MASTER THESIS IN COMPUTER SCIENCE
INF-3981

Frag: A Distributed Approach to Display Wall Gaming

Åsmund Grammeltvedt

June 1, 2006

Faculty of Science
Department of Computer Science
University of Tromsø, N-9037 Tromsø, Norway

MASTER THESIS IN COMPUTER SCIENCE
INF-3981

Frag: A Distributed Approach to Display Wall Gaming

Åsmund Grammeltvedt

June 1, 2006

Faculty of Science
Department of Computer Science
University of Tromsø, N-9037 Tromsø, Norway

Abstract

Computer games with distributed functionality, such as most modern multiplayer games, provide a rich environment in which one can experiment with high performance distribution of computational and graphical resources. Their extremely high requirements for processing power and consistent visual output create a platform with unique demands.

Current tiled display walls provide, as a consequence of their architecture, a large amount of computational resources in the form of a networked cluster of computers, driving the individual tiles. Existing forays into creating games intended for display walls have unfortunately made little use of this power, opting instead to centralise computation on a single host and distributing the resulting graphical information, or by running the same computations on all hosts and displaying different parts of the locally generated information.

We seek to remedy this situation by distributing computation and visualisation in order to exploit available resources to a greater degree. Achieving a higher level of resource utilisation will allow increasing the complexity of games to involve larger environments with more interactive entities than on comparative single host systems. These are approaches that are used to some degree in Massively Multiplayer Online Games, though rarely with the same focus on distribution or the same demands for visual consistency between adjoining tiles that on a display wall.

This thesis discusses techniques for determining and distributing a continually evolving set of information to a cluster of heterogeneous hosts connected by a network. The hosts will be able to simulate a set of mobile entities, where all entities on all hosts share a single virtual environment in which they can interact with each other while keeping a shared and consistent view of the world state. Furthermore, the approach is tailored to visualisation on a display wall.

To avoid having to design games specifically for the distributed system presented herein, a library is introduced as a method of removing all distribution-related responsibilities from the game itself, as long as the game supports a simple common interface and a limited set of callbacks for the library to modify the game. Do demonstrate the feasibility of this approach and study some of its properties, the Frag library is implemented as a prototype according to the specified design. An ordinary computer game is modified to use Frag, after which the system is tested and measured, leading to comparisons between the original and the Frag-enabled game in terms of scalability in game complexity, in addition to observations about the suitability of the approach.

Acknowledgements

This concludes my stay at the Department of Computer Science, University of Tromsø. It has been five great years and I wish to express my gratitude to the staff and all my fellow students for creating an excellent atmosphere, both socially and scientific.

There are, however, a few people whom I will mention in particular: I wish to thank my supervisors, Otto J. Anshus and John Markus Bjørndalen, your suggestions and discussions have been invaluable. Ph.D. students Daniel Stødle and Espen S. Johnsen have been great sources for discussions and help regarding technology in various stages of instability; best of luck with your degrees.

Finally, I must thank my girlfriend, Marte Karidatter Skadsem. Despite what you may think, you have been appreciated.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and background

Display walls offer the possibility of an unprecedented amount of information displayed simultaneously on a single screen. The main physical implications of display wall technology is twofold: The technology allow user to rapidly change the visual context by looking at another part of the screen or moving about the room. Several users may also interact with the system at once, working in separate context while still being able to see what other users are doing, or collaborating in the same context whether working on a document or a computer game.

Computer games can benefit greatly from being played on such a device. Most games have a social aspect, as the point is to play with or against other humans and most modern PC games include at least some form of multiplayer mode, usually allowing players to play together by the Internet. Before Internet games took off, the usual mode of multiplayer gaming was *hotseat* or *shared screen* modes. These were based on players sharing a single computer and either alternated being in control (sitting in front of the monitor and keyboard) or sharing the keyboard and screen between themselves. Obviously, both approaches have little scalability, either because of waiting for one's turn or because a keyboard and a monitor are both spatially very limited resources.

The distributed, yet tightly coupled architecture of current display walls creates many interesting problems when it comes to making use of all available resources. Regarding simulation and visualisation of large data sets, the challenge is twofold.

Firstly, there is an abundance of computing power available, but it is rarely available as a single, easily reachable resource, although the display wall front end has the appearance of a single unit. The usual environment is a heterogeneous one, where many computers with workstation-equivalent computing power are connected in a local area network (LAN). To make use of the environment, one has to parallellise applications to the point where they can run on separate computers, while keeping communication to a minimum. Latency on a normal LAN is several orders of magnitude higher than access to local memory, as to make the speedup negligible if frequent accesses to other computers' memory is needed.

Secondly, if a tile in the display wall needs to display information generated on another host, this information, which can potentially be very large, must be made quickly available. Even more important: Since the tiles generate a composite, but ideally seamless, image, information that is shared between tiles visualising it must have a high level of temporal consistency. If, for instance, a moving entity crosses an area of space spanning several tiles, the parts of the entity located on different tiles must all match up to create a correct image without tears or missing parts. Graphical inconsistencies, such as broken lines, can be very easy for observers to discover and must therefore be avoided.

As should be quite obvious, and indeed has been showed to be the case[45], it is possible to achieve a visually consistent display by running the same program in lockstep on all tiles, all the while ensuring that the tiles receive the same input and pseudo-random variables, resulting in processes that execute the exact same sequence of events, thus staying consistent. This does solve the problem of distributing visual information by making it unnecessary as all the information already has been generated locally. However, this also means that the entire system runs at the speed of the slowest participant, not offering much scalability in terms of game complexity. This approach is not completely meaningless, however, as

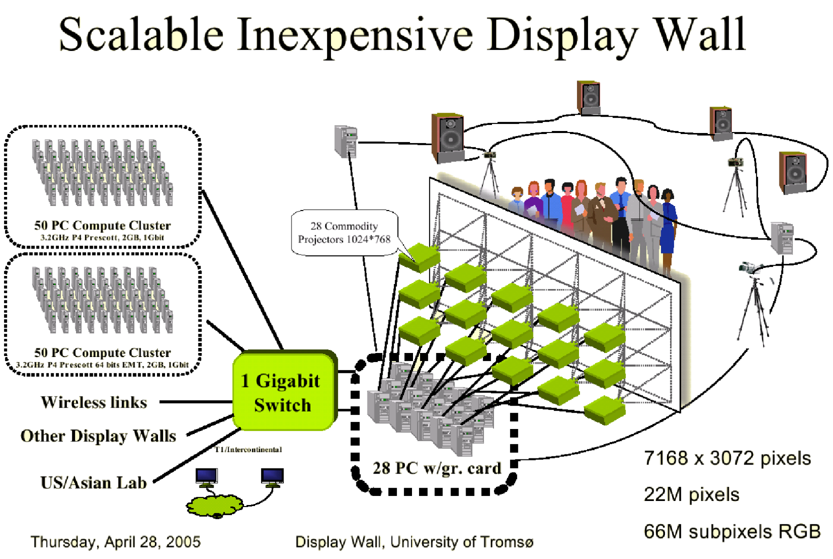

Figure 1.1: Tromsø Display Wall

it renders the game view in a resolution which is very hard to achieve with a single, ordinary graphics processor. WallSpring[45] was shown to run smoothly at a speed of 20 frames per second at the Tromsø Display Wall's native resolution, $7168x3072$, equivalent to 28 normal $1024x768$ screens.

To make maximal use of the available resources, a middle ground must be found where the simulation can be sufficiently distributed to exploit the available computing power, while still keeping the network bandwidth usage manageable. Achieving this does for a big part rest on finding a minimal, necessary set of information that each node needs to receive and also in rearranging the distribution of tasks to reduce the size of the minimal set even further.

1.1 Tiled display walls

One of the motivations for creating display walls is observation that one is unlikely to ever have too much space in which to display information. To the contrary, there always seems to be too much information to display at once in any comprehensible way. Display walls[43] attempts to remedy the problem by creating displays covering as large surfaces as possible, such as a wall. One can, of course attain a similar effect by scaling up a normal screen image with a projector, but this does not really improve the amount of displayable information. In an upscaled image, the number of pixels is constant and independent from the image size, so the information capacity would is constant.

The display wall does not simply scale up the image, it increases the number of pixels. This results in a wall-sized display with about the same resolution as a normal monitor. A tiled display wall achieves this by, as the name indicates, tiling several screens next to each other, either placing normal screens in a custom rack or by using projectors and back-projecting images onto a canvas, taking care to align them to create the illusion of a single, continuous screen, as illustrated in figure 1.1.

The continuous display is, however, just an illusion. Each tile in the display is usually driven by a separate host computer, connected to the others by a local area network. Newer architectures allow a single host to power a large amount of displays[1], but this obviously restricts the maximum flow of information to what a single host can handle. If used correctly, the distributed nature of the tiled

[1]http://www.wedgwood-group.com/xentera_gt8_pci_multi-screen_graphics_card_cable.htm

Figure 1.2: The board game Chess

display wall can be exploited to great effect by having each host focus its computing power on the information displayed on the tile it controls.

Regarding the distributed display wall architecture, there exists several studies on the distribution of large data sets for visualisation[15, 38]. Some of the techniques discussed there do indeed lead to efficient distribution of large amounts of data. However, the techniques generally assume a single source of information from which the visual data is propagated to the tiles. This leads to a focus on how best to distribute complex visual data containing very large amounts of information, while generation of the data is generally not considered to be a problem, or at least tractable using a single host or at most some easily parallellised algorithm. What we are looking for is conversely the distribution of computation tasks in a complex simulation, while still allowing efficient distribution of the resulting graphical output.

1.2 Gaming

1.2.1 A model for games

Games of all kinds plays a significant role in most peoples lives, to the point where modern humans have been nicknamed *homo ludens*[2]. In his book with the term-coining title, Huizinga states the importance of play thus[24]:

> Now in myth and ritual the great instinctive forces of civilised life have their origin: law and order, commerce and profit, craft and art, poetry, wisdom and science. All are rooted in the primaeval soil of play.

Huizinga also defines the concept of play as following[24]:

> Play is a voluntary activity or occupation executed within certain fixed limits of time and place, according to rules freely accepted but absolutely binding, having its aim in itself and accompanied by a feeling of tension, joy and the consciousness that it is "different" from "ordinary" life.

[2]The playing man

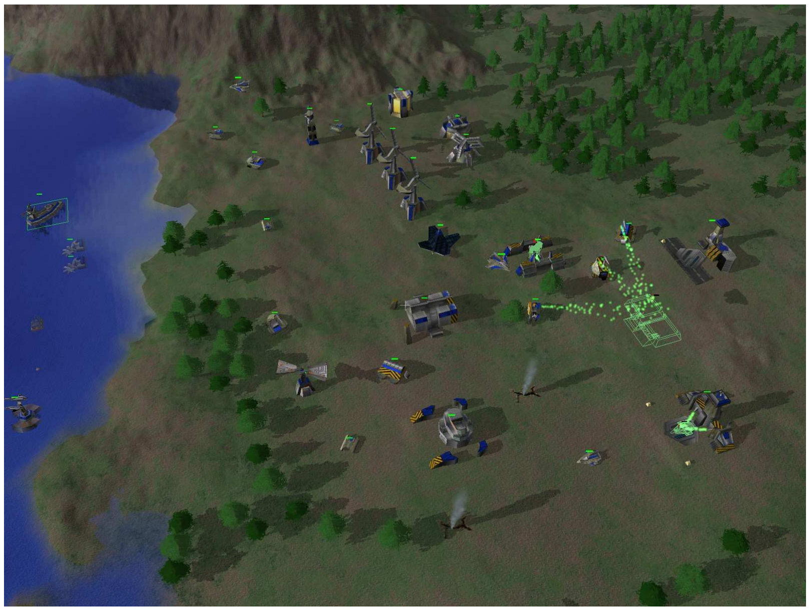

Figure 1.3: Spring, a modern computer game, with mobile entities and a continuous terrain, shown in part

Games come in an endless variety of forms and systems, but they all have the principle above in common. That means that in addition to providing some emotional reaction, all games have at their core some set of rules which govern how the *game world* functions. The game world is the context within which the game occurs and it can be everything from a painted board to the real world with some added rules regulating allowed behaviour. The rules are generally necessary to provide an amount of challenge which again can provoke and emotional response by being overcome or failed.

If one looks at chess, for example (shown in figure 1.2), the model is obvious: The game world exists as a board with 64 squares, making up exactly 64 discrete spatial locations, inhabited by the pieces of which only one can occupy a given location at any time. This extremely stylised representation of the real world is regulated by a set of rules that explicitly regulate how the pieces can move between the squares and attack each other. Following the rules, it is impossible to construct a situation where the rules are not applicable, as the game is internally consistent. The feelings of tension or joy, as Huizinga describes them, are triggered by the players' competitive instinct as the battle for control of the game world.

From this description, one can also observe that chess is a finite state machine, with states consisting of all combinations of pieces and locations, reachable by legal moves.

1.2.2 Computer games

Transferring games to computers maintains the basic properties of games, but allows the games to expand in the areas where the computer offers additional resources. Where to spend these resources has been the basis for a long-lasting discussion among computer game designers, however. The prime contenders are visuals and rule complexity[37].

The former seek to use the computer to realise something that is very hard to do in non-computer games, namely vividly visualising the game world and the events happening within it. The static and inflexible representations offered by gaming boards and pieces pale in comparison to the graphics a computer can create. Indeed, computers are increasingly being used to augment traditional board games to create a richer game experience[28].

The other element is extending the game rules and adding more computational complexity. As the computers excels at computing problems, the platform is ideal for implementing complex rule sets, which would otherwise have been impossible to use. A computer can, for example, with relative ease, implement a complex physics simulation model, making possible games which come close to real life play while still allowing things that would not happen in the real world, such as removal of gravity or the possibility of shooting other players with large weaponry.

Finally, a computer can also act as a virtual opponent, increasing the challenge by making the game more unpredictable. The computer thus creates a synthetic player which can provide pseudo-random input to the game.

We have already discussed the basic model for games, but let us look at it again in the light of the increased rule complexity offered by computers. There does, of course, exist fairly straight conversions of traditional board games to computers, such as the aforementioned game of chess. In these cases, nearly all of the improvement goes into the synthetic player, providing opposition without a real opponent, although there are also computerised games which replicate the original perfectly, adding almost no additional features[3].

Many computer games, however, attempt to expand on the highly stylised world offered by boards and pieces. The game world is then constructed from a variety of elements. It could for example consist of a rolling outdoors terrain, with hills, forests and mountains, in which players could move freely. Alternatively, the world might consist of a large building complex, with rooms, stairs and hallways. The "terrain" thus defines the physical limits of the game. Terrain, or the map, as it is often referred to, is usually immutable to keep certain aspects of the game's complexity in check. In cases where the map is modifiable, there is nearly always some firm boundary which cannot be altered, and which serves to keep players or other mobile entities within the designated game area.

Playing pieces are usually replaced by simulated entities, representing interactive objects or creatures. The degree of complexity in the simulation varies greatly, but it is common to have mobile entities with behavioural-governing logic and several properties which regulate what they can do and how they can be interacted with. While not yet approaching true artificial intelligence, computer games is an area in which application of some elements gathered from AI research is common[26].

1.2.3 Game state

As the map is rather static, it is usually possible to distribute it before the game commences. That is, the terrain data is treated as other resources the game uses, such as audio or graphical data which is used for the game's output. This also applies to some of the entity data, such as graphical models and certain fixed attributes.

The other entity-related information is created as the game goes along and either real or synthetic players manipulate entity properties. In most games, this involves moving the entities directly or giving them commands which the entities then try to fulfil with their internal behavioural logic. These orders are usually limited to indicating a location the entity should move to and modifying behavioural attributes, such as aggressiveness or willingness to take detours in order to reach a destination.

Returning to the finite state machine model, computer games are thus in essence not very different from board games such as chess. However, the computational possibilities of the computers leads to state machines that are many orders of magnitude more complex than those classical games where the players themselves must be able to keep track of the state. Entity positions can, instead of being confined to only 64 separate coordinates, be any of several million unique coordinates. In addition, entities can have a rather complex internal state, not only describing rotation or contortion but also behavioural state.

Describing the state of a running game, therefore, can be a time-consuming task, especially since the flexibility of a computer simulated environment leads to an extremely high amount of possible new states.

[3]The notorious solitaire implementation which accompanies Microsoft Windows is a good example of such games

1.3 Why games?

One might ask if games warrant the special attention given to them here. Should it not be possible to encounter the same challenges and solution spaces in dealing with more classical problems or indeed newer, network based application groups, such as web services?

Computer games are of course not completely unlike any other applications. If anything, computer games use elements form a very wide range of disciplines as they adapt to fit various environments and roles. Computer games can in some instances be seen as a driver for new technologies and techniques, and are often among the first to implement new functionality. The focus on creating the latest and greatest, preferably with the additional feature of offering something completely new, combined with the often more relaxed attitude towards correctness, both in algorithms and implementations, leads to a creative environment where there is little hesitation in pushing the available technology to its limits.

In the context of display walls, we can single out two application domains that appear to have several similarities with games, namely high performance computing and content delivery systems, like certain web services.

As described in 1, the kind of game we wish to see appears to be quite like a normal HPC application. Indeed, one could classify this type of games as a subclass of parallellised simulations, such as parallellised ELCIRC[29]. However, parallellised simulations used in high-performance computing and computer games usually have a different set of goals and therefore use a somewhat different approach. Many applications used in parallel computations are constructed in order to have clean and simple separations of responsibility, for example by studying a problem as a set of cells with certain properties, being affected by their neighbours. This results in a clear definition of which data must be exchanged between participants in order to keep the simulation consistent, usually the cells along the “border” between the areas different participants’ responsibility.

As games usually deal with objects moving around the world and interacting with each other, possibly from great distances, it becomes harder to find a fitting model which allows for simple parallellisation.

More importantly, games are fundamentally different from simulations in that their goal is not to provide scientific data, but highly interactive environments for play. This means that in addition to just keeping all participants loaded with useful work, a game domain application also requires low latency to allow real time operation. Scientific computation can often spend quite long times on each single operation and several can be queued and executed as capacity becomes available, since it is only required that the results become available eventually. A game will need to process objects rapidly and provide the results to other interested parties immediately leading to an onus on very rapid data distribution and computation scheduling, since the execution of tasks cannot be held back without delaying the game.

As such, the demands for interactivity and low latency in games may have more in common with high performance web services, such as the various services provided by Google, which indeed also relies heavily on parallellisation and distribution[4]. These services are structured around allowing large numbers of users rapid access to a large amount of data, much like the requirements of games with distributed computation and visualisation.

Content services are, however, largely directed, as in Google’s search service, where users are offered access to the database, but are unable to directly influence it. In fact, generation of the content itself is seen as rather straight-forward, with distribution being the main challenge. Indeed, one of the aspects allowing the extreme amount of distribution used by Google is the fact that their servers are stateless[4], something which likely does not map very well unto games with distributed computation.

Distributed computer games, then, instead of being clearly placeable within one dicipline, appears as an amalgam of the mentioned diciplines and many more, creating an interesting platform for experimental development.

Chapter 2

Related work

Games have seen a steadily increasing amount of interest from academic circles during the last decade. Especially after network-based gaming began achieving real popularity in the late 1990's, games have been at the leading edge when it comes to creating high-performance distributed systems with strong demands on latency, consistency and bandwidth. Games have therefore become a convenient medium for studying such systems, and the efforts have certainly not been hampered by the increasing amount of resources spent on electronic entertainment.

The joint, steady increase in both processing power and network bandwidth are constantly stretching the limits of available resources and there is a constant push for more distribution in order to handle larger and more complex environments.

However, the concept of large scale, distributed simulations is not a completely new phenomenon. As in many other areas, military research has been leading the way. The Distributed Interactive Simulation[17] (DIS) protocol provided the basis for a geographically distributed system of autonomous simulators working together in a shared, virtual environment. The use of autonomous nodes with no central arbiter can be seen as an early implementation of a peer-to-peer architecture, although the trusted nature of the nodes meant that cheating was not a problem, something which is a major problem for Internet-based games. The main thrust of DIS, which has later been improved upon by the High Level Architecture[25] (HLA), is the development of a common format for exchanging information about "ground truth", or an entity's current status. As long as nodes agree on the information exchange format, their internal implementations can vary.

DIS had the following among its main features:

- No central system for event scheduling or conflict resolution
- Autonomous simulation nodes
- Sending nodes send "ground truth" data, receiving nodes are responsible for how it is actually perceived.
- Each system dead reckons the position of all entities in the simulation, including itself. Entity state data is sent when the simulated and dead reckoned positions diverge by more than an agreed threshold or when a timeout interval is exceeded.
- The shooter determines whether and where a target was hit, the victim determines the amount of damage and its effect.

These properties are emphasised, as they are, at least to some degree, seen in almost all newer distributed simulations, Frag included. One can claim that Frag to some degrees contains a reimplementation of the DIS/HLA protocol. However, we are not just presenting a protocol, but constructing a full and workable system able to work with modern games, something no DIS applications appear to do. Furthermore, our implementation allows us to do extensive testing in a tiled display wall environment.

MiMaze[21] is a classical, fully distributed game, using bucket synchronisation to achieve distributed consistency in a real-time environment. Bucket synchronisation works by gathering and assigning player input into time-separated slots, and having participants wait until a slot has been closed, all new input going into the next slot, before operating on the given input. This is based on the state machine model of the games, where the same input applied to the same state, always results in the same, new, state. All participants can therefore run their own instances of the game, while still ensuring that no inconsistencies arise. This technique have been used for several Internet-based games, such as X-Wing vs. Tie Fighter[27], Age of Empires II[7] and Spring[45], where the complexity caused by large number of active entities made sending state updates infeasible, instead resorting to only sharing input. Common for these systems is that there is no speedup gained from adding additional computers, as all parties execute all operations regardless. It is even required for all parties to lower their computation frequency if one of the connected parties cannot maintain the current frequency, to avoid diverging the parties' game state.

In the display wall context, most relevant studies have looked at the problem of distributing visual information to the tiles of the display wall fast enough to, for instance, allow smooth animation. As display walls offer very high resolutions, visuals that attempt to fully utilise the available proverbial information bandwidth of the display must by definition contain large amounts of data and therefore need clever techniques in order to accomplish fast network distribution[15, 38]. Again, these works do not consider increasing the speed of generating the visual information, only that of the distribution. Conversely, the latter is our main goal with Frag.

Distributed Entertainment Environment[34] (DEE) presents the model that has become rather common in modern multiplayer games, with a three way division of the game. The visual model exists at, and is unique to each client, allowing them to view the world as they see fit. The conceptual model describes the properties of objects, how they behave and interact with the environment. Finally, the dynamic model handles the in game interaction between objects, such as collisions and interactions changing objects' state. This model provides a very useful approach to distributing games, as it shows how we can isolate and distribute only a part of the game, that making up the dynamic model. The conceptual model will usually be agreed on an distributed before the fact, while the visual model is generated by each participant on the fly, letting it choose what to display for itself.

OpenPING[31] describes a gaming middleware system, providing many of the features implemented in Frag. OpenPING does not focus on performance issues as much as the value of a common, open and flexible architecture to which games can delegate communication tasks. Although this system is aimed at games constructed for distributed use, where Frag is intended to distribute games with minimal participation from the game itself, the values of an open and easily modifiable platform, which can be shared by many applications are a common factor.

On the subject of general techniques in game distribution a study by Gil & al.[22] contains a review of some of the issues involved in creating a scalable distributed game system. The study touches upon both the elements reviewed in chapter 4 and the general architecture of Frag.

One of the main issues related to distribution of game state, multiple works deal with the reduction of information distribution by overlaying a partitioning scheme on the game world. The general tendency is for servers to take control of one or several partitions, computing everything that happens within, with some protocol for migrating entities between servers when the move between partitions. Adjusting to server load, partitions can either change size or the number of partitions assigned to a server can be changed according to a load balancing algorithm[16, 42].

Yamamoto & al.[44] describes an event delivery system using publish-subscribe and a dynamically overlay partitioned world, limiting information distribution to interested parties. This model or delivering events is quite close to that used by Frag, except for the fact that Yamamoto's system is designed for a server to inform large numbers of players about events and such has a more sophisticated network protocol, while Frag has a rather straightforward solution encapsulated in the Shout system (see chapter 6).

Mercury[8], based on Scribe[14] features a similar publish-subscribe system, but uses dynamic matching instead of fixed partitions. Although scalable, the Mercury, as an effect of its Scribe base architecture as a quite complex routing system involving many hops, something that leads to high latency on event delivery. As Frag is dependent on delivering a high number of events very fast, in order to keep nodes updated, the multi-hop approach used in Mercury is not ideal.

Matrix[3] is a complete system designed to alleviate the load problems that many massively multi-player online games by providing a minimal intrusion middleware which can dynamically partition the world into overlay regions and start servers to which regions are assigned. Player clients communicate with the server managing the region they are in and the Matrix middleware routes player input to other regions when players get close to them, or migrates the player transparently to a new server when the player crosses into a new region.

The Matrix system appears to be somewhat close to what we are trying to achieve with this project. However, it uses simplifying assumptions at a few key points where we wish flexibility: It assumes that most entities have the same area of interest only allowing different values as exceptions, whereas we want to model a game where entities can have radically different range of view. Matrix also assumes that a single server can handle all entities within a certain geographical area. While sound for a homogeneous game, this is not ideal for the design with autonomous nodes which we wish to use. If a certain participant is the only one in the system which can simulate a certain entity, this entity cannot be migrated, but it must nevertheless be allowed to move freely in the world.

On the matter of performance analysis, Bjørndalen makes some interesting observations regarding the tuning of the PATHS system, noting that costs related to network transport, normally considered the main bottleneck, can become insignificant compared to costs related to packet processing[9].

Chapter 3

Requirements

As has been said, what we want to achieve is the ability to play games where the simulation is distributed over many hosts connected by a network. “Many” is intentionally loosely defined, but as a minimum, 28 nodes (the number of tiles in the Tromsø Display Wall) should be supported. Ideally, the system should also be able to draw on nodes located anywhere connected by a reasonably fast local area network. In light of this, allowing several hundred nodes to cooperate seems reasonable.

Connecting a set of nodes together must result in a system which can compute the world state for a single, continuous game environment. This means that an entity should potentially be able to move across the entire game world without having to perform any special actions beyond those imposed by the terrain or other entities. As a result of this requirement, the game world cannot be partitioned into discrete sections in any noticeable way. Forcing the game to be partitioned in this way may reduce the problem to a set of independent problems with no intercommunication, but it also removes the impression of a continuous world, as borders are inevitably noticeable.

As the goal of creating the system is to increase the amount of active entities, the system needs to offer more processing power than an equivalent non-distributed system. Since a non-distributed system probably involves concurrent lockstep computation on all nodes, this basically means that the system should be able to compute more tasks than a single node.

However, only being able to perform the computations is not enough. The system must also be able to keep the game world sufficiently consistent. The use of the world “sufficiently” is defensible since the system is not intended to run distributed scientific *simulations*, but rather game applications. This means that the result does not need to be “correct” or verifiable, only reasonable. This increases the system’s freedom significantly, since the node coupling can be released somewhat and simulations can proceed independently at separate nodes, as long as conflicts and interactions can be discovered and resolved before they become significant. As long as significant inconsistencies do not arise, minor inconsistencies can be accepted.

Still, the system needs to communicate quite heavily between nodes to keep the system sufficiently consistent. It is, for example, not acceptable for entities to spend significant time in each others’ presence without reacting, if they ordinarily should have done so. Even more important, separate nodes cannot maintain a state where significant events, such as the destruction of an entity, has happened in one node’s world, but not in the others. If this occurs, the nodes’ world view, while originally shared, will rapidly diverge, as the unit which only exists on one node performs other significant actions or other entities react to it.

As has already been indicated, the system should exist as a set of nodes which together compute the state of the game world. There is thus no central server responsible for computing and disseminating the results of all actions. This model is receiving an increasing amount of attention in multiplayer games[36], but requires all game-related computations to be executed, or at least verified, at more than a single location, in order to detect errors or cheating attempts. Although the server can be replicated or otherwise share its load with other servers, this still does not reach the potential power of having the clients compute their respective parts of the game. Of course, in an environment such as the Internet, where most other parties are untrusted, letting a client control what happens is extremely prone to abuse. This problem is possible to remedy by having multiple clients double check computation results and has indeed been studied in peer-to-peer gaming contexts[20]. However, as the environment of deployment for this platform is controlled, we can assume that the clients are trustworthy and reliable. Security is thus not given much consideration.

A great advantage of moving responsibility out to the clients, in addition to harnessing more available computing power, is that one can implement the game rules in the clients and even have the clients using differing rules. One could then connect a client to the system, which uses a radically different behavioural logic for an entity type and see how this entity behaved in the shared world. One could also have a client which used a different physical model, but this would likely cause strange and useless effects unless the clients are assigned separate areas, in which they have total authority.

To make the system as flexible as possible, it should also support nodes joining and leaving at any time. Supporting dynamic disconnecting is a virtual necessity, as the system otherwise will have multiple points of failure, leading to an increasing probability of failure as the number of connected nodes rises.

In addition to computing the game world in a distributed fashion, the visualisation of the world must also be taken into account. As the game is to run in a display wall context, care should be taken to actually allow the visualisation of the game world on a display wall. Utilising the mentioned attributes of display walls, this likely entails showing an overview of the world, allowing users to see all or a large part of what is going on. This is again reasonable, as the tiled nature of the display wall means that no single tile needs to see an overwhelming part of the world.

Finally, it is rather optimistic to expect games to be designed with a distributed system, like that described above, in mind. It must therefore be a goal to create a system which can enable distributed operation in existing games with minimal changes changes to the game. Such a library would take all responsibility with regards to configuring the distributed system, distributing appropriate updates and updating local game world copies.

3.1 Performance

The previous effort in display wall gaming scaled to around 1000 objects in WallSpring[45]. As the performance was limited to that of a single computer, should expect more from a distributed system. Since we want to create a generalised library, it is hard to specify some exact target for performance, as the library could be used by a game that scales very poorly even though it does not need to perform much object state generation. Given an unfortunate choice of algorithms, a game could, for instance, see a significant amount of time spent for a single object if some of its computation is a complex function of the number of other visible objects.

As we already have access to and have tested WallSpring, however, it makes for a tangible target for comparison. Adaption of Spring to use the new library will also reveal whether the new functionality is something which can be easily adapted into existing games, without needing the game to be designed around the new functionality.

Considering a single entity, a participants needs a certain minimal amount of state regarding this entity to correctly model it to the extent that other entities can react to its presence or actions. This state varies a bit with the entity's possible actions. A projectile is for example quite a bit more predictable and simple than an autonomous unit. Nevertheless, a rough estimate of a unit's state would be around 64 bytes (not considering compression or delta updates). Given an update rate of 30 world updates per second, this amounts to a stream of $1,83$ megabytes per second for 1000 visible objects.

Even though it it hard to make assumptions when the complete application environment is not known and the library could be put to use in virtually any kind of game, we can indicate some wanted targets for the performance of the library itself. The library is potentially going to perform quite a lot of actions, so making these as efficient as possible is a must. We can therefore stipulate that for a sum of local and non-local objects, the time used in the library per frame, added to the time used by the game itself, should not exceed the time used by the non-distributed version of the game for the same number of units. This is likely to be reasonable for most games with some complexity, although it would likely lead to a problem in games with very simple, and therefore fast, object simulation and huge numbers of objects. Nevertheless, a distribution library should attempt to at least not worsening game performance.

Chapter 4

Reducing transferred state

Observing the sheer amount of information that makes up the complete game world, it is unlikely that all this state can be manipulated in a distributed environment without severe performance costs. Network capacity itself might not be a direct problem, especially not as gigabit and faster networks are becoming common in local networks. Using applications across the Internet can of course still impose direct limitations on the amount of information transferable data per second, but in a cluster- and display wall environment, processing costs, rather than bandwidth, are likely to be a larger problem.

This assumption is based on the notion that received information must be processed to some degree before being available and useful in the game. How much processing is required will of course vary somewhat with implementation, but at the very least, deserialisation of game state having been serialised for network transfer will be necessary. If the receipt of a message from the network means that an entity specified by the message must be located and updated with the rest of deserialised message by various member functions, it is clear that there must exist some limits on how many messages can be processed without noticeably affecting the game performance, for example by causing lower frame rates.

One must also consider the impact on game logic by having to consider excessive numbers of entities. A naive implementation might use entity behaviour logic having execution time being a function of the total number of units. Informing a participant of the existence of vast number of units can therefore also have negative effects on the game internals if the numbers are scaled far beyond what the game was designed for.

It will therefore be necessary to reduce the amount of information distributed to each participant, both to keep message processing overhead reasonable and also because it, after all, would be possible to saturate even the most powerful network if the size of the game world and the number of entities were increased sufficiently.

Fortunately, there are some techniques which we can apply to the distribution of game state in order to keep the information flow under control.

As has been pointed out, a participant who sees everything that happens in the game world will be overloaded when activities in the world passes a certain limit. To remedy this, we can postulate that a participant has no need of knowing about everything that happens. Indeed, by applying data dependency analysis, we see that for an entity to function correctly, it usually has a set of information it needs to receive, which can include information about the immediate game world surroundings or certain kinds of other entities. If we therefore study a single participant managing a number of units, it is possible to recognise updates which none of the managed entities have any interest in and which does not affect them in any way. These updates can then be discarded without costly processing or possibly without even being sent to the participant.

Finding the set of information that is necessary, allows us to discard all other updates. However, even with this optimisation, the necessary updates may pose a problem. Game entities are likely to change their position or other state very rapidly, potentially leading to a large number of updates. Even if we were to assume that the dissemination and application of these updates did not pose any problems, there is still a problem of latency and temporal separation of updates. Small delays in delivering updates can

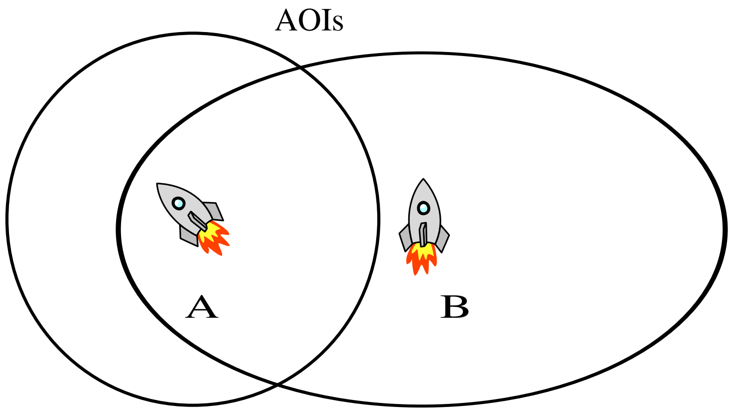

Figure 4.1: Area of interest filtering: B receives updates for A, A receives no updates

be quite easily noticed by human observers[33] and can also confuse behavioural logic of entities, if they believe the other entity to be in the wrong place.

This is of course assuming that entities not managed locally are only modified when a corresponding update message is received. What can be done is to realise that entities are usually quite predictable, at least for a short period of time into the future. We can therefore reduce the rate of updates and instead replace some of the updates with predictions of what the would contain.

When considering the notion of predictability, we make the assumption that games do not need to behave as true physical simulations. While the latter must produce a correct and verifiable result, a game only needs to be *sufficiently* consistent as to be entertaining for the players. A certain amount of inconsistency is usually acceptable to players of network games, both because the human eye and mind might not notice small and temporary errors and because such glitches are an accepted part of the network gaming experience. Indeed, in combing real-time simulations with interactivity and a network, keeping all participants consistent all the time is nearly impossible. One can execute all clients in lockstep and not allow any effects of input until everyone has received the update and acknowledged it, but this would result in a rather low level of interactivity, if one considers that latencies in multiplayer games on the Internet often ranges from 50 to 350 milliseconds[2].

4.1 Interest filtering

In most games, it is reasonable to assume that an entity is more interested in some entities than in others. By identifying the interesting entities, one can discard the rest and thereby reduce the need for updates. Interest can be both spatial and attribute oriented[30].

The spatial approach is probably the most common, and depends on entities only interacting with other entities close to themselves. In this case, we can identify a distance beyond which an entity is unable to interact with or see anything. If an entity is beyond this distance, it has no consequence for the local entity and can be safely ignored. Note that this does not mean that if, for instance, the remote entity fires a projectile in the direction of the local entity, there cannot be any consequences for the latter. If the projectile nears the local entity, it does indeed become of interest, as it might even hit. Obviously, the local entity needs to know about the projectile. The trick is to not focus upon the remote entity which produced the projectile, but rather to consider the projectile as its own, separate entity. To the simulation, it does not matter exactly where the projectile came from, but it can be treated as any other entity as soon as it enters the local entity's area of interest[1].

[1]It might, of course, be relevant to know who fired the projectile in order to count points or such, but this only requires an abstract handle to the originator and not knowledge of its in-game representation or current state.

It is worth noting that an area of spatial interest does not have to be centred upon the owning entity. If one, for instance, imagines a person looking through binoculars, that person will observe an area at a remote location, while perhaps not paying attention to his immediate surroundings. Areas of interest should therefore be rather freely placeable so that different requirements can be supported. The area of interest can also be shaped, if an entity looks in one particular direction, as illustrated in figure 4.1.

As an alternative to spatial filtering, one can also filter information by subject. This can be useful if one models different sets of sensors which have large coverages but are used to detect different entities. This approach is quite prevalent in military systems, where the need to simulate radically different weapon and sensor systems is great[11].

For all methods of filtering by interest, it is important to keep in mind that the defined interests are a means to reduce the amount of game state which the host needs to see. It is therefore extremely important to keep interest to a minimum. If an entity decides that it needs to see the whole world, all updates must be propagated to it, and the network load may diminish the advantages of computation distribution.

4.2 Dead reckoning

Dead reckoning is a technique using the assumption that entities are predictable to reduce the number of updates that needs to be transferred from the generator, simulating the units, to interested participants[36]. The notion of predictability is based on the fact that most games, as they simulate the real world to some degree, use Newton's First Law, which states that an entity in motion stays in the same state unless acted upon by a net external force. The basic interpretation of this is that entities tend to keep moving in whichever direction they are moving at the moment.

Because of this, we can make the simple optimisation of not sending updates as long as an entity is not affected by any unexpected forces, where unexpected forces are those caused by the entity's internal behaviour or by other entities and thus cannot be discovered without evaluating the entity's state. Constant forces, such as gravity, at least in games where the gravity is immutable, are known to all, are quite inexpensive to calculate and changes caused by such forces thus does not need to be distributed, as everyone easily can predict their results.

As has been said, such prediction invariably leads to inconsistencies, as predicted movement will place an entity at the wrong position if the entity changes direction midway during the movement. In systems where each participant runs the entire game, computing all entities in lockstep, inconsistencies are very dangerous. Making an entity behave slightly different on a single participant will rapidly cause this participant's game state to diverge from that of the others. An entity placed in a slightly different spot can cause other outcomes form behavioural algorithms or might cause a projectile to miss. The single inconsistency then causes a rapidly expanding series of other inconsistencies as it interacts with other entities, bringing the participant along a completely different path than the other participants.

If participants simulate different entities or no entities at all, however, the problem is reduced to visual inconsistencies and causing wrong reactions. This is for example common in Internet-based games where computation is done on a single or a set of servers to which players connect as clients[23]. In these cases, dead reckoning is used to generate fluid animations of movement even though updates are received less than once per frame, but entities' states are promptly overwritten when updates are received. The described scenario of cascading inconsistencies is therefore avoided, although the inconsistencies may affect the player or other client-side processes using the client view of the world state. If she aims an another player and fires what appears to be a precise shot, she may nevertheless miss because the other player's location was only predicted and a received update changed his position enough to throw the aim off.

In doing some entity computation locally on each participant, inconsistency problems appear like those in normal client-server applications as long as not more than one participant generates state for each entity. There is, however, a price to pay in accuracy and correctness. Using approximated data generated by predictions will, as mentioned, be wrong some of the time and calculations, such as collision detection, will therefore be "wrong" some of the time, as a single server computing all entities would have

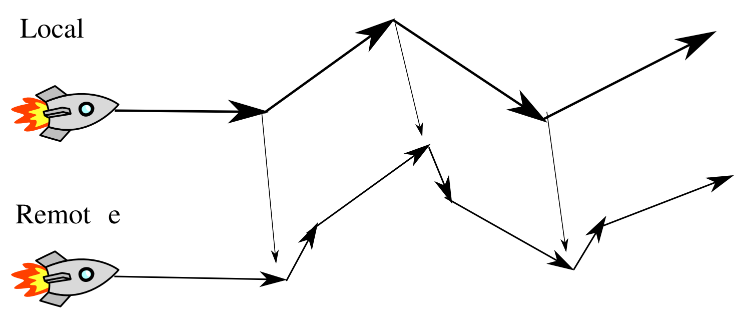

Figure 4.2: A local moving entity and it's dead reckoned remote counterpart.

found another result. As put forth in the requirement for "sufficient" correctness, however, we do not need exact result, as long as they appear plausible, meaning that collisions, for instance, can be allowed to occur even though they technically should not, as long as all parties involved can agree that a collision happened. A system architecture using this approach must therefore ensure that parties involved in an interaction agree on what happened.

Of course, one could extend the concept of dead reckoning by adding further sophistication to the prediction, such as evaluating a simple version of the entity's behaviour logic. However, adding complexity to the prediction algorithm means increasing the time spent on computing that entity's state when the whole goal is to do as little computation as possible. Such an approach would in any case not work if the entity was interactive and controlled by a player or it's behaviour was unknown. This equation represents the fundamental trade off in a distributed simulation, where a certain amount of state must be processed per frame and that state can be generated locally at the cost of processing entity logic or be transferred over the network, costing bandwidth and message processing.

There is also another issue here, which relates to filtering by area of interest. Once we have accepted that some computation is to be done locally, it becomes obvious that reducing the number of entities for which this computation must be done is a good idea. Filtering out parts of the game world and the corresponding entities will therefore also be useful in limiting the amount of computation needed to be done for remote entities.

A reasonable way of determining an appropriate rate of sending updates would be to calculate a delta between the local, and by default correct, model and the result of the predicted model[5]. This approach does require participants to keep copies of their locally managed entities, but this should be bearable, as it needs to keep equivalent representations of all remote entities it is interested in. The added computational complexity will also be negligible, compared to the "real" simulation and it is again done for all remote entities anyway.

Thus, when a participant updates its state, it performs a normal simulated evaluation of all locally managed units. Additionally, it performs dead reckoning updates on all remote entities it is aware of so that local entities can react properly to them and to make a possible visualisation appear as correct as possible. In addition to making predictions for local entities, the participant should also make the same predictions for special copies of its local entities. These special copies do not exist in the game world as such, but are used as a reference to compare local entities with. As one can assume that all other participants see something closely resembling the local prediction model, it is reasonable to send updates when an entity in the local model diverges from the copy in the predicted model by more than a certain threshold. Using this method and by tweaking the threshold value, we can make sure that updates are withheld as long as possible, thereby reducing the message count in the network.

A locally computed entity and its remote counterpart at another host is illustrated in figure 4.2.

4.2.1 Corrections

When it comes to visualising networked games, dead reckoning is considered to be a virtual necessity[36]. If one were to demand fluid movements just based on updates from other participants, these would have to arrive at least as fast as the eye is able to register them and without the temporary delays one often experiences on networks. If any on these conditions are not fulfilled, entities will seem to "warp" between locations, suddenly disappearing, and reappearing at a new spot.

Dead reckoning can be used to extrapolate the intermediate locations between two updates and hopefully place the entity at approximately the same location as the next update on the correct time. An observer can therefore see a smooth movement even though the movement does not really exist as anything else than a guess.

However, the prediction might turn out to be wrong, which is why we want to receive an update as soon as possible when an entity and it's predicted counterpart becomes inconsistent. Even though inconsistencies can be rapidly discovered and corrected, latency or high thresholds can cause the predicted model to be noticeably out of sync before it can be corrected. In such cases, dead reckoning will actually be causing warping instead of preventing it, as the entity must be moved back to its correct position.

Nevertheless, it is possible to remedy this problem by applying even more prediction. Discovering an inconsistency, the dead reckoning algorithm can find and intersection point along the updated entity movement vector, but ahead of the updated point, and continue to move the entity smoothly from its current and erroneous position towards the intersection point. The entity will move along a path which is known to be somewhat wrong, but as long as the error is reasonably small, this will likely appear better than if the entity were to warp to its correct position immediately.

4.2.2 Further optimisations

It is also possible to adjust the dead reckoning algorithm to take into account the need for precise knowledge. One could stipulate that entities only seeing each other across a large distance does not need very precise information and that they would not notice corrections below a certain size. If the entities can tolerate larger inconsistencies and the corresponding corrections, the threshold for sending updates can be increased. Cai & al., for example, suggests a model with four discrete levels of precision, decreasing the update threshold as entities get closer, based on their area of interest and a *sensitive region* within which collisions are likely[10].

4.3 Discarding obsolete updates

Depending on the architecture used to distribute state updates, dropping messages without processing them might lead to reduced bandwidth usage if messages can be filtered before being sent to uninterested receivers. As a minimum, it can lead to reduced processing in the receiver as it can discard messages it recognises as irrelevant. As the game progresses according to wallclock time, it is rarely of interest to receive updates regarding something that happened a minute ago. The information in the message is likely wrong at the current time and might therefore not even improve the local model.

As we have already accepted that participants only see a partially correct view of the game world, we can discard messages if they happened more than a certain time frame ago, as we can expect newer messages lying in the input queue or being sent shortly[32]. However, care must be taken in order to not drop the last update message, as we then may have to wait for a periodical update message. To that effect, the newest update for an entity can be kept, possibly replacing the old, until we are quote sure that no newer updates exist. At that point, the newest update cached can be applied to bring the entity up to data with a minimum of message processing.

This method could be especially efficient in an architecture where updates are routed or redistributed, as they could be stopped at an early point and never reach networks further along the route.

4.4 Data packet manipulation

Looking at the concrete end of the network system, it is also possible to apply some simple optimisations, although they do have certain drawbacks.

The first technique is packet bundling, which is based on the fact that there is a lot of overhead associated with sending small sets of data. The overhead is caused both by packet headers in the chosen transport and network layer protocols that will make up a significant portion of the total used bandwidth if updates are small and separate, and by the increased strain on network components caused by having to process more packets, of which all are treated individually. To lessen the load and attain a higher data to header ratio, several messages can be bundled into one packet before being sent on the network. One method would be to gather updates for all entities and send a single packet for the whole iteration, given that computing the entire iteration does not cause significant latency between the earliest messages being created and being sent.

While this approach would both reduce bandwidth usage to some degree and reduce the load on components doing packet processing, one must also consider both the increased harm done by lost packets and the increased latency caused by gathering data. If many updates are collected into a single packet, having the network drop this packet would mean losing all the included updates instead of just a single one. There might, as mentioned, also occur a certain latency while waiting for enough updates to send a bundle. It is clear that an algorithm for bundling messages must have an upper bound on how long it will wait before sending the data, at least providing a “flush” operation which can be called if we know that no new messages will be sent for a while.

A second technique is the use of compression to reduce the amount of data sent. By doing this, messages would be compressed just before being encapsulated in the transport layer protocol. It is, however, vital that compression is fast. It is not a good idea to have hosts spend significant amounts of time processing messages, a cost which would certainly increase if compression and decompression was slow. The issue of compression also has a bearing on the choice of protocol. Various formats have rather different entropy values, containing various amounts of “unnecessary” information[35]. A Extensible Markup Language (XML)-based protocol, for instance, will naturally tend to be verbose, due to requirements for readability and strictness. Such a protocol will also, because of its verbose and therefore low entropic format, be more susceptible to compression. This in contrast to binary formats which usually start out being quite compressed.

The choice of protocol can therefore also have some bearing on a system’s effectiveness. Aside from bandwidth concerns, there may also be other costs tied to the protocol. Using XML and the Document Object Model[2] to represent messages, for example, is likely to incur extra overhead as nodes must be created to fill the model.

[2]http://www.w3.org/DOM/

Chapter 5

Architecture

This chapter describes an architecture, seen as appropriate for the defined problem, and forming the basis of the Frag library. The first section gives an overview of the components and their functions, while the rest of the chapter describes them in detail and explains the rationale behind them.

5.1 Overview

The architecture of the Frag system consists of three main components, the game, the Frag node and the router, in addition to the router client, their relation shown in figure 5.1.

The game is the base application running all game logic, creating updating and deleting entities, in addition to possibly creating audiovisual output. How the game works and which features it provides is basically not of much interest to the rest of the system, as long as it supports the notion of entities and has been modified in order to call the Frag module. It is nevertheless assumed that the game will not attempt any entity distribution itself. A multiplayer game which already supports some form of distribution is likely to interact badly with Frag, causing unnecessary network communication or creating synchronisation errors as a result of temporary inconsistencies caused by Frag.

Our main interest lies beyond the game, from where it connects to the Frag module. Frag receives entity registrations, updates and deletions as they occur in the game and is responsible for conveying these to the other active Frag nodes. This is done by dividing the world into overlay zones, having no effect on the game world itself, but instead allowing Frag to locate the "home zone" of each entity. The zones act as channels when passed to the router client together with game updates, allowing the system to limit messages to those interested. Interest is expressed as spatial areas of interest, derived by Frag from the entities the module manages.

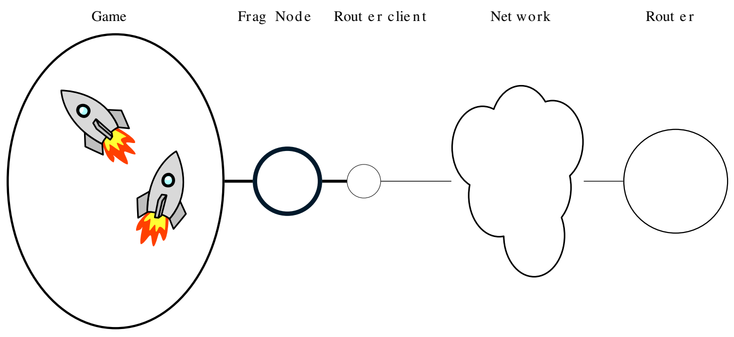

Figure 5.1: Detailed Frag architecture

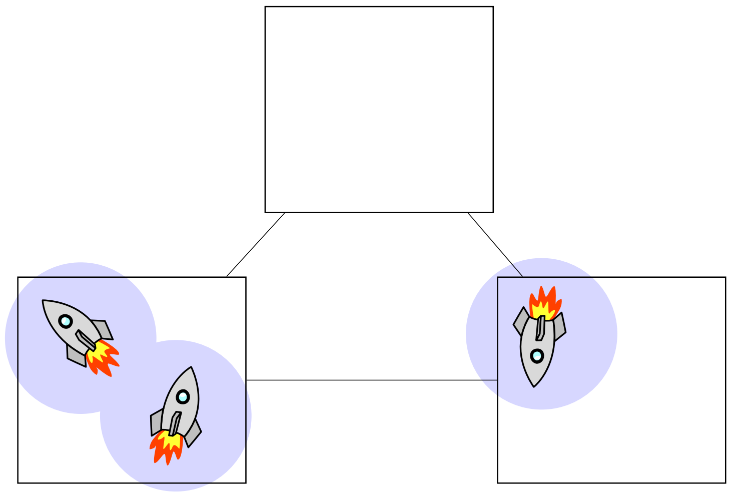

Figure 5.2: A discrete partitioning scheme

Updates are sent and received on the network through the router. The router is responsible for receiving game updates originating in various zones and disseminating the updates to interested Frag nodes.

5.2 Simple approach to zones

Using the described techniques for reducing transferred state extends the area that can be safely viewed without overloading the network. However, there is still a need for an architecture that supports efficient use of the techniques. If we are to implement interest management, for instance, we must have some way of getting updates from producers to interested consumers. The point of limiting the update rate is to reduce incoming bandwidth load, so it is implicit that updates must be filtered before reaching the client. However, it is unlikely that we can use a single server which can collect all updates before sending them on to interested parties, as this server then would have to possess a drastically higher network bandwidth capacity than the common nodes.

The traditional way of constructing this architecture has been to partition the world into a set of discrete *zones*, as show in figure 5.2. The zones are in practise separate worlds, between which no interaction is possible. This approach efficiently regulates the maximal state amount, especially when combined with regulation concerning the number of entities in each zone. Although the zones are separate, they are often used to split up a landscape, where one would expect the world to be continuous. To keep players from noticing that the world is not continuous, barriers are usually placed along the edges of the zone and thereby keep the players “walled in”. It is usually a good idea give such barriers a natural appearance, such as a mountain range or similar. Of course, this introduces some obvious limitations on the available terrain. Indeed, in cases such as a rolling plain, the infamous “glass walls” are sometimes used, where players are suddenly hindered from going any further by invisible barriers and cannot observe any events on the other side even though something should happen there. As should be obvious, this rarely improves game immersion. To move between zones, the common case is for players to explicitly pass through a gateway such as a door or magical portal, beyond which the player is transported to the connected zone. The portals are again usually constructed such that one cannot observe what happens on the other side or impose any effect there.

As the ideal system has no such artificial partitioning of the world, this architecture, while simpler and easier to implement, must be discarded in favour of a more dynamic system allowing the world to be continuous.

5.3 Overlay zones in a continuous world

As massively multiplayer online games have become common and complexity has increased, much energy have been spent on creating more immersive worlds. One of the areas of focus aims at removing barriers used to partition the world into manageable units, while at the same time allowing scalability by distributing load across several hosts. Approaches using continuous worlds where the zone partitioning scheme is not inherent in the game world has been suggested several places[1, 44, 3] and is a rather natural result of combining a continuous world with spatial interest management.

This approach has mostly been studied in the context of player nodes visualising state generated by a cluster of servers to which nodes connect by way of a single front end, providing name transparency. In that context, each player acts as a single node which connects to the server, which is responsible for modelling the game world and computing all effects. The player's area of interest covers what she can see on her display, and updates from her node are generated when the player orders her avatar to perform some action. The server will then compute the results of the input and send updates to interested player nodes. Usually, the only computation happening on a player node is that related to graphics which have no direct effect on the game state. In contrast, what we are interested in is to do computations at the leaf nodes, propagating the effects to interested parties, thus creating a more decentralised structure.

To reduce the amount of information distributed to participating nodes, the game world is partitioned into a set of discrete zones. When all parts of the world thus has been assigned to a zone, all entities can be assigned exactly one "home" zone. If positioned at a border between zones, an entity can be partially within another zone, but as we will show, this is not important as long as there is a clearly defined home zone.

The obvious thing to do here is to only let entities know about other entities in the same zone. If one were to assume a random distribution of entities over the entire world and that all zones were of equal size, this would reduce the amount of information a node needed to S/z, where S is the total game state and z is the number of zones. Indeed, this is almost exactly the simple architecture described in section 5.2. As mentioned, this solution is quite common, especially in MMORPG-type (Massively Multiplayer Online Role Playing Game) applications, where the zones are completely separate and transitions between zones are usually done by portals or some other barrier which prevents interaction between zones.

The problem gets more complex if the zones are required to overlay a continuous world. In this case, entities may well need to interact with entities an a neighbouring zone. Having invisible, but impassable, barriers at random points in the world is not a particularly preferable technique. Entities situated near the borders of a zone and possessing an area of interest, will therefore also need to get information from neighbouring zones, as the area their interest crosses the border.

This scheme, illustrated in figure 5.3, allows us to use zones as ways to identify communication channels in which nodes can share information. The zones define the lowest resolution at which the world is visible to the nodes, so that entities should always see what happens in their home zone. Additionally, entities should also receive updates sent to nodes that are covered by their areas of interest.

The zone system somewhat resembles the technique of "ghost regions", often used in parallel, cell-based, applications[12], where data not updated by the local process are nevertheless fetched and stored at the expected position. This enables the process to use these cells for its computation without having to request the cell values from another process every time one of the cells is accessed. One could say that the home zones and surrounding interest zones together form a ghost region outside which no local entities will need any information, although the lack of task division by area means that the ghost entities will be intermingled with locally updated entities.

5.4 Computation nodes

Computation of entity state happens in game clients, connected to the system as a Frag *nodes*. A node is responsible for a set of zero or more entities, which it updates as necessary. This means that instead of just sending request and attempted operations to a central server which then computes the result, the nodes themselves simulate and update their respective piece of the game state. Each entity, therefore, is

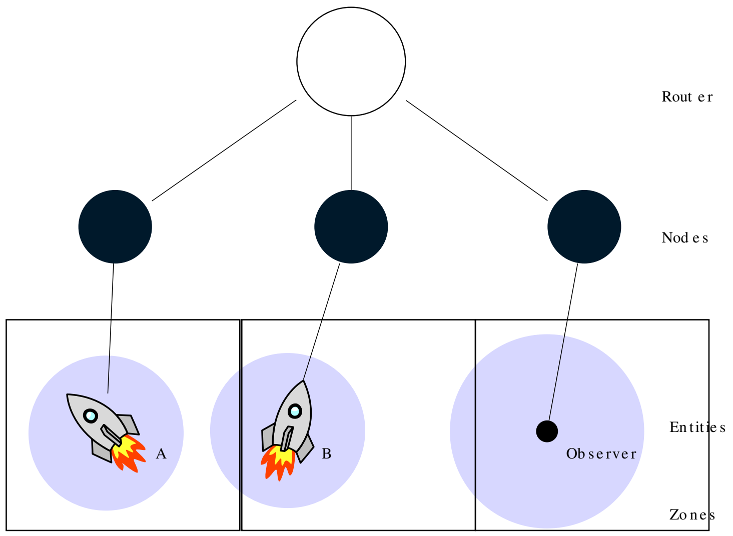

Figure 5.3: The world is partitioned into a set of zones, between which entities can look and move freely. Updates are send to the router which all nodes are connected to and which distributes the update to nodes which have subscribed to the originating zone. A node can observe an area by creating an observer entity which does not generate objects, but ha an area of interest, resulting in incoming data.

owned by a single node, where this node is the only one that can update the entity's real state. A node can therefore ensure that an entity does not enter an inconsistent state, as it is the sole arbiter of that piece of the game state.

The system does not put any limitations on the number of nodes per host. As far as the game implementation allows, there is no problem with multiple nodes and games running on a single computer. This will of course be limited by performance, as computation-intensive games are likely to completely expend the computer's processing resources if several are running at once. There may also be resource conflicts regarding graphical output, if the game insists upon reserving the only available hardware acceleration resource and so on.

To make entity simulation possible in the context of a larger world, nodes need to keep a copy of parts of the game state and therefore also copies of some remote entities, owned by other nodes. To allow dead reckoning, the state of these entities must also be modifiable, although with the understanding that this is only a local and temporary modification, which will be immediately overridden by updates from the entity's owner.

As a result of this, the architecture supports plugging in nodes with arbitrary behaviour, and allows them to insert entities into the game world. The published information regarding entity state must of course conform to an agreed standard, but the local representation is entirely up to the node. A node can then perform whatever simulation it wants to in the context of the larger world.

One can imagine cases where the client does not wish to perform simulations for itself. This could be in cases such as testing an embedded component meant to control a robot, whose mode of operation is to issue control signals to a motor module in a simulated environment. The component is unlikely to have enough processing power to perform some complex simulation on its own. Placing computation in the nodes at the edge of the system does not preclude this option, but it does require the addition of an intermediate layer. The “game” holding the Frag node must simply be modified to accept input from the control component and factor this into its decision making algorithms. Usually there will exist some algorithm which decides how entities react to changes and what they try to achieve. When controlled by an external device, this algorithm must be replaced by a communication layer which accepts input, attempting to operate the entity, and output, providing feedback to the control component through

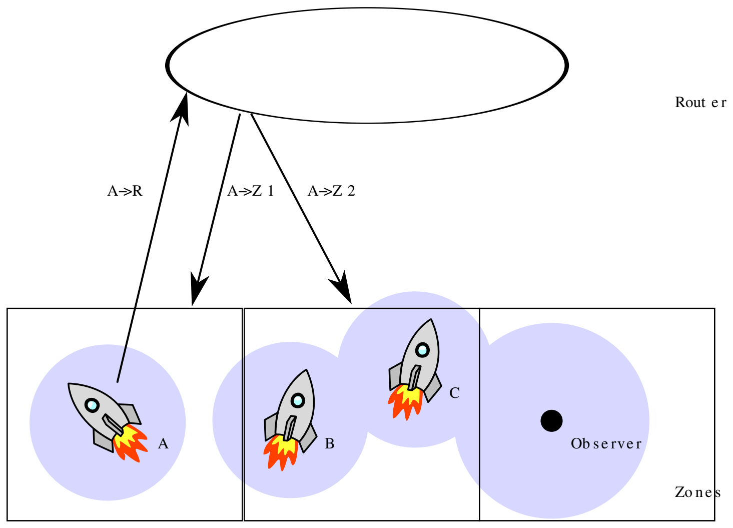

Figure 5.4: A is updated

whatever sensors it may support.

Communication is done by way of the router client. The client is responsible for supporting the various operations that the Frag node may wish to perform, such as updating entities. The node will indicate to the router client which zone the communication is meant for and the router client will contact the router and ensure that the information reaches the correct receivers. A normal entity update sequence is shown in figures 5.4 and 5.5.

In order to receive updates, Frag nodes must take inventory of all managed entities and instruct the router client to receive updates in all zones covered by entity home zones or entity interest zones. Theoretically, it is the individual entities that need to see the specific information, but as several entities are managed at a single node running a single instance of the game, this is the natural way to handle communication.

The amount of information that a node receives is thereby defined by itself, although it has no control over the frequency at which other nodes send their updates to the subscribed zones. If a node subscribes to several zones, it has the possibility of reducing its load by removing its interest from some of the zones and thus receiving fewer updates which it needs to process. This does of course require that the node takes some action regarding the entities causing the interest in those zones, such as migrating them to another node.

As each node may handle any number of entities, there is a quite significant risk of having these entities spread out over the game world. Managing many spatially distributed entities, can lead to a large number of subscribed zones, again leading to an influx of messages, potentially impeding the node's performance. In order to reduce the potential impact of such problems, Frag supports migrating entities between nodes. This mechanism is used when a node discovers that it is becoming overloaded, allowing it to shed some of its load to a node with free capacity, preferably reducing the set of subscribed zones in the process.

5.4.1 Observer nodes

As nodes can own zero entities, they can also act as passive observers which do not produce any information, only consuming that produced by other nodes. An observation node can therefore connect to the system and visualise the game world without affecting the game itself in any way. Like the ordinary computation node, the observer must decide for itself which information it is interested in, which it can then present through some output interface.

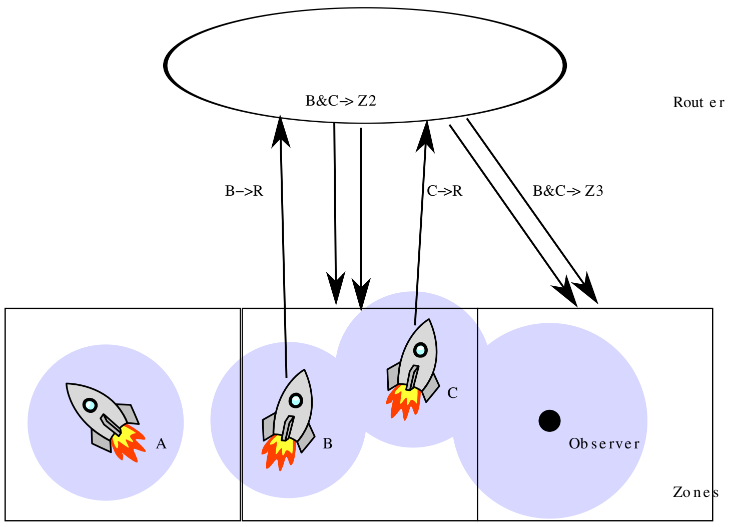

Figure 5.5: B and C are updated

Since the observer node has no local entities which it uses to determine what information it is interested in, it instead uses the confines of the area it displays to set its area of interest, acting like a camera inserted into the virtual world. The architecture's reliance on spatial context is somewhat limiting, as the observer needs to define a spatial area to watch. Showing information about a particular type of entities, for example, becomes hard without class or attribute-specific ways of requesting information.

5.5 Router

Distribution of information is handled by the *router*. Combined with the zone partitioning scheme, a publish-subscribe system is natural for distributing game state updates to interested parties. As described, each entity resides in one zone at a time and has an area of interest, which potentially covers a set of other zones. This means that as entities are updated by their owner nodes, updates can be sent to the home zone by way of the router, which then forwards the update to all nodes which have subscribed to the zone. The router thereby enables the use of zones as rendezvous points, allowing producers of information to disseminate their products to interested consumers.

Introducing a single component responsible for all communication does lead to a rather obvious single point of failure, somewhat misplaced in a system designed to be decentralised. Yet there is no strict reason for keeping the router as a single component. Its tasks should be quite possible to replace the router with a peer to peer network between the Frag nodes, enabling them to collectively establish the zone channels and deliver the necessary messages.

Chapter 6

Implementation

To demonstrate the plausibility of the described approach and measure its performance, a library has been developed which allows distribution of game state and interactions between entities on various hosts connected to the system. The library, Frag, is implemented in Python for convenience. This means that the set of potentially compatible games is limited to those written in languages with Python or C bindings[1]. Such bindings are available for a wide range of languages. Perhaps most importantly, C++[39] a very common language, especially for writing games, is able to interact with Python through several interface implementations, such as Boost.Python[19], SWIG[6] or Python's own C API[41].

Frag also assumes that the game can handle entities as objects so that references to entities can be passed around. For games written in Python or C++, this should be the norm and this limitation can therefore be counted on to not exclude many candidates.

In order to make the distribution library portable between applications, it attempts to make as few assumptions about the underlying game as possible. Indeed, a game does not necessarily need to be aware that any distribution is going on at all. The game must, however, behave in a predictable manner and must have a notion about external events and remote entities.

Thus, for a game to be compatible, it must both provide certain requirements in its entity objects and in its game logic. A system that would allow distribution to work with completely unmodified games would be nice but is unfortunately rather unrealistic.

6.1 Game behaviour

There are very few limitations placed on the game logic, allowing modified games to run with no discernible effects, besides those caused by delays or errors in the network, which is outside of the system's control.

The most important task delegated to the game by Frag is to stop the computation of remote entities. As the internals of the game is outside Frag's control, but the game is likely to track all entities through some common data structure, the game itself needs to decide when to stop computation according to entities' locality.

As an example, consider a game with a list of all active entities. The game updates its state by iterating over all these entities and calling each entity's update-method, which again performs behavioural logic. This logic may include rules such as reacting to other entities in the entity list, perhaps ordered into a more geometrically representative structure.. An entity will necessarily want to react to all entities within range, so it is clear that all nearby entities, whether local or remote must be available. However, updating remote entities' state is not necessary, since these state updates are received and applied by Frag. The game's main loop can therefore skip those entities during the entity list iteration.

[1]As Python can interact quote smoothly with C, another program accessible from C can also be accessed from Python with some circumvention.

```
    def registerEntity(self, entity): ...
      return newId()

    def removeEntity(self, global_id): ...

    def registerEvent(self, event): ...

    def registerObserver(self, entity): ...

    def removeObserver(self, id): ...
```

Table 6.1: Registrar API

6.1.1 Entity handling

For the library to function correctly, managed games must make sure to inform Frag when entities are created or removed. As long as the library is told about such changes and is passed references to the entity objects, it can perform the other necessary operations by itself. Entity registration is done through Frag's registrar interface, shown in table 6.1.

Frag is informed of the local creation and removal of entities through *registerEntity* and *removeEntity*, where *registerEntity* returns a global identifier which the entity must use. Observers are registered and removed the same way, but Frag treats them specially, only using entities registered through these methods to determine the subscription set and never generating any updates for these entities.

Additionally, the game must support callbacks from Frag in order to create and insert remote entities into the game. This should be quite easily accomplished by offering a single or several entity factories which can be accessed through a function named *createEntity*, which takes a single string as its argument and returns an entity. This string is an identifier the originating game uses to uniquely identify the entity type and which can be used by the factories to create a new entity of the same type. The new remote entity can then be updated by the usual deserialisation methods.

After the entity creation process has completed, the entity should exist in the game world, able to interact and be interacted with by other entities.

The game must also support the callback *removeEntity*, taking a global identifier and removing the corresponding remote entity from the game. This call will need to be made when remote entities are removed from the game and therefore needs to be removed at all nodes who have made copies of them.

6.1.2 Remote interaction

Interaction between local entities is quite straightforward and is handled exactly as if the game runs completely in one place, without interference from Frag, except for distribution of updated entity state. Interaction between local and remote entities, however, needs to be handled explicitly, since updates to remote entities are discarded as soon as updated information about the entity becomes available.

It would have been possible to have Frag detect modifications to remote entities and send these updates to the node owning the entity. However, this would break the responsibility model in addition to making such changes hard to verify. This approach is a break with the responsibility model because is would require a node making changes to an entity which it did not own. Remote entities is by definition imperfect copies of the original and can therefore function differently from what the non-owning node expects. This node should therefore not attempt to update the entity, even though something appears to be a reasonable operation. Changes can be hard to verify as simply sending the updated state back to the owning node does not indicate what caused the update. If the local entity would have reacted differently to the interaction than the non-owning node believed, the update might be breaking the rules and have to be discarded, even though it could have been applied correctly at the owning node if the context was known.

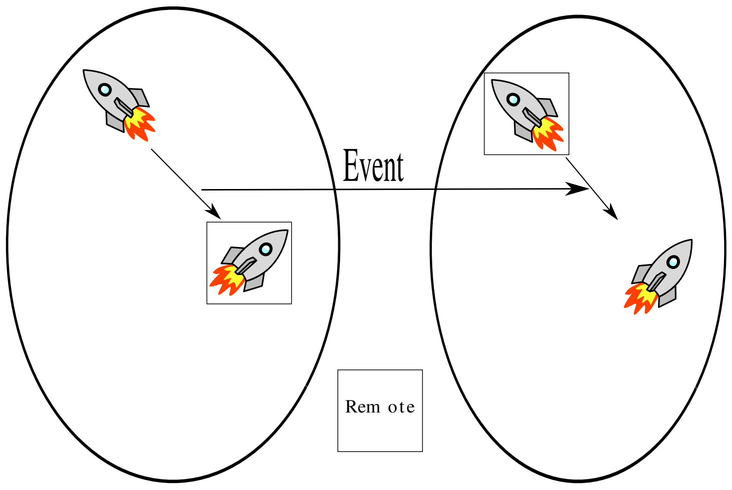

Figure 6.1: Event

Thus, to allow interaction between local and remote entities, while still keeping to the entity-specific rules, nodes must be explicitly notified of interaction attempts so they can decide if and how to change the relevant entity's status. These notifications are transferred in the form of *events*, as shown in figure 6.1, which acts as a request from a remote entity to perform an interaction with a local entity with a set of parameters.

Assuming that the underlying game performs interaction through specific entity methods, the game must at the point of entering the method check if the entity is remote. If this is the case, the update cannot be processed as usual, but an event has to be generated. An appropriate event object for the interaction is created and registered with the Frag handler (see figure 6.1). Depending on the game's source code, this could be a time-consuming operation if entities have many points of entry during an update. All these points must be identified and handled to avoid computing remote entities. Using a suitably advanced language, such as LISP, this might have been done by modifying the code dynamically, wrapping method calls in extra logic which would return the call or generate an appropriate event if the method belonged to a remote entity. Using C++ as the main "game language", however, makes such techniques hard to implement.

The game must also implement a function named *evalEvent*, which is called from Frag when an event is received. This function is responsible for calling the appropriate entity method that was aborted on the originating node and completing the interaction by updating the local entity.

Interactions are always performed between two specific entities. If some action, such as an explosion, affects several entities, this is represented as a series of events between the initiator and the affected. It is also worth noticing that the initiator should be defined as the entity most directly connected to the event. This means that if a tank entity fires a shell, which is also modelled as an entity, the event of the shell hitting a third entity is an interaction happening between the shell and the third entity. The tank entity, although the original initiator of the interaction is not directly involved, unless there is no separate projectile entity. In that case, the tank is figuratively reaching out and touching someone. This could be imagined to be the case if the projectile had an instantaneous effect, such as a laser beam at non-relativistic range.

6.2 Application Program Interface

This section gives an overview of the objects Frag uses in cooperation with the underlying game. For Frag to function properly, the game entities must provide certain attributes and must be handled correctly by the game at certain key points. Technical details on the Frag's interface to C++ games is described in appendix A.

6.2.1 Entity objects

Entity objects must supply a few properties through appropriate access methods, of which most are likely to already exist in a normal game.

First, an entity needs a field indicating whether it is local or remote. This is the only property related to the distributed aspect of the system and is thus a slight break of transparency, though a necessary one, as described in section 6.1.

Secondly, game entities must support a global identifier. The global identifier serves to identify the entity in the system as a whole and an the usual internal and instance-specific entity identifier used in most games is therefore not sufficient, since this is usually something as simple as an offset in the list of existing entities. The global identifier is randomly generated by a call to the local Frag library at entity creation. True globally unique identifiers is hard to achieve due to the decentralised architecture of the system, but a sufficiently low probability of collisions is achievable by using large enough pseudo-random values. Assuming an even distribution, the chance of a collision occurring is equal to 0.5 after a little less than $1.2\sqrt{H}$ attempts, where H is the number of possible permutations[18]. For a 128 bit identifier, this means that a collision is likely to occur after approximately $2.2 * 10^{19}$ entity instantiations. Even if the system were to generate that many entities, we still wouldn't have a problem unless both entities were alive at the same point, which is even more unlikely. We therefore assume that collisions will not happen. The random identifier was chosen in order to minimise the bandwidth usage of messages, but by using more space, it would have been possible to create an even safer method of identification by using host identifiers. If all hosts can be assigned a unique identifier, this could form the entity identifier together with a sequential number belonging to that host, being increased each time a new identifier was needed. On an addressable network, this could be as easy as using the hosts' IP-addresses and using a counter sufficiently large as to avoid the risk of it overflowing and restarting while there still existed active entities with low identifiers. As explained in section 6.3, however, nodes do not have to be globally addressable, and so would need another way of finding host identifiers, perhaps using a centralised service. Regarding the counter, its size would generally depend on the rate at which new entities were created. If 1000 local entities were created every second, a 32 bit counter would last for 1193 hours ($\frac{2^{32}}{1000*60*60}$) until it wrapped, at which point things would go wrong if any of the original entities were around. At 64 bits, the counter would have lasted a little less that 600 million years ($\frac{2^{64}}{1000*60*60*24*365}$), during which it is quite reasonable to assume that the game would have been halted at least once.

It is assumed that the game occurs in a three-dimensional space where entities have a position, indicated by Cartesian coordinates. Additionally, an entity has a speed property which may be non-zero, indicating a vector by which its position changes for each time unit that passes.

As a simple way of representing an entity's field of view, it must also be able to report its interest expression as three values, indicating a distance, each way, along each of the three coordinate axes, where the entity is interested in happenings within the described volume. This is a rather primitive approach to the area of interest concept and lacks methods for having an asymmetric area of interest. It is however worth noticing that this does not impact game modelling of fields of vision, only the amount of remote data available to the game. As such, this method ensures that all data an entity might possibly be interested in is available, by forcing the subscription zone to cover more than the entity's in-game, "real", area of interest. This also serves as a form of prefetching if the entity is able to rapidly move its asymmetric field of vision, bringing new areas into sight without changing its position significantly. Using a constant shaped area of interest, the Frag node ha already subscribed and received entities of interest within the local entity's range, and the game algorithms can thereby make use of them without any delay. Nevertheless, this does incur the cost over oversubscribing, potentially by a significant amount if an entity has a very long but narrow field of view. Using the naive approach, Frag's calculated area of interest will be a volume covering the entire length of the entity's field of view, in all directions.

Finally, the entity must support serialising and deserialising a set of basic properties, with the entity's

```
class BaseObject:
  def setLocal(self, global_id = -1, local = True):
    self.global_id = global_id
    self.local_compute = local
    self.coordinates = (0, 0, 0)
    self.interest = (0, 0, 0)
    self.speed = (0, 0, 0)

  def dumpState(self, extra_state = ''):
    return serialise(self.global_id, self.coordinates,
                     self.speed) + extra_state

  def updateState(self, message):
    coordinates, speed, rest = deserialise(self, message)
    self.coordinates = coordinates
    self.speed = speed
    return rest
```

Table 6.2: BaseObject methods

gid	category	type	pos-x	pos-y	pos-z	spd-x	spd-y	spd-z

Table 6.3: Serialised BaseObject

identity and location as an absolute minimum. Knowing the position of remote entities is enough to react to them in a sensible way, but additional information is required if the remote host is to know which model to represent the entity with or how to orient and animate it. The speed must be known if dead reckoning is to be performed, without which, the remote host will have large problems with old and incorrect state data. The entity is itself responsible for the serialisation and deserialisation, since the Frag library does not know what properties a unit has beyond those required and described above.

The Frag library provides a basic *BaseObject*, shown in table 6.2, on which the library can perform all its operations, and which acts as a template for all game entities, which must provide the same methods as BaseObject. BaseObject also supports serialisation of the required properties, meaning that entities can delegate some responsibility to the library and only have to deal with serialisation if it wants to give away additional information.

Although not strictly required, all entity classes must functionally act as subclasses of BaseObject, supporting the same calls. Because of this assertion, one can assume that a common ground exists between all entities, and that they can be treated in the same manner by the library without it having entity-specific knowledge. Furthermore, this allows representation of entities on remote hosts, even though the entity type is unknown. If no extra information is available at all, the entity can be represented by some placeholder, since at least the entity's position is known. Thus, local entities can interact with remote entities, even though they know very little about the remote entity. It is, however, important to note that this is not a preferable method of operation. Local entities may want to choose reactions from a wide range of options based on more specific knowledge about the remote entity. Visualisation of the unknown entity will also suffer, as it will be hard to identify by the placeholder representation.

6.2.2 Event objects

Like the entity objects, events are also structured as a hierarchy, providing increasingly specialised data fields for representing different events and corresponding methods for serialising and deserialising the data. The base class is *BaseEvent*, whose serialised properties is shown in table 6.4.

The category indicates the general type of event, hopefully enabling the event's use even if the receiver

category	type	source gid	target gid

Table 6.4: Serialised BaseEvent

doesn't support the full type. The categories thereby allow entities from different systems to interact, at least in part. One can, for example, imagine an event from a category called "damage" being created as a specific type defined by a certain game. This event is then delivered to an entity implemented in another game which does not have the same damage implementation, but still understands the notion of "damage". The receiving entity can then deliver a reasonable amount of damage to itself, even though it does not understand the specifics of the interaction. This can result in unexpected reactions, such as entities taking catastrophic damage from slight nudges, but then again, if the games do not have any common ground beyond the basic concepts, this does not explicitly break any rules.

To assist game logic and event validation, events also contain identifiers for both entities involved in the interaction. This can both be useful for internal bookkeeping, such as reacting accordingly to the entity that initiated the interaction and for verifying the validity of the event. If, for instance, an event requiring physical contact is received and the source entity is too far away, the event may be discarded as at apparently was issued in error and thus should have no effect.

6.3 Frag operation

The Frag library is divided into a set of modules, according to the architecture description shown in figure 5.1, where the game, Frag node and router client form the "client" application. These modules form four separate modules where the router and router client communicate over the network, while the router client, Frag node and game communicate by functions and attribute access. The location of the "client" side components are not important, as long as the router is reachable. As the Frag nodes themselves are responsible for connecting to the router, they can also be placed at addresses that are not reachable from other networks. If the router were to be replaced with a peer to peer network, this property would not necessarily hold.

6.3.1 Dead reckoning

To deliver as few updates as possible, while still maintaining good coherency between nodes that can see each others' entities, dead reckoning is used to determine the rate of updates as described in section 4.2. The dead reckoning system consists of two parts, the normal dead reckoning done for updating remote entities in-game, and the control module, performing dead reckoning on copies of local entities to determine when the dead reckoned model's error passes a certain threshold.

When a new entity is registered, a new BaseObject is created and initialised with the relevant part of the entity's properties, without being inserted into the game. The new entity is then tied to the original entity's global id in an internal dictionary belonging to Frag. The entity copies in this dictionary is used to discover discrepancies between the predicted entity and the local entity, by performing normal dead reckoning operations on the copy and comparing its position to the original, which is being updated by the game. If the delta passes a certain threshold, an entity update is issued to the home zone, and the copy is reset to the original's current position and speed, to resemble other node's new view of the entity. To account for entities that are not moving, an update will also be issued if a certain amount of time passes without the entity producing any updates. This ensures that newly joined nodes will get information about entities which have been stationary since the node subscribed to the relevant zone. It is also useful in avoiding some other issues, described in section 6.3.2.

The normal dead reckoning system works in the same way as the control module, but operates directly on the in-game remote entities. That is, Frag iterates over all remote entities and updates their position according to their movement attributes.

The prediction itself is done simply by adding the entity's speed at the time of the last update to its position to derive the new, predicted, position. The dead reckoning model is quite simple as it just performs first-degree prediction, which basically assumes that the speed vector is constant. This is fine as long as entities are moving in an environment without terrain or gravity and are not manoeuvring. However, as soon as an entity starts manoeuvring, it will usually exert some power to accelerate until it has achieved the change in direction or speed it wished for. During this period, the entity is unpredictable using the simple dead reckoning scheme. Consider a projectile travelling along a ballistic path. The projectile is continually accelerating downwards and therefore changes its vertical speed with each update, rapidly causing it to divert from the dead reckoning model. To include an acceleration vector and thereby achieve second-degree prediction could have been useful. The increased rate of updates caused by not having this precision is likely acceptable, but could still have been possible to avoid.

6.3.2 Remote entity removal

We have up to now described how Frag allows games to discover and track remote entities existing inside the area of interest belonging to one or more local entities. This is however not enough to maintain the required level of consistency. If a zone is temporarily subscribed to and remote entities discovered in it, they will exist as copies in the local game. It would be possible to delete these objects as soon as their home zone was not in the set of subscribed zones any longer, but this could prove quite inefficient if an entity is moving as to constantly add and remove the relevant zone to and from its area of interest. In that case, the copy of the remote entity would be created and destroyed continuously, quite possibly causing some overhead.

Instead, remote entities are kept in the game even when their home zone is no longer updated, to act as a cache if the zone again becomes of interest, something quite likely, given many game entities' nature of erratic movement. The remote entity will stop being updated and appear to freeze in place, but this should not matter to the game, as no entities are interested and therefore able to interact with the area in which the remote entity is frozen.

Caching remote entities in non-subscribed zones may cause some problems, however, as *delete*-messages regarding those entities may be lost while the zone is out of subscription. If a remote entity is deleted while the zone is unsubscribed, it will never be updated again, even tough the zone may become subscribed again. Local entities moving into range may therefore attempt to interact with a remote entity that does not exist and which may have been non-existent for a good period of time. This situation is clearly unacceptable and must be accounted for.

The way this is handled is to have remote entities time out after a certain period of time has passed without any updates having been received for that entity. Since the system specifies the maximum amount of time that can pass without sending update messages for an entity, not receiving any such messages for significantly longer than this period, in all likelihood means that the entity has been deleted and will results in the deletion of its copy.

Deleting remote entities after a certain period of inactivity also serves to clean previously visited zones which have not been subscribed for a while, avoiding cluttering the game with inactive entities. The technique also solves another critical issue, that of nodes dropping out of the system. Whether due to lost network connectivity or a crash, a node disappearing from the system will lead to its local entities not being updated.

6.3.3 Migration

Frag implements a mechanism for migrating entities between nodes. Since node capacities can vary, each node must examine its own load and decide when it is being overloaded. At that point, the node selects an entity to migrate. This entity will ideally be the only entity causing the node to maintain a subscription to a given zone, so that the subscription can be cancelled once the entity leaves.

When an entity has been selected, the node generates a migrate query message containing information about the entity and sends this to the entity's home zone, resulting in the dissemination of the query to all nodes having an interest in the zone.

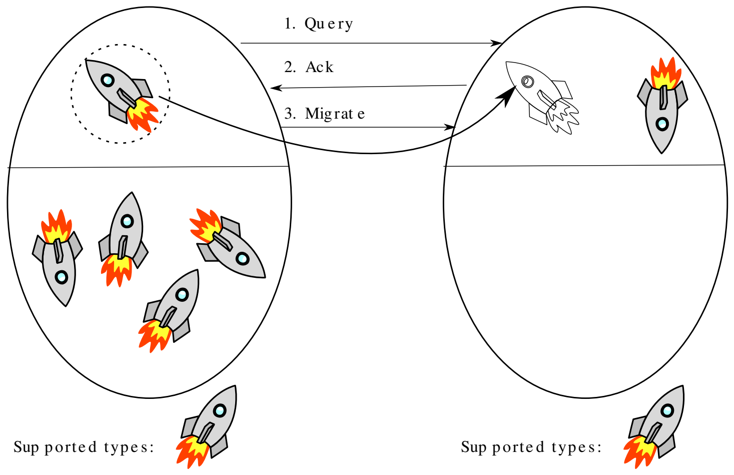

Figure 6.2: Entity migration

Upon receiving a migration query, nodes check whether they can receive the entity without unreasonably increasing their own load. If a node has several entities within the relevant zone or is managing few entities, it is likely that receiving the entity will not cause any problems and a migrate accept message is returned to the originator.

To keep the system in a consistent state, an entity can only be controlled by a single node at a time. If this is not the case, the system will contain several nodes generating updates for the same entity, quite likely gradually becoming more inconsistent. To ensure that this does not happen, the originating node will always select the first node responding to the query, issuing a migration message directed at the accepting node, before setting the entity's status to remote. Upon receiving the migrate message, the new owner updates or creates the entity and sets its status to local.

Sensing the migrate query to the home zone is suboptimal in that it risks not reaching nodes which could have accepted the entity. Nevertheless, distributing the query to all zones would not be a scalable solution if the number of zones can be large. It is also more likely to find suitable hosts among nodes already having an interest in a zone. Receiving a new entity for that zone does not increase the node's workload beyond that of computing the new entity, since it is already receiving updates for the zone. Adding an entity in a zone not already subscribed to would mean a potentially larger increase in load, as the node would have to begin processing all updates in that zone.

6.3.4 Zone subscriptions

As has been said, the amount of incoming messages is tied to the number of subscribed zones and the number of entities existing in each of them. When connected to the router, the router client sets a filter, defining the zones it wants to receive updates in. This filter is dynamically updated by the router client as the Frag node discovers changes in the managed entities' interests.

As long as a connected client has set a zone in its filter, the router considers that client as subscribed to the zone and sends all updates it receives for the zone on to the client.

Since the goal of the zone partitioning system is allowing nodes to receive a minimal amount of updates, the maintenance of zone subscriptions is a main priority for Frag. The subscription maintenance is divided into two tasks: Identifying entity home zones and entity zones of interest.

The first tasks is simply a matter of checking each entity to see if it is still within its home zone. If it is not, a new home zone will need to be located and subscribed. Finding entities' area of interest is slightly more complicated as in involves identifying a volume of the game world, possibly including several zones. The entity's area of interest is retrieved from its *getInterest* method, returning the three-dimensional interest vector which then describes the covered area when centred upon the entity. As an entity can be

type	timestamp	source id

Table 6.5: Binary protocol headers

located at different places within a zone, its coordinates must be taken into account so that it is sure to see across a zone border it is closer to than the limits its area of interest.

The entity home zone is saved in a dictionary keyed to entities' identifiers so that the home zone for any given entity can be found quickly. This is necessary, as the model requires entities to send updates to their current home zone. Zones covered by interest do by default include entities' home zones, but can also extend beyond this. These zones do however not need to be tied to any specific entity, since it is only required that the node is connected to them in order to receive updates. As all updates are applied to the local model, all local entities see remote entities in all subscribed zones, even though the zone may be out of a particular entity's area of interest.

When the interesting zones for all local entities have been identified, the zone filter is updated by the router client.

6.4 Communication

6.4.1 Router and router clients

The communication system uses the Shout rapid event delivery library[40] which resembles our proposed solution closely. Shout provides a server which can receive *events* categorised according to an integer type identifier. Shout clients can then connect to the Shout server and set filters for various event types by their identifiers. The Shout library is quite lightweight and requires little adaption of the Frag node. Shout supports events containing a binary field of arbitrary size, meaning that we can tunnel the Frag protocol transparently through Shout.

Shout's event filtering matches nicely to our demands for zone filtering, allowing us to specify each zone as a separate event type and have the server handle the filtering. The only problem with this approach is that the event type field is limited to 32 bits, leading to a rather limited number of possible zones. As zones are treated by frag as a set of three coordinates, bottom left and foremost corner of a cube, it is natural to use 10 bits of the identifier to store each coordinate, allowing very quick conversion back and forth between the normal and serialised version of a zone's coordinates.

This is obviously somewhat wasteful in that in discards two bits and thereby reduces the total number of zones by 3/4. Using all of the bits would require defining more zones to be available along two axes, spending 11 bits on each of those while only keeping 10 bits for the last.

Applying the filtering at each Frag node would also have been a possibility, but limits the system's scalability somewhat, as all updates data would be sent to all nodes. As a compensation, this approach would have yielded the possibility of more complex filtering, if so desired.

6.4.2 Protocol

Communication between applications in Frag is based on a simple binary format with messages containing a few header fields and, possibly, the serialised form of an entity or event.

The binary protocol header contains three fields as show in table 6.5, where type indicates message type, allowing it to be parsed correctly. The timestamp indicates when the message was sent and the source id identifies the source, which can be either a computation node or a coordinator. The source field is mainly used by nodes to filter out updates which originates from itself and has been redistributed by the coordinator. The timestamp can be used to drop late messages if they are of a message type with stream properties, such as entity updates.

The protocol currently supports the seven message types shown in figure 6.4.2. These messages represent the five different actions that can occur in Frag. The four first messages are related to the

- UPDATE
- EVENT
- DELETE
- MOVE
- MIGRATE_QRY
- MIGRATE_ACCEPT
- MIGRATE

general publish-subscribe system, originating from an entity's owning node and being disseminated to the entity's home zone in order to update the state on all listening nodes. The last three messages are used by the migration system, which is two-way, resulting in the need for a more complex protocol.

Shout provides both lossy and non-lossy modes of transfer, meaning that stream type messages may be dropped by shout if it becomes overloaded. In the Frag protocol, all messages except updates are non-lossy per default.

The update message indicates that a unit has updated its state. These messages are generated as soon as the unit's position changes by a significant amount and are sent at regular intervals if the unit is stationary. Entity updates can therefore be considered to be a stream, where it is safe to drop messages now and again. Dropping update messages may be prudent, as these are expected to make up the vast majority of messages sent and queueing of messages because of delivery delays will cause old information to be used long after it was produced. These messages are generated at the owning node and sent to the coordinator controlling the zone the entity resides in. The coordinator thereafter propagates the message to all connected nodes so they can update their remote entity. Since the originating node does not want to apply what by now might be an outdated update, it discards the update if it finds its own identity in the source field.

Event messages are similar to update messages, representing a serialised interaction. Events are, however, one time occurrences and should therefore not be dropped. A late event might potentially be dropped by the node at a later point, if the source and target entities are found to be in positions where they cannot interact.

Delete messages signifies that an entity has been removed from the game. To execute the deletion, no further information than the entity's global id is required and the message therefore consists of only the common message headers.

Move messages are a special case, used instead of *DELETE* message when an entity changes zones. The goal is to let all observers of the home zone know that the entity has left, enabling them to delete it immediately instead of having to wait for the timeout, the latter causing a longer period of inconsistency. Using *DELETE*, however, means that the entity, in addition to being removed from nodes only observing the home zone, is also removed from all nodes observing both the old and the new home zone. Any such nodes will first delete the entity and then respawn it immediately when the first *UPDATE* to the new zone is received. Instead, a *MOVE* message is tagged with the identifier of the new zone, resulting in receiving Frag nodes keeping the entity if the new zone is subscribed, only deleting the entity if this is not the case.

A *MIGRATE_QRY* is used when a Frag node wishes to offload an entity to another node to avoid becoming overloaded. Migration, as mentioned, is more complex than the other types of communication, since it requires a two-way protocol.

The need to communicate both ways introduces a problem: The system has no inherent way of identifying nodes, with no global view aside from the shout server, which does not support such functionality. Fortunately, we can resolve the problem by having nodes decide on a random identifier with the same rationale as for entity identifiers, described in section 6.2.1. Using Shout's feature of tagging messages with a reference, a node can tag the *MIGRATE_ACK* message, indicating its willingness to take control of the entity, with its identifier and the originator can tag the final *MIGRATE* message with the same identifier. All nodes subscribing to the migrating entity's home zone will receive the messages, but will only react to an accept if the entity identifier included in the message is listed in the node's list of currently migrating entities. Likewise, nodes will only react to a *MIGRATE* message if the message is tagged with their own identifier. Since there is always at most a single node having an entity marked as local, one would think checking the status flag was sufficient to decide whether to react to a migrate accept. However, as there is no way to control how many accepts will be created and how delayed they may be, there is a risk that the node where the entity migrated to receives an accept after setting the entity's status to local. In this case, we likely do not want to migrate the entity further. Therefore, a list of currently migrating entities is used, with entities appended when their query is sent. If an accept is received for a local entity that is not on the list, the accept is ignored.

Chapter 7

Experiments

In order to test the viability of a distributed approach, Spring[45] has been modified in order to use the Frag library for distributing the game computations.

The modifications done are along the lines described earlier, inserting calls to Frag in order to register or delete entities and events, on which more details are available in appendix A. For ease of use, a python wrapper for Spring was implemented, allowing some behaviour modifications without recompilation in addition to extending the entity classes by adding serialisation and deserialisation methods implemented in python.

To investigate the potential differences in performance between non-Frag and Frag-enabled operation, several test cases were executed and the relationship between the number of active entities and frame rate were recorded in addition to message transfer rates and corresponding network load. The frame rate is the easiest way of measuring system performance, as it provides the time taken to execute a complete single iteration of the game. The message numbers also serves to indicate how much network activity is induced and may suggest vectors of optimisation.

It is also useful to look at the frame rate of the non-Frag enabled version of the game for itself and not only as a basis for comparison with Frag. The basic game frame rate as a function of the number of entities gives a good indication about the complexity of entity simulation. We would expect that a game with very complex simulation would benefit more from distribution as the number of active entities at any time would generally be lower than that of a game with less complexity, since games are usually designed to be playable on current computers. A high-complexity game is therefore likely to put less strain on the distribution library as it has fewer entities to keep consistent.

The test cases attempt to both give a general idea of Frag's scalability by keeping entities spread out over the world, and to test performance under heavy load by creating a worst case scenario, moving all available entities into a small area and keeping them in continuous motion. The latter case creates a complex environment for entities as they will see and react to a large number of nearby entities. Due to the way Frag is structured, it is possible to maintain a very large number of entities in the game world, as long as each node contain its entities within areas were they have limited contact with remote entities. As nodes have no knowledge of the total system, the interesting metric to look at is therefore the effect of total visible entities.

7.1 Test environment

The tests are run in the Tromsø Display Wall environment. The display wall cluster consists of 28 workstations, each with the characteristics displayed in table 7.1.

Each of the workstations uses a projector to drive a $1024x768$ tile in the display wall. All tile workstations are interconnected by a switched 1 Gb/s Ethernet.

- Dual 64-bit Pentium 4 3.2 GHz CPU
- 2042 MB RAM
- 1 Gb/s Ethernet interconnect
- NVIDIA Quadro FX 3400 GPU
- Linux 2.6.9 SMP kernel

Although using a 64-bit processor, all programs and libraries are run in 32-bit compatibility mode. Problems with 64-bit versions of central libraries has lead to the use of pure 32-bit OS installations.

Table 7.1: Display wall tile characteristics

L	All entities local to the node.
R	All entities in the system.
r	Remote entities known to the node
M	The number of messages received per frame.
α	The movement threshold after which an update is produced
β	The update timeout after which an update is produced even though the entity has not crossed the movement threshold.
$\overline{m}$	Average entity movement per frame.
Z	Total number of zones containing entities.
z	Number of subscribed zones.

Table 7.2: Variables

7.2 Expected loads

As has been pointed out, the total system performance is hard to generalise about, since will be heavily influenced by the game implementation. We can, however, make some rough estimates for the amount of remote entities a node will see, which in turn will influence the amount of incoming messages and dead reckoning needed to be done. In the experiments, entities generally produce updates at the same rate, meaning that higher message rates can be linked to higher numbers of remote entities. These estimates will only be rough, however, considering the number of variables which can affect the number of messages a node sees. A node's set of subscribed zones will vary with where its local entities are positioned relative to each other, the size of their area of interest and where remote entities are located in the world. Additionally, the patterns of movement for remote entities will also affect how often they produce updates, spanning to the specified update timeout if stationary, to producing updates every frame if entities move rapidly and the dead reckon threshold is small enough to make the entity exceed it with each position change.

Let us define a set of variables as shown in table 7.2. Given a uniform distribution of entities over the world we then have the following:

$$M_R = \sum R * \frac{\overline{m}}{\alpha} + \sum (R - R_{\text{already updated}}) * \frac{1}{\beta} \qquad (7.1)$$

$$M_{\text{uniform distribution}} = M_R * \frac{Z}{z} \tag{7.2}$$
$$M_{\text{worst case}} = M_R \tag{7.3}$$
$$M_{\text{best case}} = M_R * 0 \tag{7.4}$$

The total number of messages in the system is expressed in equation 7.1. This shows us the relationship between entity movement speeds, the movement threshold and the update timeout. Note that the timeout update only occurs if the entity has not produced any updates for the entire timeout period.

Given this expression, we can make some estimates about system behaviour as a function of various entity distributions. Equation 7.2 is the general rate of messages, showing that with a uniform distribution of entities, the amount of incoming messages is directly tied to how many of the zones with entities in the current node can see. Subscribing to empty zones naturally cause no additional messages to be received. Having a completely uniform distribution of entities is rather unlikely, however, as most entities attempt to interact with each other and therefore create some pattern. In a worst case scenario, as shown in equation 7.3, all entities are gathered within the set of subscribed zones, meaning that the node sees all messages in the system. The converse is also true: If z grows to encompass Z, the same effect will be achieved, but by spreading out the local entities in order to cover all zones, or possibly by having entities with very large areas of interest.

In the best case scenario, shown in equation 7.4, z and Z are disjunct, resulting in a message rate of zero, since there are no remote entities within the area of interest. In this case, the game is functionally almost equivalent to running without Frag, as no distribution of entities is done, as seen from the node's perspective. Frag still sends updates to the home zones, and other nodes may subscribe to these, of course, so there is still a possible need for Frag, but this is not something the node needs to concern itself with.

The total amount of processing at a node is therefore as displayed in equation 7.5, consisting of game processing (7.5), Frag dead reckoning processing (7.6) and Frag message processing (7.7). Note that the dead reckon control expression also includes time spent on serialising entity state when an update is needed.

$$C_{\text{total}} = G_{\text{main game loop}}(L) + G_{\text{extra overhead}}(r) + \tag{7.5}$$
$$N_{\text{dead reckon ctrl}}(L) + N_{\text{dead reckon}}(r) + \tag{7.6}$$
$$N_{\text{message processing}}(M) \tag{7.7}$$

As we have pointed out, this shows the influence of remote entity numbers upon node frame rates. The points affected are dead reckoning, having to be evaluated for all seen remote entities and message processing, also dependent on the seen remote entities, but additionally influenced by the dead reckoning threshold and timeout.

7.3 Non-Frag enabled Spring

Spring with Frag disabled is basically running the same executable as Frag-enabled Spring, but with a run-time option which disables all Frag evaluation besides the entity insertion and deletion hooks. These use a negligible amount of time compared with the other parts of the program, and so are unlikely to affect the benchmarks. The python interface is of course not part of the original Spring and might as such have some impact compared to benchmarks with the pure C++ version. However, the time spent on calling the Spring main function, which does all game evaluation, once per frame is next to nothing.

To measure the load imposed by increasing numbers of entities, the game is started and 5 mobile entities are spawned every two seconds and given an order to move continuously in a simple pattern. The time to run each frame is logged together with the number of active entities, and the experiment is run until the number of entities reaches 1000. The relatively low number of entities was chosen because informal tests indicate that game performance in Spring deteriorates so badly for higher number of

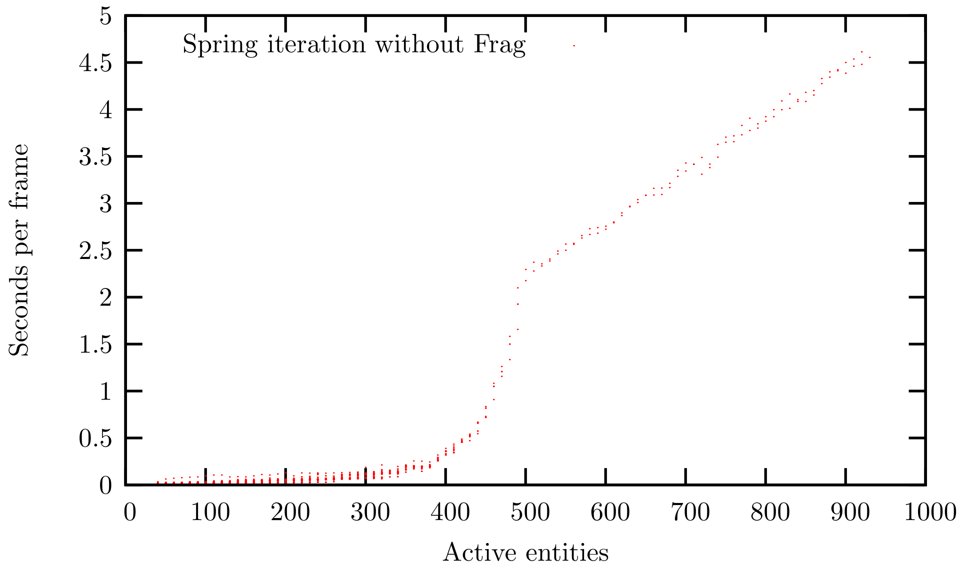

Figure 7.1: Non-Frag Spring

entities as to make the game both completely unplayable and hard to benchmark as each frame takes very long to compute.

The results are displayed as a graph in figure 7.1.

7.4 Frag-enabled Spring with congregation

To test Spring with Frag support, it is useful to model a game unfolding as close as possible to the test case without Frag support. Starting with a low number of local entities, the number of visible, remote entities is slowly brought up to around 1000. At that point, the node we are measuring at will be modelling a game with that number of entities, akin to the original version, but will only be making computations for a small number of these.

The scaling of visible units is done by first launching a single node with logging capabilities turned on. This node spawns 5 local entities every 2 seconds until it reaches 100 entities, and gives them the same orders as in the previous test case. Then, new nodes are started every 16 seconds, each spawning entities in the same pattern as the logging node. All nodes spawn their entities in approximately the same location, so that the other nodes have a good chance of discovering them. This leads to a gradual increase in the number of visible entities, until the number approaches 2800. Due to visibility limits imposed by the game world partitioning into zones, it is possible that some entities will be invisible to the logging node at any given point and thereby ignored. All measurements are therefore relative to the number of *visible* entities, as opposed to the number of *existing* entities in the system.

Results from the test run of Spring with Frag support is displayed in figure 7.3. The graph shows time spent per frame and how it was distributed over the various components. Spring's performance with Frag enabled is compared to the non-Frag run in figure 7.2.

Measurements taken at the Shout server are shown in figures 7.5 and 7.6, respectively quantifying the total number of packets in the system and the associated CPU usage by Shout (these figures also displays the results from the test described in section 7.5.1). As shout is only a transport, it has no conception about the game state. The measurements are therefore not directly linked to the number of units, but do instead show measurements as a function of time into the test run. Since the number of entities are steadily increasing during the test, once can still observe the results of an increasing number

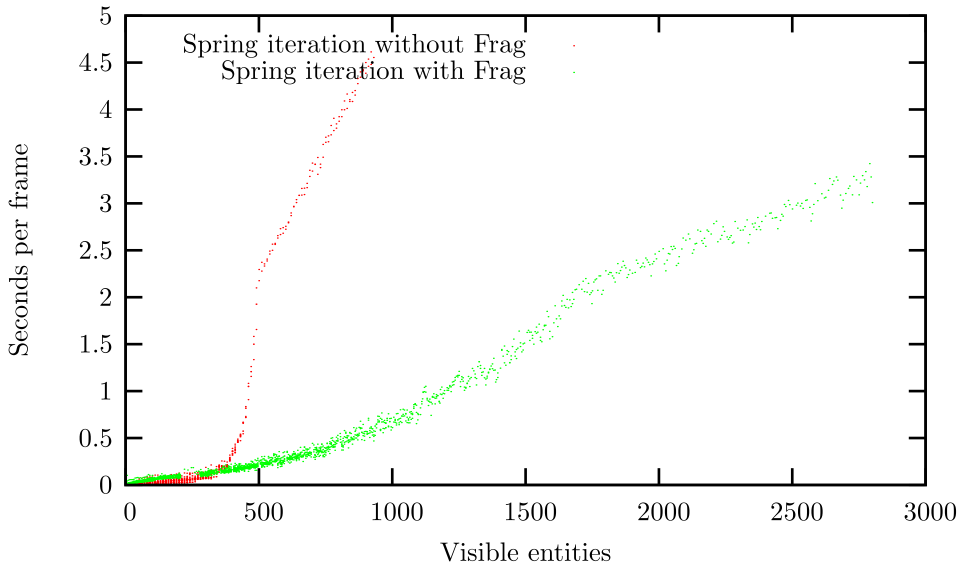

Figure 7.2: Non-frag Spring and Frag-enabled Spring

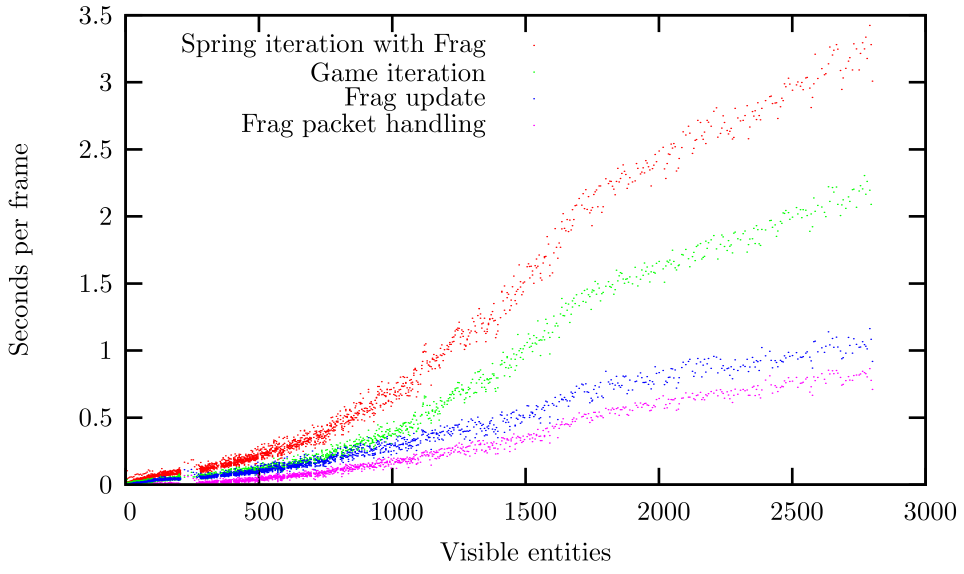

Figure 7.3: Frag-enabled Spring. Each plot, except the top plot shows the time spent in executing the designated module. The top plot is the sum of these, showing the complete time taken per frame.

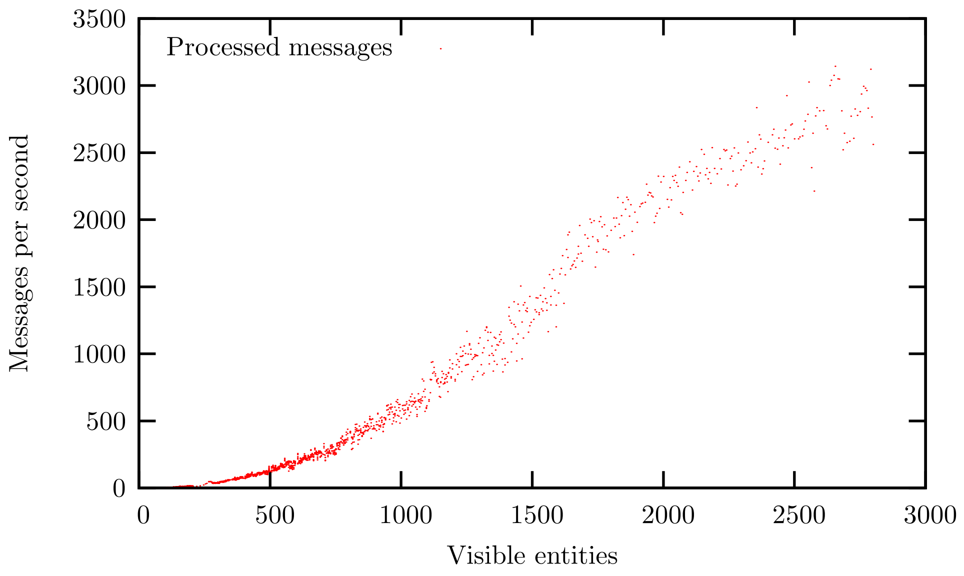

Figure 7.4: Frag message rate

of entities.

7.5 Other feature tests

Finally, two other experiments look at Frag under less demanding circumstances, highlighting some of its intended functionality.

7.5.1 Frag-enabled Spring with ideal entity placement

Having tested Frag under conditions expected the stress the system, it would also be interesting to see how it handles more benign situations and if the system as a whole can scale to greater numbers of entities in such circumstances. As shown earlier, the node processing load is to a good degree influenced by the amount of message processing. To lessen the message rate, an experiment similar to the last one is set up, but instead of spawning all entities in the same general area, they are spawned into separate parts of the game world, according to their owning nodes. The spawned entities are given move orders that keep them active, while not straying too far from the starting point. This setup should lead to all nodes only being subscribed to a few zones each as all their local entities are in more or less the same spot. These subscribed zones should also not contain too many remote entities, as they are spread out over the entire game world. In this environment, the system should be able to handle a larger number of units while keeping load at each node reasonably low.

7.5.2 Spring content generators with simple observer applications

As interoperability of applications is touted as one of Frag's features, this final test aims at showing a real use of this functionality.

Du to time constraints, the Spring 3d engine itself has not been modified to display a single viewpoint on the display wall using interest management to only receive updates about those entities displayed. As an alternative way of visualising the game on the display wall, a very simple visualisation application, using Frag, has been developed, taking on the appearance of a radar display which covers the entire wall.

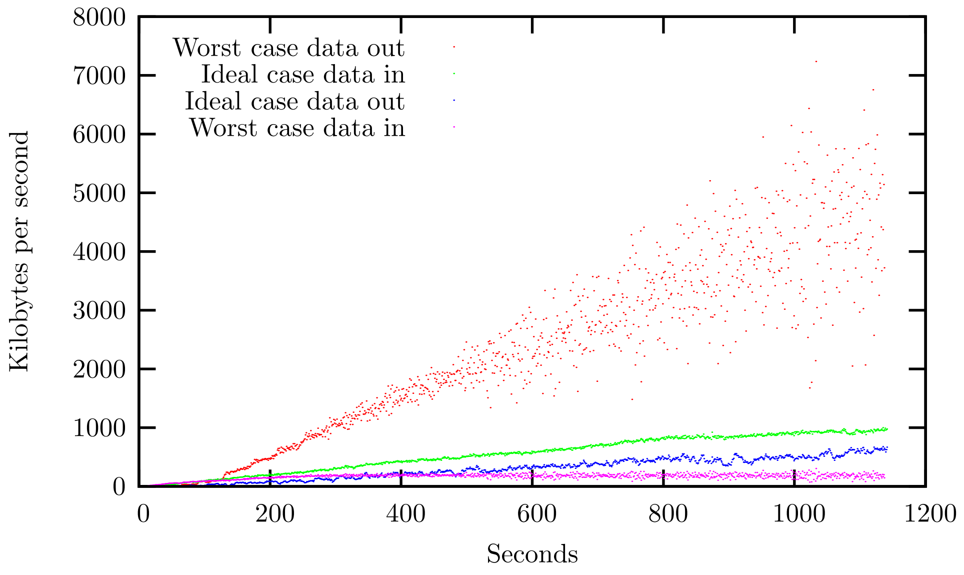

Figure 7.5: Shout packet loads

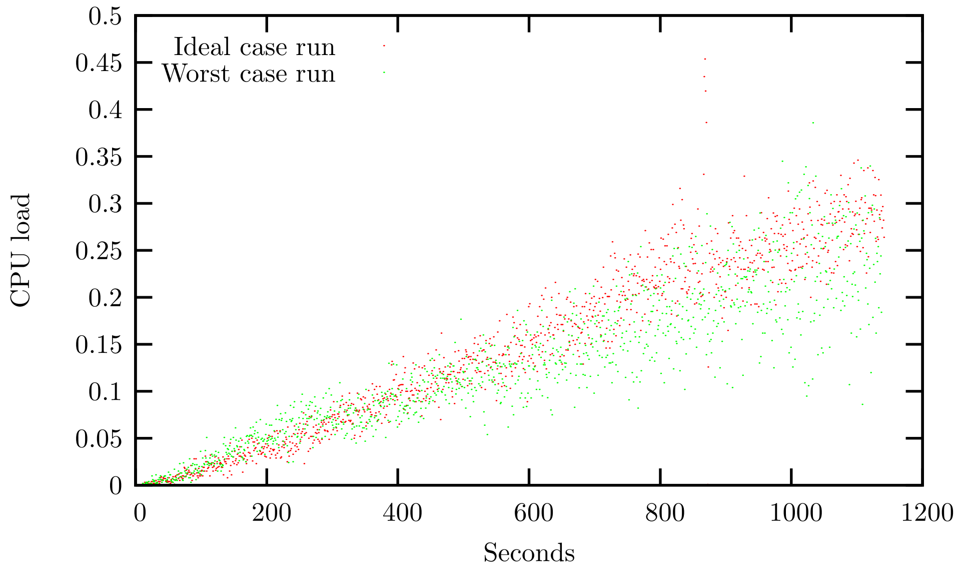

Figure 7.6: Shout CPU load

The radar program is started on all display wall nodes, being instructed at startup about their position in the tiled display in addition to the game world dimensions. Each program then places an observer at appropriate coordinates, setting its interest area to cover the space to be visualised.

The radar program does not care about entity details besides the properties of the *BaseObject* class, only requiring position data. As the program receives updates from Frag, all entities are drawn as points at screen positions appropriate for the programs placement in the display.

Chapter 8

Discussion

The tests revealed some rather surprising performance results, in addition to some not quite so surprising results. They do, however, indicate the plausibility of the chosen solution although several improvements are obviously in order.

8.1 Non-Frag Spring

First of all, the reference test itself, as shown in figure 7.1, underperformed significantly compared to what was expected. The benchmarks done on WallSpring indicated that an active entity count of around 800 was feasible, although performance rapidly degraded for higher numbers[45]. The new test results replicate the odd development in frame rates seen in the previous experiment, a rapid increase in processing time after a certain number of active entities, causing the time spent on processing each frame to become around 400% larger. After this sharp rise, frame times continue to rise, but does so at a pace which appears to be more linear.

What is disconcerting, is that the rise in the new experiment appears at 400 entities, half of what was achieved in the previous experiment. The cause of this is unknown and can have its root cause in a wide range of locations as the game code base has been changed, along with its method of invocation and experimental setup. It was speculated that some of the WallSpring results were due to some of the active entities being stationary for at least part of the experiment and it was observed that stationary entities generally required much less processing. In the new experiment, we ensure that all entities are moving continually and so may strain the game to a greater degree. The Spring implementation used with Frag is also of a newer version than that used in WallSpring, in addition to having been modified slightly in order to interface Frag. None of this can be expected to have such drastic results, but nothing can be excluded as there are no experimental results with which we can cross-check.

It is unclear what causes the rapid increase in processing times and why it occurs for the given number of entities, but it appears as if some of the algorithms used in the game have geometric complexities depending on the number of entities. As the results show, however, the geometric development is not sustained and so there does not appear to be a simple, single solution. Whatever might be the cause, we need not concern ourselves excessively with this, aside from noting that certain games may display rather serious problems with scaling. From the Frag library perspective, we have no choice but to assume that the developers of the underlying game have made their implementation efficient. It can nevertheless be interesting to study Spring in this context, as we can observe Frag enabling a partial circumvention of the initially observed scaling problems.

8.2 Frag-enabled Spring

Disregarding discrepancies between versions of Spring should not be a large problem, since we are mainly interested in studying improvements in a specific program by adding Frag support. In this regard, Frag has proven a moderate success, showing that processing times indeed can be reduced.

Comparing with the non-Frag test of Spring as shown in figure 7.2, the Frag-enabled test reveals a marked reduction in computation time per frame, perhaps most significantly avoiding the fast rise around 400 entities exhibited by the original. Although not exactly playable at two frames per second, Frag-enabled Spring still keeps total frame time below one second for the entirety of the test, at which point the original is completely unplayable.

We cannot speak of an extremely effective parallellisation, with a speedup of far less than the ideal when adding 27 nodes. That was however not expected, as the benchmarked scenario was expected to represent the worst case, with all entities congregating in a single area. As shown in 8.3, performance is indeed much more favourable in less demanding circumstances, although some network issues are encountered, as discussed in section 8.4.

Looking at figure 7.3, the different components' influence on execution time us quite clear and based on knowledge of the implementation, it appears that Frag's internal performance leaves quite some room for improvement. Primarily, we can point to the choice of programming language as a limiting factor. Python is usually not considered to be a particularly "fast" language[1], so as the amount of work which Frag needs to do increases with the number of remote entities, performance suffers notably. It is usually suggested that computation-heavy parts of python applications be implemented in C or C++, as these points are usually only a small part of the code and one thus can draw both upon Python's advantages in development speed and correctness, while exploiting the raw speed of C at critical points[19].

It is also interesting to observe the performance of the game itself, where the time per iteration appears to rise somewhat fast. Compared to the rather linear rise in time spent by Frag, Spring appears to contain algorithms with quadratic behaviour, regardless of whether all entities were updated or not. This is obviously not a good thing when we aim to lower processing load by distributing tasks. With this development in processing times, Spring will impose a limit on the number of entities that can be seen at the node, even if Frag was to spend zero time per iteration, causing the game's frame rates to become unbearably low. Nevertheless, the results show a clear improvement over unmodified Spring, yielding, as expected, far lower iteration times when only some of the entities are evaluated.

Another point of notice regarding implementation of the underlying game appears when dead reckoning timeouts are adjusted to a minimal value, causing remote entities to become deleted almost immediately when their home zone falls out of the node's interest set. As discussed in section 6.3.2, waiting a while for entities to time out works as a method of caching in case the entity's home zone becomes interesting again soon, as is likely when the local entities move back and forth. Reducing the timeout interval causes a much higher rate of "cache misses", where a zone falls out of interest, its remote entities deleted, then comes back into interest only moments later. This causes a continuous cycle of rapidly deleting and creating new entities, which results in quite heavy slowdowns in the game loop. Even worse, when the frame rate drops below a certain level, the nodes are unable to produce entity updates in time to avoid the entity timeout, or the receiving nodes takes too long to process the packet. This results in entities being deleted at the subscribing nodes, the nodes making the assumption that the entity has moved or been killed, only to have it reappear as the delayed update message is processed. This creates a cascading effect where nodes are further delayed by continually having to respawn entities, again leading to them issuing and processing updates too late, causing even more delays. In the worst case, all remote entities at a node are deleted and respawned every frame, the node being continually too late and having no chance of recovering.

It appears that instancing at least certain kinds of entities in Spring is costly. In the normal game, this is not a problem at all, since the rate at which new entities are spawned is quite low. Adding the Frag functionality, however, changed game behaviour drastically by suddenly requiring it to instance several hundred entities at once every few seconds, introducing a new choke point in the program.

The previous two points go to prove that distribution will not necessarily be a magic bullet that solves all scalability problems. In fact, patterns in game execution might be significantly altered without making any major changes to its implementation.

[1]http://shootout.alioth.debian.org/gp4/benchmark.php?test=all&lang=python&lang2=gpp

8.3 Ideal and cross-application behaviour

Running Frag-enabled Spring under more benign circumstances does, as expected, yield far better results than the worst-case scenario. The system easily scales to several thousand entities while maintaining a per-node performance like that illustrated in figure 7.3. That is, as long as nodes keeps their local entities from making contact with too many remote entities, the additional, but unknown, entities existing elsewhere has no impact on performance. This is of course at some point limited to the bandwidth and processing capacity of the router, but the Shout system has been shown to handle far greater loads than that imposed by Spring([40].

Running the radar application to visualise entities running in the Spring engine nicely demonstrates the possibilities for cross-application usage of Frag. While requiring more implementation in the radar application if more information from Spring were to be used, Frag provides the basic information necessary to display a simple view of the game world.

8.4 Shout

The measurements taken at the Shout server are also quite interesting, giving some perspective on the system's bottlenecks. Figure 7.5 gives an idea of the network bandwidth used by Frag as the test progress, both for the congregation test and the idealised situation test. As is clear from the figure, the congregation test rapidly produces large amounts of network traffic, where the idealised test is significantly less demanding. The difference is caused by Shout's point to point approach for communication, requiring the same packet to be sent to all subscribed parties. The congregation test data shows the result of an increasing number of nodes requiring information about an increasing number of entities, in what amounts to an all-to-all communication system.

The idealised test, however, shows the results of entities being spread out. There is still quite a lot of state to be disseminated, but there is a marked reduction compared to the first case.

The second chart showing CPU load (figure 7.6) sheds some light on another side of the system. Comparing the two tests, the congregation test shows Shout being generally lower loaded than the idealised test. It seems reasonable to assume that the lower load in the worst case scenario is caused by the fact that nodes rapidly come under such heavy load that they are unable to issue updates at their normal pace, resulting in reduced amounts of packets for shout to process, an assumption supported by the lower rate of incoming data in the worst case scenario shown in figure 7.5. Conversely, when nodes are under less load, we see that the stress on the Shout server increases, as the number of event-producing entities rises.

Summarised, the performance of Shout thus leads us to two observations: First, as was our assumption in chapter 4, the network bandwidth is indeed not the bottleneck in this distributed system. In a scenario introducing stress on the nodes by forcing them to receive updates from a significant number of entities, the nodes become overloaded long before the network infrastructure. The capacity of a gigabit Ethernet, although quite higher than the average end-user Internet connection, is nevertheless unlikely to become rarer in the future. Second, even though bandwidth usage in the idealised scenario is relatively low compared to the worst case scenario, due to the same data not having to be distributed to many subscribers, the CPU load of the Shout server rapidly becomes quite high. This indicates than even though we may avoid overloading individual nodes, the infrastructure used to distribute information to the correct receivers, the mechanism by which we reduce the nodes' load, is at risk. This second observation to a certain degree therefore repeats the first observation in showing that network bandwidth is not the hampering factor, the bottleneck being tied to packet processing, either at the nodes at the network's edge, or in the Shout server, the network transport's internal mechanism.

Chapter 9

Limitations and future work

Although the prototype Frag implementation presented herein is functional, it leaves quite a lot to be desired. As has been pointed out, the speed of the implementation is sub-optimal and it is quite likely that a reimplementation of the most time-critical parts in C++ would reduce computing time significantly. There are also several things that could be done for general optimisation. The dead reckoning algorithm is quite primitive, only considering the first-degree factor of current speed. The question of how complex to make the dead reckoning is of course a difficult one, where it is easy to make it overly complex. A second-degree prediction algorithm, taking entity acceleration into account would likely be useful in reducing packet rates, without costing too much in computing.

Another issue with the dead reckoning system is that the update threshold has to be set manually, making it necessary to guess about the needed accuracy. As Frag does not know anything about the underlying game's world model, except for the fact that it uses Cartesian coordinates, the meaning of "one unit" in the coordinate system might carry a different meaning in different games. To illustrate, one could say that given a defined two-dimensional view such as in the radar application, one unit equals one screen pixel (the real radar application avoids the problem by requiring the parsing of the game world size and the application's view constraints as parameters). If we introduce a viewpoint with zoom functionality, however, we see that dimensions in the game world can no longer be defined reliably in pixels, since a given view of the world always contains the same amount of pixels. In Spring, a normal entity may have a width of 10 units, although this varies. Nevertheless, this gives a somewhat fixed point relative to which we can make assumptions, like that the threshold should be reached when an entity's predicted location differs from its real position by more that half it's width. It could therefore be a possibility to have the game register some base size with Frag upon startup, which Frag then could use to calculate values such as the threshold.

The problem becomes more complex and wide-ranging if we consider that Frag ideally should support independent applications interacting, although the main requirement was for versions of the same game using different behavioural logic et cetera. If these applications use different sized units, keeping them consistent will become very hard. The ideal solution would of course to have all games use the same unit definitions, but as this is somewhat unlikely, having Frag translate units between the different systems could be a solution.

Considering the migration algorithm, it is also somewhat too simple, not taking more that one entity into account when deciding which entity to remove. A more clever algorithm would perhaps recognise limited sets of entities where migrating all would remove one or several zones from the nodes subscribed set.

Regarding the architecture itself, there are also some interesting options that have not been investigated yet. An idea that has been considered, but not explored in depth, is that of a tuple space-based system. The paradigm of inserting, retrieving and updating objects existing in a shared space seems quite close to the way Frag operates[13]. Using a tuple space could mean all entities were available in the shared space, from which nodes could copy, compute and update them. A tuple space approach also brings with it a simpler way of doing load balancing, as nodes can retrieve objects to compute as they see fit. Tuple spaces implementations have, however, traditionally had performance problems when scaled. Especially in a system such as Frag, information needs to be available immediately, as computations are done many times per second. The advantages in the abstraction the model offers is nevertheless a good reason for not dropping the idea completely.

Observing the load imposed on Shout, it is also clear that the current message distribution system has limited scalability. Shout is still under development, and may be optimised, but it would also be a good idea to look at other mechanisms for distributing information to interested parties. Multicast is an obvious network technology, which in theory offers what the system needs, with different multicast groups assigned to the various zones. The multicast approach was discarded in favour of a wish to test the Shout system, but may prove to be a good alternative. The most obvious drawback with multicast, is the somewhat low proliferation of support for the protocol, meaning that a system based on multicast will be restricted to intra-lab or Mbone-connected systems.

Another point of contention is that Frag's fault tolerance is limited to smoothly handling crashes in Frag nodes, although the entire system will grind to a halt if the Shout server becomes unavailable. In the case of a Frag node going down, the other nodes will simply stop receiving updates from its entities, eventually timing them out. Although it is hard to tell whether an entity has been consciously deleted or if its controlling node has crashed, it is nevertheless not ideal if entities disappear from the system without any good reason. Given the presence of a migration system, it would be nice if the entities belonging to the crashed node could have been moved to other nodes when it became clear that the original node was down. However, this would require having some component outside the node which would keep track of entities managed by the node and that could detect if a node becomes unavailable, migrating these entities. Given the impossibility of knowing whether a remote computer has crashed or is just temporarily unavailable, this would also require additional safeguards to ensure that the supposedly crashed node cannot come back and start emitting updates for an entity which by then is controlled by another node.

Continuing on fault tolerance, there is also, although with only a slight chance of occurring, an issue with the migration protocol. If a node crashes after emitting an accept, but is still selected for migration by the originator, there is currently no way for the originator to discover that the migration did not work correctly, effectively removing the migrated entity from the game. This is of course not ideal, but is an effect of the rendezvous model of communication, which does not lend itself too easily to such operations. Solving the problem would likely entail an additional acknowledgement from the node receiving the migrated entity and a timeout at the originator, after which it would start the migration process again. In this case, there is still a risk of the original recipient coming back from a temporary unavailability and taking control of the entity, after which there can be two nodes controlling the same entity. This would again require a scheme for conflict resolution, where several nodes controlling the same entity for example could defer to the node with the lowest identifier.

A final note that has been mentioned in the introduction, but not incorporated into the design of Frag, is that of world persistency. There is currently no way, short of defining parts of the world to be entities, to keep things like terrain deformations consistent between nodes. Such world modifications are generally caused by events that interact with the world itself, such as a projectile hitting the ground. Letting all nodes which see the impact know about the change is quite easily done by issuing a normal event (although Frag does not support events without an entity target at the moment). Other nodes who do not see the impact area, however, will not know about the modified terrain even if they become interested in that area. Making this happen is likely to require some notion of persistency in the system. The game instances that have seen the change, do of course contain the correct information, but not in a format recognisable to Frag. As mentioned, modelling parts of the world as entities would work around the problems by keeping the changes in a format that Frag could understand. Not taking this route would mean extending Frag with a notion about game state other that just entities and having a way of distributing such changes to nodes after the fact, as they need information about an affected area.

Chapter 10

Conclusion

This thesis has described a novel approach to parallellising the complex problem of simulating, in real time, a computer game world populated by mobile entities, yielding better scalability.

The real time constraint, implicit in many computer game genres, is a problem when distributing computation tasks and gathering results. Processes cannot afford high latencies when accessing entities, but must still be relieved of a significant amount of their computation tasks if they are to achieve good scalability.

This thesis introduces the Frag library in order to solve this problem. A game, which does not even have to be aware of any distribution going on, is modified slightly, only needing to follow a certain contract in how it deals with entity objects. Calls to Frag are inserted at key points, allowing Frag to keep track of entities managed at the local host while communicating with other hosts using Frag, receiving information about their entities and applying this information to the local game.

Frag eschews the approach of assigning discrete areas of the game world to designated hosts while moving simulation of entities between them based on the entities' location. In stead, the Frag approach has hosts control a semi-fixed set of entities that are free to roam the entire game world. This supports the freedom of heterogeneous participants, who can use individual rules to model "their" entities, rules that can differ between hosts, something not possible if entities are to move seamlessly from participant to participant as they traverse the game world.

Frag is founded upon two assumptions. Firstly, entities does not need to know about what happens beyond a certain range from them, as there is some limit to the distance at which they can interact with or observe other entities or happenings. Entities, and thereby their controlling hosts, not needing to have knowledge of the entire game world, is used by Frag in order to reduce the amount of updates each hosts receives to those happening in a restricted area around each entity. Secondly, a game does not need to be an exact and repeatable simulation as long as it *appears* consistent, something which allows us to tolerate brief inconsistencies if they are kept from diverging game states at different hosts noticeably. This allows Frag to use only periodical updates with "real" information, using predictions for entity movements for the rest of the time.

The system is tested and measured using a real game, where the game has no notion of the distribution that Frag manages. We show significantly lowered computation times per frame and thereby clearly increased scalability in numbers of entities in the same game. It is also shown that network bandwidth is not the main limiting factor when distributing game state, the bottleneck primarily lying in the software at the hosts themselves and the speed at which they can apply and create state update messages. Connected to this, there is also a bottleneck in the network logic managing update filtering.

The need for such filtering is showed by testing a worst case scenario in which the game instances are overloaded and grind to a halt as they receive too much state information and must spend most of their time processing it.

Bibliography

[1] Gideon Amir. Massively multiplayer game development 2: Architecture and techniques for an mmorts. www.gamasutra.com, 2005.

[2] G. J. Armitage. An experimental estimation of latency sensitivity in multiplayer quake 3. In *11th IEEE International Conference on Networks*, 2003.

[3] Rajesh Krishna Balan, Archan Misra, Maria Ebling, and Paul Castro. Matrix: Adaptive middleware for distributed multiplayer games. Technical report, IBM, October 2005.

[4] L.A Barroso, J Dean, and U. Holzle. Web search for a planet: The google cluster architecture. *Micro, IEEE*, 23(2):22–28, March-April 2003.

[5] Mostafa A. Bassiouni, Ming-Hsing Chiu, Margaret Loper, Michael Garnsey, and Jim Williams. Performance and reliability analysis of relevance filtering for scalable distributed interactive simulation. *ACM Trans. Model. Comput. Simul.*, 7(3):293–331, 1997.

[6] D. Beazley. Swig: an easy to use tool for integrating scripting languagqs with c and c++. In *Proceedings of the Fourth USENIN Tcl/Tk Workshop*, pages 129–139, 1996.

[7] Paul Bettner and Mark Terrano. 1500 archers on a 28.8: Network programming in age of empires and beyond. In *The 2001 Game Developer Conference Proceedings*, 2001.

[8] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 353–266, New York, NY, USA, 2004. ACM Press.

[9] John Markus Bjørndalen, Otto Anshus, Tore Larsen, and Brian Vinter. Paths - integrating the principles of method-combination and remote procedure calls for run-time configuration and tuning of high-performance distributed applications. NIK, 2001. Norsk Informatikk Konferanse, Tromsø, Norway.

[10] Wentong Cai, Francis B. S. Lee, and L. Chen. An auto-adaptive dead reckoning algorithm for distributed interactive simulation. In *PADS '99: Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 82–89, Washington, DC, USA, 1999. IEEE Computer Society.

[11] James O. Calvin, Carol J. Chiang, and Daniel J. Van Hook. Data subscription. *12th Workshop on Standards for the Interoperability of Distributed Simulations*, pages 807–813, 1995.

[12] W. J. Camp, S. J. Plimpton, B. A. Hendrickson, and R. W. Leland. Massively parallel methods for engineering and science problems. *Commun. ACM*, 37(4):40–41, 1994.

[13] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[14] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 20(8):1489–1499, 2002.

[15] Han Chen, Yuqun Chen, Adam Finkelstein, Thomas Funkhouser, Kai Li, Zhiyan Liu, Rudrajit Samanta, and Grant Wallace. Data distribution strategies for high-resolution displays. *Computers & Graphics*, 25(5):811–818, 2001.

[16] Jin Chen, Baohua Wu, Margaret Delap, Björn Knutsson, Honghui Lu, and Cristiana Amza. Locality aware dynamic load management for massively multiplayer games. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 289–300, New York, NY, USA, 2005. ACM Press.

[17] DIS Steering Committee. The dis vision, a map to the future of distributed simulation. Technical report, Institute for Simulation and Training, Orlando, FL., 1994.

[18] W. Price D. Davies. *Security For Computer Networks*. Wiley, 1984.

[19] Ralf W. Grosse-Kunstleve David Abrahams. Building hybrid systems with boost.python. Boost Consulting, 2003.

[20] Chris GauthierDickey, Daniel Zappala, Virginia Lo, and James Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *NOSSDAV '04: Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 134–139, New York, NY, USA, 2004. ACM Press.

[21] L. Gautier and C. Diot. Design and evaluation of mimaze, a multi-player game on the internet. In *ICMCS '98: Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 233–233, Washington, DC, USA, 1998. IEEE Computer Society.

[22] Rui Gil, José Pedro Tavares, and Licinio Roque. Architecting scalability for massively multiplayer online gaming experiences. In *proceedings of DiGRA 2005 Conference: Changing Views - Worlds in Play*. Digital Games Research Association, 2005.

[23] Tristan Henderson. Latency and user behaviour on a multiplayer game server. In *NGC '01: Proceedings of the Third International COST264 Workshop on Networked Group Communication*, pages 1–13, London, UK, 2001. Springer-Verlag.

[24] Johan Huizinga. *Homo Ludens*. 1938.

[25] Richard Weatherly James O. Calvin. An introduction to the high level architecture (hla) runtime infrastructure (rti). In *Fourteenth Workshop on Standards for the Interoperability of Distributed Simulations*, March 1996.

[26] John E. Laird and Michael van Lent. Human-level ai's killer application: Interactive computer games.

[27] P. Lincroft. The internet sucks: Or, what i learned coding x-wing vs. tie fighter. In *Proceedings of Game Developers Conference*, volume 1999, 1999.

[28] Carsten Magerkurth, Maral Memisoglu, Timo Engelke, and Norbert Streitz. Towards the next generation of tabletop gaming experiences. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 73–80, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.

[29] John markus Bjørndalen. *Improving the Speedup of Parallel and Distributed Applications on Clusters and Multi-Clusters*. PhD thesis, Department of Computer Science Faculty of Science University of Tromsø, 2003.

[30] Katherine L. Morse, Lubomir Bic, and Michael Dillencourt. Interest management in large-scale virtual environments. *Presence: Teleoperators and Virtual Environments*, 9(1):52–68, 2000.

[31] Paul Okanda and Gordon Blair. Openping: a reflective middleware for the construction of adaptive networked game applications. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 111–115, New York, NY, USA, 2004. ACM Press.

[32] Claudio E. Palazzi, Stefano Ferretti, Stefano Cacciaguerra, and Marco Roccetti. A rio-like technique for interactivity loss-avoidance in fast-paced multiplayer online games. *Comput. Entertain.*, 3(2):3–3, 2005.

[33] Lothar Pantel and Lars C. Wolf. On the impact of delay on real-time multiplayer games. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 23–29, New York, NY, USA, 2002. ACM Press.

[34] S. J. Powers, M. R. Hinds, and J. Morphett. Distributed entertainment environment. *BT Technology Journal*, 15(4):173–180, 1997.

[35] C.E Shannon and W. Weaver. *Mathematical Theory of Communication*. University of Illinois Press, 1963.

[36] J. Smed, T. Kaukoranta, and H. Hakonen. A review on networking and multiplayer computer games. Turku Centre for Computer Science, 2002.

[37] Jouni Smed and Harri Hakonen. Towards a definition of a computer game, 2003.

[38] Oliver G. Staadt, Justin Walker, Christof Nuber, and Bernd Hamann. A survey and performance analysis of software platforms for interactive cluster-based multi-screen rendering. In *EGVE '03: Proceedings of the workshop on Virtual environments 2003*, pages 261–270, New York, NY, USA, 2003. ACM Press.

[39] B. Stroustrup. *The C++ programming language*. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 1986.

[40] Daniel Stødle. Shout: A distributed and extensible system for event propagation in pervasive and ubiquitous environments. 2006.

[41] Guido van Rossum. Extending and embedding the python interpreter. Python Software Foundation, 2004.

[42] 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 *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[43] Grant Wallace, Otto J. Anshus, Peng Bi, Han Chen, Yuqun Chen, Douglas Clark, Perry Cook, Adam Finkelstein, Thomas Funkhouser, Anoop Gupta, Matthew Hibbs, Kai Li, Zhiyan Liu, Rudrajit Samanta, Rahul Sukthankar, and Olga Troyanskaya. Tools and applications for large-scale display walls. *IEEE Comput. Graph. Appl.*, 25(4):24–33, 2005.

[44] Shinya Yamamoto, Yoshihiro Murata, Keiichi Yasumoto, and Minoru Ito. A distributed event delivery method with load balancing for mmorpg. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–8, New York, NY, USA, 2005. ACM Press.

[45] Åsmund Grammeltvedt. Wallspring - collaborative tactical gaming on a display wall. Term project, December 2005.

Appendix A

Python-C++ interface in Spring/Frag

Python is supplied with a comprehensive C interface, allowing both importing specially designed shared libraries written in C to be imported into the Python interpreter, and extending a C program by embedding Python functions within it[1]. The second approach, python extension, is usually selected, as it allows most of the application to be written in pure Python, avoiding recompilation and writing C code in general.

Python extension is done by writing shared libraries in C, including certain macros defined by the Python header files which designate the shared library as a Python module, defining the available functions, classes and variables. After compiling the library, the Python interpreter may import it as a module, accessing the designated features.

This allows us to implement computationally complex parts of a program in C, potentially giving a performance boost, while keeping most of the implementation in Python, allowing higher implementation speed, lower chance of bugs and easier debugging. An example extension module is shown in table A.2, wrapping the simple function shown in table A.1, which returning a string decided by the function argument.

As is quite obvious, the wrapping procedure is rather verbose, forcing the programmer to handle things like argument extraction and reference counting manually. Fortunately, there exists several more or less automatic wrapper generators which can alleviate some of these problems. For this project we are using Boost.Python, which is part of the Boost libraries[2] and provides a drastically improved level

[1]http://docs.python.org/ext/ext.html
[2]http://www.boost.org/

```
char* greet(unsigned int x)
{
  static char* msgs[] = { "hello", "Boost.Python", "world!" };

  if (x > 2)
  {
    return "";
  }

  return msgs[x];
}
```

Table A.1: C function

```
#include <Python.h>

extern "C"
{
  // Wrapper to handle argument/result conversion and checking
  PyObject* greet_wrap(PyObject* args, PyObject * keywords)
  {
    int x;
    if (PyArg_ParseTuple(args, "i", &x))     // extract/check arguments
    {
      char const* result = greet(x);        // invoke wrapped function
      return PyString_FromString(result); // convert result to Python
    }
    return 0;                                // error occurred
  }

  // Table of wrapped functions to be exposed by the module
  static PyMethodDef methods[] = {
    { "greet", greet_wrap, METH_VARARGS,
      "return one of 3 parts of a greeting" },
    { NULL, NULL, 0, NULL } // sentinel
  };

  // module initialization function
  PyMODINIT_FUNC init_hello()
  {
    (void) Py_InitModule("hello", methods); // add the methods to the module
  }
}
```

Table A.2: C++ function exposed as a Python module through Python API (example from Boost Consulting)

```
#include <boost/python.hpp>
using namespace boost::python;

BOOST_PYTHON_MODULE(hello)
{
  def("greet", greet, "return one of 3 parts of a greeting");
}
```

Table A.3: C++ function exposed as a Python module through Boost.Python (example from Boost Consulting)

of abstraction in dealing with Python's C interface.

A.1 Boost.Python

Relying heavily on preprocessor macros and C++ template metaprogramming, Boost provides a highly abstracted way of dealing with Python[3], automatically finding type requirements and handling reference counting. Boost also provides implementations of Python's *object* class and common subclasses, such as *list*, *dict* and *tuple*, providing access to their public methods directly and all other attributes indirectly through the *attr* method, which takes a string as parameter and returns the named attribute as an *object*.

Wrapping the function from table A.1 in Boost.Python is displayed in listing A.3.

As can be seen, this is a vast improvement over manual wrapping, the only drawback being somewhat increased compile times associated with Boost's metaprogramming facilities.

Where Boost.Python truly shines is in interaction with C++'s object orientation, enabling wrapping of classes, enabling transparent access to these and their instantiated objects, allowing access to features like C++'s run time type information from Python.

A.1.1 Wrapping Spring objects

Spring is created in the object oriented fashion which is a quite common way to model games, with basic classes representing physical objects and specialisations for these into projectiles, vehicles and so on, according to their properties and usage within the game. A partial view of Spring's class hierarchy is shown in figure A.1.

Given the C++ class definitions, we can then register the classes with python, exposing the attributes and methods which need to be visible (we could easily expose everything, but in the spirit of briefness, we will keep to the bare necessities. The result is shown in table A.4.

This efficiently allows Python access to the Spring classes, both for manipulation and creation. When a class constructor requires parameters, these are specified with the *init* function. The *property* function creates a Python *property*, which was introduced in Python 2.2[4]. These appear as normal attributes, but in reality call *get* and *set* methods upon attribute access, creating a transparent access layer which for example can allow different external and internal representations of an attribute. In the Spring wrapping, this is for instance used to make coordinates and vectors appear as tuples to python, while using a custom object for internal storage.

Once a class has been registered with Boost, any subclass of that class can be passed to Python, although it will appear as an object of the superclass, breaking RTTI. If the class is registered with the *bases* template argument, however, methods such as *killObject*, which is implemented for all classes and handles object destruction and cleanup, will resolve to the correct class implementation, even though it is only defined for *WorldObject*.

[3]http://www.boost.org/libs/python/doc/
[4]http://www.python.org/download/releases/2.2/descrintro/#property

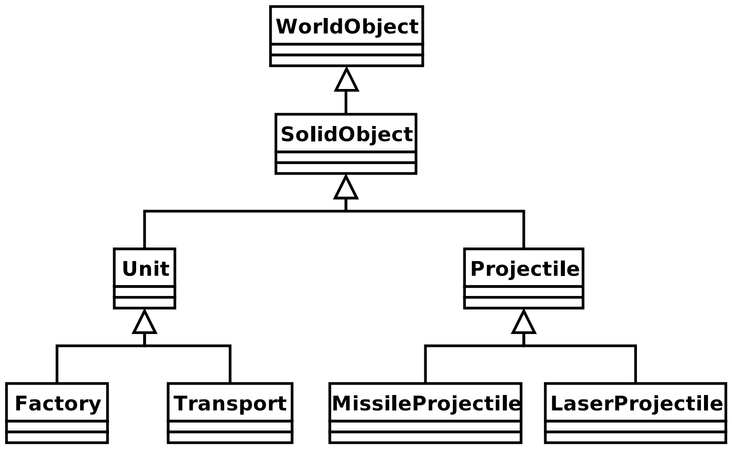

Figure A.1: Game class hierarchy

```
#include <boost/python.hpp>
using namespace boost::python;

BOOST_PYTHON_MODULE(spring)
{
  class_<CWorldObject>("WorldObject")
    .def("initObject", &CWorldObject::initObject)
        .def("killObject", &CWorldObject::killObject)
    .def_readwrite("global_id", &CWorldObject::gid)
    .def_readwrite("local_compute", &CWorldObject::local_compute)
    .add_property("coordinates", &CWorldObject::getPos, &CWorldObject::setPos)
  ;

  class_<CSolidObject, bases<CWorldObject> >("SolidObject");

  class_<CUnit, bases<CSolidObject> >("Unit")
    .def("doDamage", &CUnit::DoDamage)
  ;

  class_<CTransport, bases<CUnit> >("Transport");

  class_<CFactory, bases<CBuilding> >("Factory")
    .def_readwrite("opening", &CFactory::opening)
  ;

  class_<CProjectile, bases<CWorldObject> >("Projectile",
                                            init<float3, float3, CUnitPtr>())
    .def_readwrite("owner", &CProjectile::owner)
  ;

  class_<CMissileProjectile, bases<CProjectile> >("MissileProjectile");
  class_<CLaserProjectile, bases<CProjectile> >("LaserProjectile");
}
```

Table A.4: Boost.Python-wrapped Spring

```
from spring import Unit

def setInterest(self, span): pass

def getInterest(self): return (200, 200, 200)

Unit.interest = property(getInterest, setInterest, None,
                         'The units area of interest')
```

Table A.5: Spring class extended in Python

```
using namespace boost::python;

SpringInterface::SpringInterface(object frag) :
  frag(frag)
{}

frag::IdType SpringInterface::registerEntity(CWorldObjectPtr entity)
{
  return extract<frag::IdType>(frag.attr("registerEntity")(entity));
}

void SpringInterface::removeEntity(CWorldObjectPtr entity)
{
  frag.attr("removeEntity")(entity);
}

void SpringInterface::registerEvent(EventPtr event)
{
  frag.attr("registerEvent")(event);
}
```

Table A.6: Callbacks to Python

Having been registered, classes can also be extended in Python, although these changes will naturally not be visible in the C++ code, as they can happen at run-time. In Python, this is simply a matter of adding methods or variables to the class, which results in the new methods being able to all instantiated objects of that class. An example of this usage is shown in table A.5

A.1.2 Spring frontend

The final part of the wrapping process, is to allow access to the game itself from Python. As Spring has a sequential design, with a main loop performing all updates to the game state, this is simply a matter of exposing this function in the shared library through an interface object, allowing it to be called from Python. The constructor of this object simply takes the frag instance as its parameter, later allowing it to call the Frag registry functions as shown in table A.6.

A.2 Python access

Compiling the above mentioned source results in a Python module which gives access to the essentials of Spring, including access to entity objects and selected attributes belonging to them. Integrating with Frag is then simply a matter of writing a small Python program which initialises Frag and Spring, passing the Frag instance object to Spring, completing the connection. The program then loops Spring's and Frag's main loops alternatingly.

The Python frontend may also, by enabling the proper methods in the Spring shared library, interact with Spring in various ways, modifying game behaviour without requiring recompilation of the C++ source. This is for example used to spawn entities at intervals when benchmarking the system.

Appendix B

CD-ROM

The accompanying CD-ROM contains this document in PDF format in addition to LaTeX and BibTeX sources.

The CD-ROM also contains all code created and modified in relation to this thesis, including Frag-enabled Spring, the radar application, Shout and the Frag library. The root contains a README file giving a more detailed view of the contents.