By changing

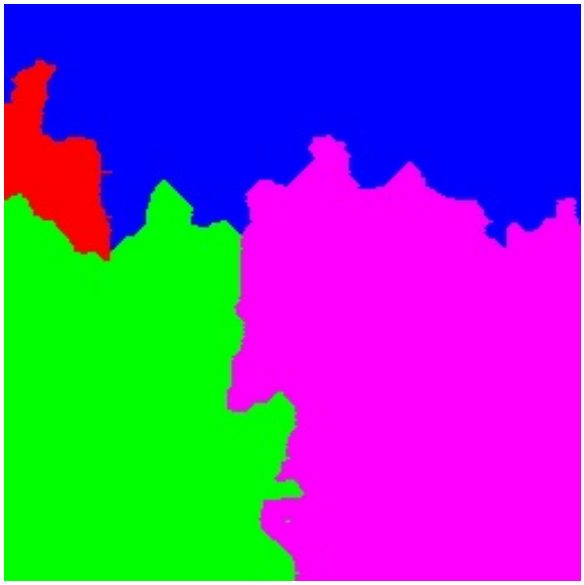

Figure 3.7: A final stage of one of the tests. The red region shrunk to an area spanning only the microcell with the most traffic. Blue has taken over the red microcells which were adjacent to it, but shed the microcells which were adjacent to purple to reduce its own traffic. Green has taken over formerly red regions that were adjacent to it. The algorithm has now reached maximum density of the clients and will not proceed further.

the ownership of a piece of terrain from server to server, the load can be adjusted. The overloaded server will pick sufficient number of border regions to offload to the underloaded server to bring itself to below 90% load, without taking the underloaded server over 90% load. If not enough of clients were offloaded, the server repeats the process with other underloaded neighbouring servers. The 90/90 metric was arbitrary. The best ratios could be found by testing a real system to detrmine how much shed load gives the longest good quality of service.

Chapter 4

Experiments

Dynamic load balancing algorithms attempt to address the issue of severely uneven distributions of players during certain events. As game terrains and player interactions vary greatly in each game, the solution for this problem in commercial titles is often custom. The academic solutions attempt to find a general solution for a specific map type.

We attempt to provide an optimized solution for 2.5D height map based terrains. This type of terrain is popular in many titles, most notably World of Warcaft. Our hypothesis is that a visibility–based segmentation using horizon counts would outperform a simple rectangular segmentation, for servers where the cost of shedding regions is low.

It was not necessary to test true dynamic load balancing using a real network. The complications presented by testing on a physical network, should not affect the condition we are testing. The traffic is a function of total number of clients and would affect any microcell based load balancing algorithm similarly. Instead, the new algorithm was tested using a virtual experiment where real network load was approximated by using a metric and user movement patterns were approximated by

forced congregation, forcing the worst case scenario which is necessary for testing this algorithm.

The cost of the offload was *not* taken into account. Such costs would depend heavily on the server distribution solution used. This cost would largely depend on factors such as the location of user data and the size of user data. If each user has only a few things that need to be remembered about it, the cost of offload of a densely populated region would still remain negligible. For example in a first person shooter game like Halo, the data would consist of the chosen appearance, which two weapons player is using, and how much ammunition he has. If players have large and readily accessible inventories with hundreds of items and many abilities (the states of which have to be remembered), the cost would have to be taken into account. The issue with taking the cost of offload into account is that of expectations. If one expects the overload to persist then it is still worthwhile to pay the offloading cost since, in the long term, that would create less load on the servers than suffering from overpopulation. If the cost of offloading is worth paying in the majority of the situations (with hard–to–predict short term poplation spikes being the only exception) we consider disregarding the offloading costs as an acceptable simplification.

4.1 Experiment Setup

4.1.1 Datasets

Our test datasets consisted of an artificially generated mountainous terrain, a section of downtown Houston, Texas, and a Martian canyon.

Figure 4.1 shows a mountainous terrain in which a ring of mountains surrounds a central depression. This was generated using free World Builder software package.

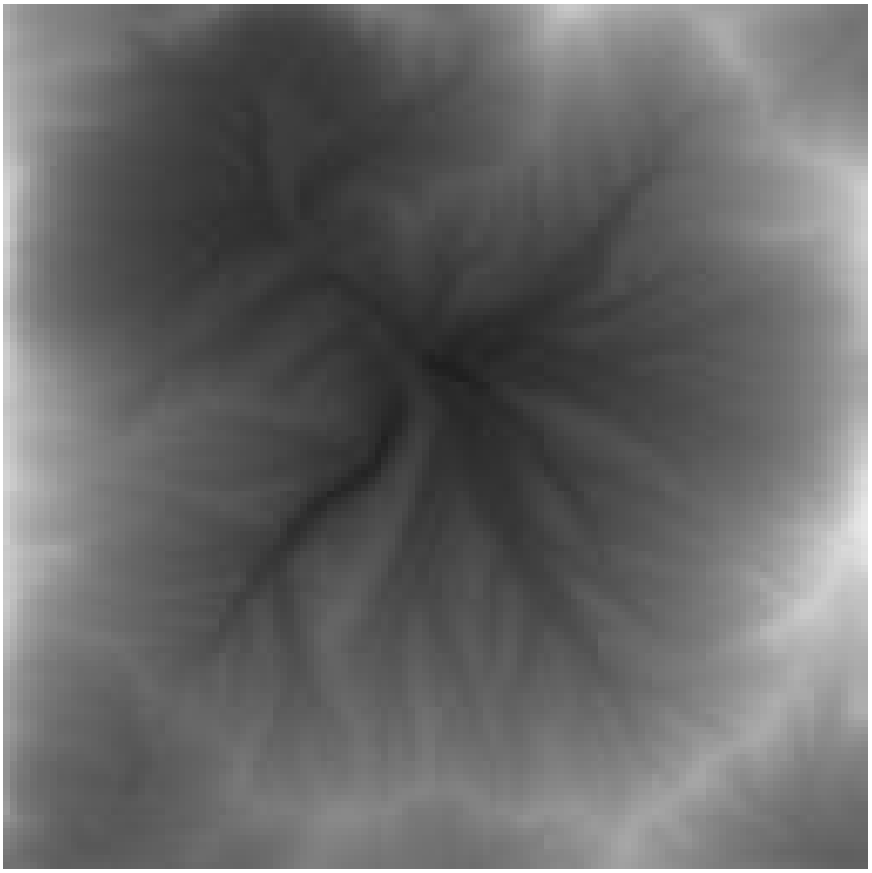

Figure 4.1: Mountainous terrain. Brighter pixels are higher.

This terrain permits players to see other players far across the depression, but not to see players in adjacent valleys.

Figure 4.2 shows a cityscape consisting of a LIDAR digital elevation map (DEM) of downtown Houston, Texas. City terrains are common in modern and postmodern genres of MMO games featuring large cityscapes. However, cities are typically not generated using terrains, but using separate models placed on top of a flat terrain. We have retained this map as a stress test. The city terrain provided a greater challenge to visibility–based microcells, as long, straight streets provided little opportunity for occlusion. Thank you to TerraPoint, LLC for providing the lidar data of Houston,

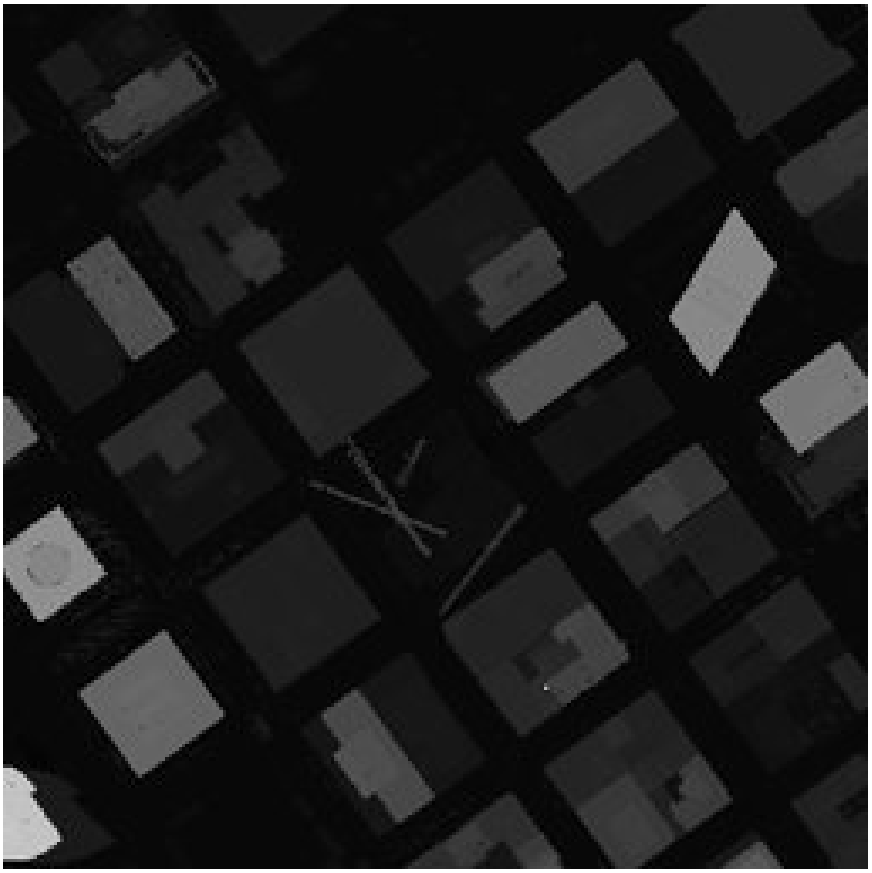

Figure 4.2: Houston, Texas. Brighter pixels are higher.

Texas.

Figure 4.3 shows a LIDAR DEM of a Martian canyon. The canyon terrain represents the classical "fantasy" game terrain where there are few large buildings and most region separation is done by mountain ranges. The canyon also provided an almost ideal case for visibility–based microcells, as it had many semi–enclosed visibility regions. This data set is also interesting because it is a real world example of the type of terrain that is the intended target for this algorithm. The Matrian terrain data provided by Michael Caplinger of Malin Space Science Systems was processed

Figure 4.3: Martian terrain. Brighter pixels are higher.

from the data on the Planetary Data System cd-rom volume MG_2007, "Mars Mosaicked Digital Image Model(MDIM) and Digital Terrain Model(DTM), version 2,0," assembled by Eric Eliason, Raymon Batson, and Anthony Manley. This cd-rom is available from the National Space Science Data Centre, Code 633, Goddard Space Flight Center, Greenbelt, MD 20771

4.1.2 Client Movement Patterns

The players were represented by entities that knew their present position, their final destination and an optimal path to the destination. At the start of the algorithm all

entities were randomly distributed throughout the map. Clients were assigned their final destination chosen randomly somewhere inside a "flocking region." The optimal path from start to finish was determined by an A* algorithm [7], chosen because it is one of the most common AI path finding algorithms used in games and because it has been shown [5] that computer controlled AI players using A* produce metrics that are close to those of real players. All players moved one pixel width along the precomputed A* path at each simulation step. The A* path was weighted so that uphill movements were penalized, and downhill movements rewarded, to simulate a player's natural movement cost. The A* algorithm used a based weight of one for movement in any direction, plus or minus ten percent of the slope of the terrain. With these metrics, clients would favour traveling downhill as it would cost less than one, and avoid travelling uphill as it would cost more. If movement would change elevation by more than ten (resulting in the potential cost of either 0 or 2) the path would be deemed illegal and client movement in that direction forbidden.

4.1.3 Metrics

In order to approximate server costs, we have to count the total number of messages being transmitted at any given snapshot in time. The message is considered transmitted if both clients are in each other's RoI and terrain allows them to see each other. The visibility distance for the radial RoI was chosen to be 10 units for a 256x256 map. In practice commercial titles have much smaller RoI radii relative to the map size, but this distance is continuously being expanded to provide greatest amount of immersion that technology will allow.

For all the test players on the map, we record the number of clients seen in the

same region (i.e. controlled by the same server) but in other microcells, in the same region within the same microcell, and in other regions. The distinction is drawn between clients within the same microcell and clients outside, so that during the offload the server can check if offloading a region does not produce more network traffic than it offloads, which can happen if most of the traffic is between microcells in the same region.

The overall server load is calculated as

$$\text{load} = \text{internal messages} + \text{cross server messages} \times \text{cross server multiplier} \quad (4.1)$$

For the purposes of this experiment, the cross server cost multiplier was chosen to be *two*. This is to account for the cost of forwarding a message from a player on one server to the other server, as well as some negotiation costs that could arise when dealing with objects appearing on both servers. The multiplier in a production environment would depend heavily on the underlying architecture.

4.1.4 Region Offloads

The algorithm only offloads microcells bordering on a different server in order to maintain some locality of regions, as there is a cost associated with transferring players from region to region during the players' movement. This metric is not part of the problem we are trying to minimize and, as such, does not factor into computations. This metric, however, makes offloading only on border regions most favorable, as a "checkerboard" of regions would incur the highest costs associated with normal player migrations.

The algorithm for off-loading microcells has two different modes. In one mode,

called **non-cost-effective offloading**, microcells are offloaded as long as the region is overloaded. The only check performed in this case is to verify that the region receiving microcells does not go over its load limit. The other mode, called **cost-effective offloading**, also checks whether it is cost effective to offload a region. In this mode, each microcell checks whether it will create more cross server traffic than the internal traffic it would create, by taking into account the messages that are addressed to clients inside other microcells on the current parent server. If that number, multiplied by the cross-server multiplier, is higher than cost of all internal messages plus the cost of cross server messages that this cell creates, the microcell is not overloaded.

Chapter 5

Results

Each run of the algorithm consisted of fifty movement steps and produced twelve measurements per step. In each step, each of the four servers produced three measurements: an number of internal messages, a number of external messages, and an overall server load calculated from Equation 4.1.

The reduction in the number of cross-server messages is the primary goal of this work, in the hope the such a reduction would lead to better load-balancing that other algorithms.

To evaluate the quality of load balancing, we measured the percentage of time each server load was above a particular threshold.

We performed sixty runs of each algorithm. In each run, the same client starting positions were used but with a different congregation point, resulting in a different movement pattern for each congregation point and each terrain. The congregation points were created using a Sobol quasi-random low-discrepancy sequence [14] to ensure sufficient spread over the terrain.

The same starting positions and the same sixty congregation points were used for all algorithms and all terrains, so two runs using two different algorithms with

the same congregation point and the same terrain could be directly compared. In all tests, the number of clients was kept constant at 300. We chose this number of clients based of several factors. At the low end this is more than double the number of clients that can be found on the largest capacity non massive multiplayer games, which stands at 128. We also wanted to keep the number manageable to reduce the computation time of the test harness. Our map is significantly sparser than what a real game map would be, with only 256 discrete points a client can occupy in either dimension. If the sampling of the map was increased by at least an order of magnitude in each dimension, and the clients number scaled accordingly, the resulting number of roughly 30000 clients would represent maximum load greater than best existing solutions, providing a good stress test. The starting number of clients is not important in these tests as the stress level is a product of the client number. If we had increased the client load tenfold we would simply had to increase the maximum load threshold tenfold to achieve the same stress level.

Our reasons for choosing four servers are similar. Four servers are easy to manage, and represent a reasonable downsclaing of a real production server number, given the low sample rate of our terrain. Additionally, it was also observed that most scenarios end with either two or three servers participating in load sharing, with fourth server not needing to take any extra load. We would need a significantly large map to benefit from having more servers.

During a run, the clients traversed the map toward the congregation point. A client's path was computed using a deterministic A* algorithm. Therefore, given the same start and end points and the same underlying terrain the path would remain the same in all runs. Note that a client might not reach the congregation point if, for

example, the congregation point is too close to the edge, forcing some of the clients to stop at the edge of the map or if, for example, the client could not reach the destination due to unclimbable obstacles.

Three algorithms were considered:

- We implemented a **static** algorithm to serve as a base comparison, representing a system with no load balancing algorithm in place. With the static algorithm, the map is divided into four quads, one for each server. These quads are not adjusted in any way during a run.

- We implemented the **terrain** algorithm described in Chapter 3.

- We implemented a simple **blocks** load balancing algorithm that exchanged rectangular microcells between servers. This allowed us to test whether the proposed terrain based algorithm outperforms an existing dynamic load balancing algorithm.

  The blocks algorithm divided the map into four regions, one per server, just like the terrain algorithm. It then divided each region into equally-sized rectangular microcells in the same number as the visibility-based microcells used by the terrain algorithm (this number varied with the particular terrain). During a test run, the blocks microcells were handled in the same manner as the terrain microcells.

Two variants of the blocks and terrain algorithms were tested: one a cost-effective (**CE**) variant and the other a non-cost effective (**NCE**) variant. The cost effective variant would not unload a microcell if that action would increase the total number of messages in the system.

The CE/NCE distinction was created to evaluate the potential of a load-balancing algorithm that seeks to minimize the load and not simply prevent failures by keeping the load below a given threshold.

The same number of microcells was used for both the block and terrain algorithms, while the static algorithm used no microcells.

Three different types of terrains were tested: a real canyon (Figure 4.3), a large artificial crater (Figure 4.1) and part of a city (Figure 4.2).

For each of the three terrains, the five algorithms were run 60 times each and the three measures (number of internal messages, number of external messages, and percentage of time overloaded) were compared using a two-tailed paired t-test. P values less that 0.05 were considered to be significant.

5.1 Canyon Test

The canyon represents a semi self-enclosed terrain, based on real life Mars data (Figure 4.3).

In the worst configuration of clients, a total load across the four server of about 6000 could be reached. If the congregation point is exactly at the centre, this load would be evenly divided between four servers using the static algorithm (with some additional load from cross server messages) resulting in no overload on any servers. This number is further reduced by natural client-to-client occlusion provided by the terrain. But this ideal situation is rarely reached.

We tested using three load thresholds (2500, 3500, and 4000) corresponding to conditions of frequent, occasional, and rare overload, respectively.

5.1.1 Frequent Overloads

This set of tests used 2500 as a threshold, creating frequent overloads. Threshold in this case refers to the maximum cost of messages any individual server can take before it starts to fail quality of service. The load is computed using the Formula 4.1. Table 5.1 shows the results for a load threshold of 2500.

Internal Message Counts

For internal message counts we find that terrain and block algorithms in both their varieties (CE and NCE) are significantly better than a static subdivision. There was a significant difference between the static and block algorithms ($p < 0.001$). But the block and terrain algorithms were almost identical ($p = 0.97$), suggesting that the terrain algorithm has no significant effect on internal message counts.

External Message Counts

For external message counts, the static algorithm had significantly less traffic than the other algorithms ($p < 0.003$). The static algorithm does not adjust its borders dynamically, so cross-server messages will be generated only by clients in close proximity to one another across a border, or when the congregation point happens to be very close to the server boundary.

It is interesting to note that dynamic algorithms will move the server boundaries closer to the congregation point as part of their load balancing process and would often attempt to unload a portion of the congregation area to a different server to maintain better load balance.

The terrain algorithm significantly outperforms the blocks algorithm ($p = 0.01$),

Table 5.1: Test results for the canyon with a load threshold of 2500. The average and 95% confidence interval are shown for each measure.

Algorithm	Internal Message Count		External Message Count		Percentage Time Overloaded	
Static	1557	(1456,1658)	16.5	(8.1,24.9)	27.2	(23.6,30.7)
Blocks-CE	1285	(1183,1386)	35.9	(26.5,45.3)	14.3	(10.3,18.4)
Terrain-CE	1283	(1173,1392)	21.6	(16.2,27.0)	13.3	(9.3,17.4)
Blocks-NCE	1285	(1183,1386)	35.9	(26.5,45.3)	14.3	(10.3,18.4)
Terrain-NCE	1283	(1173,1392)	22.0	(16.6,27.4)	13.2	(9.2,17.2)

strongly supporting our hypothesis that cross-server traffic is reduced when terrain is taken into account.

Percentage Time Overloaded

For percentage time overloaded, the static algorithm was significantly worse than either of the terrain and blocks algorithms ($p < 0.001$). This (and the other overload tests presented below) shows the benefit of dynamic load balancing algorithms and their necessity in cases where server infrastructure could be easily overwhelmed by user congregations.

The terrain algorithm was not significantly different from the block algorithm ($p = 0.72$). These tests appear to indicate that cross-server traffic does not play a significant enough role in server overload. Only if the relative weight of external messages (2 in our load equation) were much greater would a difference emerge.

5.1.2 Occasional Overloads

For these tests the threshold was chosen to be 3500. This reduces frequency of overloads. Table 5.2 shows the results for a load threshold of 3500.

Table 5.2: Test results for the canyon with a load threshold of 3500. The average and 95% confidence interval are shown for each measure.

Algorithm	Internal Message Count		External Message Count		Percentage Time Overloaded	
Static	1557	(1456,1658)	16.5	(8.1,24.9)	9.5	(6.6,12.3)
Blocks-CE	1416	(1308,1524)	25.3	(17.0,33.6)	3.7	(1.4,6.0)
Terrain-CE	1344	(1231,1457)	17.4	(12.6,22.2)	4.2	(1.4,6.9)
Blocks-NCE	1416	(1308,1524)	25.3	(17.0,33.6)	3.7	(1.4,6.0)
Terrain-NCE	1341	(1228,1453)	17.7	(12.8,22.6)	4.0	(1.3,6.6)

Internal Message Counts

Results of this evaluation show a very similar picture to the one presented in the Table 5.1. As the frequency of the overload decreases, the static subdivision begins to approach the performance of the dynamic algorithms. The difference between static and blocks subdivisions is almost statistically significant ($p = 0.058$). The difference between static and terrain is statistically significant ($p = 0.006$).

External Message Counts

As the frequency of overload decreases so does the number of times that the inter-server borders are moved. Because of this, the difference between the static and the dynamic algorithms also decreases (static vs. block has $p = 0.13$ and block vs. terrain has $p = 0.10$).

Percentage Time Overloaded

The static algorithm is still easily outperformed by the dynamic algorithms ($p = 0.002$ for static vs. blocks and $p = 0.008$ for static vs. terrain). The difference between the blocks and terrain algorithms is not statistically significant ($p = 0.80$).

5.1.3 Rare Overloads

In this test the overload threshold is set to 4000 to further reduce the frequency of overloads. Table 5.3 shows the results for a load threshold of 4000.

Internal Message Counts

At this stage there is no significant statistical difference between all tests. The difference between static and blocks is not statistically significant ($p = 0.24$), nor is the difference between blocks and terrain in both cost effective ($p = 0.23$) and non cost effective versions ($p = 0.23$).

External Message Counts

There are no significant differences in external messages (static vs. blocks has $p = 0.60$ and blocks vs. terrain has $p = 0.44$), likely because the threshold is high enough that the borders are rarely moved.

Percentage Time Overloaded

Despite the limited difference in previous tests, the blocks dynamic algorithm still perform statistically better than the static algorithm ($p = 0.02$). The difference between static and terrain algorithms fails to be statistically significant ($p = 0.44$). With overloads less frequent, the difference between the two dynamic algorithms increases, but does not become statistically significant ($p = 0.24$).

Table 5.3: Test results for the canyon with a load threshold of 4000. The average and 95% confidence interval are shown for each measure.

Algorithm	Internal Message Count		External Message Count		Percentage Time Overloaded	
Static	1557	(1456,1658)	16.5	(8.1 ,24.9)	3.3	(1.5,5.2)
Blocks-CE	1467	(1353,1581)	19.6	(11.3,27.9)	0.9	(0.0,2.1)
Terrain-CE	1368	(1250,1486)	15.9	(11.1,20.7)	2.3	(0.3,4.4)
Blocks-NCE	1467	(1353,1581)	19.6	(11.3,27.9)	0.9	(0.0,2.1)
Terrain-NCE	1368	(1250,1486)	15.8	(11.0,20.6)	2.3	(0.3,4.4)

Table 5.4: Test results for the crater with a load threshold of 2500. The average and 95% confidence interval are shown for each measure.

Algorithm	Internal Message Count		External Message Count		Percentage Time Overloaded	
Static	1863	(1745,1981)	20.2	(10.3,30.0)	35.9	(32.6,39.1)
Blocks-CE	1490	(1370,1609)	47.3	(35.4,59.2)	24.7	(20.5,29.0)
Terrain-CE	1482	(1374,1591)	41.7	(31.2,52.2)	21.6	(17.3,25.9)
Blocks-NCE	1490	(1370,1609)	47.3	(35.4,59.2)	24.7	(20.5,29.0)
Terrain-NCE	1474	(1366,1582)	43.3	(32.9,53.7)	21.1	(16.7,25.5)

5.2 Crater Test

The crater was an artificial terrain generated with watershed erosion lines running towards its centre (Figure 4.1). This created locally occluded regions but with more open global visibility (an average client can see most of the map but not necessarily nearby clients).

5.2.1 Frequent Overloads

Table 5.4 shows the crater results for a load threshold of 2500.

Internal Message Counts

As the Canyon tests (Tables 5.1 and 5.2) established, the static algorithm is significantly worse than the dynamic algorithms, especially as the frequency of overload increases. The crater tests confirm the trend with static vs. block $p < 0.001$. There is no significant difference between the two dynamic algorithms ($p \approx 0.90$).

External Message Counts

External messages are once again higher for dynamic algorithms by a strong margin ($p = 0.0001$ for static vs. blocks and $p < 0.0001$ for static vs. terrain). There is no statistically significant difference between dynamic algorithms ($p = 0.48$ for cost effective and $p = 0.61$ for non cost effective).

Percentage Time Overloaded

As with all previous tests, the static algorithm fails on this test when compared to blocks or terrain ($p < 0.001$). Similar to Table 5.1, the terrain algorithm has measurements that are slightly lower than the blocks algorithm, but not by a statistically significant margin ($p = 0.31$ for cost effective and $p = 0.24$ for non cost effective).

5.2.2 Occasional Overloads

Table 5.5 shows the results for a load threshold of 3500.

Internal Message Counts

The static division still has more internal messages than blocks ($p = 0.002$) or terrain ($p = 0.0002$). There are no significant differences observed between the two dynamic

Table 5.5: Test results for the canyon with a load threshold of 3500. The average and 95% confidence interval are shown for each measure.

Algorithm	Internal Message Count		External Message Count		Percentage Time Overloaded	
Static	1863	(1745,1981)	20.2	(10.3,30.0)	21.0	(17.5,24.5)
Blocks-CE	1590	(1471,1709)	37.8	(27.3,48.3)	8.8	(5.7,11.9)
Terrain-CE	1558	(1452,1664)	35.2	(25.6,44.8)	8.8	(5.7,11.9)
Blocks-NCE	1590	(1471,1709)	37.8	(27.3,48.3)	8.8	(5.7,11.9)
Terrain-NCE	1551	(1446,1656)	36.3	(26.7,45.9)	8.4	(5.4,11.4)

algorithms ($p = 0.69$ for cost effective and $p = 0.62$ for non cost effective).

External Message Counts

The picture in this test has not changed from the 2500 threshold. Dynamic algorithms are still better than the static algorithm (static vs. blocks has $p = 0.02$ and static vs. terrain has $p = 0.03$). This time there is a statistical difference between the two outcomes. Blocks vs. terrain show no statistical difference ($p = 0.71$ for cost effective and $p = 0.84$ for non cost effective).

Percentage Time Overloaded

The static algorithm is far worse than either of the dynamic algorithms in this test ($p < 0.001$ for both static vs. blocks and static vs. terrain). The two dynamic algorithms are virtually identical for cost effective ($p = 0.99$) and for non cost effective ($p = 0.85$). The difference between cost effective and non cost effective was wider in this test but none were statistically significant.

Table 5.6: Test results for the city with a load threshold of 600. The average and 95% confidence interval are shown for each measure.

Algorithm	Internal Message Count		External Message Count		Percentage Time Overloaded	
Static	1863	(1745,1981)	20.2	(10.3,30.0)	21.0	(17.5,24.5)
Blocks-CE	1590	(1471,1709)	37.8	(27.3,48.3)	8.8	(5.7,11.9)
Terrain-CE	1558	(1452,1664)	35.2	(25.6,44.8)	8.8	(5.7,11.9)
Blocks-NCE	1590	(1471,1709)	37.8	(27.3,48.3)	8.8	(5.7,11.9)
Terrain-NCE	1551	(1446,1656)	36.3	(26.7,45.9)	8.4	(5.4,11.4)

5.2.3 Rare Overloads

As the frequency of overloads decreases, both dynamic algorithms perform more and more like the static algorithm, with the ultimate case of no overloads making the dynamic algorithms behave identically to the static algorithm. Due to this overall trend, the rare overload tests were not performed on this or any successive datasets.

5.3 City Test

The city is a map of small portion of downtown Houston, Texas (Figure 4.2). This map was selected to stress test the algorithms. This is reflected in extremely low maximum threshold values that congregation was able to achieve without frequent overload in this terrain. The maximum server load this test ever achieved was 1452. This is primarily due to the large occlusions of the buildings. The clients start in the same spots as all other tests and proceed to the same spots. If either spot is on the rooftop the client will either stop short of destination or will not start at all, as the clients are unable to climb such extreme cliffs (the sides of the buildings are essentially vertical cliffs). The threshold for this test was lowered to 600 to accommodate this behaviour. Table 5.6 shows the results for the city map.

Internal Message Counts

This test still has static perform worse than blocks, but by a smaller (and statistically insignificant) margin ($p = 0.17$). There is a statistically significant difference between static and terrain ($p = 0.07$). There is no significant difference between the dynamic algorithms ($p = 0.69$).

External Message Counts

As was the case with all the external message tests, the static algorithm generates fewer messages than either blocks ($p = 0.26$) or terrain ($p = 0.23$), but not significantly so. The dynamic algorithms once again behave similarly ($p = 0.98$).

Percentage Time Overloaded

The only statistically significant difference found with the city was that the dynamic algorithms outperform the static algorithm ($p = 0.047$ for static vs. blocks and $p = 0.004$ for static vs. terrain). Similar to all previous high frequency of overload tests (Tables 5.1 and 5.4), the terrain algorithm's mean appears lower but not by a significant margin ($p = 0.42$).

5.4 Discussion

5.4.1 Cost Effective vs. Non Cost Effective

The difference between the cost effective and non cost effective modes tests to determine if the overall load will increase if a region is given away to a different server. This only happens when clients within a microcell see more clients in other microcells on their own server than on other servers. The only terrain where there was

any difference was the crater, yet the difference was far from statistically significant ($p \approx 0.80$). One can stipulate that the crater had a slightly higher incidence of the condition necessary for non-cost effective offload to not transfer a region. This may happen because it was designed as a test where there is more near occlusion (due to a multitude of erosion lines on the crater) and less distance occlusion (due to the hemispherical shape of the crater (Figure 4.1).

Ultimately, the distinction between cost effective and non cost effective does not appear to be frequent enough to merit taking it into account when evaluating microcell based dynamic load balancing algorithms.

5.4.2 Threshold Effects

If the threshold for overload is so high that the rate of overload is zero, then a dynamic algorithm begins behaving like a static algorithm: There is never a need to unload anything. As we increased the threshold in our tests, the internal messages of the dynamic algorithms would increase to approach the values of the static algorithms. The cross server traffic would also decrease and start to approach the values seen in the static algorithm.

Interestingly, with internal and external messages approaching the static values and the overload percentage in low single digits, the difference in the overload percentage still remained statistically significant in favour of the dynamic algorithms. This shows that one would benefit from a dynamic load balancing algorithm even if the frequency of critical events is very low.

5.4.3 Blocks vs. Terrain

Out of all the performed tests, the only one with a statistically significant advantage for terrain algorithm was the cross server traffic test in Table 5.1. Cross server averages were lower on all tests. However, on most of these, they were not lower by a statistically significant margin.

This indicates that the value of terrain occlusion in reducing cross server traffic in load balancing scenarios is situational at best. It may prove to be beneficial on specific maps and for a specific frequency of overload (and as such, a high frequency of region offloads), but it is unlikely to be beneficial by a large enough margin to warrant its use.

When we add the analysis of percentage over threshold (which is the most important metric we are trying to minimize) to the evaluation, we see that the terrain algorithm fails to gain any benefit from potentially lower cross-server messaging and, overall, is not statistically better than a much simpler block algorithm.

Aside from its simplicity, the block algorithm also has the advantage of uniformity. The terrain algorithm is limited to the natural topography of the map when creating the clusters, whereas the blocks algorithm is not. Thus the terrain algorithm produces uneven numbers of clusters. In natural, noisy data sets, it produces a large number of very small clusters which need to be merged into bigger clusters. Without additional subdivision of large clusters, it also has the maximum number of regions that it can produce. If the terrain has only a few simple and very smooth features, the number of clusters can be very limited.

The block algorithm does not have this problem. Server designers can easily adjust the number of clusters based on factors that would be dictated by the hardware setup

of the server cluster. As presented, the blocks algorithm is a better choice for dynamic load balancing than the terrain algorithm.

Chapter 6

Summary and Conclusions

We have presented a new technique for creating microcells in a terrain for dynamic load balancing of massively multiplayer online games. The technique attempted to use visibility to create microcells, with the aim of reducing the quantity of cross server messages in the events of sudden player congregation in small regions. We used a horizon count algorithm to generate the visibility clusters. We had hoped that by locating the server boundaries along visibility ranges we will reduce overall load significantly enough to affect the frequency of Quality of Service failures during player congregations.

In some cases, we found that external (cross-server) messages were significantly reduced by this algorithm. However, the overall load, consisting of both internal and external messages, was *not* significantly better with this algorithm. We have found that cross server-traffic made up only a small portion of overall traffic and as such a minor reduction would not have a large enough outcome on the overall frequency of quality of service failures.

More frequently, the use of our algorithm did *not* change the cross-server load to a significant degree. This may be due to a the fact that visibility based microcells

produce much longer borders than the rectangular microcells, which in turn increases the opportunity for the clients to see each other across the borders. Our regions also had varying densities, so for tests where congregation point was surrounded by smaller regions, the server boarder could be located much closer to the densest client congregation than the rectangle based algorithm would allow. If the server border was closer to the congregation center it would also contribute to more cross server traffic.

Dynamic load balancing algorithms were shown to be less prone to Quality of Service failures (i.e. overloading) in all tested terrains and at all failure thresholds. We conclude that there is a strong benefit to using a dynamic load balancing algorithm in multi-server Massively Multiplayer clusters. A microcell subdivision based on simple geometric shapes, such as rectangles, performs similarly to the terrain-based microcells, so there appears to be no advantage to using the more complicated terrain-based microcells.

6.1 Future Work

The problem of better microcell creation remains an open question. We believed that visibility-based microcells should have increased performance over simple rectangular subdivisions. Some of the avenues of research that we chose not to pursue could yield better results, as could an augmented algorithm to further subdivide the regions which this algorithm fails to segment.

A hybrid algorithm which combines horizon segmentation with rectangular or other simple segmentation for regions with dimensions significantly longer than the visibility range could prove to be more effective than simple rectangles.

A different approach to solving the problem could be to abandon horizon counts as a metric and attempt a statistical analysis of the potential visibility set of the terrain to create the visibility regions. This technique would be more time consuming but it would be a one-time-only precomputation step.

It is possible that cross-server traffic could be reduced by using a locality heuristic to keep the regions compact. Such a heuristic could, for example, compute the center of mass of any server and only allow a region to be shed when the offload would not change the centre by more than a certain portion of standard deviation. This compact regions technique should reduce the length of the perimeter between regions, which should in turn reduce the total amount of cross-server traffic.

Bibliography

[1] Jesse Aronson. Dead reckoning: Latency hiding for networked games. `http://www.gamasutra.com/features/19970919/aronson_01.htm` (viewed December 2008), September 1997.

[2] M Assiotis and V Tzanov. A distributed architecture for MMORPG. In *ACM SIGCOMM Workshop on Network and System Support for Games*, page 4, 2006.

[3] Marios Assiotis and Velin Tzanov. A distributed architecture for mmorpg. 2006.

[4] blizzard.co.uk. Press release: World of Warcraft reaches 9 million. `www.blizzard.co.uk/press/070724.shtml` (viewed December 2007).

[5] J-S Boulanger, J Kienzle, and C Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *ACM SIGCOMM Workshop on Network and System Support for Games*, page 6, 2006.

[6] J Chen, B Wu, M Delap, B Knutsson, H Lu, and C Amza. Locality aware dynamic load management for massively multiplayer games. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 289–300, 2005.

[7] R Dechter and J Pearl. Generalized best-first search strategies and the optimality af A*. *Journal of the ACM*, 32(3):505–536, 1985.

[8] M Ferguson and M Ballbach. Product review: Massively multiplayer online game middleware. `www.gamasutra.com/view/feature/2908/product_review_massively_.php` (viewed December 2007), January 2003.

[9] Chris GauthierDickey, Daniel Zappala, Virginia Lo, and James Marr. Low latency and cheat-proof event ordering for peer-to-peer games. 2004.

[10] Shervin Shirmohammadi Ihab Kazem, Dewan Tanvir Ahmed. A visibility-driven approach to managing interest in distributed simulations with dynamic load balancing. 2007.

[11] Zona Inc. Terazona whitepaper. `http://www.zona.net/whitepaper` (viewed 2002), 2002.

[12] Aaron McCoy, Declan Delaney, and Tomas Ward. Game-state fidelity across distributed interactive games. `http://www.acm.org/crossroads/xrds12-1/gamestatefidelity.html` (viewed December 2007), 2004.

[13] Source Multiplayer Networking. Valve software. `http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking` (viewed April 2008), 2008.

[14] I.M. Sobol'. Distribution of points in a cube and approximate evaluation of integrals. *Zh. Vych. Mat. Mat. Fiz.*, 7:784802, 1967.

[15] J Stewart. Fast horizon computation at all points of a terrain with visibility and shading applications. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):82–93, March 1998.

[16] B. De Vleeschauwer, B. Van Den Bossche, T. Verdickt, F. De Turck, B. Dhoedt, and P. Demeester. Dynamic microcell assignment for massively multiplayer online gaming. In *ACM SIGCOMM Workshop on Network and System Support for Games*, pages 1–7, 2005.

[17] worldofwarcraft.com. Map resources for World of Warcraft. `www.worldofwarcraft.com/info/#world` (viewed December 2007), 2007.