SCALABLE COLLISION DETECTION FOR DISTRIBUTED VIRTUAL ENVIRONMENTS

A THESIS
SUBMITTED TO THE SCHOOL OF COMPUTING SCIENCE
OF THE UNIVERSITY OF NEWCASTLE-UPON-TYNE
IN PARTIAL FULLFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Kier Storey
April 2007

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Doctor of Philosophy.

Dr. Graham Morgan (Supervisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Doctor of Philosophy.

Prof. Rynson Lau

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Doctor of Philosophy.

Dr. Paul Ezhilchelvan

Approved for the School of Computing Science:

Prof. P. A. Lee
Head of the School of Computing Science

ABSTRACT

SCALABLE COLLISION DETECTION FOR DISTRIBUTED VIRTUAL ENVIRONMENTS

Kier Storey
Ph.D. in Computing Science
Supervisor: Dr. Graham Morgan

December 2006

Distributed Virtual Environments (DVEs) provide a mechanism whereby dispersed users can interact with one-another within a shared virtual world. DVEs commonly allow users to interact with one-another in ways analogous to the real-world, e.g. mimicking Newtonian physics. A scalable DVE should enable large numbers of users to participate simultaneously, regardless of the geographical location and hardware configurations of individual users. In addition, these users should perceive a mutually-consistent virtual world in which each user perceives a consistent series of events in real-time.

Collision detection and response is a fundamental requirement of most virtual environments and simulations. It is a computationally-expensive operation which must be performed at frequent intervals in all virtual environments which simulate the motion of solid objects. Collision detection has received large amounts of research interest and as a result a number of efficient collision detection algorithms have been proposed. However, these collision detection approaches are designed to detect collisions efficiently in simulations run on a single machine and are not capable of overcoming problems associated with scalability and consistency, which are of paramount importance in DVEs.

This thesis presents a new collision detection approach, termed distributed collision detection, which provides high-levels of scalability, consistency and responsiveness. This thesis presents the algorithms and theory which underpin the distributed collision detection approach and provides experimental results demonstrating its scalability and responsiveness.

Acknowledgements

I would like to thank my wife, Fengyun Lu, for her patience, support and advice while writing this thesis.

I would also like to thank my mother, Diana Storey, for supporting me throughout my education.

I would finally like to thank my supervisor, Dr. Graham Morgan, for his assistance throughout my Ph.D studies and his advice while writing this thesis.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

A Virtual Environment (VE) [Burdeau03][Singhal99] is a world simulated in computer software. In general, these systems are designed to allow a user to interact with the virtual environment and the entities which inhabit it through a set of world-specific rules. These rules often mimic the interaction rules observed in the real world, such as Newtonian physics[Palmer][Bourg] and verbal communication, although they are limited only by the imagination of the VE designer and the processing resources an memory available in the target platform. Virtual Environments are a heavily-researched and exceptionally commercially-successful family of software systems. Due to the commercial success of VEs in the computer games market, research is ongoing to improve both algorithmic and hardware performance of VEs on a global scale in both industry and academia.

A Distributed Virtual Environment (DVE)[Singhal99][Morgan05][Zhao01] is a VE which allows multiple dispersed participants to interact with the same VE in real-time. Each user can inject one or more objects into the DVE, which they

can control; in many DVEs, a user injects just a single object, termed an avatar. An avatar is a virtual representation of the user, which they can use to interact with the DVE, the objects and avatars which reside within it. Initial DVE research was conducted with the goal of developing distributed military simulation and training tools. As the cost of hardware capable of rendering three-dimensional images in real-time was prohibitively expensive, alternative applications for DVE technology were not extensively explored until consumer hardware became capable of 3D rendering. DVEs currently find applications in training and simulation, virtual classrooms, entertainment and e-commerce.

A DVE is required to ensure that each user experiences the same, or a very similar, environment to one another and that the actions they perform in the DVE have the same semantics and result in the same reactions being observed by all users. In a DVE, satisfying real-time requirements while ensuring all users maintain a consistent view of the shared state is difficult. Although the actual behaviour of users is non-deterministic, they can only interact with the environment in set ways meaning that the behaviour of players can be predicted to a certain level of accuracy based on heuristics [McCoy03]. However, due to network delays and limited bandwidth, DVEs adopting such techniques can become inconsistent and chaotic, with many participants viewing the world in a number of different ways. In addition to consistency requirements, it is also desirable to develop scalable DVEs. A scalable DVE should allow large numbers of dispersed participants to interact with one-another simultaneously. The machines which the dispersed users participating in the DVE use may exhibit large variations in performance and variations in network transmission delays and available bandwidth. As a result, designers are responsible for ensuring that their DVE enables large numbers of users to interact with a consistent virtual world while dealing with heterogeneity in real-time.

The current state of the objects inhabiting a DVE is shared between machines participating in the DVE through the use of message passing over a network. The scalability of a DVE can be affected by a number of factors, including:

- The volume of messages required to be transmitted
- The processing overhead associated with managing the DVE

As the number of participants increases, the volume of messages transmitted between participants may increase to reflect the additional objects inhabiting the virtual world. The choice of communication architecture and network protocols can significantly affect the scalability of the DVE. The communication architecture defines which machines are responsible for transmitting state update messages to each other throughout the lifetime of the DVE. In addition, the choice of network protocols can affect the volume of messages each machine is responsible for transmitting. These issues are discussed in detail in the subsequent chapters.

As the number of participants increases, the number of objects inhabiting the DVE will also increase. With an increased number of objects in the DVE, the processing overhead associated with managing these objects will also increase. These processing overheads include: Rendering; Animation; Collision Detection; Physics; AI. This list is by no means exhaustive but offers many of the larger processing overheads associated with most DVEs.

This thesis concentrates on collision detection [Ericcson05][Bergen04] for DVEs. Collision detection is required to determine which objects inhabiting a virtual space are intersecting. It is an extremely computationally-expensive operation which must be performed frequently in all DVEs which model the motion of solid objects. It is frequently the second-most computationally-expensive operation in VEs; rendering is usually the most expensive operation. The following section will introduce a number of concepts and collision

detection algorithms and identifies a shortcoming in traditional collision detection algorithms and approaches when adopted in DVEs.

1.1 Collision Detection in DVEs

Collision detection is a heavily-researched topic. It is a classic $O(n^2)$ problem for which, through the exploitation of temporal and spatial coherence, a number of algorithms have been developed which offer better than $O(n^2)$ performance [Lin98] [Storey03] [Morgan04] [Morgan05 2] [Watt01] [Bergen04] [Ericcson05]. Most VEs exhibit both temporal and spatial coherence. The presence of temporal coherence implies that the state of objects at a given time, t_t, will be similar to the state of objects at time t_{t+1}. The presence of spatial coherence implies that the configuration of the objects inhabiting the VE is such that there exists space separating non-intersecting objects.

Traditionally collision detection algorithms have been developed, implemented and optimised for single-processor platforms. However, recent developments in consumer hardware have moved towards multi-core processors, for which many of the available algorithms cannot easily be adapted to exploit. This thesis provides an evaluation of current collision detection algorithms and their applicability on multi-processor platforms.

Current DVEs commonly perform current collision detection between all objects in the DVE on each machine participating in the DVE [Watt01]. This approach can limit responsiveness and scalability by repeating collision detection and can result in inconsistencies in object states between machines; small inconsistencies in object states as a result of collision response can compound on each-other resulting in chaotic, unpredictable object behaviour in DVEs.

The factors contributing to inconsistencies in object states in DVEs include message transmission delays and limited network bandwidth. Limitations in network bandwidth place restrictions on the volume of messages which can be physically transmitted over the available network infrastructure. For the purpose of DVEs, the available network bandwidth places restrictions on the size of and frequency which state update messages can be transmitted. In order to compensate for relatively infrequent state update messages, DVEs commonly utilise state prediction approaches, such as dead reckoning, to extrapolate on object's current state from its previous state using information such as previous position, velocity, acceleration etc. These approaches avoid users noticing objects "jumping" from one state to another. However, these approaches cannot be guaranteed to predict an object's correct state and, as such, are required to use convergence paths to smoothly correct deviations between an object's predicted and actual state. This, however, can result in significant inconsistencies arising. For example, if a collision is detected involving an object while it is travelling on a convergence path it is probable that this collision may be detected differently, or not at all, on one or more machines participating in the DVE. Message transmission delays introduce delays between an object's state changing and this change being realised on all machines participating in the DVE. As a result of this, the state of one or more objects on each machine participating in the DVE may be out-of-date. Any such inconsistencies in the state of objects can result in collisions being detected and responded to in an inconsistent manner between machines participating in the DVE. As previously stated, these inconsistencies can compound on one-another and result in significant differences developing between the state of objects on each machine in a relatively short period of time.

In addition to inconsistencies, the current approach of performing collision detection for all objects on all machines participating in a DVE results in significant processing repetition. As the number of objects inhabiting the virtual world increases, the processing overhead associated with collision detection for

the DVE on each machine increases accordingly. The processing overhead associated with collision detection may restrict the number of, and detail of, objects inhabiting the DVE in order to achieve an acceptable level of responsiveness on the target machines; it may be necessary to sacrifice scalability in order to achieve acceptable levels of responsiveness and vice-versa.

1.2 Thesis Contribution

It is desirable for DVEs to offer high-levels of consistency, responsiveness and scalability. Current DVE research has concentrated on promoting scalability by employing a number of techniques, including interest management/message filtering [Greenhalgh] [Morgan01], dead-reckoning [Watt01] and the use of dedicated server farms [IBM07]. Consistency has been promoted in DVE research by the use of centralised servers [Watt01], lock-step time-stepping schemes [Sweeney99] and state roll-back and correction schemes [Sweeney99]. However, all of these approaches sacrifice one or more of the three requirements (consistency/responsiveness/scalability) in order to improve their desired requirement(s),e.g. an application may restrict responsiveness by imposing that all users interact using a turn-based approach in order to improve consistency. Currently, research into consistency in DVEs has not focussed on one of the major sources of inconsistency, collision detection and collision response. Current research into collision detection has focussed on providing improved performance and accuracy in determining points of intersection between a system of objects. However, little research has been undertaken into collision detection for systems in which the current state of all the objects being simulated is not known by all machines participating in the simulation. This thesis will present a collision detection approach which provides high-levels of scalability, consistency and responsiveness suitable for both single-user VEs and large-scale DVEs. The approach leverages the parallel processing power offered by multi-

core processors and the set of machines participating in the DVE to distribute the processing overhead associated with collision detection to provide improved responsiveness. The approach utilises a peer/server hierarchy network architecture which increases scalability by reducing the volume of messages each machine participating in the DVE is required to transmit and receive. This approach provides improved consistency by reducing the number of machines which are responsible for collision detection for a given object, thereby reducing the probability of conflicting responses being initiated as a result of a collision. It is capable of adapting to variations in network transmission delays, machine failure and network congestion to maintain high-levels of consistency, responsiveness and scalability. Through the use of a peer/server hierarchy network architecture, many of the processing burdens are removed from the main server to localised servers for given territories or networks thereby improving scalability; these local servers are termed group leaders and are dynamically assigned from the machines participating in the DVE based on the network transmission delays perceived between the machines participating in the DVE.

To summarise, it is desirable for a DVE to provide high-levels of consistency to its users. However, current research into collision detection has not yielded approaches which address the problem of consistency in DVEs. This thesis presents a general-purpose network architecture and collision detection approach capable of exploiting both multi-core processors and multiple machines participating in a DVE. The approach enables participants separated by large geographic distances to participate in the same DVE in real-time; every user will perceive high-levels of responsiveness and the level of consistency between users will be adapted depending on the observed network transmission delays and bandwidth restrictions, which are monitored and adjusted during run-time. Experimental results will be presented to demonstrate the performance and applicability of this approach.

Chapter 2

Background

2.1 Virtual Environments

A Virtual Environment (VE) [Burdeau03][Singhal99] is a world simulated within computer software. In general, these systems are designed to allow a user to interact with the virtual environment and the entities which inhabit it through a set of world-specific rules. These rules often mimic the interaction rules observed in the real world, such as Newtonian physics and verbal communication, although they are limited only by the imagination of the VE designer and the processing resources and memory available in the target platform. Due to the commercial success of VEs in the computer games market, research is ongoing to improve both algorithmic and hardware performance of VEs on a global scale in both industry and academia.

2.1.1 Challenges in Virtual Environments

There are a number of challenges associated with developing VEs. In order to attract consumers in a highly-competitive market place, the quality of the graphics is often viewed as having paramount importance. After all, the consumers' first impression of any new product being released onto the market is usually formed by observing its graphics. This has resulted in an on-going competition between developers and manufacturers to produce the most impressive graphics engines [Epic06][Id06][Valve06] and hardware respectively [ATI06][NVidia06]. Animation, which is closely related to rendering, is highly important in producing an immersive virtual environment as smooth and realistic animation can significantly improve a user's immersion. Often receiving less research attention than the two previous challenges, Artificial Intelligence is required to provide the illusion of intelligence in software-controlled entities within the VE. While research into AI for VEs is increasing, the level of research interest previously received was directly related to the level of sophistication of the AI routines employed in VEs.

This thesis will concentrate on the problem of collision detection [Lin98] [Storey03] [Morgan04] [Morgan05 2] [Watt01] [Bergen04] [Ericcson05], which will be discussed in detail later in this chapter. Collision detection is a computationally-expensive operation which must be performed at frequent intervals in all VEs which simulate the motion of solid objects. In terms of computational expense, collision detection is usually the second most expensive operation in a VE behind rendering the VE. Collision detection is used directly within animation, but collision detection techniques are also used extensively in rendering and artificial intelligence.

2.2 Distributed Virtual Environments

A Distributed Virtual Environment (DVE) [Singhal99][Morgan05][Zhao01] is a VE which allows multiple users to interact with the same virtual world in real-time. Each user can inject one or more objects into the DVE, which they can control; in many DVEs, a user injects just a single object, termed an avatar. An avatar is a virtual representation of the user, which they can use to interact with the DVE, the objects and avatars which reside within it. For example, an avatar could be a three-dimensional model of a human which a user controls within a virtual world.

Initial DVE research was undertaken to develop real-time military training and simulation systems [Miller95]. However, due to the wide-spread adoption of the Internet and the affordability of consumer broadband Internet connections, DVEs are becoming more commonly used in a wide-range of non-military applications including:

- Entertainment
- Simulation
- Training
- E-commerce

2.2.1 Challenges in DVEs

There are a number of challenges associated with DVE development. These challenges include those associated with VEs while introducing additional challenges associated with the distributed nature of DVE deployment. A DVE is required to ensure that each user experiences the same, or a very similar, environment and that the actions they perform in the DVE have the same semantics and result in the same reactions being observed by all users. In a

DVE, satisfying real-time requirements while ensuring all users maintain a consistent view of the shared state is difficult [Watt01][Singhal99]. The behaviour of users in VEs is unpredictable. However, DVEs commonly restrict the range of interactions which a user can have with a DVE. Therefore, using heuristics and application-specific interaction rules, it is possible in many circumstances to estimate a user's future interactions with a DVE based on previous behaviour [McCoy04]. However, due to network delays and limited bandwidth, such DVEs can become inconsistent and chaotic, with many participants viewing the world in a number of different ways.

It is desirable for a DVE to scale to allow large numbers of heterogeneous machines to participate in the same DVE. This heterogeneity includes, but is not limited to, the machine architecture and specifications, the operating system and software being executed on the machine and the speed and type of network connection being used.

2.2.2 DVE Implementation Challenges

There are a number of properties which it is desirable for a DVE to possess. Among these are scalability, consistency and responsiveness. A scalable DVE is capable of supporting large numbers of simultaneous participants. A consistent DVE enables the users interacting in a DVE to perceive mutually-consistent states of the objects inhabiting the DVE. A responsive DVE is able to respond to user input sufficiently quickly that these responses are perceived to occur instantaneously.

2.2.2.1 Scalability

A scalable DVE is capable of supporting large numbers of simultaneous users. While there are a number of factors contributing to scalability, scalability in

DVEs is largely affected by message dissemination [Morgan03][Abrams98] [Bharambe02][Singhal99][Watt01]. It is necessary to exchange state update messages between all the machines participating in a DVE to ensure a consistent view of the DVE is perceived by all users. Message exchange must be frequent enough that any event triggered by a user is perceived by all users sufficiently quickly that the DVE appears to be operating in real-time. However, the more frequently messages are exchanged, the more network bandwidth will be consumed. Additionally, as the number of nodes and objects in the DVE increases, the network bandwidth consumption will increase due to the increased number of nodes transmitting state update messages. As the volume of messages transmitted and received by a node increases, the processing overhead associated with handling these messages will rise; this increased processing overhead will restrict scalability and responsiveness. In addition, increased network bandwidth consumption can contribute to network congestion and, therefore, increased message transmission latency, which can affect responsiveness and consistency. As mentioned previously, it is desirable for a DVE to allow large numbers of heterogeneous nodes to participate in the same DVE. As the speed of each node, and the network connection they use, cannot be pre-determined, it is necessary for a DVE to be able to scale to all forms of network connections.

2.2.2.2 Consistency

DVEs can contain a large number of participants, separated by large geographical distances and connected via unreliable, high-latency network connections, e.g. the Internet. As such, it is possible for each user to perceive a different view of the current state of a DVE due to state update messages being subject to varying message transmission delays. It is possible to ensure that any events occurring in the DVE are perceived correctly by participants by appointing an arbitrator, or oracle, to act as a definitive view of the current state of the DVE. However, this can lead to a bottleneck in system performance

because the arbitrator may have limited processing, memory and network resources. Additionally, participants exhibiting large message transmission delays between themselves and the server can observe a significant delay between the detection of an event and its manifestation on their machine; significant delays can detrimentally affect interactivity and user immersion. This problem is termed the consistency-throughput trade-off, because, in general, consistency can be improved at the detriment of throughput and vice-versa [Fischer83][Singhal99][Watt01]. However, currently there exists no solution which allows high-levels of consistency and throughput in DVEs. Current DVEs either restrict the number of users which can interact with each-other or reduce the level of interactivity permitted to allow larger numbers of users to interact with one-another.

2.2.2.3 Responsiveness

A responsive DVE exhibits low latency between a user issuing a command and the response to this command being manifested. Depending on the applications of the DVE, e.g. real-time action computer game, an unresponsive DVE can detrimentally affect user interaction, immersion or suitability of the DVE for its purpose. In order for high-levels of responsiveness to be achieved, the DVE must be capable of processing user input sufficiently quickly that the user cannot perceive a delay between issuing a command and the response being manifested. This requires the DVE to be able to listen to user input, listen to and transmit state update messages and display the updated DVE state to the user at a relatively high-frequency.

The illusion of smooth motion in full-motion video can be achieved at frame rates greater than 25FPS [Watt01][Wiki06]. The smoothness of this motion is partly due to the effect of motion blur, a side-effect of recording continuous motion into discrete video frames using a camera; without the presence of motion blur, frame rates of 25FPS will not give the appearance of smooth

motion; this effect is most noticeable when displaying high-velocity objects. Motion-blur is not present in computer graphics and techniques to reproduce its effects into rendered images are computationally expensive. As such, real-time computer graphics must be rendered at higher frame rates in order to achieve smooth motion; a common target frame rate in DVEs and computer graphics is 60FPS. Additionally, user interaction is usually obtained once per frame drawn. As a user can issue a command (e.g. press a button) at any time between one frame and another, it takes on average 1.5 frames, within the range [1,2] frames for a user command to be received and the response to the command displayed to the user. Therefore, DVEs rendered at higher frame rates will manifest user interaction with less latency than DVEs rendered at lower frame rates. If the frequency of rendering is low, the delay between issuing a command and its effect being realised may become noticeable to the user, which can compromise user interactivity and immersion.

2.2.3 Distributed Application Architecture

There are a number of different architectures [Tenenbaum96] which can be used in distributed applications. Each of these structures has different performance characteristics, including scalability and message delivery latency. Four common architectures are introduced in the following section and evaluated in terms of these performance characteristics.

2.2.3.1 Peer-to-Peer

Peer-to-peer message transmission involves the direct communication between all nodes participating in a DVE. Each node must have knowledge of the network address of every other node participating in the DVE. When a node leaves or a new node joins, every node must be informed of this event and adjust their message recipient data structures accordingly. State update messages are

sent directly between nodes, resulting in no additional transmission delay due to messages having to be processed by an intermediate machine. However, this architecture results in each node having to transmit and listen to incoming messages from and transmit state update messages to the remaining n - 1 nodes participating in the DVE. This can introduce a relatively large communication overhead as the number of nodes in the DVE becomes large, which may limit scalability by not only exhausting available network bandwidth but also consuming significant processing resources, compromising the responsiveness of the DVE.

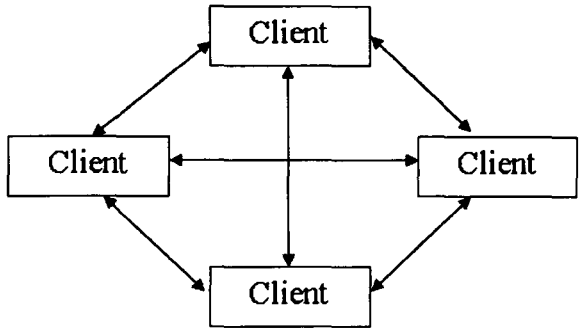

Figure 2.1 Peer-to-Peer Architecture

2.2.3.2 Client-Server

The client-server structure is currently the most popular architecture for distributed applications. It has been widely adopted for use on the Internet. The architecture is simple; one machine acts as a server. The server waits for a client to connect. When a client connects, the server services the client's requests. Once the client's requests have been completed, the connection is severed. This architecture is easy to implement, and is capable of supporting small-to-medium size DVEs, depending on the message dissemination requirements and the server's processing resources. In this architecture, a single machine acts as a message dissemination server. All nodes participating in the DVE connect to their state update messages to the server. The server then forwards these state updates to the relevant nodes in the DVE; the recipients of these messages may

be affected by application-specific message dissemination criteria, e.g. one user is wearing an invisibility cape and, therefore, its state update messages need not be disseminated to the other users. Each client is required to maintain only one connection with the server, and is not required to have any knowledge of the other nodes connected to the DVE. This allows nodes to join and leave the DVE without affecting any other nodes connected to the server. However, this architecture introduces a single bottleneck in the system, the server itself. If the server is not capable of forwarding the incoming stream of messages quickly enough, a backlog of messages may build up, resulting in inconsistency in the clients. This could be alleviated by dropping messages if they cannot be processed within a time threshold, although this may compromise the fluidity of the DVE due to objects changing state radically as a result of the server dropping a number of the object's state update messages. Additionally, the client-server architecture introduces an inherent delay in message delivery; a message must pass from the client to the server, and then from the server to the other clients. This imposes an additional network transmission and processing delay, compared to if the nodes were to transmit messages to one-another directly.

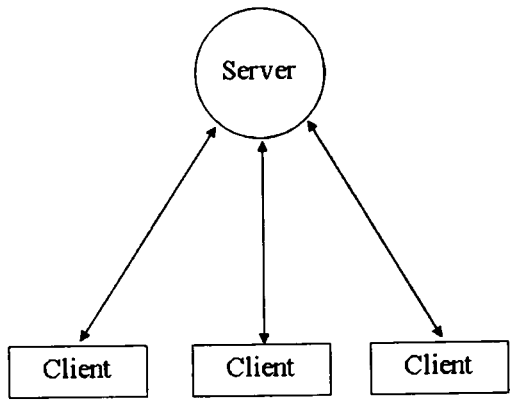

Figure 2.2 Client-Server Architecture

2.2.3.3 De-centralised Server

The de-centralised server architecture is an extension of the client-server model. In the de-centralised server model, a number of machines act as servers. Each server is connected to one-another using the peer-to-peer architecture. Nodes connect to a one of the servers; the choice of server can be arbitrary or based on network or geographic properties, e.g. physically closest server. Each server processes the state update messages it receives and passes them on to the relevant clients and other servers, which will subsequently deliver these messages to their clients. The clients are unaware that there are multiple servers, as the client-side implementation is identical to that of a client-server model. This architecture helps to alleviate the problem of a single bottleneck in the communication subsystem. However, it introduces further transmission delays on top of the client-server architecture, as messages must now travel through more than one server.

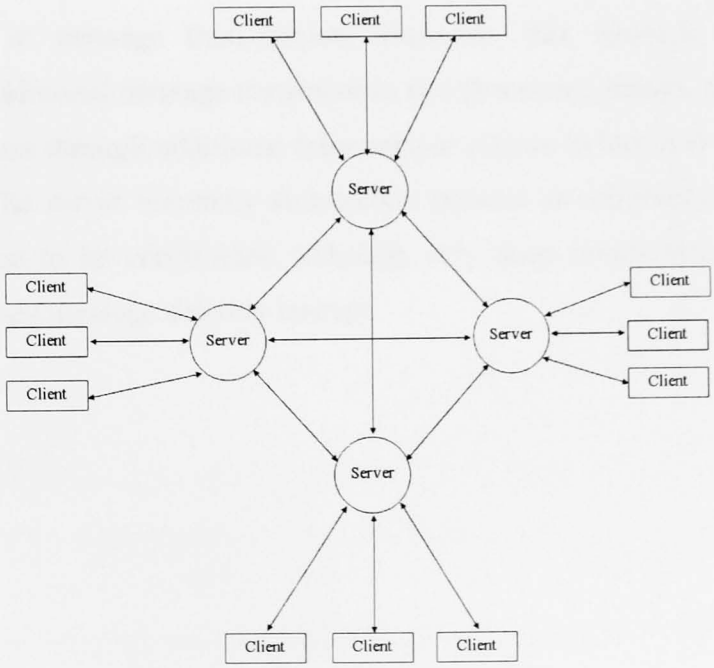

Figure 2.3 De-Centralised Servers

2.2.3.4 Server Hierarchies

Server hierarchies offer the most scalable approach to message dissemination, at the detriment of message transmission delays. A server hierarchy, from the client's perspective, is a client-server model, in which a client connects to a single server and transmits its state update messages. The server delivers these state update messages to the nodes participating in the DVE. In the server hierarchy architecture, the server that a client is connected to can act as a client to another machine. This "super-server" is responsible for delivering the messages received from each of its servers to the other servers in the DVE, which will in turn deliver these messages to their clients. This architecture avoids the need to maintain a peer-to-peer connection with servers, which allows servers to become available and unavailable during the lifetime of the DVE without each server needing to be aware of it. In addition, this architecture helps to reduce the number of connections each server must maintain with the other de-centralised servers in the DVE, helping to reduce the likelihood of bottlenecks in message transmission. However, this network architecture introduces additional message transmission and processing delays, as a message must now pass through additional intermediate servers before it is received by the client. The server hierarchy architecture permits an arbitrarily-deep server hierarchy tree to be constructed, although very deep server hierarchies may result increased message delivery latency.

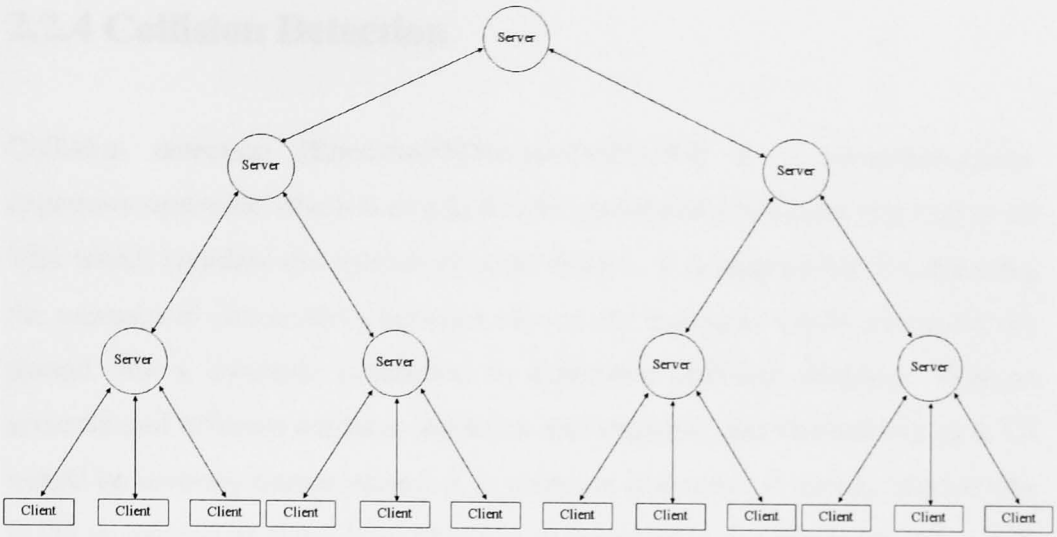

Figure 2.4 Server Hierarchy

2.2.3.5 Summary of Distributed Application Architectures

Four distributed application architectures were introduced:

- Peer-to-Peer
- Client-Server
- De-centralised Server
- Server Hierarchies

The peer-to-peer architecture generally offers the fastest message delivery speed at the cost of scalability; the architecture can be made scalable through the use of hardware multicasting, but hardware multicast protocols are only available on LANs. Client-server offers improved scalability but message delivery latency is increased due to messages having to be processed by an intermediary. The de-centralised server architecture provides further scalability by employing multiple servers. This enables additional users to participate in a DVE at the cost of increasing message transmission latency. Server hierarchies offer the highest level of scalability but can result the largest message transmission delays.

2.2.4 Collision Detection

Collision detection [Ericcson05][Bergen04][Lin98] is a computationally-expensive operation which is required to be performed at frequent intervals in all VEs which simulate the motion of solid objects. It is responsible for detecting the presence of intersections between objects, the results of which are commonly passed into a dynamic simulation to determine collision response. Without accurate and efficient collision detection and response, the interactivity of a VE would be severely compromised; it is common that most, if not all, interactions in DVEs depend on some form of collision detection, e.g. walking up stairs, pick up a box, push something. However, collision detection and response often goes unnoticed because users will normally only notice errors in collision detection and response. As such, inconsistencies in collision detection and response in DVEs are most noticeable by users and can detrimentally affect user interaction.

It is common that collision detection is performed for all objects in the DVE on each machine participating in a DVE using the most recent object state information received by each machine [Watt01]. The results of local collision detection can be passed into a physics simulation to generate collision response. However, due to limited frequency state update messages and variations in network latency between the machines participating in the DVE, the state of the objects on each machine participating in the DVE may differ significantly from one-another. These deviations can result in differences in collision response, ranging from minor to completely different responses.

Dead reckoning is a commonly-used approach to reduce message transmission frequency by predicting the future position of objects based on their previous state; messages are only transmitted when the node hosting the object detects a deviation between the object's true state and its predicted state in excess of some pre-defined threshold value. The use of predictive approaches such as this can cause further inconsistencies in object states and therefore in collision response

due to objects straying from their predicted paths. While it is possible to reduce deviations by exchanging additional messages describing the state of objects and correcting deviations between machines, this can result in undesirable visual anomalies as objects change state arbitrarily to correct deviations, e.g. an object jumping from one position in the world to another. In addition, it is difficult to correct significant deviations elegantly between the state of objects on different machines, e.g. a deviation which result in a player being killed on one screen but surviving on another.

In order to achieve high-levels of responsiveness, it is necessary for the DVE to complete all of the processes required to display the next frame to the user at high-frequencies, e.g. 60FPS [Wiki06]. As one of the major overheads, collision detection must be computationally-efficient and must not miss collisions or allow objects to be rendered penetrating one-another. In addition, it is desirable for collision detection in DVEs to be able to overcome inconsistencies between the perceived states of objects on each machine participating in the DVE to produce a consistent virtual world.

2.2.5 Responsiveness and Consistency in Collision Detection for DVEs

A number of collision detection approaches have been proposed [Lin98][Watt01][Bergen04][Ericcson05], which offer high-levels of performance and accuracy. Current approaches are designed to perform accurate and efficient collision detection on single-user VEs. The same approaches used in VEs are adopted for use in DVEs. However, due to network transmission latency, each machine participating in the DVE may detect and respond to collisions using inconsistent object state information. Inconsistencies in collision detection and response can significantly compromise user immersion. In addition, as the number of objects inhabiting a DVE increases, the

computational overhead associated with collision detection may detrimentally affect responsiveness due to limited processing resources on a single machine. The exploitation of the highly-parallel distributed processing resources made available by DVEs may offer an opportunity to improve scalability, responsiveness and consistency.

The following sections introduce a number of existing collision detection algorithms and evaluate their applicability in DVEs. An ideal collision detection approach for DVEs will:

- Be easily parallelisable to take advantage of multi-core processors and the presence of multiple distributed processing resources to offer improved responsiveness
- Not require excessive message exchange between machines participating in a DVE and be suitable for deployment using the server hierarchy network architecture to offer improved scalability
- Offer high-levels of consistency between machines participating in the DVE

Currently, a collision detection approach does not exist which fulfils these requirements. As such, the following section examines the possibility of modifying existing algorithms to fulfil these requirements.

2.3 Collision Detection Algorithms/Approaches

Collision detection is a highly computationally-expensive process which is required to be performed at frequent intervals in any VE which models the motion of solid objects. Being usually performed each frame, and often more frequently, it is one of the major overheads in VEs and, as such, performance is of the utmost importance. Collision detection is a term which is used with a wide range of meanings. In its strictest sense collision detection is a

computational geometry problem which is required to determine if a number of objects intersect to a certain level of detail. As the two operations are so intrinsically linked, the term Collision Detection is sometimes used, slightly inaccurately, to describe the act of detecting and responding to a collision, a process which usually requires the presence of a dynamic physics simulation.

It is common in computer graphics for each object to be represented as a set of polygons [Gottsman05]. This could result in a naïve collision detection approach comparing every polygon in two objects to determine if the two objects are intersecting. This leads to collision detection between two objects composed of p polygons requiring p^2 polygon-polygon comparisons. Similarly, in a virtual world which consists of n objects, a brute force approach to collision detection between these objects would require n^2 object-object comparisons, where each object-object comparison requires p^2 polygon-polygon comparisons. As p and n become larger, even the fastest computers will struggle to maintain real-time performance [Sedgewick96].

The problem of real-time collision detection has received large amounts of research interest. A number of algorithms and approaches faster than brute force collision detection have been presented. Many of the earlier algorithms were targeted towards static meshes, i.e. objects which could undergo affine transformations (translation, rotation, scaling), but whose vertices in model-space do not change. More recently, new algorithms and extensions to previous algorithms were developed to provide efficient collision detection for animated objects, such as model humans with moveable limbs. These were often pre-scripted animations, in which the animation was simply a sequence of pre-calculated poses the object could be drawn in. Recently, further research has been performed into collision detection for objects which can undergo arbitrary movements, e.g. objects which can change shape as a result of external influences. These collision detection algorithms must deal with not only

collisions with other objects but also self-intersection, in which an object's polygons intersect with one-another.

In the following sections, a number of collision detection approaches and algorithms will be introduced. Wherever possible, these algorithms will be discussed in terms of the performance they offer and their applicability in parallel or distributed execution. This consideration is due in part to the recent adoption of multi-core processors in consumer PCs and next generation consoles. Additionally, a collision detection algorithm which is capable of being executed in parallel may also provide opportunities to execute collision detection in a distributed fashion, providing a scalable collision detection approach. A number of the approaches discussed in this section are not suitable for parallel execution by themselves. However, these approaches may be suitable for parallel execution when used in conjunction with other algorithms.

2.3.1 Parallel/Distributed Execution

Software systems utilising multiple threads of control [Jaja92][Ben-Ari06] can be categorised in terms of the proportion of program which can be run in parallel compared with the proportion of the program which must be run sequentially. It is common that applications are separated into sections which can be run in parallel and sections which must be run sequentially. By abstracting away a number of lower-level issues regarding parallel execution, the parallel execution of an application can be categorised in terms of forks and joins. A fork is a stage in a program in which one process can be forked into multiple parallel processes which can execute concurrently. At the end of such a parallel section, a join can be performed which waits for all the processes to terminate before the main thread of execution can proceed further. Each fork and join operation can be seen to incur a cost. A fork operation has the cost of generating a number of new processes, which requires a number of kernel-level operations to be performed

by the operating system. Conversely, a join operation requires the main process to wait for the processes which are being joined upon terminate; this can be a costly operation if the main process must wait for a large number of processes to terminate.

In order to reduce the performance costs of executing parallel threads within an application, it is common that synchronisation primitives are used to control the execution of threads by:

- Placing locks on resources, e.g. function or access to a given memory address
- Using message passing to cause processes to wait, sleep or wake-up depending on application-definable conditions
- Control the priority of processes and yield processing resources when it is desirable to do so

These synchronisation primitives can result in more efficient parallel performance compared to forking new processes and terminating them whenever a parallel section of code is completed. This performance gain is achieved because synchronisation primitives are generally more lightweight operations than kernel-level process-management. However, the level of parallelism offered by an algorithm remains consistent regardless of whether these synchronisation primitives are used as opposed to kernel-level process management.

An important factor in determining whether it is beneficial to parallelise an algorithm is the granularity of the parallelism. Fine-grained parallelism implies that the amount of work that is to be executed in parallel is relatively small; coarse-grained parallelism, conversely, implies that the proportion of work that is to be executed in parallel is relatively large. Very fine-grained parallelism is undesirable as the processing overhead of spawning and managing a new

process can outweigh the performance improvements offered by performing the processing in parallel.

2.3.2 Two Phase Collision Detection

Collision detection within a animated virtual environment is separated into two inter-related problems. Given a virtual world with n objects each consisting of p polygons, a brute force collision detection approach would require n^2 object-object comparisons, each requiring p^2 polygon-polygon intersection tests. Rather than performing p^2 polygon-polygon intersection tests for each pair of objects, a more efficient approach to collision detection could be to use some less computationally expensive technique to quickly disregard a pair of objects from further consideration. Any pairs of objects which are not culled by this process must then undergo further consideration to determine if the objects do in fact collide. These phases are termed *Broad Phase* and *Narrow Phase* respectively [Lin98][Bergen04][Watt01][Ericcson05].

2.3.2.1 Broad Phase Collision Detection

Broad Phase collision detection is a class of collision detection algorithms which is intended to reject pairs of objects from further consideration using computationally inexpensive techniques. If a pair of objects is found to be colliding in broad phase collision detection, this does not definitely mean that the pair of objects is in fact colliding; only that the objects warrant additional inspection. There are a wide-range of broad phase collision detection algorithms which have been developed. Most broad-phase approaches use bounding volumes, which will be descried in the subsequent section. The most popular broad-phase collision detection techniques are Sweep-and-prune [Lin][Lin98] [Bergen04][Ericcson04] and spatial-subdivision approaches [Watt01][Bergen04] [Ericcson04]. These two approaches to broad-phase collision detection will be

introduced and evaluated for their applicability in distributed/parallel environments.

2.3.2.1.1 Bounding Volumes

A popular broad-phase collision detection technique is the use of bounding volumes to enclose a set of polygons [Watt01][Bergen04][Ericcson05][Lin98]. This can provide a highly efficient culling strategy because if two objects' bounding volumes do not intersect, then the polygons which the bounding volumes enclose cannot possibly collide with one-another. This provides an efficient collision detection strategy, providing performing collision detection between bounding volumes can be performed inexpensively. To this end, a number of shapes have been proposed as bounding volumes for collision detection:

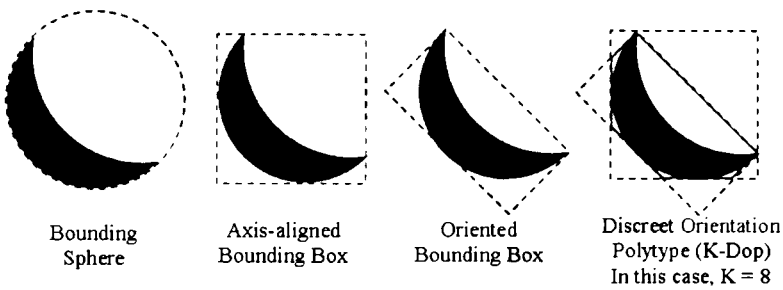

Figure 2.5 Bounding Volumes

The different bounding volumes exhibit different benefits and weaknesses. A bounding sphere occupies the smallest memory footprint, being represented by its centre-point and its radius. It also is relatively computationally cheap to perform intersection tests between two spheres; if the distance between two spheres' centre points is less than the sum of their respective radii, then the spheres overlap. Bounding spheres are rotationally independent. This means that the object can undergo any form of rotation and translation without needing to re-calculate the bounding sphere. Bounding spheres, unfortunately, often exhibit

low bounding efficiency. High bounding efficiency means it is less likely a collision detection engine will need to unnecessarily enter the expensive, narrow-phase collision detection stage with a pair of objects. It can be seen that a sphere can provide high bounding efficiency for spherical objects. However, spheres will provide extremely low bounding efficiency for long, thin objects.

Axis-Aligned Bounding Boxes (AABBs) require slightly more memory to store than bounding spheres, being represented by two vectors containing:

- Its extremes along the coordinate axes
- Its centre and half-extents along the coordinate axes

The former representation offers faster collision detection whereas the latter offers faster transformation, as only the half-extents must be rotated provided rotations occur around the object's centre. An AABB offers efficient collision detection, and often exhibits better bounding efficiency than spheres. AABBs are not rotationally independent; they must be recomputed after an object's rotation. Recalculation can be performed in one of three ways:

- Recalculate from scratch by clearing the AABB and adding each transformed point to the AABB
- Update the AABB by rotating the 8 corners of the AABB and using these to determine the extremes of the AABB.
- Rotating the extents of the AABB using the absolute rotation matrix (the sign bit of each member of the rotation matrix is dropped)

The first option will result in the best possible bounding efficiency for an AABB, but it is computationally expensive. The second and third options are computationally cheap, but may result in larger bounding boxes, lowering bounding efficiency. It is also possible to ensure an AABB is rotationally-independent by expanding the AABB to encompass all possible rotations of the object it encloses; this approach results in an axially-aligned bounding cube.

Oriented Bounding Boxes (OBBs) require additional storage space. In three dimensions an OBB is represented by its extents along its local x-, y- and z-axes, its position and the orientation of these axes (a 3x3 matrix); in total this requires 15 floating point numbers. OBBs are relatively expensive to perform collision detection between, requiring 15 potential separating axes to be considered to determine whether a pair of OBBs intersects. However, OBBs will generally provide better bounding efficiency that AABBs or bounding spheres. They should be pre-calculated either algorithmically or manually during object modelling. OBBs are not rotationally independent. However, unlike AABBs, the OBB's planes are not restricted to being axially-aligned. Therefore, the OBB's planes can be transformed with the object they enclose. As the OBB is oriented to be the tightest possible fit to the object, the OBB after object rotation will also be the tightest possible fit.

Discrete Orientation Polytypes (K-DOPs) are a generalisation of AABBs. The value of K indicates how many candidate planes are available to enclose the object. If K is 4, then the bounding volume is an OBB (an AABB if the K planes are axially-aligned). Essentially, a set of candidate planes is chosen to enclose the object. The combination of these planes which provides the highest bounding efficiency is selected as the bounding volume. The memory footprint and computational overhead of collision detection on K-DOPs depends entirely on how many planes constitute the bounding volume. As K increases, the bounding efficiency and computational cost of collision detection increases. K-DOPs can be used as a narrow phase collision detection scheme as well as broad phase. Similar to OBBs, K-DOPs are not rotationally independent. They can, however, be rotated in conjunction with the 3D object. As with OBBs, K-DOPs can be pre-calculated either algorithmically or manually during object modelling.

2.3.2.1.2 Coherence

The use of bounding volumes can have a great effect on the performance of a collision detection engine, as they can substantially reduce the number of objects which must have more detailed collision detection performed upon them. However, a naïve approach to collision detection using bounding volumes will still require every object's bounding volume to be compared with each-other. Given a world containing n objects, this will require $\frac{n(n-1)}{2}$ comparisons. As n becomes large, this will become a severe performance drain. In order to alleviate this problem a number of collision detection approaches have been proposed which can reduce the number of bounding volume comparisons required for broad phase collision detection. These algorithms exploit coherence in the virtual world: spatial coherence and temporal coherence. High levels of spatial coherence [Li01] in the virtual world means that there exist planes or empty space which separate objects from one-another. With this type of coherence, it is possible to separate a set of objects into smaller groups of potentially colliding objects. A virtual world exhibits high levels of temporal coherence [Lin98] if there is a high level of similarity between object states from one time-step to another. With this type of coherence, it can be said with some level of confidence that if a pair of objects do not collide at time T_t, it is unlikely that they will be colliding at time T_{t+1}, providing the change in object positions between T_t and T_{t+1} is relatively small. These two forms of coherence commonly occur in most virtual worlds, and can be exploited to reduce the number of comparisons required to perform collision detection. A number of collision detection algorithms will be presented which exploit a combination of temporal and spatial coherence. Initially, two algorithms will be introduced which utilise the properties of two bounding volumes: Sweep and Prune [Lin] and Expanding Spheres [Storey04].

2.3.2.1.3 Sweep and Prune

The Sweep and Prune algorithm [Lin][Lin98][Bergen04] operates on AABBs. It is currently the most popular broad-phase collision detection algorithm used in commercial physics simulation and computer games due to its $O(n)$ average run-time performance and memory usage. A property of AABBs is that a pair of objects intersects if and only if their projections onto the coordinate axes overlap. Sweep and Prune is a coordinate-reducing strategy, in which the objects' extremes are sorted along the x-, y- and z-axes into three lists of start and end-points, which are traversed in turn. The algorithm provides a best-case $O(n)$ performance, average-case $O(n)$ and worst-case $O(n^2)$ performance. The worst-case occurs when the objects clump along an axis. A list of currently active objects is maintained, which is initially empty. In addition, a list of the number of times a pair of objects has been found to be overlapping along an axis is maintained, in which all entries are initially set to zero.

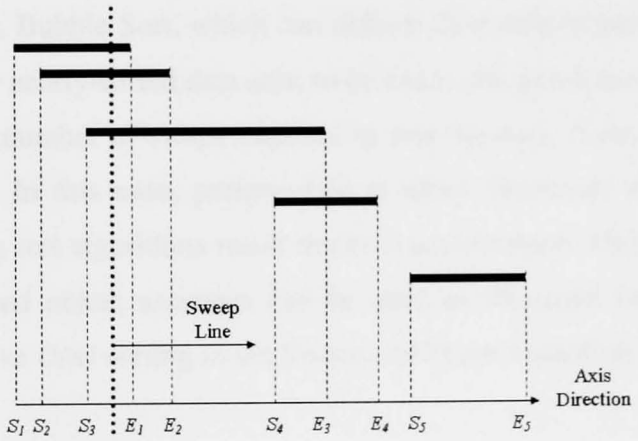

Figure 2.6 Sweep and Prune along a single axis

The first axis to be swept is selected, e.g. the x-axis. The list of AABB extremes is traversed in order. When a start point is found, the corresponding object is added to the active list. When an end point is found, the corresponding object is removed from the active list. At any point in the sweep, the set of objects in the active list overlap along the given axis. Figure *2.6* shows sweep and prune along a single coordinate axis. As can be seen, at the current stage of sweeping, the current set of objects on the active list is $\{obj_1, obj_2, obj_3\}$. When an end point is

reached, the set of objects currently in the active list is recorded as overlapping along the axis and the corresponding object is removed from the active list. In Figure *2.6*, the resulting set of overlapping objects should be: $\{\{obj_1, obj_2, obj_3\}, \{obj_3, obj_4\}\}$. The process is repeated for all three axes. If a pair of objects overlaps along all three axes, then the AABBs overlap. In a virtual world with n objects, each list will contain $2n$ points. As three lists must be traversed, this gives the algorithm $O(n)$ performance; $6n$ to be precise. The major workload for this algorithm is in maintaining the three sorted lists. Sorting has a complexity of $O(n^2)$ [Sedgewick96], with the very best sorting algorithms able to sort data in $O(nlogn)$ time. However, it has already been identified that VEs exhibit high-levels of temporal coherence, as the configuration of objects only changes slightly between time-steps. This implies that the set of object extremes sorted along an axis at time T_t is likely to be similar or identical to the sorted extremes at time T_{t+1}. There exists a class of sorting algorithms, called *sifting sort* algorithms, e.g. Bubble Sort, which can deliver $O(n)$ sorting performance when given sorted or nearly-sorted data sets; to be exact, the performance is $O(n + c)$, where c is the number of swaps required to sort the data. If the data is already sorted, $c = 0$. In this case, performance is $O(n)$. However, if the dataset is random, sifting sort algorithms result in $O(n^2)$ performance. This means that the previously-sorted object extremes can be used as the input for a sifting sort algorithm to give $O(n)$ sorting in the Sweep and Prune algorithm.

2.3.2.1.3.1 Parallelism in Sweep and Prune

The Sweep and Prune algorithms offers considerable performance advantages over brute-force collision detection. However, it offers only limited opportunities to exploit parallelism and, due to the nature of the algorithm, can offer only constant performance optimisation as opposed to increasing performance by orders of magnitude. The sweep and prune algorithm can be separated into two constituent sub-processes: sorting and sweeping.

The sorting process requires that three lists of object extremes be sorted. Trivial parallelism can be exploited by sorting each individual list in parallel, whereby three processes could be used to sort each list. The list sorting process could be further parallelised with the use of divide-and-conquer sorting algorithms such as QuickSort. Divide-and-conquer sorting algorithms offer O(nlogn) performance by sub-dividing the list into sets of smaller lists and recursively sorting these lists; each of the sub-lists can be sorted in parallel. However, this offers very fine-grained parallelism, whereby the cost of creating a new process or thread to sort the list may outweigh the benefits of performing the operation in parallel. In addition, due to the presence of temporal coherence, better performance can be achieved using sifting-sort algorithms; such sorting algorithms can not be parallelised.

The sweeping process can only be commenced once the sorting process has been completed; this requires a join operation to be performed waiting for the sorting to complete before the sifting can begin. The sweeping process requires that each of the sorted arrays be traversed, recording from each array which objects overlap along a given coordinate axis; if a pair of objects overlaps along all three coordinate axes, the objects' AABBs intersect. It is possible to sweep each of the three lists in parallel. However, this incurs the additional cost of traversing the three lists of overlapping objects along each coordinate axis to determine which objects are intersecting. If the sweeping process is performed sequentially, this additional step can be avoided by examining the results from the previous coordinate axis' sweep to determine if a pair of objects overlapping on a given coordinate axis is potentially intersecting. This additional step can also be avoided in parallel execution at the cost of memory by utilising a matrix counting the number of axes a pair of objects overlaps upon. Every time a pair of objects overlaps along a coordinate axis, the corresponding value for this pair of objects is incremented. If this value reaches 3, the objects are intersecting and successfully pass from the broad phase to the narrow phase. This matrix must be

NxN, where *N* is the number of objects and each element must be set to 0 before the sweeping phase is begun.

The Sweep and Prune algorithm offers the capacity to be executed in parallel across 3 processors, but offers little capability to improve performance if more than 3 processors are available. This is a reasonable performance optimisation on current home computers, where it is uncommon to have large numbers of processors. However, next-generation games consoles, such as the Sony PlayStation III, have up to 8 processing cores. In addition, the sweep and prune algorithm does not lend itself to distributed collision detection in DVEs as such applications may host hundreds or thousands of simultaneous participants; the exploitation of this number of distributed processors for use in collision detection requires algorithms capable of executing in parallel across arbitrary numbers of processors.

2.3.2.1.4 Spatial Subdivision

Spatial coherence is most easily exploited by subdividing the VE into unit cells [Watt01]. In VEs where the objects move on the ground, this can be simplified into a 2D grid. This can potentially provide highly-efficient broad phase collision detection as objects need only be compared with one another if they occupy the same subspace. The major problem with this approach is choosing an optimal size for the cells, as it is undesirable for an object to occupy multiple cells simultaneously. It also requires an algorithm which is capable of determining which cells an object occupies efficiently. A number of approaches have been proposed to solve this problem. These generally involve the use of recursive data structures, such as Quadtrees [Samet84], Oct-trees [Watt01] and Binary Space Partitioning (BSP) trees [Fuch80][Naylor90][Wiley97]. An approach will also be presented, Spatial Hashing [Ericcson05], which does not require a recursive data structure. Each of the spatial subdivision approaches

will use an object's approximate bounding volume to reduce the computational overhead.

Spatial subdivision provides a strong broad phase collision detection approach. In order to demonstrate the efficiency of spatial subdivision, it is assumed that narrow phase collision detection is performed in a brute force manner within each region; this may not be the most efficient approach, but it illustrates the strength of spatial subdivision sufficiently. Given a VE with n objects and s sub-regions, each sub-region would contain, on average, n/s objects. A brute force approach to collision detection in a single cell would require:

$(n/s)((n/s)-1)/2 = \frac{1}{2}\left(\frac{n^2}{s^2}-\frac{n}{s}\right)=\frac{n^2}{2s^2}-\frac{n}{2s}$ comparisons between objects. Given that this must be performed s times, this gives a total cost of: $s\left(\frac{n^2}{2s^2}-\frac{n}{2s}\right)=\frac{n^2}{2s}-\frac{n}{2}=\frac{n^2-ns}{2s}$.

Given that n = 100 and s = 50, this would require, on average, 50 comparisons between objects. This is significantly less than the 4950 comparisons between objects required if there was just a single sub-region. While spatial subdivision explicitly exploits the presence of spatial coherence, it can also exploit temporal coherence because objects will usually remain in the same sub-regions for more than one time-step. As such, it is not necessary to re-insert objects into the spatial subdivision every time-step.

2.3.2.1.4.1 Quad-trees and Oct-trees

A Quad-tree [Samet84] is a tree structure in which each non-leaf node in the tree contains four child nodes. It is used primarily for subdividing two dimensional VEs, although it can also be applied to three dimensions. Figure ***2.7*** shows the mechanism whereby the space is subdivided into four equal-sized subspaces along the x- and y-axes. This subdivision is applied recursively until some termination criterion is reached, e.g. minimum cell size.

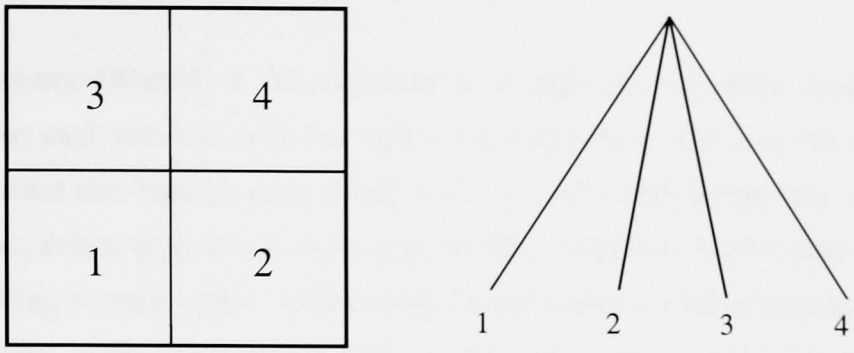

Figure 2.7 Quad-Tree Structure

Figure ***2.8*** shows a Quad-tree decomposition of a VE containing a sphere, a rod and a box. In this diagram, the termination criterion is that a cell contains only a single object, or part of object. This is a reasonable approach for VEs in which the configuration of objects does not change. This, however, is not applicable in VEs where the configuration of objects is not known a priori. In this case, a different termination criterion should be used, such as minimum threshold cell size. If such an approach is used, run-time optimisations could be used to further subdivide overcrowded cells or merge under-populated cells into a single cell.

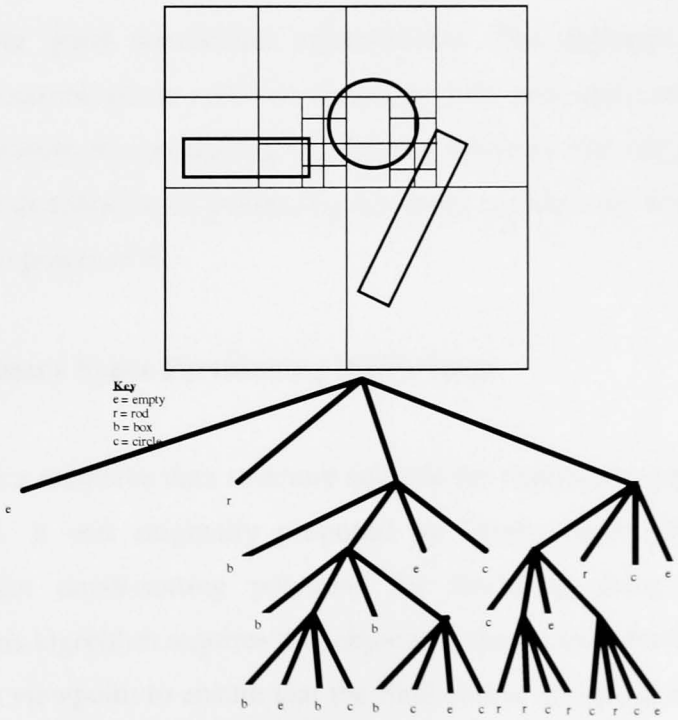

Figure 2.8 Quad-tree decomposition of a VE

An Oct-tree [Watt01] is the extension of a Quad-tree into three dimensions, whereby each non-leaf node has eight child nodes. In an Oct-tree, the world is subdivided into uniform cells along the x-, y- and z-axes recursively. An Oct-tree, as with a quad-tree, can be used to fully subdivide the VE into regions containing a single object. Alternatively, it can maintain a list of objects in each leaf node. In this case, objects sharing the same cells are candidate colliding objects.

In order to perform broad phase collision detection on a VE using an Oct-tree, it is necessary to insert each object into the tree. It requires *O(logn)* comparisons to insert a single object into the tree, where n is the number of cells the VE is subdivided into. Given a VE consisting of p objects, this leads to a broad phase collision detection cost of *O(plogn)*.

2.3.2.1.4.1.1 Parallelism in Oct-trees

Oct-trees offer good parallelism opportunities. The approach allows both insertion and narrow-phase collision detection to be performed in parallel. Oct-trees can distribute the processing between an arbitrary number of processors. However, the distribution of processing resources is only even if the number of processors is a power of 8.

2.3.2.1.4.2 Binary Space Partitioning (BSP) Trees

A BSP tree is a recursive data structure suitable for exploiting spatial coherence within a VE. It was originally proposed by Henry Fuchs [Fuchs80] as a mechanism for depth-sorting polygons for rendering using the painter's algorithm. This algorithm requires that objects be drawn from back-to-front with respect to the viewpoint to ensure that the final colour of a pixel is correct. This

was prior to memory becoming cheap enough for depth-buffer values to be retained; currently, depth sorting is only necessary to correctly render transparent objects.

A BSP tree contains, at each non-leaf node, a partitioning plane, which subdivides the scene, or part of the scene, into two subspaces. This partitioning is recursively performed until some termination criterion is reached, such as maximum tree depth or minimum number of objects in a subspace. Each leaf node contains a list of all the objects which it contains.

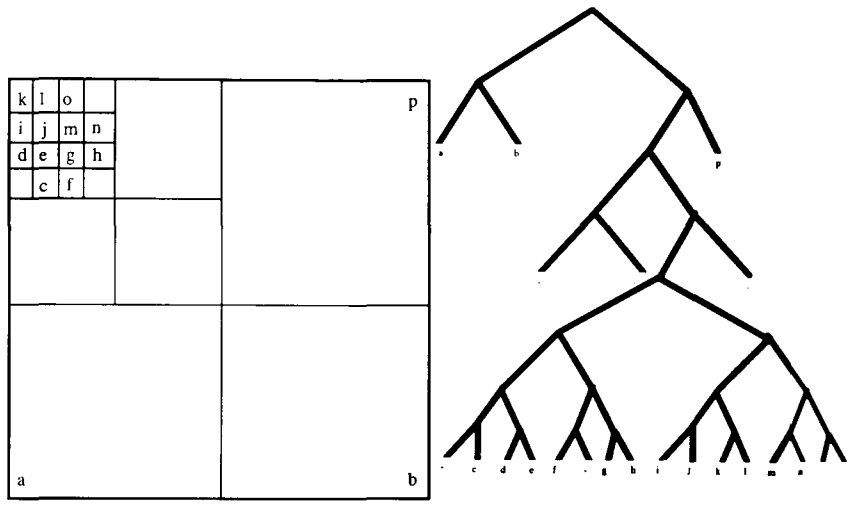

Figure 2.9 BSP Tree of a VE

Figure ***2.9*** shows a BSP tree using axially-aligned partitioning planes to divide the VE into equal-sized cells; this process is termed *static subdivision*. This results in an unbalanced binary tree of the VE with maximum height 8. This demonstrates an issue with spatial subdivision: objects are not always uniformly distributed throughout the VE. A problem with Oct-trees and Quad-trees is that they are likely to produce uneven trees with large numbers of empty cells. When a BSP tree is used to represent a subdivision of space into cubic cells, it shows no significant advantage over a direct data structure encoding of an Oct-tree. However, a BSP tree is not required to subdivide the space into uniform cells; in fact, the main advantage of BSP trees is that the space can be divided using an arbitrary plane. As BSP trees prescribe the use of an arbitrary partitioning plane

at each level of subdivision, rather than an Oct-tree's restriction of axially-aligned partitioning planes, BSP trees can be considered a generalisation of Oct-trees.

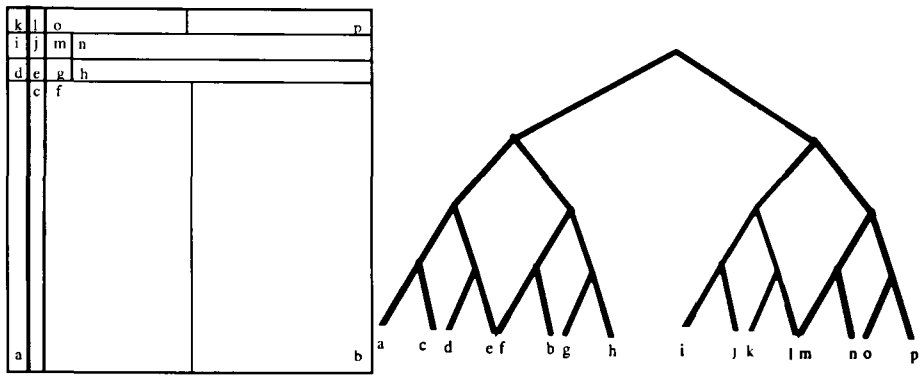

Figure 2.10 BSP Tree of a VE with Adaptive Subdivision

As mentioned previously, the partitioning plane used in BSP trees can be arbitrarily oriented. As such, it is possible to use *adaptive subdivision* to generate a balanced tree. In adaptive subdivision, a partitioning plane is selected which subdivides the objects in the VE into two approximately equally-sized sets of objects. Figure ***2.10*** demonstrates a two dimensional VE in which adaptive subdivision was used; this is the same VE as shown in Figure ***2.9*** using static subdivision. These diagrams illustrate why adaptive subdivision is advantageous: Figure ***2.9***'s BSP tree has a maximum height of 8, whereas Figure ***2.10***'s BSP tree has a maximum height of 4.

Adaptive subdivision requires that at each non-leaf node, a partitioning plane is selected which divides the objects in the space into two roughly-equal groups. This is, however, a computationally expensive process, as there are an infinite number of partitioning planes in any space. This property rules out a brute-force approach to select appropriate partitioning planes. However, it is still possible to select good partitioning planes; it is just not possible, in all but the simplest cases, to prove that these partitioning planes are optimal. An algorithmic

approach to performing adaptive subdivision is provided later when BSP trees are re-introduced for use in narrow-phase collision detection.

As the configuration of objects within the VE is not constant, adaptive subdivision may not be appropriate for such an environment, as a good partitioning plane at time T_t may become a bad partitioning plane at time T_{t+1}. Instead, it may be more appropriate to use static subdivision, followed by update operations to adjust the depth of the tree depending on the distribution of objects. This is another benefit of BSP trees over Oct-trees; it is much easier to update a BSP tree than it is to update an Oct-tree or Quad-tree.

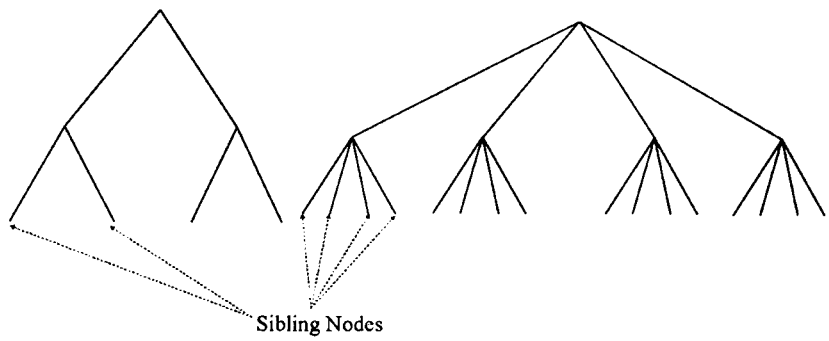

Figure 2.11 Sibling Nodes in BSP Trees and Quad-trees

Figure ***2.11*** shows sibling nodes in a BSP tree and a Quad-tree. Sibling nodes are child nodes which share the same parent node. It is quite logical that a non-root node in a BSP tree will have one sibling; a non-root node in a Quad-tree will have three siblings and each non-root node in an Oct-tree will have seven siblings. In order to update the spatial subdivision of a VE, two threshold values can be used:

- The minimum number of objects allowed in a group of sibling leaf nodes
- The maximum number of objects allowed in a leaf node.

If there are fewer than the minimum number of objects in a group of sibling nodes, then the sibling nodes can be merged together into one node. In a BSP tree, this requires only 2 nodes to be considered, whereas an Oct-tree requires 8

nodes to be considered. Similarly, if there are more than a maximum number of objects in a single node, then that node should be subdivided. In a BSP tree, this would result in the formation of one additional leaf node; in an Oct-tree, this would result in the formation of seven additional leaf nodes. This implies that BSP trees are more likely to produce fewer empty, wasted leaf nodes than an Oct-tree, therefore providing better performance and memory utilisation.

In order to perform broad phase collision detection on a VE using a BSP tree, it is necessary to insert each object into the tree. It requires $O(lgn)$ comparisons to insert a single object into the tree, where n is the number of cells the VE is subdivided into. Given a VE consisting of p objects, this leads to a broad phase collision detection cost of $O(plgn)$. Following this, narrow phase collision detection is required to determine which objects residing in the same leaf nodes do in fact collide with one-another.

2.3.2.1.4.2.1 Parallelism in BSP Trees

BSP trees offer good opportunities for parallelism. The approach allows both insertion and narrow-phase collision detection on each sub-region to be performed in parallel. In addition, BSP trees allow an arbitrary number of parallel processors to be used. However, best performance is yielded when the number of parallel processors is a power of 2. BSP trees allow dynamic spatial subdivision to be performed, in which sub-regions are further subdivided or merged depending on the distribution of objects in the DVE. Arbitrary partitioning planes may be employed in BSP trees to leverage the best subdivision performance; this is desirable if the BSP tree's configuration is to be pre-calculated based on the shape of the environment which the users participate in, e.g. partitioning planes coplanar to the walls separating rooms or the ceilings separating floors in a high-rise block.

2.3.2.1.4.3 Spatial Hashing

Spatial Hashing [Lefebvre06] is a non tree-based technique for spatial subdivision. It requires uniform subdivision of the VE into unit cells; varied-size cells could be used, but this would complicate the algorithm considerably and may sacrifice performance. The essential notion in spatial hashing is to define a hash function which identifies the cells each object occupies, ensuring that potentially-colliding objects are hashed to the same cells for further consideration. The benefit of spatial hashing lies in its *O(1)* performance to place an object in its respective cells. However, as previously mentioned, this approach does not easily permit variable-size cells; this may affect memory efficiency. Regardless of this minor issue, provided a computationally inexpensive, accurate hashing function can be defined, spatial hashing will be capable of placing n objects in their respective sub-regions in $O(n)$ time. Following this, narrow-phase collision detection can be performed by comparing objects sharing the same subspace.

2.3.2.1.4.3.1 Parallelism in Spatial Hashing

Spatial hashing offers good opportunities for parallelism. While the process of object insertion cannot be parallelised, as it is a single operation, narrow-phase collision detection performed on the objects in each sub-region can be parallelised. This approach allows an arbitrary number of processors to be used evenly because each processor can be allocated an appropriate set of sub-regions, which can be adjusted to achieve load balancing. Unfortunately, this approach is best-suited to static subdivision. While it is theoretically possible to dynamically resize sub-regions, this may overcomplicate the hashing function, reducing performance. In addition, the hashing function may be complicated further with the use of non axially-aligned partitioning planes; this may restrict the flexibility of the subdivision approach.

2.3.2.1.4.4 Multi-resolution Spatial Hierarchies

Spatial subdivision encounters problems with virtual worlds in which the size of objects are significantly different. In this situation, it is not possible to pick an appropriate sub-region size; either the sub-region will be too large for an object, possibly causing many unnecessary comparisons, or it will be too small for an object, causing the object to occur in a large number of sub-regions. To overcome this, multi-resolution spatial hierarchies have been proposed. Multi-resolution spatial hierarchies subdivide the virtual world into a hierarchy of sub-regions, each at different resolutions (dimensions), relative to the dimensions of the objects they will contain. For example, given a VE with two classes of objects, a tank and a human, the virtual world would be subdivided using two subdivision resolutions: one appropriate for human-sized objects (in the order of meters); one suitable for tank-sized objects (in the order of tens of meters). Each object is classified in such a way that they are inserted into an appropriate-resolution sub-region. This approach can be easily integrated into spatial-hierarchy approaches, e.g. Oct/BSP trees, without the need additional storage space. However, this approach requires additional storage space if integrated into Spatial Hashing.

2.3.2.2 Broad Phase Collision Detection Summary

Collision detection is often subdivided into two phases, termed broad phase and narrow phase collision detection. Broad phase collision detection is required to reject pairs of objects which cannot possibly be colliding using computationally-inexpensive operations. These approaches are relatively inaccurate but are conservative in that broad phase collision detection must never reject pairs of objects which are intersecting. A number of broad phase collision detection algorithms and approaches were presented:

- Bounding volumes
    - AABB

    - OBB
    - Sphere
    - K-Dop
- Sweep-and-prune
- Spatial subdivision
    - Quad-trees/Oct-trees
    - BSP trees
    - Spatial Hashing

These algorithms were evaluated based on their performance characteristics and their ability to be parallelised. From this evaluation, it was determined that spatial-partitioning approaches offered the best opportunities for parallelisation as these approaches allow for parallelisation across arbitrary numbers of processors. Spatial partitioning subdivides the virtual world into cells. Objects are inserted into the cells which they occupy in the VE and detailed collision detection must only be performed on objects which share the same cell. There are a number of spatial-partitioning approaches, including BSP trees, Oct-trees/Quad-trees and spatial hashing. The cost of inserting objects into the tree structures is $O(lgn)$, whereas the cost of inserting an object using spatial hashing is $O(1)$. All spatial-partitioning approaches offer good parallelisation opportunities. However, BSP trees and spatial hashing make exploiting arbitrary numbers of processors easier than Oct-trees/Quad-trees. Spatial hashing requires that the virtual world be subdivided into uniform-sized cells; the use of non uniform-sized cells adds additional complexity and may compromise performance. BSP trees, however, implicitly allow the use of arbitrary-sized sub-spaces, which can be adjusted to reflect the distribution of objects in the virtual world. Due to these properties, it is assessed that axially-aligned BSP trees are best-suited for the distributed collision detection approach (see ***Fig*** 2.13).

2.3.3 Narrow Phase Collision Detection

Narrow phase collision detection is the second phase of two phase collision detection. It takes, as its input, a pair of objects which survived broad phase collision detection and is responsible for determining whether and how the objects collide with one-another. Whereas broad phase collision detection is required to reject pairs of objects from further consideration in a computationally-inexpensive way, narrow phase collision detection is required to determine which, if any, components of a pair of objects collide.

2.3.3.1 Review of Narrow-Phase Collision Detection Algorithms

Narrow phase collision detection is required to detect whether a pair of objects are intersecting to some degree of accuracy. It is also required in many applications to provide additional information, such as points of contact, penetration depths and contact normals. Objects in current DVEs are commonly constructed from a set of primitive objects, e.g.:

- Lines
- Splines
- Triangles
- Rectangles
- Spheres
- Boxes
- Cylinders
- Cones
- Tetrahedra
- Sphere-swept lines
- Sphere-swept rectangles
- Patches

Narrow-phase collision detection can therefore be separated into two parts: enumerating potentially intersecting primitives and intersection tests between a given pair of primitives.

Given two objects constructed from n and p primitive shapes respectively, a naïve narrow-phase intersection test would require np primitive intersection tests. However, the number of primitives which are actually intersecting, in most simulations, will be considerably smaller than this. Therefore, in order to efficiently perform collision detection between objects constructed from a set of primitives, it is necessary to efficiently reject pairs of primitives which are not intersecting. A number of approaches have been proposed to achieve this. The most popular of these approaches are bounding volume hierarchies [Bergen04] [Ericcson05][Gottschalk][Watt01] and local-space BSP/Oct trees [Watt01] [Fuchs].

Bounding volume hierarchies are a hierarchical approximation of the object being simulated. The primitives which the objects are constructed from are recursively subdivided into smaller sets of primitives until some termination criteria is met, e.g. maximum tree depth or the number of primitives in a set falls below some threshold. A bounding volume is constructed each level of the subdivision to enclose the sub-set of primitives. Therefore, at each subsequent level of subdivision, a bounding volume hierarchy forms a more accurate approximation to the shape of the object it encloses. Bounding volumes can be constructed as a pre-processing step prior to simulation and offer fast, average-case $O(logn)$ identification of intersecting primitives.

Local-space Binary Space Partitioning (BSP) trees recursively subdivide the primitives an object is constructed from using arbitrarily-oriented partitioning planes. At each level of subdivision, a partitioning plane is selected which roughly separates the primitives into two equal sets until the termination criteria is met. BSP trees will commonly result in more evenly-distributed trees than

bounding volume hierarchies, resulting in better average-case performance. Similar to bounding volume hierarchies, BSP trees can also be constructed as a pre-processing step and offer *O(logn)* average performance.

The primitives from which the objects are constructed can be tested for intersection using a number of techniques. These techniques can be roughly classified into two distinct types: explicit intersection tests and implicit intersection tests.

Explicit intersection tests must be specifically implemented to detect intersections between a given pair of object types, e.g. triangle-triangle, triangle-rectangle, triangle-box etc. This approach offers efficient and accurate collision detection. However, the use of such an approach in modern DVEs requires a large amount of code to implement specific intersection tests for the wide range of primitives which objects may be constructed from. This additional work-load may deter developers from adopting this approach if the number of primitives they wish to support is large.

A number of implicit intersection approaches have been proposed. The most popular of these are the Lin-Canny closest feature tracking algorithm [Lin] and the Gilbert-Johnson-Keerthi (GJK) distance computation algorithm [Bergen04]. Both of these algorithms operate on convex shapes and do not provide support for concave shapes; concave shape support can be emulated by subdividing the concave shapes into a series of convex shapes. Both of these algorithms, by default, provide the closest features between a pair of convex shapes and the distance; distances of zero indicate that the objects are intersecting. However, these approaches have been adapted in the literature [Bergen04][Watt01][Ericcson05][Lin] to provide additional information, such as points of contact, penetration depths and contact normals.

2.3.3.1 Parallel/Distributed Execution of Narrow-Phase Collision Detection

A number of narrow-phase collision detection algorithms offer some opportunities for parallel execution, e.g. bounding volume hierarchies. However, real-time narrow-phase collision detection between a pair of objects should be completed within a very small quantum of time, in the order of microseconds. Therefore, this offers very fine-grained parallelism. As such, the use of multiple processes/threads within narrow-phase collision detection between a pair of objects may detrimentally affect performance. Therefore, the best performance improvements through the use of parallel/distributed execution in collision detection can be achieved by executing broad-phase collision detection in parallel. The effectively executes narrow-phase collision detection in parallel because narrow-phase collision detection between different pairs of objects may be executed in parallel.

2.3.4 Summary

The following section will summarise the collision detection algorithms discussed in the previous section and describe the role of collision detection in DVEs. Following this, a discussion will be provided onto the requirements of a collision detection algorithm suitable for DVEs.

A number of collision detection algorithms were described in the previous sections. The notion of two-phase collision detection was introduced: broad-phase and narrow-phase collision detection. Broad-phase collision detection is required to cull away pairs of objects which cannot possibly be colliding in as computationally-inexpensive way as possible. Narrow-phase collision detection is performed on the pairs of objects which are not culled by the broad-phase; this more accurate collision detection phase determines, to the desired level of detail,

whether and how a pair of objects is colliding. Broad-phase and narrow-phase collision detection occur in separate phases and, as such, it is usual that any combination of broad-phase and narrow-phase collision detection can be used.

The collision detection algorithms presented in the previous section were categorised in terms of their general performance, memory requirements and their suitability for executing in parallel. The algorithms' suitability for parallel execution was assessed to reflect both the move towards multi-processing core architectures and the availability of large numbers of distributed processors in DVEs, which will be discussed in more detail later in this section.

From the analysis of parallelising broad phase collision detection, it was found that spatial subdivision approaches offered the most beneficial opportunities for parallel execution. While other algorithms such as Sweep-and-prune offered some possibilities for parallel execution, these approaches required a number of join operations and limited, relatively fine-grained sections which can be executed in parallel. In addition, the level of parallel execution in these algorithms may be limited, e.g. the sweep-and-prune algorithm can support up to 3 parallel processors to sort 3 lists, but could not use additional processors if more than 3 were available. The spatial partitioning approach can support an arbitrary number of parallel processors up to the number of sub-regions the virtual world is divided into. However, binding a processor to each sub-region in the virtual world would result in too fine-grained parallel processing. In practice, a virtual world will usually be divided into several thousand sub-regions; it is unlikely that there will be sufficient processors available to lead to fine-grained parallel processing in virtual environments populated by large numbers of objects.

The use of a parallel implementation of spatial partitioning as a broad-phase collision detection approach implicitly leads to parallel narrow-phase collision detection without the need for a join operation between the broad-phase and

narrow-phase. Spatial partitioning places objects into their respective sub-regions of the virtual world; once the objects have been placed in a sub-region, collision detection can be performed on each sub-region independently. With the use of hierarchical data structures to represent the spatial subdivision, e.g. BSP tree or Oct-tree, the act of inserting the objects into the tree can also be parallelised.

It is possible for narrow-phase collision detection between a pair of objects to be performed in parallel with the use of bounding volume hierarchies or BSP trees. However, this can lead to very fine-grained parallel performance because it is common that only a small proportion of the trees are traversed before either it is determined that the objects do not intersect or the point of contact is found. It should be noted that if the entire tree was required to be traversed in order to satisfy a collision query, it is likely that brute-force collision detection would perform better than hierarchical collision detection due to the cost of traversing the tree in addition to the cost of performing intersection tests on all the polygons contained in the model. In practice, best performance is yielded by parallelising broad phase collision detection such that narrow-phase collision detection for a given pair of objects is executed entirely on one processor; this approach will distribute the processing overhead of performing collision detection between the pairs of potentially-colliding objects between the available processors.

2.3.5 Requirements of Collision Detection for DVEs

A Distributed Virtual Environment should be able to support a large number of simultaneous participants. These distributed participants should be able to experience a relatively consistent virtual world. While small degrees of inconsistency are acceptable in a DVE, inconsistencies which cause significant differences to be perceived between participants can detrimentally affect users'

immersion. One of the most noticeable and, therefore, significant form of inconsistency in DVEs stems from differences in responses to collisions between participants. This ranges from slight differences in response to collisions being responded to by one participant's machine but being missed by others. The levels of inconsistency become increasing worse as the number of participants and/or the average message transmission delay rise. While considerable research effort has been put into developing efficient collision detection techniques, relatively little research effort has been put into developing collision detection techniques suitable for alleviating inconsistencies in DVEs. In addition to the inherent inconsistencies, DVEs also provide a platform consisting of a large number of distributed processors. This can be exploited by executing collision detection in parallel to leverage improved performance and enable more complex DVEs inhabited by larger numbers of objects to be simulated in real-time.

2.4 Chapter Contributions

This chapter introduced Virtual Environments (VEs) and the challenges and considerations associated with their development. Following this, Distributed Virtual Environments (DVEs) were introduced as an extension to VE research which incorporates the challenges of building virtual environments with the issues of developing a distributed system. These additional complexities include:

- Message dissemination
- Scalability
- Consistency
- Responsiveness

Message dissemination is responsible for ensuring that messages are delivered to the appropriate recipients. This can be achieved using a number of network-

layer protocols and communication models, coupled with the use of application-dependent message filtering techniques.

Scalability is a property which described how the performance of an application is affected by an increase in work-load. In DVEs, scalability is affected by the number of users simultaneously participating in the DVE, the number of objects inhabiting the DVE and the volume of messages which must be transmitted; it is likely that there is a direct relationship between the number of participants and the number of objects in a DVE, but this relationship is not guaranteed. It is desirable to produce a scalable platform for DVEs to enable large numbers of users to interact simultaneously within a virtual world. A number of approaches have been developed to improve scalability in DVEs by reducing the volume of message transmission. However, to date little work has been done to improve the scalability of collision detection in DVEs.

Consistency is a property of a DVE which describes how similar each user's perceived view of the virtual world is. Consistency is affected by the frequency of state update message transmission and the message transmission delay. The frequency of state update messages affects how accurately the motion of a remote object is approximated on a user's machine, whereas the message transmission delay governs how long the delay is between an object's state changing and this change being realised on a given user's machine. Low-frequency state updates and high-latency message transmission delays cause high-levels of inconsistency. Techniques have been developed to conceal the effects of low-frequency message transmission, such as dead reckoning. However, these techniques attempt to predict the motion of the objects and can often result in increased inconsistency if the motion of the object cannot be predicted correctly. It should be noted that, while low-frequency state updates can contribute to inconsistency, increasing the frequency of state update messages will not remove inconsistency in the presence of large message transmission delays.

Responsiveness is a property of a VE which describes the delay between a participant instigating an event and the event occurring, e.g. the delay between a user pressing a button and the response to the button press being manifested. This property is usually dependent on the frame rate which can be achieved on the user's machine; higher frame-rates can be achieved by reducing the computational overhead of simulating the DVE, e.g. improving the efficiency of the VE engine. The responsiveness of a DVE depends not only on the computational overhead involved in simulating the DVE but also depends on the delay between a user instigating an event and a remote user perceiving this event; this is affected by the message transmission delay. Responsiveness is important to users' immersion as an unresponsive DVE can make a user feel as though they are not in control of their avatar. Humans will usually not notice delays less than 60ms, but delays greater than 300ms may significantly limit interaction. To counteract this, many commercial DVEs employ interaction techniques which conceal lacks of responsiveness. However, this usually compromises the levels of interaction users are permitted.

Collision detection was introduced. Collision detection is a highly computationally-expensive operation which must be performed at frequent intervals in all VEs which model the motion of solid objects. Collision detection is usually subdivided into two distinct phases: broad-phase and narrow-phase collision detection. Broad-phase collision detection is responsible for efficiently rejecting pairs of objects which are not colliding from further consideration. Narrow-phase collision detection operates upon the pairs of objects which the broad-phase cannot discard and is required to determine, to the desired level of detail, if and how the pair of objects collides. A number of collision detection approaches and algorithms were described and analysed in terms of performance and their suitability for executing in parallel. It is the opinion of the author that broad-phase collision detection is most suitable for parallelisation and that the

most suitable broad-phase collision detection algorithms for parallelisation are spatial subdivision approaches.

While collision detection has received a large amount of research effort, the algorithms which have been developed have been targeted towards efficient performance in single-user VEs. Until recently, the majority of collision detection algorithms were designed with the assumption that they would be executed in a single processing thread; with the recent move towards multiple processing cores in home computers and next-generation games consoles, many of the previously popular collision detection algorithms which are not capable of being parallelised must be replaced with algorithms capable of exploiting the performance offered by these new platforms. In addition to providing algorithms suitable for new platforms, research is being undertaken into exploiting hardware Graphics Processing Units (GPUs) to perform collision detection upon.

Current collision detection research trends do not address the problem of consistency in collision detection in DVEs. Current DVEs usually either adopt central-server architectures, whereby a server acts as an arbitrator to determine how a collision should be responded to, or peer architectures, where collision detection and response are performed by all machines participating in the DVE. The former approach provides consistency at the cost of throughput, whereby the central server not only imposes delays between an event being initiated and it being realised by the other machines, but also becomes a performance bottleneck and single point of failure. The latter allows events to occur near-instantaneously but can result in significant deviation in how events are perceived by users. This problem is termed the consistency-throughput trade-off and governs performance and consistency in most distributed applications. While DVE research has recognised this problem, little work has been undertaken to alleviate it.

2.5 Thesis Purpose

A number of highly-efficient collision detection algorithms have been proposed in the literature. However, while these algorithms can be efficiently implemented in single-user virtual environments, they provide no consideration for the problems and potential optimisations available within DVEs. The purpose of this thesis is to develop an accurate general-use collision detection algorithm which provides efficient performance in single-user VEs while also exploiting the characteristics of DVEs to facilitate the development of highly-complex, consistent distributed virtual worlds. The collision detection approach will adopt the server hierarchy network architecture and provide mechanisms to reduce the network bandwidth required to maintain the state of objects in the DVE, thereby providing improved scalability. It will exploit spatial partitioning to distribute the processing overheads associated with collision detection, improving responsiveness, scalability and consistency.

Chapter 3

Theory

3.1 Introduction

It is desirable to be able to develop scalable, consistent and responsive DVEs [Singhal99]. A scalable DVE will allow a large number of users and objects to interact with one-another simultaneously. A consistent DVE will result in very little variation in the perceived state of objects between different participants. A responsive DVE will manifest user interactions without any perceived latency between the user issuing a command and it being executed. It is an accepted fact in DVEs that these three requirements can not be achieved simultaneously and that; instead, it is necessary to sacrifice one or more of these properties to improve the third. For example, in order to achieve high-levels of scalability, it may be necessary to sacrifice consistency and responsiveness. In DVE research, this property has been termed the consistency-throughput trade-off [Singhal99][Bosser04]. This issue has been well-investigated in the field of distributed systems, in which a number of consistency protocols have been developed which sacrifice throughput speed in return for guaranteed consensus [Fischer83].

This chapter introduces the underlying theory and basic concepts required to distribute the processing overhead of collision detection across multiple addressable spaces. To achieve this, the problem will be approached iteratively, whereby a number of candidate solutions will be presented; each solution building on previous solutions' functionality, addressing new problems to reach a general model for distributed collision detection across heterogeneous nodes in an asynchronous, unreliable network.

The basic principle of this approach is to subdivide the collision detection problem domain into a set of sub-problems, each of which can be solved independent of one-another. This approach, termed divide-and-conquer, provides an extremely efficient mechanism for reducing the overhead of computationally-expensive operations. The use of divide-and-conquer algorithms is well-understood and commonly exploited in the field of collision detection through the use of spatial subdivision and bounding volume hierarchies [Ericcson05][Lefebvre06][Watt01][Bergen04].

The theory outlined in this chapter utilises spatial subdivision to uniquely map sets of objects to different addressable spaces. Initially, this approach is introduced in the domain of reliable, instantaneous communication between addressable spaces. Following this, a number of common problems related to distributed systems are addressed:

- Limited bandwidth in communication channels between addressable spaces
- Unreliable communication between addressable spaces
- Variable-latency communication between addressable spaces

The problems of both machine and network failures will be addressed. Throughout this chapter, as each iteration is introduced, the DVE model will be analysed in terms of scalability, consistency and responsiveness. With the use of the models presented in this chapter, it is possible to improve scalability.

consistency and responsiveness simultaneously, although the improvements depend largely on the geographical location and network properties of the users participating in the DVE. However, the final approach presented in this chapter offers a model for the distribution of collision detection in DVEs such that the scalability, consistency and responsiveness of a DVE developed using the model will never be worse than the models used in current DVEs and in most circumstances will be far superior.

3.2 Background Theory

3.2.1 Bounding Volumes and Spatial Subdivision

Bounding volumes were introduced in Chapter 2 and form an integral part of the system described in this thesis. In this section, bounding volumes will be formalised by considering a 3D world to be an infinitely fine-grained three-dimensional grid of points, termed a lattice. Using a lattice, a three-dimensional object can be defined by the set of points in the lattice it contains, i.e. the space it occupies in the 3D world.

Let L be the set of points in the world. Given a pair of objects, A and B, let P_A and P_B be the set of points which A and B occupy in L respectively. Therefore:

$\forall p_a \in P_A, \forall p_b \in P_B, p_a \neq p_b \Rightarrow A \; disjoint \; B$

Following this, given bounding volumes BV_A and BV_B, which encompass objects A and B respectively:

$\forall bv_a \in BV_A, \forall bv_b \in BV_B, bv_a \neq bv_b \Rightarrow \forall p_a \in P_A, \forall p_b \in P_B, p_a \neq p_b \Rightarrow A \; disjoint \; B$

Given that it is usually far less computationally expensive to perform collision detection between two bounding volumes than it is to perform collision detection between the objects themselves, bounding volumes offer an efficient mechanism to determine disjointedness between a pair of objects.

Using a similar approach to that of bounding volumes, spatial subdivision can be formalised using the lattice model, where each sub-region in the spatial hierarchy can be defined by the set of points in L it contains. From the spatial subdivision strategies outlined in Chapter 2, it can be seen that each sub-region is disjoint from all nodes except from those it contains (its descendants) or those which contain it (its ancestors). As such, if a pair of objects do not share a common terminal sub-region, i.e. they do not share a sub-region which has no descendants, they cannot be intersecting.

3.2.2 Broad Phase Collision Detection

Broad phase collision detection [Lin98][Watt01][Bergen04][Ericcson05] was introduced in Chapter 2 as the first stage of two phase collision detection. Broad phase collision detection is responsible for enumerating pairs of potentially-colliding objects in a computationally-inexpensive way. This form of collision detection is often inaccurate and can fail to reject pairs of objects which, upon further examination, are found not to be intersecting; it is valid for broad phase collision detection to fail to reject pairs of objects which are not intersecting, but it must never reject pairs of objects which are intersecting.

The distributed collision detection approach presented in this thesis subdivides the virtual world into discrete sub-regions. Initially, the virtual space is subdivided into fixed-size cells. Subdivision is performed recursively using axially-aligned partitioning planes along the X-, Y-, and Z-axes, subdividing along the largest axis at each stage of subdivision; subdivision is terminated

when sub-regions become smaller than some threshold value. Each sub-region maintains a list of objects occupying the space enclosed by the sub-region. Given this pre-computed set of regions, broad-phase collision detection places each object inhabiting the DVE into the sub-regions which its bounding volume occupies. An object may cross boundaries between sub-regions but, as ***Fig* 3.1** depicts, assuming the sub-regions are at least as large as the object's bounding volume, the number of sub-regions an object can occupy is at most 8. To clarify, a sub-region is at least as large as an object if the object's projection onto each coordinate axis is smaller than the sub-region's projection onto the same coordinate axis.

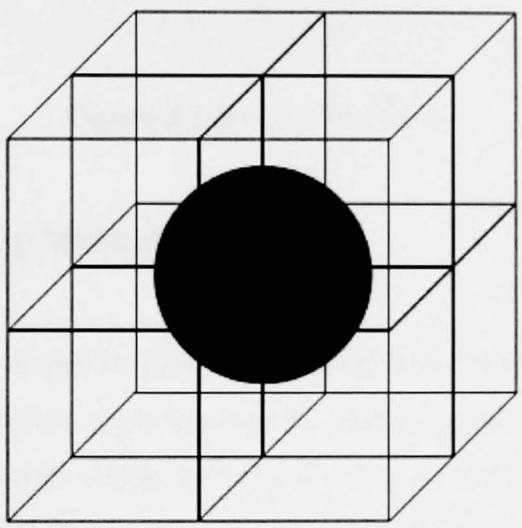

Figure 3.1 An object in 3D space occupying 8 regions

Once the objects have been placed into their sub-regions, each sub-region is iterated through and narrow phase collision detection is performed on each pair of objects found to occupy the same sub-region. As can be seen in ***Fig* 3.2**, the top-left region has two objects within it (*A* and *B*) which are colliding. As these two objects share the same region, they will not be rejected by the broad phase and progress to narrow phase collision detection. Additionally, objects *C* and *D* traverse boundaries between different regions and, as such, inhabit a common region. These objects will also pass into the narrow phase although, upon further

inspection, they will be found not to be colliding. Finally, objects *E* and *I* share the same regions and will be compared with one-another and found to be colliding.

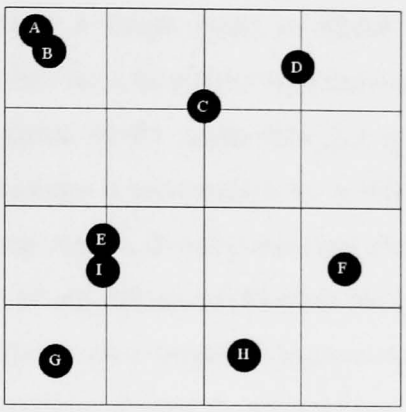

Figure 3.2 Spatial Subdivision

3.2.2.1 Occupying Multiple Regions

In ***Fig* 3.2**, Objects *E* and *I*'s bounding volumes both traverse the same region boundaries and, therefore, share two regions. This situation could lead to narrow phase collision detection being performed twice on these objects; as narrow phase collision detection can be computationally expensive, this is extremely undesirable. Therefore, an additional piece of information is used to determine in which region the potential collision occurs. The mid-point between the two objects is determined and narrow phase collision detection is only performed between objects in the region which contains the mid-point between the two objects.

To clarify, a region is a space enclosed by an axially-aligned bounding box (ABBB), which is denoted by its minimum and maximum extremes along the coordinate axes, *Min* and *Max*. The mid-point between two objects can be obtained trivially from their transformation matrices by taking the average of the

two vectors which represent the objects' respective positions in world space. This assumes that the local/model space origin of the objects is located somewhere in the object itself, which is standard in computer graphics; if this was not the case, additional computation may be required. The mid-point between the two objects is a single point in world space. It can, however, represent a space which lies on a boundary between two regions. To overcome this, a region, r, is the space in the range $Min \leq r < Max$, along each of the coordinate axes. This provides a mechanism by which a point in world space corresponds to exactly one region, thereby ensuring that narrow phase collision detection between a pair of objects occurs at most once. This can be formalised using the lattice model. Given two neighbouring sub-regions, A and B, denoted by the set of points they enclose, P_A and P_B respectively:

$i \in P_A, k \in P_B, i \neq k$

This holds providing each region, r, is the space in the range $Min \leq r < Max$, i.e. every point in space is contained by at most one region of the virtual world. It follows that any point V within the boundaries of the world must be contained within at most one sub-region. Therefore, the centre point between two objects can be used to ensure that collision detection between a pair of objects is performed at most once.

3.2.3 Narrow Phase Collision Detection

Once the broad phase has been completed and all objects are placed in their corresponding sub-regions, narrow phase collision detection is performed by iterating through each sub-region and performing more detailed collision detection between the objects which occupy the same region. As mentioned previously, to avoid situations where pairs of objects appear in more than one region, the detailed collision detection is only performed if the mid-point between the objects occurs in the region. If the mid-point is not within the region, then the pair of objects must occur in another region, which contains the

mid-point, and therefore detailed collision detection should be performed elsewhere.

Given a DVE with n objects, a brute force approach to collision detection would require $\frac{n(n-1)}{2}$ object/object comparisons. However, if the world is subdivided into r regions, it will require, on average, $r\left(\frac{p(p-1)}{2}\right)$ object/object comparisons, where $p = n/r$. For example, if n = 100 and r = 10, brute force collision detection would require 4950 comparisons, whereas collision detection utilising spatial subdivision would require, on average, only 450 object/object comparisons. This occurs when objects are randomly distributed throughout regions in the DVE. This distribution of objects may not be a realistic assumption as, in many DVEs, users often group together in the virtual world. Regardless of object distribution, region-based collision detection approaches will never require more object/object comparisons than a brute force approach. However, collision detection utilising spatial subdivision incurs an additional overhead: that of placing the objects into the appropriate regions in the DVE. Techniques to do this will be discussed later in this chapter, which will include a brute force approach, a tree based approach and a spatial hashing approach.

There is inherent temporal coherence in most virtual worlds, as animation is performed at frequent intervals between which the objects generally do not travel a great distance. This allows an optimisation to be performed on spatial subdivision-based collision detection; region sizes can be grown and shrunk, by dynamically adding and removing partitioning planes, based on object population density in each sub-region of the virtual world. For example, if two neighbouring regions contain less than a threshold number of objects, the partitioning plane separating the two regions can be removed, thus merging the two regions into one. Similarly, if a region contains more than a threshold number of objects, a partitioning plane can be introduced which subdivides the

region into two. This approach helps to optimise the broad phase collision detection scheme by merging sparsely-populated neighbouring regions and subdividing densely populated regions. To avoid constantly subdividing and merging the same sub-regions, different threshold values should be used for the merging and subdivision criteria, e.g. merge a pair of sub-regions if the number of objects in two neighbouring sub-regions is less than 4 and subdivide if the number of objects in a sub-region is greater than 6.

3.3 Distributed Collision Detection

The collision detection technique described previously provides efficient broad phase collision detection. However, the performance of the algorithm can be significantly improved with the exploitation of parallel/distributed execution, in which collision detection for different regions may be performed in different addressable spaces. The parallel variant of this algorithm can be used to exploit the recent adoption of multi-processor architectures in home computers and next-generation games consoles. This will allow increasingly complex virtual environments, in terms of number of objects and complexity of object models and interaction, to be simulated in real-time on current and next-generation architectures. In addition to improved performance, a distributed implementation of this algorithm provides the foundations of a scalable, responsive DVE with improved consistency. The distributed execution model will be explored in additional detail in the following sections while the collision detection model is refined to reflect the problems of reliability and latency.

3.3.1 Glossary of Terminology

During the remainder of this chapter, a number of terms will be used extensively:

- World/Virtual World/Environment
    - A virtual space/place in which a simulation occurs
    - Depending on the context, world/environment can also be used interchangeably to describe the geometry of the environment from which the virtual world is constructed, e.g. hills, floors, walls, ceilings etc.
- Object
    - Any virtual entity which projects a physical presence into the virtual world which can be interacted with, e.g. chair, table, character, vehicle etc.
- Avatar
    - A user-controlled object inhabiting a virtual world.
- Object state
    - Information required to replicate the state of an object in a virtual world. This may include, but is not limited to, position, orientation, velocity, acceleration and animation state.
- Event
    - Any form of interaction between objects and between objects and the environment
- Inconsistency
    - Any situation in which there is a significant discrepancy in how users perceive an event. This ranges from relatively minor differences, e.g. slightly different contact normal or point of contact, to major differences, e.g. collision perceived by one user but missed by another.
- Synchronisation
    - The simulation is synchronised between collision detection nodes if all collision detection nodes must complete their collision detection before the simulation time can be advanced

3.3.2 Simplified Distributed Collision Detection Approach

In this simplified model of the distributed collision detection approach, the following assumptions can be made:

- All nodes participating in the DVE have complete knowledge of all the objects' current state
- There are no inconsistencies in object states between nodes as the result of contradicting responses to collisions (an example of such a situation will be provided later).
- Message delivery is reliable and instantaneous
- All machinery and software components are completely reliable

Given a DVE consisting of n objects and a world subdivided into r regions, being hosted on d nodes, each node would be required to perform, on average, $\frac{rp(p-1)}{2d}$ object/object comparisons, where $p = \frac{n}{r}$. If n = 100, r = 10 and d = 10, this would require each node to perform, on average, 45 object/object comparisons.

The results of the collision detection must be disseminated to all other nodes participating in the DVE. This could be achieved in two ways:

- Transmit the collision events to all nodes (the two objects, the point of contact, the contact normal, and velocity and acceleration information etc.)
- Transmit the result of the collision event (the new positions of the two objects involved in the collision).

From the discussion in Chapter 2, it was shown that the number of collisions that occur in a virtual world is relatively small in most situations; implying that

either dissemination approach is valid. However, in a virtual world in which every object is colliding with one-another, the first approach would require $\frac{n(n-1)}{2}$ collision events to be reported. However, if the latter approach is adopted, and all responses to collisions are calculated before being reported, then at most only n messages must be transmitted: one new transformation per object. However, as it is rare for every object to be involved in a collision, it is only necessary to transmit the new transformations of objects which were involved in collisions.

This approach of transmitting the results of collisions, and not actual collision events, potentially reduces the bandwidth requirements of this approach immensely. However, it introduces an issue:

- Node n_1 reports a response to a collision involving objects o_i and o_k
- Node n_2 reports a response to a collision involving objects o_i and o_j

The collision reported by n_1 may result in a different position for the object o_i than the collision reported by n_2. This could result in inconsistent states being reached in the DVE. If collision detection was performed sequentially on a single computer in this situation, the response to whichever collision was detected earlier would be passed into the latter collision detection, i.e. any response to a collision between o_i and o_j would be passed into the collision detection between o_i and o_k. This may result in the latter collision not occurring at all or, if the collision was still detected, some alternative response than would be proposed by nodes n_1 or n_2. Potential solutions to this problem will be proposed later, once the model has been further refined.

3.3.3 Partial Knowledge of DVE State

The following assumptions are made in this refinement of the distributed collision detection approach:

- All nodes participating in the DVE have complete knowledge of the current state of the objects which they are responsible for collision detection on
- There are no inconsistencies in object states between nodes as a result of contradicting responses to collisions
- Message delivery is reliable and instantaneous
- All machines and software components are completely reliable

In the previous model, each node participating in the DVE was assumed to initially have complete knowledge of all the objects' current state. However, this is not always necessary. It is, in fact, only necessary for a node to have knowledge of the objects within its sub-region(s). As such, the following refinement of the collision detection model assumes that a node has absolute knowledge of the objects within its sub-region. However, each node is not required to have knowledge of any objects in regions other than those the node is responsible for determining collision detection within.

It can be seen that this model produces the same collision detection results as the previous model. However, the response to a collision may "*push*" an object from its current sub-region, hosted on node n_i to another region, hosted by a different node. In this case, it is necessary for n_i to inform the node responsible for the sub-region the object has moved into that it is now responsible for collision detection for that object. There are two ways in which this kind of event can occur:

- The object moves partly into a new sub-region, but partly remains in its original region (it straddles a region boundary). In this case, the object

must be *replicated* in two or more nodes/regions. Given that a region is larger than an object, an object can exist in at most 8 regions in a 3 dimensional world.

- The object moves completely into a new sub-region. It no longer occupies its original sub-region. In this case, the object must be *transferred* from the old node to the new node.

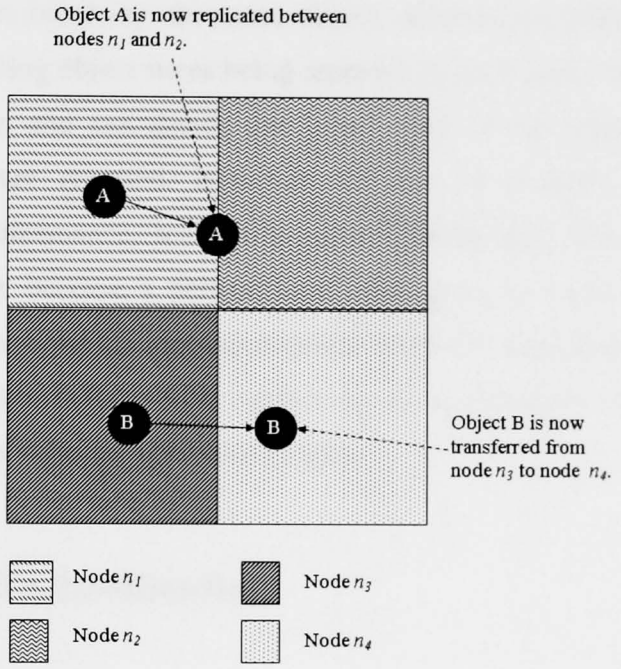

Figure 3.3: Object Transfer and Replication

In order for replication and transfer operations to take place, a mechanism must be employed which will inform the respective nodes as and when they become responsible for collision detection for a given object. This could be achieved by utilising some entity responsible for retaining the current state of all objects in the DVE, such as a server. This entity could act as an oracle by informing each node which objects they are responsible for performing collision detection upon prior to performing each collision detection iteration. Alternatively, when an object, o_n, a node, n_1, is responsible for traverses into another node's, n_2's, sub-region, n_1 could transmit a message directly to n_2 instructing the node to perform

collision detection on o_n. This requires each node to have prior knowledge of the other nodes in the DVE. However, assuming that the distance an object can travel between each collision detection iteration is smaller than the size of a region, it may only be necessary for a node to retain information about the nodes responsible for its neighbouring sub-regions. From a conceptual perspective, either solution would be acceptable. The presence of an entity which retains an absolute "*view*" of the objects' current state potentially solves the problem in which collisions involving the same object, detected on different nodes, can result in conflicting object states being reported by each node. The entity can act as an arbitrator, and can decide the "*true*" state of the object if conflicting current states are reported. However, the use of a single arbitrator may compromise performance, scalability and reliability as it is a single point of failure and may become a performance bottleneck. As such, communication between nodes may be the most appropriate means of object transfer/replication. However, this does not solve the conflicting object state problem, which will be addressed in more detail later in this chapter.

3.3.4 Object Classification

Before refining the model further, it is necessary to classify the different types of objects which can populate a DVE. The previous models presented assumed that all objects were equivalent. However, there are in general three different types of objects which can participate in a virtual world:

- Objects which do not move: static objects
- Objects which move only as a response to collisions (e.g. pushed by a force): physically-controlled objects
- Objects which move autonomously within the world

In most DVEs, the "environment", or "world model", is a static object. This implies that the walls, floors, pillars, hills, valleys, ceilings etc. constituting the environment can not be moved, no matter how much force is exerted upon them; this assumption is valid in current VEs. However, objects populating the VE such as doors, chairs, tables, boxes etc. may move when a large enough force is exerted upon them. The final classification of objects, those which move autonomously in the world, can further be subdivided into two distinct classes within traditional DVEs: objects which are controlled by external users and objects which are controlled within the DVE software.

Objects which are controlled by external users, commonly termed *avatars* [Greenhalgh][Singhal99], allow users to interact with the objects populating the DVE. Autonomous objects are generally controlled by artificial intelligence algorithms within the DVE software: a mixture of pre-computed responses, application/environment-specific rules and pseudo-random decisions, which may or may not be adjusted as a result of the behaviour exhibited by avatars in the DVE [Watt01]. The behaviour of system-controlled autonomous objects should be deterministic; it can be replicated across all nodes in the DVE, by replicating the parameters and seed values used by the AI routine. However, the behaviour exhibited by avatars is not deterministic, as it is the result of user input. This distinction between avatars and other types of objects is important when determining how the different classes of objects should be disseminated to the nodes in the DVE.

Given these object classifications, it is possible to make the following assumptions:

- All static objects will remain in the same regions throughout the lifetime of the DVE
- All other non-autonomous objects will remain in the same regions unless a force is exerted upon them which moves them to a new region

These assumptions imply that it is possible to position non-autonomous objects into their respective regions and, provided they are not moved as a result of a collision, the objects will remain in the same region indefinitely. As a result, it is not necessary to inform a node of the position of its non-autonomous objects between every collision detection iteration. Instead, it is possible for a node to retain the current state of its non-autonomous objects until the object is no longer within its region, at which point it *transfers* the object to the nodes which are now responsible for it. If the object straddles a region boundary, the nodes which are responsible for the object must ensure that the object is *replicated* consistently by transmitting current state information about the object to the nodes whose regions the object intersects. However, in order to ensure consistency with autonomous objects, especially avatars, the technique as it has been described so far will not suffice.

3.3.5 Distributed Collision Detection Architecture

This section introduces the core architecture of the distributed collision detection approach and introduces the software components which must interact with one-another to maintain a DVE. Following the description of these components, the mechanisms by which these components communicate with one-another are introduced. The following assumptions are made:

- All nodes have knowledge of the current state of their persistent objects
- Message transmission is reliable and instantaneous
- All software components and machinery are completely reliable.

The architecture can be subdivided into three logical components: servers, nodes and clients. A server is a machine at a fixed, known address which acts as a central repository and directory service to assist nodes in joining a DVE and communicating with one-another. The server can also act as an arbitrator if

disagreement over an object's state arises. A node is an entity which is responsible for performing collision detection in a given set of sub-regions in the DVE. A client is a process by which the user interacts with the DVE. It can introduce one or more objects (including avatars) into the DVE, monitors user input, translates user input into state updates and injects these state updates into the collision detection system. A diagram of the system architecture is provided in ***Fig*** 3.4.

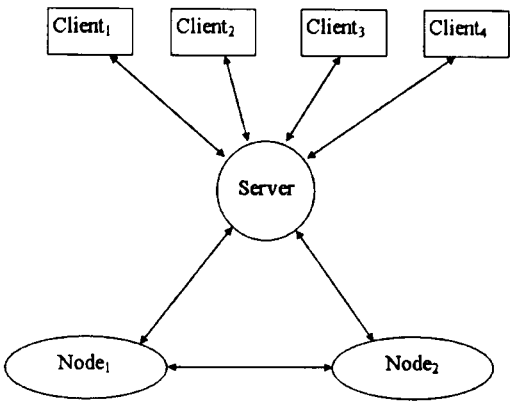

Figure 3.4: System Architecture

Upon initialisation, the server creates a virtual world, containing the objects initially populating it: static objects, physically-controlled objects and computer-controlled objects. When a new node joins the DVE, it is assigned a sub-region of the world. This causes the nodes currently participating in the DVE to be reallocated a new, potentially smaller, sub-region of the DVE. For example, when the first node joins, it is allocated a sub-region corresponding to the entire virtual world. However, when a second node joins, each node is assigned a sub-region corresponding with half of the virtual world. When each node is allocated a section of the DVE, they are informed of the current state of every object occupying that region by the server. Once this communication with the server is completed, the nodes manage the current state of their objects and transmit object transfers and replication messages to the other nodes in the DVE as and when they are required.

The objects a node is assigned by the server are termed *persistent* objects because the node manages the state of these objects until it detects that one of its persistent objects is no longer within its assigned sub-region of the DVE. The nodes report the responses to any collisions they detect to the server, allowing the server to act as a central repository for object states. When a client joins a DVE, it introduces its objects into the virtual world and periodically transmits its objects' state update messages to the server.

The server must transmit these state updates to the relevant nodes, i.e. the nodes whose sub-regions the objects intersect. However, these objects, termed *client* objects, are not handled the same way as persistent objects. As the behaviour of client objects cannot be predicted algorithmically, the nodes can not accurately predict the current state of client objects between state update messages. Instead, the server is responsible for transmitting the current state of client objects to the relevant nodes before collision detection is performed. This allows the server to act as a global timer for the DVE, ensuring that the nodes remain relatively synchronised with one-another. The server can monitor when each node completes collision detection and take additional metrics, such as the number of collisions each node reports, to monitor and tune performance accordingly. The server can also detect any conflicting collision responses and resolve them accordingly by correcting the object state in the relevant nodes. The results of collision detection/response are disseminated to the clients. This can be achieved with minimal bandwidth overhead by using techniques such as interest management. However, this model assumes that the server is a *reliable* entity, i.e. a failure in the server will cause the DVE to terminate.

3.3.6 Communication Latency

The previous model adopts the classic client/server request-response model which is widely used in distributed systems; the server and collision detection nodes are responsible for satisfying the collision detection requests made by the client when it transmits an object state updates. The model assumes that a group of reliable machines are available to perform collision detection upon which exhibit instantaneous, or negligible, message transmission delays; this assumption is reasonably valid if a dedicated cluster of high-performance machines is available. However, due to the associated cost, this may not be an appropriate assumption to make.

The next refinement to the model combines the notion of clients and nodes, such that a client and a node exist on the same physical machine; however, they are still considered to be separate components. This implies that some collision detection will be performed on the machines of each user participating in the DVE. This change does not rule out the use of dedicated collision detection nodes without users; such machines will behave in the same manner but will not introduce avatars into the DVE

An important challenge in DVE research is to be able to support a wide range of participants over the Internet, who may reside in different geographic regions. With the presence of large distances between participants, it cannot be guaranteed that the network transmission latencies between machines will be low-latency. In this situation, the previous model may not be sufficient to guarantee high-levels of responsiveness as message transmission may become a bottleneck. The distributed collision detection model will be refined to sacrifice some level of consistency in order to improve responsiveness in DVEs which have participants exhibiting high message transmission delays.

3.3.7 Consistency Groups

The following distributed collision detection architecture makes the following assumptions:

- All collision detection nodes have knowledge of the current state of the objects within their respective sub-regions
- All machinery and software components are completely reliable
- Message transmission delays are present
- Message transmission delays between a pair of machines remain constant

The previous model, as depicted in ***Fig*** 3.4, results in all nodes participating in the DVE being responsible for collision detection in a unique portion of the virtual world. However, in order to deal with network transmission delays, it is necessary to generalise this premise. A metric is introduced, which estimates the network transmission delay between a pair of nodes. This metric can be used to estimate which nodes can communicate with one-another sufficiently quickly to ensure a responsive virtual world. Although transmission delay is highly variable, depending on a number of factors, including network load, this model assumes that message transmission delays are relatively uniform. This will later be refined to deal with variable message transmission delays to adapt to changes in network behaviour in run-time.

Given a set of nodes $\{n_1, n_2, \ldots, n_i\}$ and a threshold value T, each node will need to retain transmission delay information between itself and the other i-1 nodes. From this information, it is possible to construct a matrix of transmission delays. Transmission delays observed between nodes j and k are recorded in two places in the matrix, in indices n_{jk} and n_{kj} respectively. This is because the transmission delay observed from node n_j to n_k may be different from that observed from node n_k to n_j. If both of these values are below T, then it is assumed that these

two nodes can communicate with one-another with a small-enough transmission delay to be able to maintain a responsive DVE. These two nodes can therefore be placed in a Consistency Group (***Fig*** 3.5) together. A consistency group is a group of nodes whose message transmission delay is sufficiently small that they can each maintain a highly-consistent view of the same DVE. A consistency group can contain a set of nodes if every node within the group has observed transmission delays between themselves and every other node in the group less than T. In addition, a node can be a member of only one consistency group.

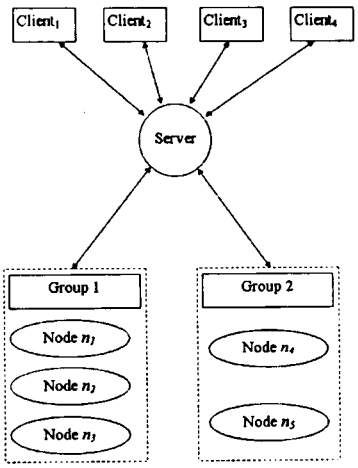

Figure 3.5 Consistency Groups

It is desirable to produce as large consistency groups as possible, as the more members in the group, the less collision detection each member must perform and the more consistent the virtual world will be. In order to produce large consistency groups, the set of all potential consistency groups is constructed (a power set), sorted from the largest group to the smallest. The largest group is initially selected to form the first consistency group. This group is removed from the power set. However, there may be a number of nodes not yet allocated to a group. If there still exists nodes which are not part of a consistency group:

- The nodes in the newly-allocated consistency group are removed from the remaining potential groups in the power set

- The power set is re-sorted so that the largest set appears first. In addition, any empty sets are removed from the power set within the sort algorithm.
- The largest potential consistency group in the power set is selected as the next group and removed from the power set.
- If there are still unallocated nodes, repeat the process until all nodes are allocated.

Once this algorithm is completed, every node will be a member of a consistency group. This algorithm creates the largest consistency groups it can. However, this may not result in the best performance for each node in the DVE. For example, given a DVE with 4 nodes, n_1, n_2, n_3 and n_4, and a threshold time of 5ms. The observed transmission delay times were:

$n_{1,2} = 3, n_{1,3} = 4, n_{1,4} = 10$
$n_{2,1} = 2, n_{2,3} = 1, n_{2,4} = 6$
$n_{3,1} = 4, n_{3,2} = 3, n_{3,4} = 2$
$n_{4,1} = 7, n_{4,2} = 4, n_{4,3} = 1$

Given these transmission delays, the resulting consistency groups using the described algorithm would be $\{\{n_1, n_2, n_3\}, \{n_4\}\}$. However, it would be beneficial for the performance of n_4 for the consistency groups to be $\{\{n_1, n_2\}, \{n_3, n_4\}\}$. In addition, due to the binary spatial subdivision approach, in a group of 3 nodes, one node will be responsible for collision detection on half of the world, while the other two nodes will be responsible for a quarter of the world each. The collision detection performance will be bound by the most heavily-loaded node, so it is beneficial to have groups whose sizes are a power of 2. This means that the latter set of consistency groups, with two groups of size two, would be more desirable than a group of size 3 and a group of size 1. However, this problem is similar to the knapsack problem, an NP-Complete problem. As such, it is not possible to determine an optimal solution in real-time as the

number of nodes becomes large and, therefore, a greedy algorithm to find an approximation to the solution is used.

3.3.8 Group Leaders

The refinement to the distributed collision detection model described in the previous section can result in potentially large numbers of consistency groups simultaneously performing collision detection. Recall that the server acted to provide synchronisation in the previous models, dispatching collision detection on all the nodes and ensuring consistency in object positions. The notion of consistency groups indicates that the server would be responsible for synchronising a large number of groups and maintaining consistency in each group. However, the transmission delays observed between the server and each collision detection node can be large. This can result in the delay in receipt of synchronisation messages from the server compromising responsiveness. The new model prescribes that each consistency group performs collision detection and response for all objects in the DVE. In the previous models, these results would be reported directly to the server by each collision detection node, which could place an additional strain on the server potentially causing a performance bottleneck. In order to overcome these performance bottlenecks, the notion of group leaders is introduced.

A group leader is intended to relieve some of the processing overhead from the server. One node in each consistency group is appointed group leader. This node essentially takes on a number of the server's responsibilities for its consistency group. Once a group leader is appointed, all non-administrative messages pass through the group leader before being delivered to either the server or collision detection nodes. The server is responsible for disseminating messages between consistency groups as there is no mechanism for nodes in different consistency groups to communicate directly.

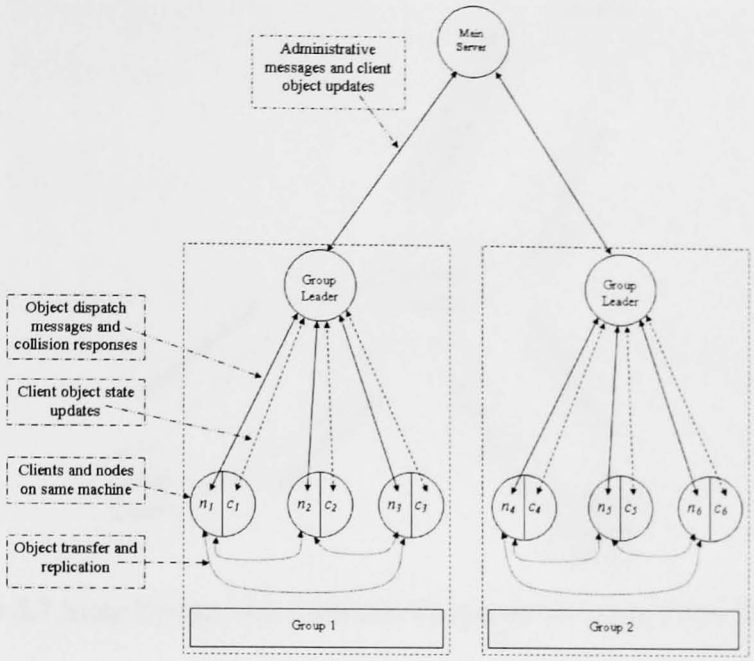

Figure 3.6 Consistency Groups with Group Leaders

The following section will describe the flow of messages between the server, group leaders, collision detection nodes and clients. Recall that clients are now physically located on the same machines as the collision detection nodes. Additionally, one of the collision detection nodes in each consistency group is appointed group leader. However, regardless of their physical location, group leaders, clients and collision detection nodes are regarded as logically separate components. In the following message flow descriptions, messages will be shown being transmitted between nodes, clients and collision detection nodes. Mechanisms are in place to bypass the process of forming and delivering messages to group leaders, collision detection nodes and clients if they are located on the same physical machine. However, to aid clarity, these mechanisms are overlooked in this chapter.

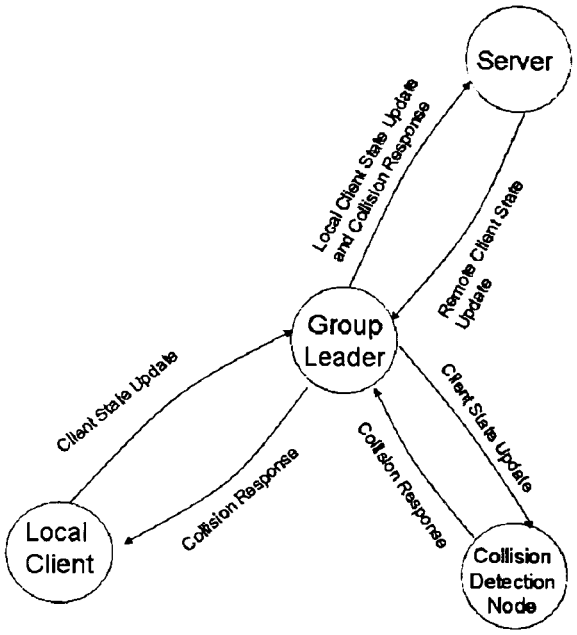

Figure 3.7 State Update and Collision Response Message Flow Diagram

The main change in the flow of messages from the previous distributed collision detection model is in the flow of client state update and collision response messages. In the previous models, client state updates were transmitted from the client to the server and then disseminated from the server to the collision detection nodes. Similarly, collision response messages were transmitted from the collision detection nodes to the server and then forwarded to the relevant clients. The responsibilities of the server, clients and collision detection nodes must change to reflect the presence of group leaders.

Clients

Clients may be hosted on machines hosting collision detection nodes. If this is the case, the client adopts the same group leader as the collision detection node it shares a machine with. The clients are responsible for:

- Transmitting client state updates to its group leader if it has one, otherwise transmitting state updates to the server

- Receiving collision response messages from its group leader if it has one, otherwise receiving collision response messages from the server

Collision Detection Nodes

Collision detection nodes are responsible for:

- Receiving client state updates from the group leader
- Transmitting collision response messages to the group leader

Group Leaders

The group leaders are responsible for:

- Receiving client state updates from its local clients
- Receiving client state updates from the server
- Transmitting client state updates to its group members
- Transmitting local client state updates to the server
- Receiving collision response messages from group members
- Transmitting collision response messages to local clients
- Transmitting collision response messages to server

Server

The server is responsible for:

- Receiving client state updates from clients without group leaders
- Receiving client state updates from group leaders
- Transmitting client state updates to group leaders
- Receiving collision response messages from group leaders
- Transmitting collision response messages to clients without group leaders

3.3.8.1 Message Dissemination

When a group leader receives an object state update message from a client, it forwards this update to the server, which disseminates this to the other consistency groups and stores the current state of the object in its own internal data structures. The group leader also updates its view of the object's state and dispatches collision detection on its group members, passing the current state of client objects (as viewed in its data repositories) to the relevant nodes, as the main server did in the previous model. When a group leader receives an object state update from the server, the same process is performed with the exception of transmitting the state update to the server.

The group leader monitors the performance of the collision detection nodes, recording which nodes have completed collision detection, and which nodes have not yet completed. The group leader buffers state updates until all nodes have completed collision detection. When a node completes collision detection, it reports the results of collision detection and response to its group leader. When all nodes have completed collision detection, the group leader dispatches the next collision detection iteration, provided state updates have been received from clients since collision detection was last dispatched. If this is not the case, collision detection is not dispatched until the next state update is received. This avoids the nodes repeatedly performing collision detection on the same data set.

The results of collision detection may cause client objects' current state to change. If the group leader receives such a state update, it must first determine if the client who owns this object is within its consistency group. If the client is within its consistency group, the group leader transmits this state update to the client. However, if the client is not within the consistency group, then this response is not disseminated to the client. It is assumed that the consistency group in which the client exists will be responsible for its update. If collision response results in a non-client object moving, the group leader updates its view

of the object's state, and passes this state on to the main server. Before it passes this state on, the group leader can perform some consistency checks to ensure that there are not any discrepancies between the states reported by each of its group members. If the group leader detects discrepancies in object states between its group members, it can instruct its members to change the state of the object to reflect what it views as being the object's true state; the true state of an object can be chosen arbitrarily by the group leader, although approaches such as taking a weighted average of object states is likely to perform well. When the server receives a state update from a group leader, the server stores this new object state and, if it finds it to differ from the state reported by the other consistency groups, the server can determine the true position and correct each group. It should be noted that minor discrepancies are to be expected between object states and that correcting an object's state should only be performed when the deviation in reported states is larger than some threshold value. From a performance perspective, it is better to use a smaller threshold values for correcting discrepancies within a consistency group than the threshold used to correct discrepancies between consistency groups.

The notion of *local* and *remote* objects is introduced, where a local object is an object hosted on a client in a given consistency group and a remote object is an object hosted on a client not in a given consistency group. The state of a local object within a consistency group is as recent as possible, in that it is highly probable that the current state of a local object as viewed by the group leader is current and accurate. However, this can not be guaranteed for remote objects. A remote object's state updates must be forwarded by the server, and may have suffered from substantial transmission delays since it was sent by a remote client. As nodes in different consistency groups have substantial transmission delays between them, possibly due to geographical distance (e.g. residing in different countries), any collisions detected involving remote objects are unlikely to be completely accurate. As such, it is not necessary, or desirable, to disseminate information about these collisions to the client which hosts this

object, as their objects' collisions are likely to have been detected with more accuracy within their consistency group. However, the result of collisions with remote objects are stored in the group leader and used for further collision detection until a state update is received for the remote objects, which will overwrite the current state. This mechanism ensures that within a consistency group, the events are perceived to be as consistent as possible. Interpolation, extrapolation and dead reckoning are used to smooth the transition from the old state to the new state, although these approximation techniques may introduce further inconsistencies.

Within a consistency group, the group leader acts as a central repository for the DVE state for that group. It may differ from the state observed in other groups but, as all members of the group have a low-latency network connection with the group leader, the state viewed on all nodes in the group will be similar or identical to that on the group leader.

Figure 3.6 shows a DVE in which two consistency groups of size 3 have been formed. The diagram shows the clients and nodes being hosted on the same machine. However, the diagram shows the group leaders as being a separate entity to the nodes/clients to aid clarity. Although logically the group leader is a separate entity, in fact the group leader will be hosted on one of the nodes/clients. In addition, the communication between the node/client on which the group leader is hosted will be implemented using inter-process/thread communication, rather than by using any real networking.

Due to the additional responsibilities of the group leader, the group leader's machine will be more heavily loaded than the other nodes in the group. The workload a group leader must perform includes:

- Receiving client object updates from clients and the main server
- Forwarding client object updates from its clients to the main server
- Dispatching collision detection on its group's nodes

- Performing collision detection and response on its subspace
- Receiving results from its nodes
- Detecting and correcting any inconsistencies in the results
- Sending appropriate current state information to clients

This additional processing overhead could lead to the group leader becoming a bottleneck in the system, similar to how a single server can become a bottleneck. As such, it may be desirable to place a limit on group size. This would result in multiple consistency groups when, without the limit, a single group could have been formed. However, this will ensure that the group leaders do not become the weak link in the system.

3.3.9 Variable Transmission Delays

The consistency group model provides potentially better consistency and performance than the standard client/server DVE model. However, the previous models have not addressed the problem of variations in network transmission delay; the previous model assumed transmission delays to be consistent. As mentioned previously, transmission delays are not constant, and can vary considerably depending on a number of conditions, such as network congestion and hardware failure. With the popularity of wireless networking, especially in modern mobile phones and gaming devices, the use of wireless ad-hoc networking in DVEs looks set to increase. Such networks can suffer from high-levels of variation in network transmission delay. As such, it is necessary to consider how this can be dealt with in the consistency group model.

In the current model, the consistency group remains constant while the nodes/clients participate in the DVE. When a new client/node joins the DVE, the main server will initiate the reallocation group membership. This requires each node to determine which nodes it can form a consistency group with. The

consistency groups are then allocated, and all nodes, including the newly-joined node, are placed in their respective groups. These groups will remain constant until a new client/node attempts to join or a client/node attempts to leave the DVE. However, this model assumes that during the time between a node being allocated within a consistency group and the groups being reallocated, the properties of the network latency between the nodes will remain consistent.

There are a large number of factors which contribute to network transmission delays, such as increased network traffic, routing hardware failures, damaged or noisy communication media. Due to the massively distributed nature of the Internet and the large number of hosts, routers and ISPs, network transmission delays can vary substantially. Wireless networking allows communication between hosts using radio signals whose physical distance is within a maximum range, e.g. 30m. In addition to the factors contributing to network delays in tradition, wired networks, wireless networking introduces the problem of signal interference which can cause significant variations in message transmission delays between hosts. Due to these factors, the previously held assumption of constant network transmission delays is not valid for wireless networking and the Internet.

As a consistency group is required to work in a relatively synchronised way, a node responding slowly (for whatever reason) may cause the responsiveness of the entire consistency group to suffer. In order to overcome this, the collision detection nodes are responsible for maintaining, during run-time, metrics pertaining to the transmission delay perceived between themselves and the other nodes in the DVE. These metrics can be gathered by storing the round-trip time taken to send a message and receive its response. This time includes not only the network transmission delay in both directions, but also any delays in processing, reading and responding to the message. A large transmission delay may imply not only a large network-induced delay, but may also indicate that the node/client may be being overloaded by processing demands. Regardless of its

cause, a large delay will cause performance degradation in the consistency group.

If a node detects that another node within its consistency group is beginning to lag, it can initiate a rejection vote with the server. The server requests the remainder of the group to decide whether the proposed evictee node is indeed lagging. If it is determined that the node is lagging, the node is removed from the consistency group and placed in a new consistency group of size one.

If a node in a consistency group determines that a node in another consistency group is responding sufficiently quickly, it can initiate an inclusion vote through the server to determine whether the node should join its consistency group. The vote will only be processed by the server if the group the proposed node is a member of contains fewer or the same number of members as group it is proposed to join. The vote requires each member of the group to determine whether the proposed new member is responding quickly enough to warrant joining the group. If the node is determined to be responding fast enough, the server instructs the node to leave its current consistency group and join its new group.

There is an inherent problem which may arise if this technique is used. Group membership may become in a state of flux, in which nodes are permanently changing groups. In order to alleviate this, two threshold values should be used: a *rejection threshold* and an *inclusion threshold*, where *rejection threshold* > *inclusion threshold*. This essentially gives a range of message transmission delays whereby the group will not request to be reallocated. In order to bring about a membership vote, a node within a group must observe:

- A transmission delay between itself and a node in another group less than the inclusion threshold
- A transmission delay between itself and a node in its group greater than the rejection threshold

The use of additional metrics, such as the mean/mode/median transmission delay and standard deviation, can be used to analyse the delays perceived by a node to avoid unnecessarily adjusting group membership as the result of minor transmission delay fluctuations. In addition, heuristics such as "three strikes and your out" could be used, in which a node can only initiate a rejection vote if it perceives a node responding slowly in three consecutive messages/samples. As restructuring groups may be expensive, it may be necessary to cap the frequency in which groups can be restructured. This could be done by restricting how frequently a client can initiate a vote or by restricting how many votes a server will allow within a given time frame. In addition, restrictions may have to be placed to ensure that multiple votes involving the same node do not occur simultaneously. A minimum time in which a node must be a member of a group could also be defined to ensure that groups do not repeatedly "poach" nodes from one-another. However, research into which approaches yield best results must be performed.

If the group leader was a reliable entity, it would be valid to not perform voting and simply piggy-back all recorded message transmission delays between a node and the other nodes in the DVE to its group leader. However, if the group leader becomes overloaded or fails, it will not initiate a group reallocation, resulting in the performance of the consistency group suffering significantly. As the group leader is a collision detection node, which is hosted on a user's machine, it cannot be assumed to be reliable. The problem of group leader reliability could be overcome by sending perceived delays directly to the server at frequent intervals. However, as the number of nodes in the DVE increases, this may cause scalability problems, resulting in the server becoming overloaded and failing to initiate group reallocations in a timely fashion, detrimentally affecting the performance of the consistency groups. Therefore, best overall reliability and performance can be achieved by adopting the voting strategy outlined in this section.

3.3.10 Reliability

One of the major concerns with distributed applications is reliability [Singhal99][Ezhilchelvan92]. Network protocols provide a range of reliability guarantees, ranging from completely unreliable to best-effort reliable. Some network protocols may offer complete reliability with the use of persistent storage of undelivered messages to deal with the event of failure in the sender, receiver or network. However, it is not possible to guarantee that a message will be received by the desired recipient(s) within a given time-frame in the event of hardware failure and, as such, reliable network protocols offer no guarantees for message delivery delay, which is of paramount importance in DVEs. It is, however, possible to state probabilistically how likely it is that the message is received within a given time-frame. Network protocols such as User Datagram Protocol (UDP) provide little reliability guarantees; UDP contains a CRC (Cyclic Redundancy Check) for error detection, but does not guarantee the delivery or ordering of packets. Transmission Control Protocol (TCP) is a best-effort network protocol, which provides best-effort guarantees of ordering and message delivery. This is achieved using message receipt acknowledgements and message retransmission after timeouts. Other higher-level protocols and services provide further reliability guarantees. For example, transactions ensure that an event either occurs or does not on all machines it is intended to. It uses commit algorithms and roll-backs to ensure that the event is either perceived to have occurred or not occurred on all machines consistently. These higher level protocols/services require the transmission of additional messages and may also incur a substantial delay between an event occurring and consensus being reached across all machines in the network.

Reliability can be discussed not only in terms of network protocols, but also in terms of machines. A machine can be described as reliable if it can be guaranteed to never fail. This, of course, is not realistic using current technology

as computer hardware will eventually wear out. In addition, the software executing on the hardware is prone to containing programming errors; the more complex the software, the greater the likelihood of undiscovered errors remaining in the software. With these facts, it is not realistic to expect a machine to be completely reliable. However, reliability can be improved in distributed applications by using data replication and back-up machines. This follows the principle that, given the likelihood of a machine failing is, for example, 1% (0.01), then the likelihood of 2 machines failing at the same time is 0.01% (0.0001). Following these probabilities, it can be seen that data replication/back-up can render the probability of failure negligible.

The systems previously described all assumed that the server is reliable. If the server fails, the DVE will come to an abrupt end. It may be possible to utilise a number of servers to alleviate this problem, although this issue is outside of the scope of this thesis. The description of the systems also assumed that the nodes/clients were reliable. These nodes/clients are free to join and leave the DVE as they require, although the unexpected failure of one of these machines must be detected by the server or other nodes, which will result in its group members being allocated a new sub-section of the world. Although it is not completely accurate, a node can be assumed to have failed if it has not responded to a message within a given large threshold time. This is termed a timeout, and will result in the server reallocating the group memberships, instructing the other nodes to cease communicating with the failed node and, finally, the server closing its connection with the node. If a persistent state is required to be maintained for the node's avatar, the server can store the avatar's current state in the DVE. If the node chooses to rejoin the DVE, it would then be able to continue from its last position, as recorded by the server. It should be noted that this is a best-effort attempt at detecting node failure as it is impossible to detect node failure in asynchronous networks due to the property of unbounded network transmission delays in these networks. However, for the purpose of DVEs, a period of extremely-large network transmission delays in

communicating with a given node will detrimentally effect responsiveness in its consistency group and, as such, it is appropriate to treat this situation as if the node has failed to avoid continually compromising responsiveness.

The reliability of network connections has so far been overlooked in this project. Due to the responsiveness/speed DVEs are required to operate, it is not possible to use high-level reliable messaging services while maintaining a highly-responsive virtual world. For the purpose of this system, we assume that a best-effort reliable network protocol, such as TCP/IP, is essentially reliable. This is not strictly true, but it will suffice for the time being. It is now necessary to categorise the messages a node or server will receive in terms of whether it is critical or not.

The first class of messages are administrative messages. These are messages exchanged between nodes and the server. These messages may contain messages such as instructing the server to add a new object into the world, instructing a set of nodes to connect to a new node or form a consistency group. These messages are vital for the correct behaviour of the DVE. If a node fails to receive a message instructing it to join a consistency group, an undesirable situation may occur. The nodes currently in the consistency group receive new sub-spaces to perform collision detection upon. However, there is a sub-space which has been allocated to the new group member. If this node does not join the group and perform collision detection on this region, then a section of the world will exist in which objects will not conform to the DVE rules regarding collision detection and response. Similarly, if an administrative message was damaged during transmission, the message could instruct a node to perform a task different from the desired task. As such, it is necessary that any administrative messages are sent over a (best-effort) reliable network protocol.

Nodes communicate with one-another in a variety of ways. It is necessary for nodes to determine the speed of message transmission between one-another.

This could be estimated using locality. For example, by examining IP addresses to determine if the nodes are part of the same network. Other heuristics could be used, such as users providing address details, which could be used to group together machines in the same towns or countries. Although these techniques could be successful, they rely on user honesty. Additionally, as IP addresses may be obtained from an ISP, two users may share the same network simply because they connected to the same ISP using a dial-up Internet connection. A more accurate technique to determine network latency, as previously mentioned, is to measure the time taken to send a message to a machine and receive its reply. This is termed the round-trip delay. These metrics could be obtained by either sending an empty (or nearly empty) message, or by piggy-backing timing messages to system messages. The former may consume additional bandwidth as it will require the transmission of many small messages, where each message must have fixed-size packet headers appended by the network protocols. The latter may reduce the accuracy of the timing data, as a response to a message will only be sent when a system message is ready to be transmitted. If the former approach was adopted, the reliability of the messages is not vital. As such, an unreliable network protocol could be used, although this may result in message loss or corruption.

Nodes are responsible for transmitting object transfer and replication messages. An object transfer message is relatively important, as it indicates that a given node is ceasing to be responsible for collision detection for a given object. The responsibility of collision detection for this object is transferred to another node, or nodes. If this transfer message is lost or damaged, it cannot be guaranteed that any node has assumed responsibility for the object; the object will become an *orphan object* in the DVE. As such, it is necessary for object transfer messages to be received correctly in order for collision detection to be performed correctly. An object replication message, however, merely indicates that the sending node has determined that an object's state should be replicated between itself and another node or group of nodes. If such a message was lost or

damaged, this may result in inconsistencies arising between nodes' views of an object. These inconsistencies, although undesirable, would not be devastating to the system. In addition, provided the group leader acts as an arbitrator, any inconsistencies could be detected and corrected.

Clients are responsible for transmitting their objects' state updates to the group leader and receiving object state updates from the group leader. If a client object's state update message was not received correctly by the group leader, the group leader's perception of this object would remain unchanged. As such, the group leader may initiate collision detection using out-of-date information. This should not cause any major issues, providing that each client object's state update messages are received from regularly enough to ensure that their object is perceived to move smoothly by the other clients and nodes. This is a reasonable assumption as, even with unreliable messaging, it is probabilistically likely that the majority of messages will be received provided the communication media is not congested. In addition, unreliable messaging does not guarantee the order of message receipt. As such, unreliable messaging may be desirable for state updates as it will allow current messages to be received and processed even if previous update messages have not yet been received. It is necessary, however, if this type of messaging is used that each message be time-stamped or assigned some logical ordering (perhaps using a logical clock). This will help to avoid the situation in which out-of-order messages result in older state updates overwriting newer data. The highest time-stamp, t^h, received could be stored so that any messages received whose time-stamp is not greater t^h is discarded. State update information must also be sent from the group leader to the clients to update their view of the DVE. This information can be transmitted identically to the state update information from the clients, as it will not disrupt the system too much if some messages are lost provided enough messages get through.

3.3.11 Unsynchronised Operation

The model described in this chapter requires members of a consistency group to operate in a synchronised manner, i.e. each collision detection iteration is synchronised within a consistency group. However, better performance can be achieved, at the detriment of consistency, if the members of a consistency group are allowed to operate in an unsynchronised manner. This also relieves the group leader from the burden of synchronising the collision detection nodes for every simulated time-step.

Unsynchronised operation requires a number of minor alterations to the model:

- Collision detection nodes store the state of client objects between collision detection iterations and do not wait to receive state updates from the group leader prior to performing collision detection
- Collision detection nodes use dead reckoning to predict object states between collision detection iterations if a new state has not been received
- Group leaders transmit client object state updates to the required collision detection nodes only when new states are received

This revision to the model introduces a new problem:

- A collision detection node, n_1, is responsible for a client object o_c at time t_0.
- The group leader receives a state update for o_c at time t_1, which results in n_1 no longer being responsible for o_c. Instead node n_2 is responsible for o_c.
- n_1 predicts the state of o_c using dead reckoning. As a result of deviation between the predicted state and true state of o_c, n_1 believes it is still responsible for o_c.

In this situation, it is necessary for the group leader to inform nodes which were previously responsible for a client object that they are no longer responsible for that object. While this adds a small overhead to the messaging requirements of the group leader, it potentially increases the performance of a DVE adopting this model by avoiding the need to synchronise nodes in a consistency group.

3.3.12 Discussion

The traditional client/server model of message dissemination in DVEs prescribes that state update messages are transmitted from clients to the server. The server forwards these messages onto the other clients participating in the DVE. High-level techniques, such as message filtering, may be employed by the server to reduce the volume of messages which the server must transmit to its clients.

The consistency group model of message dissemination in DVEs prescribes that state update messages are transmitted from clients to their group leader. The group leader, upon receiving a state update message from one of its clients, forwards the message on to the server and to the other clients within its consistency group. Upon receiving a state update message, the server forwards the message onto the other consistency group leaders, which are delivered to the remaining clients by their respective group leaders. Message filtering may be employed on the group leaders and the server to further reduce the volume of messages being transmitted.

From this brief description, it can be seen that for a client to receive a message through the client/server model, the message must pass through the server, which acts as an intermediate. For a client to receive a message through the consistency group model, the message must pass through a group leader. If the client receiving the message is not within the originator's consistency group, the

message must also pass through the server and an additional group leader before being delivered to the recipient.

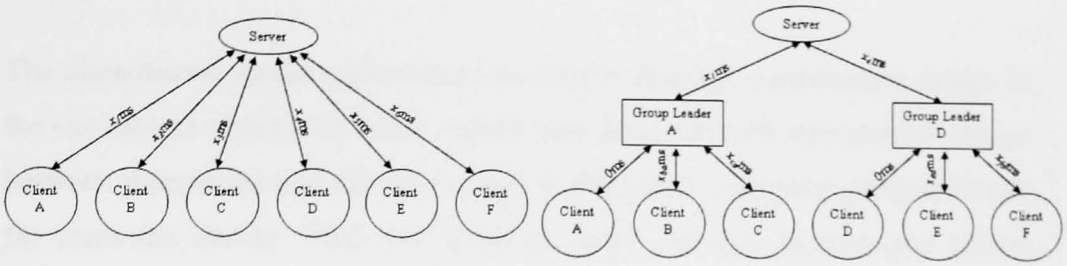

Figure 3.8 Client/Server vs. Consistency Group Models

Figure X.1 depicts the difference between the consistency group model and the client/server model. The average network latencies are shown on the diagram. In the client/server model, the delay between a state update message being transmitted by client B and being received by client C is $x_2 + x_3$ ms. Therefore, the overall message transmission delay in the client/server model is the sum of the delays between the respective clients and the server.

In the group leader model, the delay between a message being transmitted by client B and being received by client C is $x_{ba} + x_{ca}$ ms. Given that the perceived message transmission delay between a client and its group leader must be relatively small, the delivery of such a message should not suffer from a great deal of message transmission delay, assuming the absence of network or hardware failure. However, the delay between a message being transmitted by client B to client E is $x_{ba} + x_1 + x_4 + x_{ed}$ ms. In the client/server model, this delay would be $x_2 + x_5$ ms. As a result, the delay between a message being transmitted between clients in different consistency groups may be larger than the delays perceived between the same clients using the client/server model. However, the additional network transmission delay due to the message passing through the group leaders should be small. In addition, this approach may reduce the volume of messages which the server must process. This can reduce the server's

processing load, thereby reducing the delay between the server receiving a message and it processing the message and forwarding it to its intended recipients, thereby reducing the total message transmission delay.

The client/server model suffers from its largest message transmission delays in the situation in which each client suffers from large network transmission delays between themselves and the server, e.g. if the server is located geographically far from the clients. This will result in large message transmission delays between a pair of clients regardless of the clients' respective message transmission delays. To clarify, clients' who would ordinarily enjoy low-latency message transmission delays directly between each-other may suffer from large transmission delays if the message transmission delay between themselves and the server is high. The consistency group model may alleviate this problem by grouping machines together which exhibit low-latency message transmission delays; messages between these machines are routed via the group leader, which is itself a member of the consistency group, offering lower-latency message transmission.

Given a DVE in which all client's exhibit high message transmission delays with the server, network transmission delays between consistency groups will be large but network transmission delays within consistency groups will be small. This is because the consistency group model requires message transmission between consistency groups to pass between two group leaders. In this situation, the increased network latency as a result of messages between consistency groups passing through group leaders should be negligible due to the large delays exhibited between the server and its clients.

The consistency group model suffers from its worst overall performance when all clients participating in the DVE exhibit high network-transmission delays among each-other. In this scenario, every client will be placed in a consistency group containing just one client; the consistency group model results in the same

message dissemination as the client/server model, offering identical performance. Therefore, the use of consistency groups and group leaders allow collocated nodes to enjoy the available low latencies between them, promoting improves consistency and responsiveness. However, large latencies between non-collocated nodes, e.g. due to geographical location, will never be eased by the use of consistency groups. Additionally, scalability limitations related to the number of nodes that can be supported may be alleviated via the consistency group model making use of aggregated messaging.

In the consistency group model, the collision detection nodes have partial knowledge of the DVE; they know the current state of the objects for which they are responsible for collision detection but are oblivious to the state of any other objects inhabiting the DVE. The clients have partial knowledge of the DVE, as they know the current state of their objects/avatars. However, in order to display the appropriate images to the end-user, it is necessary for the clients to be informed of the current state of any objects which must be displayed onscreen. As was previously mentioned, the group leader stores the current state of all objects as perceived within the consistency group. As such, it is the responsibility of the group leader to ensure that all clients within its consistency group are informed of any objects which they must render onscreen. This could be implemented efficiently using techniques currently employed in DVEs, such as interest management [Morgan05][Greenhalgh][Watt01][Sinhal99]. However, this is beyond the scope of this thesis.

Chapter 4

Implementation

4.1 Introduction

The following chapter describes the implementation details of the distributed collision detection approach described in this thesis. This includes discussions on the implementation technologies, algorithms and structure required to implement the distributed collision detection approach described in Chapter 3. Following this, implementation-specific optimisations will be discussed.

4.2 Implementation Technologies

There are a number of different implementation technologies which are available to DVE developers, of which the developer must select the most appropriate to meet their needs based on:

- Ease of use or integration
- Performance
- Scalability
- Memory utilisation
- Platform-independence
- Stability and Reliability

The implementation technologies which will be discussed in this chapter include programming languages, system libraries and middleware solutions. Although this list is far from extensive, it covers the major considerations which are of concern to developers.

4.2.1 Programming Languages

There are a number of programming language paradigms available [Wiki06 2], the most common of which are *Structured* and *Object-Oriented* programming languages.

Structured programming languages allow developers to separate programs into a series of functions, each of which can take arguments and return values. This approach promotes reduced code duplication and allows large problems to be broken up into a series of smaller, easily-understandable sequential steps. This facilitates the development of software in groups, allowing individual functions to be implemented and tested independent of one-another. Functions in structured programming prescribe the use of arguments and local variables over global variables. While not forbidden, the reduced dependence on global variables eases bug-tracking and fixing in large software systems.

Object-oriented programming (OOP) prescribes the separation of a program into a set of individual units, or objects. An object-oriented program is essentially a set of classes (object-types). A class contains member variables and provides methods and constructors by which an instance of the class can be interacted with. Methods, similar to functions, can take parameters and can return values. While not forbidden, OOP discourages the direct manipulation of an object's data members and allows programmers to conceal implementation details of class with the use of scope operators to declare member variables and methods

as being, for example, public, private or protected. OOP allows classes to inherit functionality from one-another; the class inherited from is termed a super-class and the class which inherits the functionality is termed a sub-class. This promotes polymorphism, where a given method, inherited from a super-class, may behave differently depending on which sub-class the object is in fact an instance of.

In addition to programming paradigms, programming languages can be subdivided into two different types:

- Compiled programming languages
- Interpreted programming languages

Compiled programming languages, such as C and C++, go through a compilation process in which the high-level code produced by a programmer is transformed into platform-specific machine instructions. Modern compilers may utilise code-optimisation, which may result in the pipelining of commands, code branching optimisations and code-reordering, to provide efficient performance. However, this often sacrifices debugging facilities, meaning that the program counter may not map to the original source code accurately. In addition to debugging issues, programs written in a compiled language must be compiled independently for each target platform; this may require the re-working of parts of the program to utilise platform-specific libraries.

Interpreted languages, such as Basic, JavaScript and Python, conversely, do not go through a compilation process but are instead interpreted by a run-time engine. This ensures that programs in these languages are platform-independent, as an interpreted program can be run on any platform which has the appropriate run-time engine. This flexibility, however, comes at the detriment of performance as interpreted languages usually perform far slower than compiled programs. As such, interpreted languages are often used to produce smaller

programs or prototyping, while compiled programming languages are commonly used for the development of large applications.

4.2.1.1 Candidate Programming Languages

Following from the descriptions of the classifications of programming languages, it is necessary to categorise the candidate languages and analyse their strengths and weaknesses. Although there are a large number of programming languages available, this section will consider two object-oriented programming languages: C++ and Java.

4.2.1.1.1 C++

C++ [C++] is the successor to the C programming language. Developed at Bell Laboratories for the UNIX environment, C++ has become the de-facto programming language for large-scale applications in which performance is of paramount importance. It is a compiled programming language and, while an executable produced from C++ source code is often not as efficient as the same program written in C, the addition of object-oriented design makes C++ capable of producing much more elegant solutions to complex problems. C++ is a flexible programming language in which developers can choose to adopt a combination of object-oriented and structured programming paradigms, depending on their requirements.

C++ is fully supported across most platforms and supports applications being separated up into a series of modules, or libraries. Most modern operating systems support static and dynamically-linked libraries. However, while the C++ language is reasonably standardised, much of its library-support is platform-dependent. This means that not only must applications be compiled for target platforms, but often large portions of code must be written specifically for each target platform. Most platforms will provide libraries for networking,

multi-threading, synchronisation, I/O and GUI application building. Additionally, C and C++ are the most widely-supported programming languages for 3D graphics application development, through the use of graphics APIs such as OpenGL and Direct3D.

C and C++ are relatively low-level programming languages, whereby the developer is required to control all memory allocation and free this memory before the application terminates; failure to properly allocate the required memory can result in run-time errors, whereas failure to free all memory allocated can result in memory leaks. The developer is also required to differentiate between allocating memory on the stack and in free memory (the heap). This makes C++ memory management difficult for experienced and novice programmers alike. However, this feature ensures that C++ applications have a memory footprint roughly equivalent to the amount of memory the application requires. Automatic garbage collection has been introduced into the latest versions of Visual C++ .NET for the Microsoft Windows platform. However, the utilisation of this significantly reduces code portability and, while this mechanism utilises reference counters to remove developers' need to delete objects, they must still distinguish between allocating in free memory (so-called "GC" objects) and those allocated on stack memory (value objects); the use of this technique introduces a rigid differentiation between GC and value objects, whereby a class declared as a GC object can not be instantiated on the stack and vice-versa. Additionally, the use of automatic garbage collection can reduce application performance significantly and can also limit the programmer's ability to engage in a number of low-level memory manipulation techniques, e.g. storing offsets to memory-addresses of pre-allocated objects relative to a known memory address; the object in this memory address would be automatically deleted when their reference counters reached zero, regardless of any future intention by the developer of referencing the objects again at a later time.

4.2.1.1.2 Java

Historically, the majority of commercial applications were written in compiled programming languages, with the most common being C and C++. However, Java [Sun06] has become an extremely popular programming language which is used for large-scale software development. It is an object-oriented interpreted language with a slight difference: the source code is compiled into platform-neutral virtual machine instructions called byte code. This byte-code can be efficiently mapped onto platform-specific machine instructions thereby providing run-time performance approaching that of compiled programming languages. Java includes built-in support for multithreading, thread synchronisation and network communication; these features are only available through libraries in C and C++, which can vary significantly between platforms. Unlike C++, Java's libraries are standardised across all platform ensuring that Java source code can be written and compiled once and executed on any supported platform. While Java includes support for lower-level network programming, such as TCP Sockets, it also has integrated support for higher-level concepts such as remote procedure calls, distributed objects and message-oriented middleware through the use of Java RMI, JiNi, J2EE and CORBA. These high-level networking services are available through third-party libraries in C++ but are not integrated into standard C++ SDKs.

Java provides facilities to embed applications into web pages and to produce dynamic web pages. It also provides facilities to integrate Java code with legacy code written in compiled languages, such as C++, called Java Native Invocation (JNI). Unlike lower-level programming languages, such as C++, Java utilises automatic garbage collection to delete objects from memory without programmer intervention. Additionally, Java does not require the programmer to distinguish between allocating memory in free memory or on the stack. Java is an evolving programming language. However, rather than allowing arbitrary additions to the language which may compromise backwards compatibility,

Java's evolution is tightly regulated to ensure that all Java Run-time Engines (JREs) can correctly execute programs written to their current or previous specifications. However, Java's flexibility comes at the cost of memory consumption. Each Java thread must have its own JRE. This means that large multithreaded applications can occupy a considerable amount of memory, potentially resulting in memory page thrashing, which can sacrifice performance considerably.

4.2.1.2 Summary of Programming Languages

There are a number of different programming language paradigms, the most common being structured and object-oriented programming. The object-oriented programming paradigm can be seen as the evolution of the structured programming paradigm, providing a number of facilities which promote code-reuse and data encapsulation. In addition to these paradigms, two different classes of programming languages were explored: compiled and interpreted programming languages. Compiled programming languages are transformed into platform-specific instructions before being executed whereas interpreted languages are translated on-the-fly by a run-time engine into machine instructions. Compiled programs must be built for each target platform whereas interpreted programs can be run on any platform which provides an appropriate runtime engine.

Two candidate programming languages were introduced, C++ and Java. C++ is a compiled, object-oriented programming language which offers good support for graphics development and extremely efficient performance and memory utilisation. However, it comes at the cost of much greater development complexity. Java is an object-oriented interpreted language which utilises platform-neutral virtual machine code which can be efficiently translated on-the-fly to machine code. Java is, comparatively, far easier to develop for than C++ and offers comparable performance in most situations, with the exception of

applications which require large amounts of memory dynamically allocated in runtime.

This chapter will describe the distributed collision detection technique in a language-independent manner. Discussion of programming languages are included to outline the importance of selecting an appropriate language for application development. The distributed collision detection technique described in this thesis has been implemented in both Java and C++; the former being the initial prototype and experimental environment used to gather performance metrics whereas the latter was incorporated into a games engine developed as part of this thesis to demonstrate the application of this technique in a publicly available DVE.

4.2.2 Platforms

It is highly desirable for DVEs and their supporting technologies to be able to operate on a wide range of machine configurations and platforms. The commercial arm of DVE research, computer games, are released on a number of platforms including PCs, Apple Macintosh, Portable and Home Games Consoles, Portable Digital Assistants (PDAs) and Mobile phones. These platforms differ in their Operating Systems and hardware configurations. The differences in operating systems are manifested in variations in library support, possibly necessitating the developer to rewrite sections of code specifically for each platform. Different operating systems may also offer different levels of support for multi-threading, including different levels of granularity between pre-emptive process switching if pre-emptive multitasking is available. Differences in hardware configurations may include the processing resources available, including the clock speed of the primary central processing unit and any secondary processing resources, e.g. co-processors, graphics processing units, and additional processing cores. Variations in available memory, bus

transfer speed, and network bandwidth may also be present. These variations, coupled with machine-specific issues such as byte aligned memory, endian-ness and instruction pipelining can greatly effect the applicability and performance of collision detection algorithms.

This section will provide case studies for PC and games consoles, which will focus on two current generation games consoles, PlayStation 2 and X-Box, and two next-generation consoles, PlayStation 3 and X-Box 360.

4.2.2.1 PC

The PC comes in a wide range of configurations, varying in:

- Processor manufacturer, model and clock speed
- Number of processors or processing cores
- Instruction set extensions (e.g. MMX, SSIMD etc.)
- Amount of available memory and access speed
- Speed/bandwidth of bus
- Graphics Processor model and memory
- Operating system

The differences in PC architecture are largely concealed by the operating system. As such, differences in hardware rarely result in developers needing to change code; major code changes are only usually required to provide interoperability between different operating systems. However, DVE developers do have to target a certain minimum configuration, such as minimum CPU speed, amount of memory and graphics card. As PCs are upgradeable and use virtual memory, PC DVEs rarely need to be overly concerned with memory utilisation. Similarly, PCs and their operating systems can efficiently allocate and release large amounts of memory without compromising performance.

These factors usually allow DVE developers to allocate and release memory as required in run-time with little impact on the DVE's responsiveness.

Modern compilers are capable of transforming high-level code into efficient machine instructions. However, the assembly code which is generated by a compiler is often less efficient than can be produced by a competent assembly language programmer. As such, it is common in DVE development and, specifically, in collision detection, that many of the more frequently executed code segments are optimised in assembly language for each target platform.

Different operating systems exhibit different levels of granularity in their pre-emptive multi-tasking. Windows is generally more coarsely grained than Linux, while specialist hardware, such as games consoles, often only support cooperative multi-tasking. This means that processes are less likely to be starved of processing resources in Linux than Windows. In addition, the Windows OS tends to consume more resources than Linux, causing the same application, compiled for each platform, to require a lower-specified machine on Linux than on Windows to run acceptably. However, due to marketing successes and its ease of use, Microsoft Windows is the most popular PC operating system by far despite these shortcomings.

4.2.2.2 Games Consoles

Modern games consoles, when newly-released, represent the state of the art in affordable consumer processors and graphics. There are often a number of sacrifices which are made to ensure that games consoles are affordable, such as restricting memory and processing resources in an attempt to lower prices; in fact, many games console manufacturers accept significant losses on the sale of hardware by weighing these losses against the sales of software. Games consoles are traditionally played through home television sets, which provide significantly lower-resolution images than PC monitors. The resolution of

television sets reduces the problem of aliasing significantly, making low-resolution 3D graphics look more attractive than they would on higher-resolution screens; anti-aliasing techniques for higher-resolution displays are extremely computationally expensive. Recently, high-definition TV has become more affordable and has offered high-resolution TV content, which the next-generation of games consoles offers built-in support for. However, high definition TV still offers image resolutions far lower than those supported by PC displays.

Games consoles traditionally have a 5 year lifecycle, during which the games console's hardware is fixed; the hardware may go through evolution to make it more cost-effective to produce, to make it smaller, more robust, consume less electricity etc., but the actual hardware specifications will not change. As such, nearing the end of their lifecycle, games consoles are underspecified when compared with modern PCs, for which new hardware advances are released continuously.

Games consoles will usually have a minimal operating system pre-loaded onto the hardware which will allow the user to perform setup configurations and manage saved games when a game is not inserted into the console. As this operating system is minimal, it will not incur the overheads associated with more substantial operating systems such as Windows or Linux. As such, games console hardware often provides better performance than the equivalent specification PC. However, the lack of a sophisticated operating system means that games developers cannot rely on the presence of advanced operating system features such as pre-emptive multitasking, which may complicate development.

4.2.2.2.1 PlayStation 2

Sony's PlayStation 2 [Sony06] is the leading current generation games console. It was released in 2000 and was, at the time, extremely powerful. At its core is a

300Mhz Custom processor developed by Sony and Toshiba termed the Emotion Engine, 32Mb RAM, a graphics processor with 4Mb RAM and a DVD ROM. These specifications are significantly lower than current PC hardware. At the time of writing, current entry-level PCs contain processors with clock speeds in excess of 3Ghz, 512Mb-2048Mb RAM and hardware graphics cards with 128-512Mb RAM.

Software development for the PlayStation 2 is fraught with difficulty. It has taken developers the life-time of the console to discover how best to harness its power. Its architecture and development tools make software development and debugging difficult. The PlayStation 2 has a number of co-processors, termed Vector Units, which can be used to perform common vector operations in parallel to increase performance substantially; these operations are equivalent to the SIMD extensions introduced in the Pentium III processors. However, the use of these requires data to follow strict byte-alignment rules; failure to follow these rules will not always result in application failure but can result in unexpected results. It is only possible to perform operations on Vector Units by writing the section of code in assembly language; the development tool compilers will not perform these optimisations automatically. Additionally, the PlayStation 2 has very poor memory allocation performance, meaning that allocating memory during the runtime of an application can significantly compromise performance. In order to achieve acceptable performance, it is necessary to pre-allocate blocks of memory and provide a memory manager to assign these blocks to objects in runtime.

Due to the limited memory available on the PlayStation 2, it is necessary to break a virtual environment up into discrete, smaller levels or stream sections of the environment to main memory during run-time. However, streaming in real-time can not be guaranteed to be performed correctly as disc access speed can be slow on damaged discs and can vary significantly between consoles. This is because as DVD/CD drives age, the lens which reads the disc can become

damaged and less efficient, resulting in longer disc access times. Failure to load a section in time can result in missing textures or missing geometry. To overcome this, it is common that sections are streamed in a series of detail levels, whereby at least the most coarse detail levels are successfully loaded prior to them being needed. While this usually overcomes the problem of missing sections, it can result in noticeable detail popping if the console is late in loading the high detail version of the environment section.

4.2.2.2.2 Xbox

The Microsoft Xbox [Microsoft06] was released as a rival for the PS2 in 2002. While the console was significantly more powerful than the PS2, it failed to overtake Sony's already significant market share. Although the Xbox sold well in the USA and Europe, it failed in the Asian market, specifically in Japan. The Xbox was essentially an entry-level PC made from off-the-shelf components, being constructed from:

- An 733Mhz Intel Celeron Processor
- 64Mb RAM
- DVD ROM
- 8 Gb HDD
- NVidia NV20 Graphics Processor

The Xbox adopted a Unified Memory Architecture design, where programmers could choose how much memory to assign to holding graphics data or game data. It utilised a current-generation NVidia GeForce graphics processor, which was significantly more powerful than the Graphics Synthesizer in the PS2. The Xbox also provided a high-quality networked game system called Xbox Live.

The Xbox was programmed using similar tools and libraries to Windows PCs. This removed the need for developers to learn new tools in order to create games

for the Xbox, which resulted in numerous high-quality launch titles for the console which demonstrated its processing superiority to the PS2. The Xbox's hardware addressed a number of issue with current games consoles. The most significant of these was the adoption of an easily-programmable general-purpose Intel processor instead of proprietary technology. This, coupled with support for the DirectX SDK meant that Xbox games could be produced inexpensively; Xbox games could be created by porting PC code and reducing memory requirements by reducing texture resolution and geometry detail.

The Xbox provided an 8GB hard drive as standard. This allowed game developers to use the hard disk as a temporary, higher-speed storage medium for environment data to help data streaming. This significantly reduced the impact of the condition of the DVD lens in the runtime performance of the game.

4.2.2.2.3 Xbox 360

The Xbox 360 [Microsoft06] console was released in 2005 and adopts three 3.2Ghz Hyperthreaded IBM PowerPC processing cores. It comes with 512Mb Unified Memory, a DVD ROM, a next-generation ATI graphics processor and an optional 20Gb hard drive. It utilises similar programming APIs to the Xbox, easing the porting of games from PC to console. It supports high-definition displays and surround-sound audio. The Xbox 360 adopts less well-understood technology than the original Xbox and therefore will require time before games developers fully understand the intricacies of optimising applications for the new hardware.

Unlike the original Xbox release, Microsoft has pre-empted its rivals in releasing its next-generation hardware. Microsoft has been criticised for not giving the Xbox as long a shelf life as the PS2 has received. This criticism has been largely ignored by the software giant who ceased production of the original Xbox console and future games in early 2005. Criticism has also been levied at

the console that, while the graphics hardware is very powerful, the CPUs in the Xbox 360 are significantly under-powered and cause a bottleneck in performance. While this is the case, it should be considered that many of the games released for the console only used one of the 6 hardware threads capable of being executed simultaneously in the Xbox 360. As games engines evolve to exploit multiple processing cores, the performance of games on next-generation platforms will improve.

4.2.2.2.4 PlayStation 3

The PlayStation 3 [Sony06] is expected to launch in November 2006 in Japan and the USA and March 2007 in Europe. It adopts a single 3.2Ghz Power PC processing core and 7 Synergistic Processing Engines (SPEs), termed the Cell Processor; the cell processor was a joint-development between Sony, IBM and Toshiba. Rumour is circulating that the central processing core may be reduced to 2.6-2.8Ghz to reduce manufacturing costs. The PS3 will fully support high-definition TV standards and will adopt the new Blue-ray disc format, a high-definition replacement for DVD. It will come with 256Mb RAM, a 256Mb next-generation NVidia graphics processor and a 20-60Gb hard drive. As it has yet to be released, these specifications are subject to change but are consistent with the pre-release development kits being used to produce launch titles.

The PS3's utilisation of the cell processor offers a huge amount of power (in excess of 2 TFlops) at the cost of increased programming difficulty. The PS3's main processor is significantly slower than its SPEs, which can be seen as being secondary co-processors; its main processor is in fact the exact same processor as is used in the Xbox 360. The SPEs cannot directly operate on main memory and each contain 256Kb of cache memory. The job of the main processor, according to Sony, is to stream jobs to these SPEs and the GPU and write results back to main memory. However, this requires current games engine designs to be completely re-evaluated to adopt parallel execution, something which has

previously not been considered. This provides developers with a significant challenge: a current games engine will potentially run slower on a PS3 than on an Xbox 360, despite the fact that reports suggest the PS3's cell processor may offer performance orders of magnitude faster than the Xbox 360.

In addition to SPEs, the PS3 also adopts Vector Unit co-processors to efficiently perform vector operations; there is one VU per SPE and these VUs still impose the same strict byte-alignment rules as in the PS2. Finally, the PS3 includes within its hardware the original PS2 processor. This will operate as an I/O processor and will also provide backwards compatibility with all PS2 games.

4.2.3 Transformations

Transformations are a fundamental operation in graphical virtual environments [Lengyel03]. A point in 3D space can be rotated, translated, scaled and sheared. These operations can be represented using a 4x4 transformation matrix. With the development of hardware accelerated transformation and lighting GPUs, transformations using 4x4 matrices are performed in graphics hardware. However, these transformations are performed as part of the rendering pipeline and the transformed vertices are local to each stage of the rendering pipeline. Therefore, while the choice of representation of transformations is fixed in GPUs, it is still worthwhile exploring alternative transformation representations as part of a collision detection engine

In DVEs, it is common that only rotations and translations are used as DVEs often mimic the real world and scaling (enlarging/shrinking) and shearing (stretching along an arbitrary axis) operations generally do not occur in reality. Translations can be represented using a 3D translation vector. Rotations can be represented in a number of different ways:

- Euler angles

- 3x3 transformation matrix
- Quaternions

Euler angles offer the simplest representation of rotations. It stores three floating point numbers, representing the angle of rotation around the coordinate axes, significantly reducing the storage overhead of a transformation compared to a 4x4 matrix. However, in order to perform the transformation, this must be converted into either a 3x3 or 4x4 transformation matrix.

3x3 matrices are the upper-left part of a 4x4 matrix, representing the rotational component of the transformation. They require significantly fewer multiplications and additions than 4x4 matrices. A 3x3 matrix with a 3D translation vector require 9 multiplications and 6 additions to perform the transformation; a 4x4 matrix requires 16 multiplications and 12 additions. 3x3 matrices with a translation vector therefore offer a reduced memory footprint (12 floats as opposed to 16 floats) and require fewer operations to perform the transformation.

Quaternions offer a different representation of a rotation. A quaternion is essentially a 4D complex number. It is composed of 4 parts: one real number and three coefficients to imaginary numbers. A quaternion can be represented as: $H = a + bi + cj + dk$, where a, b, c, and d are real numbers and:

$i^2 = j^2 = k^2 = ijk = -1$

$ij = -ji = k$

$jk = -kj = i$

$ki = -ik = j$

Quaternions hold a number of different properties, the most useful in DVEs being that a rotation of angle Θ around a unit vector n can be represented by the quaternion:

$q = (s, \mathbf{v}) = \left(\cos\left(\frac{1}{2}\Theta\right), n\sin\left(\frac{1}{2}\Theta\right) \right)$, where s is a real number and $\mathbf{v}$ represents a 3D vector of coefficients for the imaginary numbers i, j and k.

It can be seen that a quaternion offers a reduced memory footprint for rotations. Coupled with a translation vector, a quaternion would require 7 floating point numbers to represent a full transformation. However, while not fully discussed in this section, quaternion maths requires a firm understanding of complex numbers and is more difficult to visualise than other representations, i.e. it results in a slight increase in development complexity. Due to floating point rounding errors, it is possible for transformation matrices to become non-orthogonal after a number of matrix multiplications have been performed; a non-orthogonal matrix is a matrix whose component axes are not mutually perpendicular to each-other. In order to overcome this, matrices must be re-orthogonalised frequently; re-orthogonalisation is a relatively expensive operation. Quaternions are by definition always orthogonal and, therefore, it is not necessary to re-orthogonalise a quaternion.

In virtual environments, it is often desirable to be able to interpolate between rotations to smoothly animate an object from one transformation to another. Although there are an infinite number of ways to traverse from one transformation to another, it is normal to use the shortest path from one to another, termed the *torque-minimal path*. An approximation to this path can be found using a number of techniques, the most intuitive being linear interpolation. Given two transformation matrices, $\boldsymbol{A}$ and $\boldsymbol{B}$, it is possible to interpolate between these matrices to smoothly transition from $\boldsymbol{A}$ to $\boldsymbol{B}$ as below:
$(1-i)\boldsymbol{A} + i\boldsymbol{B}, 0 \leq i \leq 1$

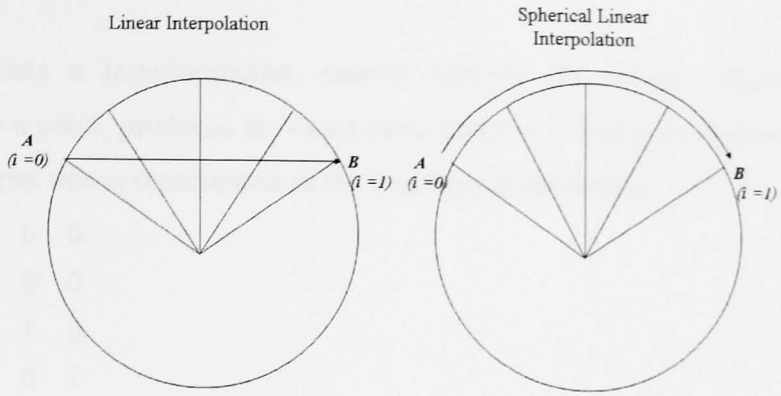

Figure 4.1 Linear Interpolation v Spherical Linear Interpolation

From ***Fig*** 4.1, it can be seen that linear interpolation can result in irregular sampling around the circumference of a circle, which results in inconsistent angular velocity. If this form of interpolation was used in animation, it could sacrifice the smoothness of the animation. Quaternions offer spherical linear interpolation (SLERP), which guarantees constant angular velocity and can provide a very accurate approximation the torque-minimal path. SLERP can be performed as below:

$\frac{\sin(1-i)\Omega}{\sin\Omega}A+\frac{\sin i\Omega}{\sin\Omega}B$, where $0 \leq i \leq 1$, A and B are the origin and end rotations respectively and Ω is the angle subtended by the arc, so that $\cos\Omega = A \cdot B$, the n-dimensional dot product of the unit vectors from the origin to the end.

However, spherical linear interpolation is a computationally expensive operation, which may result in reduced performance. As such, the choice of either linear interpolation or spherical linear interpolation must be made by balancing out the computational cost against the smoothness of animation. It should be noted that the undesirable visual effects of irregular angular velocity can be reduced significantly by increasing the sampling frequency. For example, given sampling frequencies in excess of 30 samples per second, it is unlikely that a human would be able to distinguish between both techniques. However, at sampling frequencies of, say, 2 samples per second, it is likely that a human would notice irregular angular velocity.

There exists a transformation, termed Identity (**I**), which, when used to transform a point, produces the exact same point as it was provided with; in 4x4 matrix form, this is represented as the transformation matrix:

$$\mathbf{I} = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Additionally, for any transformation, **T**, there exists a transformation, $\mathbf{T}^{-1}$, such that $\mathbf{T}\mathbf{x}\mathbf{T}^{-1} = \mathbf{I}$. This is termed the inverse transformation, where performing the transformation **T** followed by $\mathbf{T}^{-1}$ provides the same result as no transformation being applied whatsoever. Therefore, it can be seen that given two objects obj_1 and obj_2 with respective world transformation matrices $\mathbf{T}_1$ and $\mathbf{T}_2$: $\mathbf{T}_1 \times \mathbf{T}_1^{-1} = \mathbf{I}$ $\rightarrow \mathbf{T}_2 \times \mathbf{T}_1^{-1}$ transforms obj_2 into obj_1's local coordinate space. This means that it is possible to calculate a transformation such that one object can be expressed in terms of another object's local coordinate space, thereby removing the need to transform both objects into world space to compare them for intersection. As coordinate transformation is an complex operation, such an optimisation can vastly improve the performance of a collision detection technique.

4.3 System Implementation

The design and implementation of the distributed collision detection technique described in this thesis is relatively complex and a number of non-trivial optimisations were performed to improve performance and reduce memory requirements, some of which were introduced previously in this chapter. However, to provide a more understandable description of the system implementation, this section will describe the data members and behaviour expected of each component which constitutes the collision detection approach,

rather than discussing low-level implementation details; little or no discussion will be made of optimisations such as embedding flags into bit vectors or encoding variables into the same data member using lowest-significant bits to flag which data is contained. The use of these optimisations is recommended to produce the best performance possible.

The implementation details provided in this section describe the model described in Chapter 3, Section 3.3.11. The implementation can be logically subdivided into three components, the Server, the Clients and the Collision Detection Nodes. The server and collision detection node implementations rely heavily on the use of and inter-communication between multiple threads of execution. With the use of multiple threads of execution, it is very important to ensure that random thread interleaving cannot result in undesirable and non-deterministic behaviour in the application. This problem has received considerable research interest and a number of approaches to avoid undesirable thread interleaving have been developed. However, these techniques are outside of the scope of this thesis and to aid clarity, descriptions of thread synchronisation techniques and the synchronisation requirements of the distributed collision detection approach are omitted from this chapter. The use of thread synchronisation is critically important in a number of operations in distributed collision detection to achieve correct behaviour.

4.3.1 The Server

The server is responsible for handling incoming requests from collision detection nodes to join a given DVE and informing each node of the newly-joined participant. Following this, the server must inform the new node which objects it is responsible for in the DVE and re-allocate the existing consistency groups in the DVE to reflect the new participant. The server must also handle requests by existing nodes to leave the DVE and requests for membership and

eviction votes raised by members of each consistency group during the life-time of the DVE. In addition to receiving requests from collision detection nodes, it is necessary for the server to receive connection requests from clients wishing to participate in the DVE. These clients usually have a one-to-one relationship with collision detection nodes, but it is possible for a machine to join as a client but not as a collision detection node and vice-versa. This provides the possibility for low-spec machines to participate in a DVE and for collision detection to be executed using a grid-computing model if appropriate resources are available.

The server is a separate application to the nodes, which operates in a completely different addressable space. The Server is subdivided into a number of threads which execute simultaneously and communicate with one-another through shared memory. The structure of the Server can be seen in Fig 4.2. The Server can be broken up into four threads: the main server thread, the client listen thread, the object listen thread and the admin listen thread. The main server thread is responsible for listening for incoming connection requests. The client listen thread listens for state update messages from the clients, indicating that one of the client's objects has changed its state. The object listen thread listens for updates in object states from the collision detection nodes. A message received by this thread indicates that a collision detection node has detected a collision and responded to it. The admin listed thread listens for administrative messages from the collision detection nodes; such admin messages may include:

- Requests to leave the DVE
- Requests to initiate a group inclusion or rejection vote
- Communication latency reports indicating which collision detection nodes are possible candidates for group membership from a given node

The Admin and object communication streams are kept separate to allow different threads to process these messages without needing to be concerned about random interleaving between object and admin messages.

All of these threads are implemented using thread pools to ensure that events are processed in an efficient manner. The choice of thread pools over one thread per collision detection node and client is to avoid process starvation, in which the proportion of processing resources each thread receives becomes minutely small. To the right of the Admin Listen Thread and Object Listen Thread in ***Fig*** 4.2 is the Handshake Server. These are created on a per-collision detection node basis and are the mechanism by which the admin and object listen threads access the data transmitted from each individual collision detection node. Similarly, the client update server is created on a per-client basis and is accessed directly by the client listen thread. Both the handshake server and client update server contain instances of an abstract Communication interface, which will be discussed in more detail in the next section.

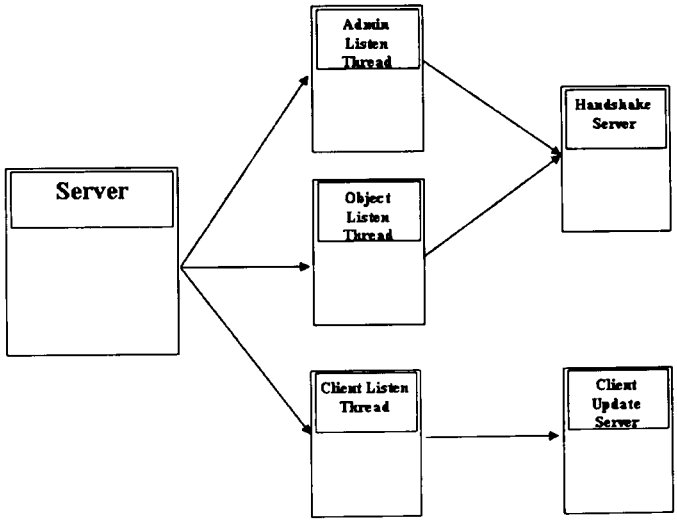

Figure 4.2: Distributed Collision Detection Threads

4.3.1.1 Communication Model

The communication model used in the distributed collision detection approach is designed to be as abstract as possible. An abstract Communication interface is

defined through which all data communication takes place. This interface shields the developer from the intricacies of network communication, such as protocol selection and platform-specific issues such as proprietary system libraries and byte-ordering. Communication objects are instantiated using a Communication Factory. The Communication Factory provides a mechanism to obtain standard Communication objects and Server Communication objects. The Server Communication object is responsible for listening for incoming connection requests and creating Communication objects to service the message exchange requirements of each request. The Communication object provides mechanisms to transmit primitive types to its recipients. In addition, all non-primitive types required to be transmitted through the Communication object must extend the Communicable interface and implement four methods:

- void encode(Communication comm)
- void decode(Communication comm)
- void encodeLightWeight(Communication comm)
- void decodeLightWeight(Communication comm)

The encode and decode methods encode and decode all member variables belonging to the object through the communication parameter. The encodeLightWeight and decodeLightWeight methods offer a mechanism whereby only the minimum amount of data needed to achieve consistency is encoded and decoded.

A number of protocols and middleware services have been integrated into the distributed collision detection approach, including TCP/IP, UDP/IP and CORBA. The use of flexible Communication Factories allows extensibility whereby any protocol capable of transmitting data can be utilised transparently within the distributed collision detection engine. The Communication interface uses the abstraction of a packet-based data-stream, whereby in order to transmit a message, it is necessary to:

- Request a handle to a message, which returns an integer identifier to a new message object
- Submit communication primitives to the data-stream of the message object, e.g. integers, floating point numbers, strings etc.
- Instruct the Communication object to transmit the message with a given identifier

Once the message is transmitted, attempting to use the message handle will result in a run-time exception; eventually the message handle will be re-used. This mechanism offers efficient access to the communication media as it allows numerous threads to transmit messages simultaneously. The only mechanisms which must be completely synchronised are requesting a handle to a thread and transmitting the data through the underlying protocol. Adding data to a message must only be synchronised around the message itself, allowing other threads to simultaneously access other messages. It is possible to remove the need for synchronising adding data to a message if it can be guaranteed that a message will only be accessed by one thread, which is the case in the current system.

4.3.1.2 Auxiliary Components

Distributed collision detection utilises a number of auxiliary components to facilitate state replication between nodes participating in the DVE. The most important of these are the EnvironmentProperties, Object3D and ObjectFactory objects. The EnvironmentProperties contains all the information required to fully describe the virtual environment. This includes members such as:

- World unique identifier: For example, identify the geometry which describes the virtual world
- World bounds: a description of the volume of virtual space the world occupies

- Participating nodes: All the collision detection nodes and clients currently participating in the DVE
- Participating objects: All the objects currently inhabiting the DVE
- Server and Peer CommunicationFactory objects

The EnvironmentProperties type implements the Communicable interface. An instance of EnvironmentProperties is created by the Server at DVE instantiation and is updated when objects, collision detection nodes or clients join or leave the DVE. The server is responsible for ensuring that all participants observe a reasonably up-to-date EnvironmentProperties object. The EnvironmentProperties type can be extended to include application-dependent information which must be disseminated to all participants of the DVE. It contains two CommunicationFactory objects: one for server communication and one for peer communication. This enables the appropriate levels of reliability to be employed in different communication scenarios. For example, it may be desirable to have low-latency unreliable messaging between peer collision detection nodes while maintaining slower, reliable (or best-effort reliable) messaging between the collision detection nodes and the server.

The Object3D type represents an object which inhabits the DVE. It contains the objects' geometry, collision data, textures, physics and behavioural properties. In addition, the Object3D type can be extended by defining a sub-class of the Object3D type which contains application-specific information.

The ObjectFactory is responsible for replicating objects across all nodes participating in the DVE. It provides functionality to take a Communication stream describing an object and return a reference to an instance of an Object3D representing the object which the Communication stream described. If an Object3D corresponding to the data contained in the Communication stream does not already exist, the ObjectFactory creates an instance of an Object3D to correspond with this object. Alternatively, if an Object3D corresponding with the data is already in existence, the Object Factory returns a reference to the

existing object. The Object Factory also acts as a resource monitor and is capable of re-cycling geometry information such that if two Object3Ds represent the same type of object in the DVE, e.g. the same make and model vehicle, the Object3Ds will reference the same geometry, texture information and some of the same collision data. This significantly reduces the amount of memory the DVE occupies. In order to implement this efficiently, it is necessary for the Object Factory to be able to efficiently find Object3D instances, geometry and texture information. In order to achieve this, a combination of hash maps and binary search are utilised to provide extremely fast searching performance.

It is assumed that the server, clients and collision detection nodes will all utilise compatible Object Factory instances. It is also assumed that the data describing objects' geometry will be stored locally on the server, collision detection nodes and clients. Streaming geometry information is not forbidden by the system described in this thesis, but it has not been implemented in the current system because streaming geometry data may incur significant communication costs as next-generation platforms are capable of rendering objects constructed of large numbers of polygons which would consume a large amount of network bandwidth. Therefore, streaming geometry data may detrimentally affect the responsiveness of the DVE.

4.3.1.3 Allocating Work to Collision Detection Nodes

The basic requirements of the server have already been discussed and have been categorised into client and collision detection node issues. The main work of the server is to ensure that the collision detection nodes perform collision detection efficiently on consistent and up-to-date object state information. In order to achieve this, the server is responsible for managing consistency group membership and routing client update messages to the group leaders of the consistency groups. While communication in normal situations occurs between the server and each consistency group's group leader, it is necessary for the

server to be able to receive messages from all collision detection nodes participating in the DVE to overcome group leader failures.

In order to efficiently utilise the collision detection nodes, each member of a consistency group is allocated a sub-region of the DVE to perform collision detection within. This is achieved by forming a spatial subdivision hierarchy of the DVE and allocating a unique node in this hierarchy to each member of a consistency group. This subdivision hierarchy is termed a distribution tree and each node within the tree is termed a distribution node. Each consistency group has its own distribution tree. To ease clarity, the following section does not discuss the group leader's role in managing its group's distribution tree, which is a shared duty between the server and the group leader.

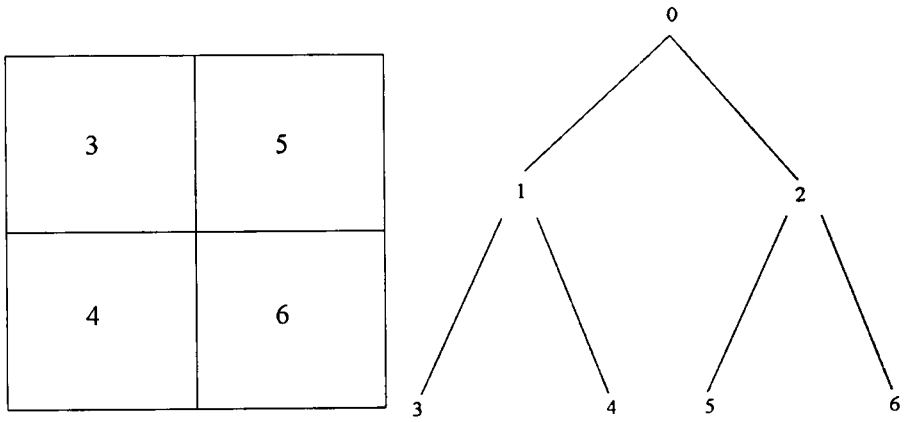

Figure 4.3 Uniquely Identifying a Sub-region

Each vertex in the tree is given a unique identifier. The root is given the Id 0. All other vertices are given Ids based on their parent node's Id:

Left child = 2 * Parent Id + 1

Right child = 2 * Parent Id + 2

This unique vertex identification allows each sub-region of the DVE to be identified by the server and collision detection nodes. All internal vertices store the criteria on which the world is to be subdivided, e.g. a partitioning plane.

Leaves can be used to store the objects which inhabit the corresponding sub-region of the DVE. From ***Fig 4.3***, it can be seen that vertex 0 represents the vertical partitioning plane separating vertices 3 and 4 from 5 and 6. Vertices 1 and 2 represent the horizontal partitioning planes separating vertex 3 from vertex 4 and vertex 5 from vertex 6 respectively.

Each collision detection node in a consistency group is represented within the consistency group's distribution tree by a DistributedServer object which is placed in the distribution node corresponding to the sub-region the collision detection node is responsible for. For example, a DistributedServer object placed at node 0 would be responsible for collision detection on objects contained in vertices 3, 4, 5, and 6, a DistributedServer at vertex 1 would be responsible for collision detection on objects contained in vertices 3 and 4 and a DistributedServer at vertex 5 would be responsible for collision detection only on objects contained in vertex 5 (or any of its children vertices if it was subdivided further). Each collision detection node is assigned a unique identifier by the server when it joins the DVE, which is stored in the DistributedServer object. The DistributedServer stores the communication information for the collision detection node and a list of objects the collision detection node is responsible for. These objects are categorised into two different types, persistent objects and client objects. The state of persistent objects is maintained by the collision detection nodes as their behaviour is controlled completely within the DVE software. It is necessary, however, to provide the relevant collision detection nodes with frequent state updates for the client objects which they are responsible for collision detection upon. As such, a collision detection node is informed of the current state of all persistent objects it is responsible for each time its consistency group is re-allocated. Following this, the state of persistent objects are maintained by the collision detection nodes within the consistency group; this will be discussed in more detail later. Objects within the DVE are referenced by two unique identifiers; the first being the index of the client who hosts the object and the second being the object's unique identifier issued by the

client client. If the object is a persistent object, the client identifier is -1 and the unique object identifier is provided by the server when the object is instantiated.

The binary tree structure lends itself towards this form of workload distribution. If a higher-order tree structure was selected, such as a tree with 4 children per vertex, the distribution of workload would be far more complex. Such a tree structure would be capable of handling situations in which the number of members in a consistency group is a power of 4, e.g. 1, 4, 16, 64 etc. However, given a more difficult number, e.g. 5, it would be necessary to place the same DistributedServer in more than one place in the tree. ***Fig*** 4.4 demonstrates the difference between a binary tree and a quad-tree for use in allocating collision detection nodes' sub-regions of a DVE. It can be seen from the diagram that a DistributedServer will appear at most once in a binary distribution tree, whereas a DistributedServer may appear more than once in a distribution tree with 4 child nodes per node. ***Fig*** 4.4 also shows the shortcomings of the tree representation, in that collision detection nodes may not be responsible for equal portions of the DVE. It can be seen from the diagram that collision detection nodes 1, 2 and 3 are responsible for regions of the world twice as large as nodes 4 and 5. This could result in nodes 1, 2 and 3 being required to do more work than nodes 4 and 5. However, the amount of work required to be completed by these nodes is still a fraction of the work required to be performed if distribution collision detection was not employed; in the case of ***Fig*** 4.4, nodes 1, 2 and 3 are required to perform collision detection for a quarter of the DVE and nodes 4 and 5 an eighth of the DVE respectively.

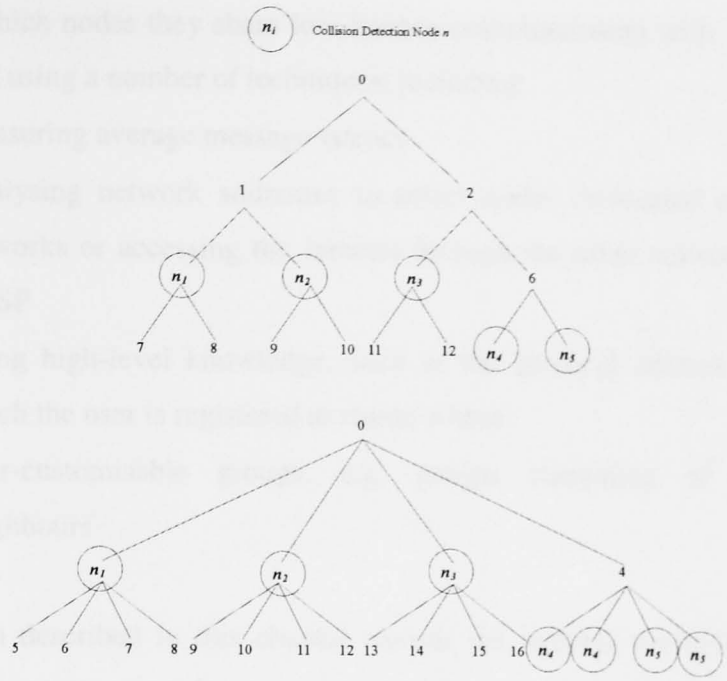

Figure 4.4 Binary Tree vs Higher-order Tree

4.3.1.4 Forming Consistency Groups

The server is responsible for organising consistency group membership. This involves re-allocating consistency groups when a new collision detection node joins the DVE or an existing node leaves. In addition, consistency groups must be modified during run-time as a result of membership votes instigated by the collision detection nodes. As discussed in Chapter 3, it is desirable to generate as large consistency groups as possible to provide the best possible performance and consistency in the DVE. However, if consistency groups become too large, the group leader may become overloaded; this problem will be discussed later when group leaders are examined in more detail.

Members of consistency groups must be able to communicate with one-another through low-latency network connections. When the server instantiates a consistency group re-allocation, it requests all collision detection nodes to

estimate which nodes they share low-latency communication with. This can be determined using a number of techniques, including:

- Measuring average message latency
- Analysing network addresses to select nodes co-located on the same networks or accessing the Internet through the same network exchange or ISP
- Using high-level knowledge, such as the physical address (town/city) which the user is registered to reside within
- User-customisable groups, e.g. groups consisting of friends or neighbours

The system described in this chapter records the average round-trip message delay over a given time-frame and uses this to approximate the appropriate group memberships. This allows group membership to adapt depending on network behaviour, e.g. network congestion. However, analysing network performance may be inappropriate in massively multiplayer DVEs, where alternative approaches may be more suitable. The use of high-level knowledge, e.g. registered address of the user, relies on the user being honest and keeping their address up-to-date. Similarly, user-customisable groups rely on the users understanding that they are creating groups of nearby users; a group consisting of members in distant countries will result in extremely poor performance. However, a mixture of these techniques can be used, where user-customisable groups, physical addresses or network address analysis can be used to prune the set of collision detection nodes into smaller sets of candidate consistency groups. This subset would allow less expensive network analysis to be performed.

Once the server has received a list of candidate group members from each collision detection node, it is necessary for the server to create groups of nodes who appear in each-other's list of members. In order to achieve this, the server creates a set of all possible groups from this data, sorted from largest group to

smallest group. For efficiency reasons, this set of possible groups avoids creating subsets of groups already created to save memory. For example, given a server hosting a DVE with 5 collision detection nodes, *A*, *B*, *C*, *D* and *E* that return the candidate group members:

A = {B, C, D}

B = {A, C, E}

C = {A, D}

D = {A, C, E}

E = {D}

The set of candidate groups created would be:

{{A, C, D}, {A, B},{D, E}}

This set is then used to create the actual consistency groups. The algorithm to do this is outlined below:

```
function createGroups(List candidateGroups)
{
  List groupMembers = getGroupMembers();
  while(!groupMembers.isEmpty())
  {
     Sort(candidateGroups);
     Group group = (Group)candidateGroups.removeFirst();
     MakeGroup(group);
     RemoveInstanceOf(group, groupMembers);
     RemoveInstanceOf(group, candidateGroups);
  }
}
```

The above algorithm selects the largest group from the candidate groups. It then removes the members of the new group from the list of unallocated collision detection nodes. All instances of the members of the new group are removed from the remaining candidate consistency groups; if this results in empty candidate groups, then these are discarded from the list. Following this, the list is re-sorted to ensure that the first element is the largest consistency group. If there are remaining unallocated collision detection nodes, the process is

repeated. Given the previous example, this algorithm would create the consistency groups: *{{A, C, D}, {B}, {E}}*.

4.3.1.5 Run-Time Consistency Group Adjustments

The previous re-allocation occurs when a new node joins the DVE or an old node leaves. During the run-time of the DVE, members of a consistency group can request for members of their group to leave the group as a result of poor communication performance, or can request another node to join their group as a result of fast communication perceived between themselves; nodes can only join a consistency group at least as large as the group they are currently in. In order to avoid nodes repeatedly requesting nodes to join and leave their group, two threshold values are defined: the *inclusion* and *rejection* values, where *inclusion* < *rejection*. This means that a node will only request an inclusion vote for a node if it perceives an average transmission latency with the node less than the inclusion threshold. Similarly, a node will only request a rejection vote if it perceives an average transmission latency with a member of its consistency group larger than the rejection threshold. In addition, the server can choose to ignore requests for inclusion/rejection votes, e.g. limit the maximum number of votes in a given time-frame.

When an inclusion vote is initiated, all members of the group must vote to determine if the node should be included in the consistency group. Each member can vote to include the node, reject the node or abstain from the vote if the transmission latency they perceive makes their decision unclear. To clarify:

- If a node perceives a transmission delay less than the inclusion threshold, it should vote for inclusion
- If a node perceives a transmission threshold greater than the rejection threshold, it should vote for rejection

- If a node perceives a transmission delay between the inclusion and rejection thresholds, it should abstain from voting

When all votes are received, if no group member rejects the node and at least one member accepts the node, the proposed collision detection node will join the consistency group. Similarly, when a rejection vote is initiated, each group member (except the node which is proposed to be rejected) must decide whether the node should be rejected from the group using a similar mechanism as the inclusion vote:

- If a node perceives a transmission delay less than the inclusion threshold, it should vote against the rejection
- If a node perceives a transmission threshold greater than the rejection threshold, it should vote for the rejection
- If a node perceives a transmission delay between the inclusion and rejection thresholds, it should abstain from voting

Once all results have been received, the decision to reject the node is based on a weighted average of the votes. Given I inclusion votes, R rejection votes and A abstain votes, the node will be rejected from the group if:

$I + 0.5A - 2R < 0$

4.3.1.6 Providing the Client with State Updates

The Client is responsible for generating frequent state updates and transmitting these to the server or group leader of its consistency group. Upon receipt of these, it is necessary for the group leader or server to ensure that these messages are disseminated to the other consistency groups and collision detection nodes which must simulate these objects. When the state of client objects are changed as a result of a collision, it is necessary for the server/group leader to inform the client hosting this object of the event. This will be further discussed later when group leaders are discussed in detail.

In a real-world DVE, it is also necessary for the server/group leader to inform all clients of the current state of the objects which are visible to them. This is beyond the scope of this thesis, but it is necessary to ensure that users experience a consistent DVE. Considerable research has been undertaken into this problem and a number of techniques have been developed to ensure that users receive object state updates, while reducing the volume of messages which must be transmitted.

4.3.1.7 Summary of the Server

The architecture and responsibilities of the server in distributed collision detection have been introduced. The mechanisms by which the server allocates work to the collision detection nodes, allocates collision detection nodes to consistency groups and communicates with the clients were discussed. While these mechanisms are required to manage distributed collision detection, they are relatively expensive operations. As the number of collision detection nodes in the DVE increases, this may overload the server. As such, the notion of group leaders, which were introduced in Chapter 3 and earlier in this chapter, will be further examined later in this chapter as a mechanism whereby the duties of the server are distributed among certain designated collision detection nodes. This helps to reduce the server's processing overheads. In addition, while it is beyond the scope of this thesis, the techniques for distributed collision detection discussed in this chapter are not restricted to single-server architectures. The server described in this section can be implemented using server hierarchies or groups of peer servers to distributed the workload appropriately and increase scalability.

4.3.2 Collision Detection Nodes

The collision detection nodes are responsible for detecting and responding to collisions and ensuring that the collision events are disseminated to the relevant clients to ensure a consistent and responsive DVE is maintained. The underlying theory behind the collision detection nodes was introduced in Chapter 3. The following section will discuss the structure required to implement collision detection nodes capable of adapting to changing network properties.

4.3.2.1 Overview

The Collision detection nodes are constructed from a number of components, some of which are common to both the collision detection nodes and the server. Both the server and collision detection nodes share a common communication model and utilise the DistributedServer class to store references to collision detection nodes.

The collision detection nodes consist of two main components, the collision detection component and the communication model component. These are kept separate by using a class hierarchy:

- The collision detection components of the collision detection nodes are implemented within a class called DistributionNode
- The communication model components are implemented in a class which extends DistributionNode called DistributedNode

4.3.2.2 DistributionNode

The DistributionNode represents a node within a binary tree representing the DVE called a distribution tree. Distribution trees are constructed on each

collision detection node to represent the sub-region of the DVE which they are responsible for collision detection within and the sub-regions of the DVE other nodes within their consistency group are responsible for. Essentially, this structure mirrors the distribution tree stored on the server for each consistency group. Each internal node in this tree represents a partitioning plane which subdivides the virtual world into two sub-regions. This subdivision is performed recursively until a number of termination criteria are reached:

- Size of the sub-region is less than some threshold value
- Depth of the tree is greater than some threshold value
- Number of leaf nodes is greater than some threshold value

These threshold values can be defined on an application-dependent basis. As this subdivision is to be adjusted dynamically, the subdivisions are performed using axially-aligned partitioning planes to yield fast partitioning performance.

Leaf nodes in the distribution tree contain a list of objects which occupy the enclosed sub-region of the DVE. Each DistributionNode contains a reference to a DistributedServer. If this reference is not null, this implies that the sub-region represented by this node (and all of its descendent nodes) is controlled by the collision detection node described by the DistributedServer. The local node is responsible for collision detection in any leaf node whose descendant nodes all contain null references to a DistributedServer.

The collision detection node is initially given a list of objects which it is responsible for performing collision detection on by the server or group leader. It maintains this list of objects and inserts these objects into their respective nodes within its distribution tree. This is achieved by traversing the distribution tree for each object and depositing the object in any leaf node which it reaches. If tree traversal for a given object reaches a node which references a valid, non-null DistributedServer, the object is inserted into the DistributedServer's list of objects. This implies that the collision detection node corresponding to the

DistributedServer is responsible for collision detection on this object; the corresponding collision detection node will be informed of the state of the object through the communication model, which will be described in the DistributedNode class later. The descendants of a node with a non-null DistributedServer reference are not traversed as the local node is not responsible for collision detection on that region. If no leaf node is reached in a given object's tree traversal, this implies that the local node is no longer responsible for collision detection for that object, and the duties of collision detection for that object are relinquished to other collision detection nodes in the DVE. This mechanism ensures that if a pair of nodes are responsible for collision detection on a given object, each node will receive state update messages for each nodes' perceived state of the object so that any inconsistencies can be overcome. Similarly, when a node is no longer responsible for collision detection on an object, the state of this object is transmitted to the nodes which are required to take over control of the object. The DistributionNode class does not manage object ownership itself; this is the responsibility of the DistributedNode. The DistributionNode simply provides methods to insert objects into their correct place in the distribution tree. Pseudocode for the insertion algorithm is provided:

```
function insert(DistributionNode node, Object obj)
{
  if(isLeafNode(node))
  {
    node.objs.add(obj);
    return;
  }
  //If this node contains a server
  if(node.server != null)
  {
     node.server.insert(obj);
     return;
  }
  if(node.child[0].contains(obj))
     insert(node.child[0], obj);
  if(node.child[1].contains(obj))
     insert(node.child[1], obj);
}
```

In addition, the DistributionNode contains a number of performance-enhancing optimisations. The root node of the distribution tree contains a quick-search reference to the DistributionNode which represents the root of the sub-region which the local node is responsible for. In addition, the DistributionNode contains a method by which a list of all leaf nodes below a given DistributionNode can be retrieved; this list can be created once and stored for quick access. This list is stored in an array, in which leaf nodes are stored in sequential memory addresses. This provides high-levels of cache-coherence to help improve collision detection performance.

Collision detection between objects is performed on a per-leaf node basis. This utilises the flattened list of leaf nodes. The algorithm for this is provided below:

```
function collisionDetection(List leafNodes)
{
   for(int a = 0; a < leafNodes.size; a++)
   {
      DistributionNode node = leafNodes.get(a);
      for(int obj1 = 0; obj1 < node.numObjs - 1; obj1++)
      {
         for(int obj2 = obj1+1; obj2 < node.numObjs; obj2++)
         {
            Point p = node.objs[obj1].pos + node.objs[obj2].pos;
            p/=2.0;
            if(node.contains(p))
            {
               if(node.objs[obj1].collide(node.objs[obj2]))
               {
                  //Collision response
               }
            }
         }
      }
   }
}
```

The collision detection algorithm essentially iterates through all leaf nodes. In each leaf node, all objects are compared with one-another using a brute force

algorithm. However, each pair of objects is only compared with one-another if the centre point between the two objects is inside the sub-region represented by the leaf node. This ensures that the same pair of objects is not compared for collision with one-another more than once. The algorithm described in this section is simplified. For example, rather than simply returning a Boolean, the collision detection algorithm used returns an approximation to the point of contact or points of contact, depending on the parameters passed to the algorithm. These points of contact can be used to determine how the collision should be responded to accurately using a dynamic simulation.

The spatial subdivision approach initially subdivides the DVE into uniform-sized discrete regions. However, the distribution of objects throughout the DVE may not be uniform. To reflect this, each collision detection node's distribution tree is dynamically updated to reflect the distribution of objects. To clarify, a leaf node is further subdivided if the number of objects in the sub-region corresponding to the leaf node is larger than some threshold value. Similarly, if the number of objects in a pair of peer leaf nodes' sub-regions is fewer than some threshold value, these two leaf nodes are merged together into a single node. This form of adaptive spatial subdivision allows the distribution tree to evolve to reflect the distribution of objects. However, to avoid nodes being repeatedly merged and split, the threshold values used for merging and splitting should be different, where *Threshold(merge)* < *Threshold(split)*. The optimal values for these thresholds are platform-dependent, as different machine architectures may perform better with different threshold values; these values are affected by machine-specific factors, including as CPU cache size and BUS bandwidth.

4.3.2.3 Narrow Phase Collision Detection

The distributed collision detection system presented in this thesis allows a wide-range of narrow phase collision detection algorithm to be employed. However,

the default algorithm used in this system is a BSP tree-based time-dependent approximation collision detection algorithm. The algorithm is capable of determining the exact points of contact to the sub-polygon level. However, it adapts the level of detail of collision detection to reflect the amount of time collision detection took in previous time-steps as, due to temporal coherence, the number of collisions detected at time t are likely to be similar to the number of collisions detected at time $t+\Delta$, provided Δ is small.

The developer or user is required to provide a duration by which collision detection is required to have completed, e.g. 10ms. This value is used to tune collision detection accuracy by reducing or increasing the depth of the tree that can be traversed before an approximation to the point(s) of contact can be found. The narrow phase algorithm takes the following parameters:

- A flag indicating termination of collision detection when a given number of points of contact are found
    - If the flag less than 1, this indicates all points of contact should be found
    - If the flag is greater than or equal to 1, collision detection is terminated when this number of points of contact are found
- An integer indicating the maximum depth of traversal
    - If the value is -1, the tree can be traversed to any depth

The flag indicating the maximum number of points of contact can greatly reduce the time taken for collision detection in certain cases. For example, if two objects are completely intersecting one-another, finding all points of contact may be very time consuming, whereas finding just one point of contact will be very fast. The integer stating the maximum depth of tree traversal can also affect collision detection performance. For example, if collision detection is allowed to progress to depth 12, up to $2^{12} - 1$ nodes could be passed through prior to collision detection completing. Therefore, if the maximum depth is decreased by 1, the number of nodes being passed through can be halved. If the objects are

found not to be intersecting in the early stages of tree traversal, the tuneable variables will have little effect on collision detection performance. However, if the objects are found to be intersecting, both of these variables can significantly speed up intersection tests at the cost of accuracy. Restricting the number of points of contact may result in:

- Noticeable jittering and failure of an object to reach restitution in a dynamic simulation
- Undesirable or inaccurate response to collision

Reducing the maximum depth of tree traversal may result in:

- Undesirable or inaccurate response to collisions
- False positives being returned, where close but non-intersecting objects are deemed to being colliding

However, these inaccuracies largely go unnoticed as humans are usually unable to recognise unrealistic physics response. In addition, the problem of objects not reaching restitution can largely be ameliorated by freezing objects when their angular and linear velocity falls below some threshold value. This, in fact, must usually be performed even with the most accurate collision detection and dynamic simulation; due floating point rounding errors it is often impossible to guarantee an object controlled by a dynamic simulation will reach complete restitution without the use of such approaches.

The narrow phase algorithm used in the system described in this thesis adapts both of these level-of-detail variables to provide accurate collision detection for the objects occupying the DVE while attempting to reduce the processing time required for collision detection below a threshold value. If it is not possible to reduce the time below a threshold value, both variables will be fixed at their lowest detail setting. Conversely, if the node is capable of performing collision detection in time shorter than the threshold time, the detail settings will be fixed to their largest setting. Commonly, however, the detail levels will fluctuate as a result of the objects' behaviours within the DVE.

The collision detection algorithm uses just-in-time polygon and bounding volume transformations to ensure that vertices are only transformed if they are required to be compared for intersection. In addition, the algorithm halves the number of transformations required by utilising inverse transformation matrices. The inverse transformation matrix for the object with the most polygonal complexity is determined. This inverse matrix is multiplied with the other object's world transformation matrix and used to transform the less complex object's vertices into the more complex object's local space. The resulting transformation matrix is stored and used only when bounding volumes or primitives must be compared for intersection. The results of this transformation are stored, along with a timestamp, to ensure that the polygon is transformed at most once in any given intersection test.

The use on-the-fly transformations usually results in far fewer transformations being performed than would be required if all vertices and bounding volumes were transformed. This is because, in most cases, either the objects do not intersect or only demonstrate a small degree of intersection. The use of inverse matrices further reduces the number of transformations that must be performed.

4.3.2.4 DistributedNode

The DistributedNode class extends the DistributionNode class to provide a mechanism for communicating between collision detection nodes and between collision detection nodes and the server. The DistributedNode occupies the root position of each collision detection node's distribution tree; each descendant node of the root node is a DistributionNode, as described previously. The structure of the distribution tree can be seen in ***Fig*** 4.5.

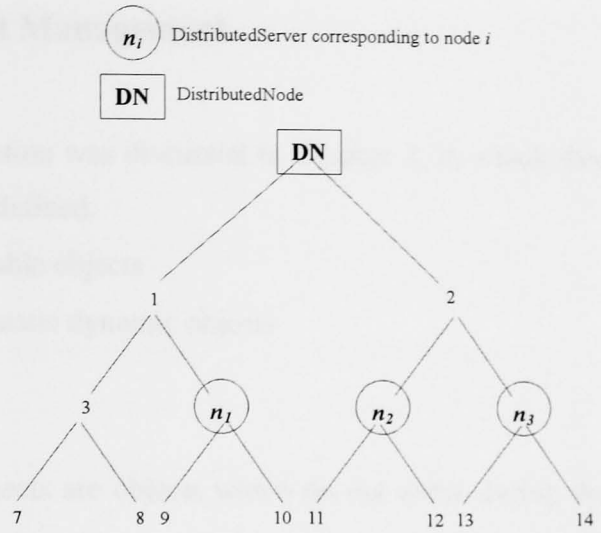

Figure 4.5 Collision Detection Node's Distribution Tree

Fig 4.5 shows a DistributedNode at the root of the distribution tree. All descendent nodes are of type DistributionNode. The tree shows that the local node is responsible for collision detection on the sub-region of the DVE corresponding to the internal node with id 3. In addition, there are three other collision detection nodes within the consistency group responsible for collision detection on the sub-regions corresponding to nodes 4, 5 and 6 respectively.

The DistributionNode class is essentially responsible for adaptively subdividing the DVE into sub-regions, placing the objects in their respective sub-regions and invoking collision detection between objects sharing the same sub-regions. The DistributedNode class is responsible for:

- Object management
- Object replication and transfers
- Consistency group performance monitoring
- Communication with group leader and server

4.3.2.5 Object Management

Object classification was discussed in Chapter 3, in which three different types of objects were defined:

- Immoveable objects
- Deterministic dynamic objects
- Avatars

Immoveable objects are objects which do not move during the life-time of the DVE; this includes objects which constitute the fixed environment, such as floors, walls, ceilings, staircases etc. Due to the fixed nature of these objects, they can be treated differently from other objects in the DVE by being placed into their respective sub-regions at initialisation and never need to be re-positioned.

Deterministic dynamic objects are objects which move throughout the DVE in response to collisions or through application-dependent AI routines. The behaviour of these objects can be deterministically modelled and, as such, can be relatively accurately replicated between collision detection nodes and consistency groups.

Avatars, conversely, demonstrate behaviour which is non-deterministic as they are controlled by external users. Therefore, the state of avatars cannot be accurately modelled or replicated without the use of high-fidelity message exchange between the collision detection nodes and the client which hosts the avatar in question.

The DistributedNode, Group Leader and Server treat these objects differently. Upon joining a DVE, the DistributedNode is informed of all immoveable objects in the DVE. As these objects will not move during the course of the DVE, these

objects will always be completely consistent between all nodes participating in the DVE.

Deterministic dynamic objects are assigned to collision detection nodes by the server during consistency group reallocation. After this, the collision detection nodes in each consistency group are responsible for managing the state of these dynamic objects without the need for intervention from the server. The collision detection nodes in a consistency group are required to transmit object replication and transfer messages between one-another to ensure that the appropriate nodes accept responsibility for collision detection during the lifetime of the DVE. Upon response to collisions, collision detection nodes are required to transmit the updated state of the colliding objects to their group leader; the group leader will, in turn, handle inconsistencies in object states within its consistency group and forward the internally-consistent state on to the server and clients. The server will, following receipt of new object states, resolve any inconsistencies between consistency groups and store the current state of all objects in the DVE. Collision detection nodes will be informed of up-to-date object states by the server during group reallocation or in the event of node failure. This mechanism reduces the server's processing and communication requirements by allowing a degree of inconsistency to emerge between dynamic deterministic objects between consistency groups.

Collision detection nodes are assigned avatar objects through the use of frequent state update messages transmitted by the group leader to its group members. The group leader receives avatar state updates from a combination of the server and the clients which are connected directly to it. To clarify, a client is a process which provides a graphical user interface to the user and disseminates the current state of the avatars it hosts. Clients may be hosted on collision detection nodes. If a client is hosted on a collision detection node, the client transmits its frequent state updates to its consistency group's group leader; if the client is hosted on the consistency group leader, the state updates are performed using

shared memory updates on the object state. If the client is not hosted on a collision detection node, its state updates are transmitted to the server. When a group leader receives state updates directly from a client, the group leader forwards these state update on to the server, which transmits the state update to any other consistency groups in the DVE through their group leaders. Upon receiving avatar state updates, the group leader transmits the latest avatar state to the collision detection nodes responsible for collision detection upon it. Additionally, the state update is transmitted to any collision detection node previously responsible for collision detection on the object to ensure the collision detection node acknowledges the object's new state and relinquishes responsibility for its collision detection.

Collision detection nodes retain the state of avatar objects between frequent state updates. The state of these objects can be updated using extrapolation techniques, such as dead reckoning, in an application-dependent manner. When a collision is responded to, the state of the objects involved in the collision are transmitted back to the consistency group leader. The group leader transmits the updated state of the avatar in the following way:

- If the avatar is hosted on a client which is directly connected to this group leader
    - Transmit the updated state to the client
- Transmit the updated state of the avatar to the server

Upon receiving an updated avatar state, the server responds accordingly:

- If the client hosting the avatar is communicating directly with the group leader who transmitted the message
    - Record the state update and disseminate it to the other group leaders for use in subsequent collision detection
- If the client hosting the avatar is not communicating directly with any group leader
    - Record the state update and disseminate response back to client

- Otherwise, do not record the state update, as the client hosting the avatar is engaged in high-fidelity communication with another group leader. This implies that the client's group will have access to more up-to-date state information and, as such, will provide more accurate and consistent collision results.

The server, clients and collision detection nodes employ message buffering techniques, where non-admin messages are transmitted at regular frequencies. To clarify, rather than transmitting a message whenever a collision event is detected, the collision detection node stores a list of objects which have been involved in collisions since the last message was sent. After a given time has passed since the transmission of the last message, the collision detection node will transmit the states of any objects involved in collisions. This mechanism better utilises the available bandwidth by avoiding sending bandwidth-expensive short messages. In addition, it reduces the computational overhead associated with message transmission. Similar techniques are employed in the server and group leader processes.

4.3.2.6 Object Replication and Transfers

Each collision detection node is responsible for object replication and object transfers within its consistency group. Object replication occurs when two or more collision detection nodes are responsible for collision detection on the same deterministic dynamic object. Object transfers occur when a deterministic dynamic object is passes from a sub-region controlled by a given collision detection node to a sub-region controlled by another collision detection node.

Object replication and transfers are performed using similar techniques. Both involve the direct communication between collision detection nodes, whereby a message is transmitted from one node to another informing the recipient of the current state of an object as perceived by the message originator. These

messages are transmitted whenever an object is passes through a node in a collision detection node's distribution tree which contains a non-null DistributedServer reference. If the object in question also occupies one or more sub-regions which the local collision detection node is responsible for, the local node retains a reference to the object. If the object no longer occupies any sub-regions which the local node is responsible for, the local node relinquishes responsibility for the object, which is transferred to the recipients of the message.

This mechanism requires that all members of a consistency group have sufficient knowledge of their group members to be able to transmit messages to one-another. Given that collision detection is performed at a relatively high frequency, as an object travels from one node's sub-region to another's, the object will most likely be replicated for a number of collision detection iterations before it is transferred completely. If the frequency of collision detection could be guaranteed to be sufficiently quick that an object could not pass completely through a sub-region, it would be possible to reduce the amount of information about its group a collision detection node is required to know. In this situation, a collision detection node must only know about the nodes responsible for sub-regions neighbouring the regions it is responsible for collision detection upon. However, the frequency of collision detection is largely application dependent and, therefore, such conditions cannot be assumed to hold.

4.3.2.7 Consistency Group Performance Monitoring

In order to adapt consistency group membership during run-time to reflect system bottlenecks and network behaviour, it is necessary for the collision detection nodes and server to continually gather performance metrics. Each collision detection node records average message round-trip delays between itself and the other collision detection nodes participating in the DVE. This

information is used to determine whether a collision detection node is capable of communicating another node sufficiently quickly to be part of the same consistency group. This information is used to create a list of candidate group members when requested to do so by the server; this occurs whenever a node joins the DVE or a membership vote is initiated.

Each collision detection node transmits a ping message at a fixed frequency and measures the time taken to receive a response from each node. The average time is constructed over the most recent n messages, where n is an application-dependent variable. The average message transmission delay is analysed as follows:

- If the pair of nodes currently share a consistency group
    - If *average message delay* > *rejection threshold* or *instantaneous message delay* > *rejection threshold* + *tolerated deviation*
        - Instantiate rejection vote for object
- If the pair of nodes do not share a consistency group
    - If *average message delay* < *inclusion threshold*
        - Instantiate inclusion vote for object

Once a vote has been instantiated, the collision detection nodes must use their most recent message transmission delays to determine whether a node should be included or rejected from a consistency group.

4.3.2.8 Group Leader

The group leader is an additional processing thread which is initialised on a collision detection node when it is promoted to group leader by the server. The choice of group leader is arbitrary; the server in described in this chapter selects the node in the consistency group with the smallest id, i.e. the consistency group member who joined the DVE earliest.

The group leader thread receives messages from:

- Clients providing avatar state updates
- Collision detection nodes providing collision responses
- The server providing avatar updates

The group leader is responsible for managing avatars within its consistency group and for collecting the results of collision detection and disseminating this to the server and clients directly connected to it. In order to achieve this, the group leader process is continually listening for messages from its group members. This can help to increase the scalability of the server at the cost of the performance of the collision detection nodes which are promoted to group leader. The benefit of group leaders is that the group members should exhibit low-latency network connections between themselves and their group leader. This will ensure that update messages within a consistency group are delivered to the appropriate clients and collision detection nodes quickly enough to provide high-levels of interactivity and consistency to the group members. The group leader strategy, however, can introduce additional delays in the receipt of messages originating from outside of the consistency group. However, this additional delay is minor and it is possible that messages may in fact be delivered more quickly as a result of the server's reduced message transmission requirements.

4.3.2.9 Collision Detection Node Joining the DVE

It is necessary to describe the steps required for a new collision detection node to join a DVE. Initially, the new collision detection node transmits a connection request to the main server of its desired DVE. The server responds by allocating the collision detection node a unique identifier. It then informs the existing collision detection nodes in the DVE of the presence of the new collision

detection node. During this time, the new collision detection node opens a ServerCommunication object which listens for incoming connection attempts from the existing collision detection nodes.

Each existing collision detection node transmits initial handshake messages to the new node and, from these and an initial ping request, predicts whether or not the new node could be part of its consistency group. Upon receiving these initial potential group allocations, the main server reallocates group membership. While this reallocation is being performed, any consistency groups whose membership is being altered are instructed to perform collision detection locally. While they perform local collision detection, additional threads perform the required handshaking to initiate communication between themselves, their group members and the group leader. Once this handshaking is completed, the group leader instructs the collision detection nodes to begin working together on collision detection.

4.3.2.10 Collision Detection Node Threads

The collision detection nodes, much like the server, are constructed from a set of components which are responsible for handling different classes of message exchange and responding to these messages appropriately. These components occupy different threads of execution and utilise shared memory to communicate between one-another and access shared objects. As mentioned previously, the collision detection nodes contain a distribution tree, where the root of the tree is a DistributedNode object and the remaining nodes in the distribution tree are DistributionNode objects. The DistributionNode objects are responsible for performing collision detection on the objects inhabiting the DVE. The DistributedNode inherits from the DistributionNode and provides the functionality to perform collision detection in a distributed manner. ***Fig*** 4.6 shows the data members and threads present in the DistributedNode object.

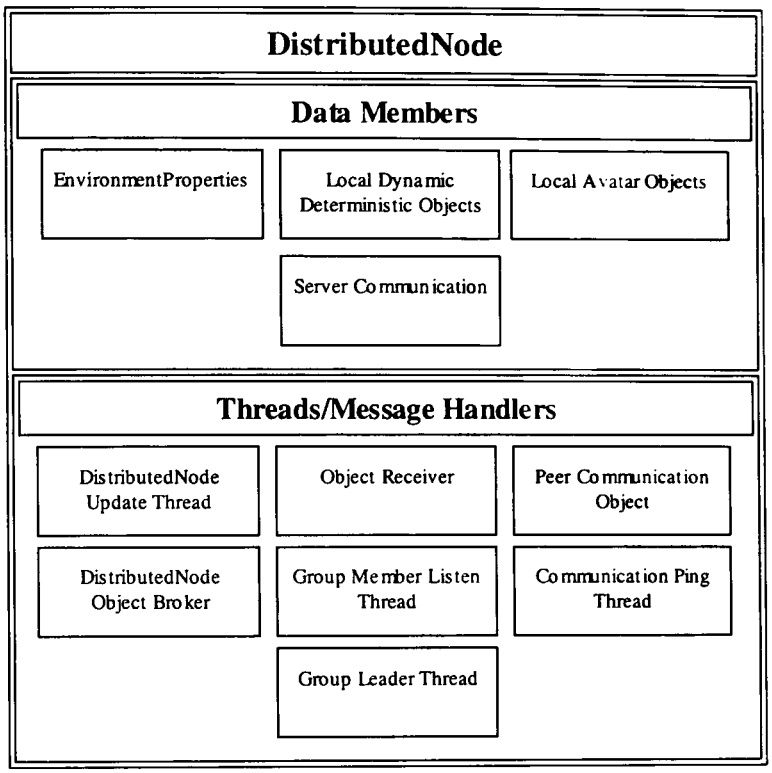

Figure 4.6 DistributedNode Architecture

The EnvironmentProperties object contains a list of all the collision detection nodes participating in the DVE, each referenced by a DistributedServer. It also contains references to all objects in the DVE, through the use of an ObjectFactory. The DistributedServer object contains the corresponding collision detection node's communication information. The DistributedNode contains the list of dynamic deterministic and avatar objects it is responsible for in the DVE. The list of immoveable objects is maintained in the EnvironmentProperties object and is pre-inserted into the distribution tree. The server communication contains the communication information for the DVE server; in practice, this is separated into admin and object communication information, which may be separated by, for example, the port number communication occurs upon.

4.3.2.10.1 DistributedNode Update Thread

The DistributedNode Update Thread listens for admin messages from the server. This includes the following messages:

- Update EnvironmentProperties
- Update EnvironmentProperties Light Weight
- New Client
- New Collision Detection Node
- Group Allocation
- DVE Exit

The Update EnvironmentProperties messages are a catch-all message type whereby the most recent EnvironmentProperties are transmitted from the server to the collision detection node. This can be performed using both light weight and standard mechanisms; the light weight mechanism does not encode data which should be constant, such as world bounds, world identifiers, immoveable and dynamic deterministic objects.

A New Client message informs the collision detection node of the inclusion of a new client and, possibly, new avatar objects into the DVE.

A New Collision Detection Node message informs the collision detection node of a new collision detection node joining the DVE. The receipt of this message implies that the new collision detection node is ready and waiting for connection attempts and ping requests; the steps required for a collision detection node to join the DVE were outlined previously. Following connecting to the new node, the nodes transmit potential consistency group allocations, which the server uses to reallocate the consistency groups within the DVE.

A Group Allocation message instructs the collision detection node to join the consistency group described in the message; this may cause the collision

detection node to become a group leader, cease being a group leader or connect to a new group leader. If the node is joining a group with the same group leader as the group it previously occupied, the node need do nothing. However, if it is joining a group run by a different group leader, the collision detection node must establish a Communication stream between itself and the new group leader.

A DVE Exit message instructs the node to exit from the DVE. This may be as a result of the node requesting to quit the DVE, in which case this message provides the node with permission to quite. Alternatively, it may be as a result of a server/DVE update or an irrecoverable error in the DVE which the server must re-start the DVE to resolve. The receipt of this message causes the collision detection node to close all Communication streams and exit the DVE.

4.3.2.10.2 Object Receiver

The Object Receiver is responsible for receiving avatar object state updates from the server. These state updates are only transmitted to the group leader. However, each node can potentially become the group leader, so each node maintains an instance of an ObjectReceiver at all times. Avatar state updates need only be received between collision detection iterations. As such, rather than executing in a separate thread, this is performed in the main thread, where the Communication stream is polled to determine if avatar update messages are waiting to be received. Avatar update messages are transmitted through a specific Object communication stream, which means that the presence of a message can be determined by testing to see if any data is currently unread in the stream. If an object update is received, the object update is passed to the ObjectFactory to find an existing reference to the object and update its current state; if no reference is found which matches the object, it is assumed that the object update has become corrupted during transmission and is discarded as the node should have received an admin message informing it of the presence of a new object prior to receiving the object's state update messages.

4.3.2.10.3 Peer Communication Object

The Peer Communication Object transmits object replication/transfer messages at regular intervals to the relevant collision detection nodes. The regularity of these messages are application-configurable; transmission frequencies can be defined in terms of time duration or number of collision detection iterations. The messages transmitted by the Peer Communication Objects are consumed by the corresponding collision detection node's DistributedNode Object Broker.

4.3.2.10.4 DistributedNode Object Broker

The DistributedNode Object Broker is responsible for receiving object state updates from the other collision detection nodes within a node's consistency group. This form of state update can be received between collision detection iterations. As such, the process is performed in the main thread of execution. The Communication streams which must be read from exclusively transmit object replications/transfers between collision detection nodes. As such, the presence of an object transfer/replication message can be determined by checking if there is unread data in the relevant Communication streams. If an object replication/transfer message is received, the data is passed to the ObjectFactory, which searches for the object and updates its current state; if the object is not recognised then the message is consumed and thrown away. If a valid object state update is received, the object which this message refers to is added to the list of objects which the collision detection node is responsible for.

4.3.2.10.5 Group Member Listen Thread

The Group Member Listen Thread receives messages from the group leader. These messages include:

- Node Update Messages

- Object Dispatch Messages

The Node Update Message is an aggregated state update message describing the objects which this node must know about but is not responsible for collision detection upon. This form of message is transmitted by the group leader when a new node joins the group leader's consistency group. This ensures that the new node's view of dynamic deterministic objects is consistent with the groups view of the objects; each group's view of the objects may differ.

The Object Dispatch Message contains the current state of the avatar objects the collision detection node is responsible for. Depending on the options prescribed for the collision detection node, this message can be used to completely synchronise the state of the DVE between all collision detection nodes in the consistency group. This can be achieved if each node waits for an object dispatch message before executing the next collision detection iteration. However, to increase the responsiveness of the DVE, an extrapolation techniques, such as dead reckoning, can be utilised to predict the position of avatar objects between object dispatch messages allowing the collision detection nodes to perform collision detection upon the objects between these messages.

If a collision detection node is promoted to group leader, its group member listen thread is disabled as it is not necessary to transmit/receive messages to/from itself.

4.3.2.10.6 Communication Ping Thread

The Communication Ping Thread is a low-priority thread which monitors ping performance between itself and other nodes in the DVE. Ping messages are transmitted at low-frequencies to all other collision detection nodes in the DVE. The frequency of ping message transmission is proportional to the average communication delay perceived between the collision detection nodes. To

clarify, collision detection nodes in the same consistency group will transmit ping messages and evaluate group membership with one-another more frequently than they will with collision detection nodes exhibiting large message transmission delays. Ping messages are sent at frequencies within a fixed interval, where:

- Pings to collision detection nodes whose transmission delays are less than some minimum threshold value are transmitted at the minimum interval
- Pings to collision detection nodes whose transmission delays are larger than some maximum threshold value are transmitted at the maximum interval
- All nodes whose transmission delays are between the minimum and maximum threshold values (*threshold*(min) and *threshold*(max) respectively) are transmitted at a proportional frequency. Given minimum interval *Ivl*(min) and maximum interval *Ivl*(max), the transmission frequency can be calculated using the following formula:

$$transmission(t) = Ivl(\min) + (Ivl(\max) - Ivl(\min))\frac{t - threshold(\min)}{threshold(\max) - threshold(\min)}$$

For example, given $t = 25$, *threshold*(min) = 10, *threshold*(max) = 1000, *Ivl*(min) = 5, *Ivl*(max) = 500, *transmission*(t) = 12.5 seconds

This mechanism helps to reduce the bandwidth consumption and computational expense of the ping messages. It is intended to react to changes in network behaviour as a result of network congestion, while specifically targeting the majority of network analysis between collision detection nodes exhibiting low-latency network transmission.

4.3.2.10.7 Group Leader Thread

The Group Leader Thread is responsible for receiving the results of collision detection from its group members. When a node is promoted to group leader, it opens up a Communication stream through which its group members can transmit messages directly to it. These messages include:

- Avatar Update Messages
- Collision Detection Results

Upon receiving an avatar update message, the group leader forwards the state of this object to the main server, which will in turn forward this message to the other consistency groups in the DVE. The group leader also uses this message to update its view of the object's state and informs the relevant nodes of the avatar's new state through an object dispatch message. However, to better utilise bandwidth, the group leader can choose to buffer incoming avatar update messages and dispatch update messages when the object's state deviates significantly from the previous state or the extrapolated state if dead reckoning is employed on the collision detection nodes; the use of dead reckoning can be enabled in the EnvironmentProperties object.

Upon receiving a Collision Detection results message, the group leader forwards the results to the main server. These results are also forwarded to the relevant clients. The group leader updates its view of the objects in the DVE and includes the updated avatar object states in any subsequent object dispatch messages. The group leader does not disseminate the updated states of the dynamic deterministic objects to the collision detection nodes as the collision detection nodes are responsible for maintaining each-other's consistent view of these objects through the use of object replication and object transfer messages.

4.3.3 Reliability and Fault Tolerance

Reliability and fault tolerance are extremely important issues in any distributed system. They are an open and popular topic of research which have received a great deal of interest. Reliability and fault tolerance are outside of the scope of this thesis and, therefore, will not be discussed in great detail. While a great deal of effort was expended to produce a reliable distributed collision detection approach, the reliability is based on the assumption that the main server is reliable and that admin messages transmitted between collision detection nodes, clients and servers are reliable. To clarify, a reliable server is a server which will not break; should the main server break, the DVE will come to an end. Mechanisms may be in place to roll back the state of the DVE to some recorded point prior to the server's failure, but this, again, is beyond the scope of this thesis. In addition, a reliable message is a message which will be delivered successfully to its intended recipient eventually. Both of these assumptions can not be guaranteed in real-world deployment due to the presence of mechanical failures and the unreliable nature of the Internet. However, best-effort reliability can be provided which offers high-probabilities of reliable service but cannot guarantee the absence of failures.

There are a number of problems which can emerge during the runtime of the DVE. These problems can, if they are not detected and responded to, cause undesirable loss of responsiveness or failure to detect and respond to collisions appropriately. The main issue of concern is if a collision detection node fails or becomes a bottleneck. In the case of collision detection node failure, its members in its consistency group will detect an increase in ping transmission delays. During this time, a number of seconds may have elapsed in which this collision detection node's sub-region did not receive collision detection/response in the consistency group. This will cause a loss of consistency. However, once the increased ping latency has been detected, the group will initiate a vote and

the collision detection node will be evicted from the group and inserted into a singleton consistency group. This will ensure that the node does not compromise other nodes' responsiveness. If the node which failed was the group leader, a new node will need to be promoted to group leader status. The failure of a group leader may result in significant inconsistency and loss of responsiveness in the consistency group. However, this will be overcome with the appointment of a new group leader and the re-dissemination of current world state by the server to the new group leader.

While the collision detection node is in a singleton consistency group, it is responsible for collision detection by itself. As such, it will not detrimentally affect the DVE if it fails to perform collision detection. The main server is responsible for detecting failed nodes; this may or may not be possible depending on the network protocols used but, for example, it is possible to detect a potentially failed node using TCP/IP by the connection timing out. If the server detects a timed out failed node, it will remove the node and any objects it hosts from the DVE.

All admin messages are assumed to be reliable. However, all non-admin messages can be transmitted either best-effort reliably or unreliably. Failure to deliver any non-admin messages will result in inconsistency until subsequent messages are received. However, it will not cause failure. However, the level of consistency within a consistency group will still be no worse than if collision detection and response was executed individually on each node in the consistency group.

4.4 Summary

This Chapter introduced the implementation issues which must be overcome in order to implement a consistent, scalable collision detection approach for DVEs.

This discussion included the choice of programming languages and platform-specific analysis of a number of current and future DVE platforms. These discussions included aspects such as:

- Programming paradigms
- Performance
- Memory requirements
- Platform-specific libraries and support
- Presence and granularity of pre-emptive multitasking
- Platform architecture and resources
- Platform-specific rules and optimisations

Following this, the architecture and requirements of the distributed collision detection approach described in this thesis were introduced. The problem of distributed collision detection was subdivided into the server, clients and collision detection nodes. Following this, the server, clients and collision detection nodes were further subdivided into their constituent components, which were analysed in terms of their purpose and responsibilities.

Following this, the algorithms and data structures required to implement the collision detection approach were discussed and potential optimisations were suggested, although the details of how these optimisations are implemented were omitted to aid clarity.

The problems of reliability and fault tolerance were discussed briefly. The assumptions which underpin the collision detection approach were outlined and some potential error conditions were described including the mechanisms by which the collision detection approach dealt with them.

Chapter 5

Experimentation

5.1 Introduction

This chapter describes provides experimental results to measure the performance and scalability of the distributed collision detection approach described in this thesis. This chapter describes the environments in which the experiments are conducted. This includes:

- Constant DVE properties
- Variable properties

The effects of these properties on both performance and scalability are discussed. The experimental platform is described. A brief discussion on the expected behaviour and scalability of the experimental DVE is discussed prior to the presentation of collected results.

5.2 Experimental Platform

Experimental results are collected from a non-interactive DVE. This allows the behaviour of the objects occupying the DVE to be replicated in subsequent experiments to obtain results from different configurations operating on a DVE exhibiting the same events. While an interactive DVE has been developed, results are conducted on a simulated DVE to obtain performance figures from a DVE inhabited by large numbers of objects (in the order of thousands); an interactive DVE with an equivalent number of participants is not currently possible for logistical reasons.

The DVE used in the experiments described in this chapter is simplified, whereby objects in the DVE can interact with one-another but there is no environment model which constrains their movement. This allows objects to move freely throughout the DVE without the need for path-finding AI to navigate through complex environment models. The world is enclosed within a cubic AABB which represents the maximum and minimum coordinates of the world. This AABB is scaled proportionally to the number and size of objects in the DVE, such that a world size can be selected to obtain a given probability of collisions occurring.

Collision detection nodes are hosted on a cluster of 2.8GHz P4 PCs with 1024Mb RAM running Red Hat Linux 7.0. The server is hosted on a 2.4GHz Intel Xeon server with 4096Mb RAM. However, the machines used in these experiments are a shared resource and, as such, the performance results gathered in these experiments may be affected by processes competing for processing and/or memory resources. In order to minimise the impact of the effect of competing processes, the averages of three independent sets of results will be presented in this thesis.

5.2.1 Probability of Collisions Occurring

The probability of a collision occurring between two objects in the DVE is the likelihood of the space enclosed by the objects overlapping. Given two objects, o_1 and o_2, with volumes $vol(o_1)$ and $vol(o_2)$ respectively, and a total world volume $vol(world)$. The probability of these two objects intersecting is proportional to the total volume of the objects with respect to the volume of the world. For example, if $vol(world) < (vol(o_1) + vol(o_2))$, the probability objects o_1 and o_2 intersecting is 1 because the volume of the world is sufficiently small that there is no position/orientation of the objects which can result in them being disjoint.

The lattice model can be used to formalise the probability of collisions occurring within a virtual world. Recall that the lattice model was introduced in Chapter 3, Section 3.2.1, to formalise bounding volumes to prove the assertion that if a pair of bounding volumes does not intersect, the objects which the bounding volumes enclosed also do not intersect. Given a virtual world represented as a lattice composed of n points and two objects, o_1 and o_2, each of which occupy p points in space designated by the sets p_1 and p_2. The probability of non-intersection is the probability of their being no common point in p_1 and p_2, i.e. $p_1 \cap p_2 = 0$. Given that object o_1 is already placed in the VE, the probability of object o_2 not intersecting o_1 is the probability of p_2 not containing any element of p_1. The total number of combinations of p points can be expressed mathematically as $\frac{n!}{p!(n-p)!}$. Given that p points are already occupied, there remain $\frac{(n-p)!}{p!(n-2p)!}$ combinations of points which o_2 can occupy which provide disjointedness. In practice, this is not entirely true as the points which an object occupies will, in most cases, be localised within a given area rather than being

arbitrarily placed around the virtual world. However, this provides a general metric as to the probability of objects intersecting, which can be expressed as below:

$$\frac{\frac{n!}{p!(n-p)!} - \frac{(n-p)!}{p!(n-2p)!}}{\frac{n!}{p!(n-p)!}} = 1 - \frac{(n-p)!^2}{n!(n-2p)!}$$

In order to obtain a valid probability, it is necessary to clip the result of this formula within the range [0, 1]. For example, given n = 10 and p = 2, the probability of two objects colliding within the world is 0.378.

With the probability of a pair of objects intersecting, it is possible to determine the expected number of collisions in a virtual world. Recall from Chapter 2 that the maximum number of collisions that can occur in a world composed of o objects is $\frac{o(o-1)}{2}$ collisions. Therefore, given that the probability of a collision occurring, P, has been determined based on the volume of the world compared to the volume of the objects which occupy it, the expected number of collisions in the virtual world is $P\frac{o(o-1)}{2}$. It can be seen from the quadratic nature of the function that, given a constant P, the expected number of collisions would exhibit $O(n^2)$ growth, i.e. If the number of objects was doubled and:

- the world size remained constant, the expected number of collisions would quadruple
- the world size was doubled, the expected number of collisions would double
- the world size was quadrupled, the expected number of collisions would remain constant

The experiments described in this chapter scale the world size such that as the number of objects is doubled, the size of the world is doubled, which leads to a

linear increase in the average number of collisions occurring in the virtual world as the number of objects is increased.

5.2.2 Experimental Virtual Environment

The DVE used in the experiments described in this chapter is initialised with a fixed number of deterministic dynamic objects. As clients join the DVE, they introduce their avatar objects into the DVE and propagate frequent state update messages for their avatar. In addition to introducing an avatar into the DVE, each client machine assumes the duties of a collision detection node for distributed collision detection.

Throughout the experiments, the following variables will be modified and performance figures will be obtained:

- The initial number of objects the DVE is created with
- The volume of the virtual world
- The number of clients and collision detection nodes participating in the DVE

Altering the initial number of objects in the DVE allows the scalability and responsiveness of the distributed collision detection algorithm to be analysed. This includes the determination of the general performance of the algorithm and the scale-up factor achieved through introducing additional processing resources.

Modifying the volume of the virtual world with respect to the number of objects allows the expected average number of collisions to be adjusted. This is expressed in the experimental results in terms of the percentage of the total volume of the virtual world which contains objects; this can be scaled by the maximum number of collisions to determine the expected number of collisions.

This can be determined by summing the volume of the objects in the virtual world and dividing this volume by the total volume of the virtual world. For example, given that the virtual world is defined as being a cube of dimension 10m and the world contains 100 objects, each cubes of length 1m:

- The volume of a single object is $1m^3$
- The volume of the virtual world is $1000m^3$
- The total proportional coverage of the virtual world is $\frac{100(1m^3)}{1000m^3}=0.1$ = 10%

Rather than determining the proportional coverage of the virtual world based on the current number of objects and size of the world, a desired proportional coverage is provided coupled with the number of objects inhabiting the virtual world. From this, the world size is generated using the following formula:

$$Volume = \frac{numObjects * objectVolume}{proportionalCoverage}$$

The above formula takes the number of objects, the volume of the objects and the proportional coverage desired and returns the total volume of the virtual world required. As stated previously, the experiments described in this chapter enclose the virtual world inside an axially-aligned cube. As such, the dimensions of each coordinate axis can be determined by taking the cube root of the volume: $Side = \sqrt[3]{Volume}$.

5.3 Expected Results

The goal of this thesis is to create a collision detection technique suitable for DVEs which offers high-levels of scalability, responsiveness and consistency. The responsiveness and scalability of the approach can be quantified by the time

taken for each collision detection iteration to be completed and the proportional performance increase respectively. Consistency, however, is difficult to quantify; it is more a qualitative property of a DVE which is perceived by a user. To clarify:

- Recording the average duration of collision detection in the DVE across all participating collision detection nodes demonstrates the responsiveness of the DVE. By modifying the number of objects participating in the DVE, the general performance trend of the collision detection approach can be observed.
- Modifying the number of collision detection nodes participating in the DVE demonstrates the scalability of the distributed collision detection algorithm as the number of processing resources increases. This demonstrates the scale-up factor associated with distributing the processing and networking requirements of distributed collision detection.

Consistency can be loosely quantified by the number of consistency groups with respect to the number of members in each group. The states of objects within a consistency group are consistent with all the group's members. This can be reasoned by the fact that collisions are detected and responded to at most once within a consistency group. However, consistency cannot be guaranteed between groups due to the potentially-large message transmission delays present between consistency group members.

As previously stated, consistency is more of a qualitative property of DVEs than a quantitative property. While it is possible to quantify the deviation in state of objects inhabiting the DVE between all machines participating in the DVE at any given time in the simulation, due to the impossibility of synchronising states in an asynchronous network in real-time, this is not possible during the run-time of the DVE and would need to be performed offline. This could be done by either by sampling the objects' states at frequent intervals, recording these

values and comparing the recorded values offline or by simulating a DVE and its related message dissemination on a single machine and taking metrics of simulated inconsistencies during run-time. The sampling option would require large amounts of storage and extremely expensive analysis; it should be noted that in order to chart inconsistencies, it is necessary to take readings over long time periods to determine the "snowballing" effect of small inconsistencies over the long-term, which may require too large amount of storage space to be feasible. The simulation option, conversely, may offer a mechanism to quantify consistency but it is unlikely that a simulation would accurately model the true behaviour of the network, server, collision detection nodes and group leaders and would, therefore, not offer a real-world representation of the consistency in DVEs.

5.3.1 Expected Responsiveness

It is expected that the performance of the collision detection approach should behave roughly O(nlogn) due to the adoption of spatial partitioning as the broad-phase collision detection approach. However, the decision to increase the size of the virtual world linearly with respect to the number of objects inhabiting the DVE may result in near-linear collision detection performance.

5.3.2 Expected Scalability

It is expected that the performance of the collision detection approach will improve near-linearly with respect to the number of members in the consistency group. This performance increase should not be perceived in DVEs with relatively small numbers of objects, in which the message transmission overhead will skew the results, overshadowing any performance improvements in collision detection performance. However, as the number of objects increases

beyond a level in which the major performance overhead in the DVE becomes performing collision detection rather than message dissemination, the scalability should begin to tend towards linear performance improvements. It is expected, however, that there will be a platform-specific limit on consistency group size at which point the performance may actually degrade if additional members join a consistency group. The value of this limit may vary depending on the number of objects inhabiting the DVE. This limit will represent the point at which the improvement in collision detection performance is overshadowed by the increased message transmission overheads. This limit will be platform-specific due to the platform-specific overhead associated with network communication, which may be influenced by the network hardware performance and the OS/application-specific implementation of network protocols.

5.3.3 Expected Consistency

It is expected that the overall consistency in the DVE will increase as the average consistency group size increases. This can be reasoned due to the fact that collision events are detected and responded to at most once within each consistency group and, as such, a collision is responded to in only one way in each consistency group. Given a DVE in which all participants are members of a single consistency group, the state of all objects in the DVE is expected to be consistent. However, results demonstrating improved consistency within DVEs utilising distributed collision detection are not presented in this thesis due to the reasons previously outlined.

5.4 Performance Experimentation

In order to determine the suitability of the distributed collision detection approach, it is necessary to determine the algorithm's performance when

operating in its worst-case; this occurs when collision detection can only be performed on a single collision detection node. In these experiments, the initial number of dynamic deterministic objects is increased from 500 to 5000 in increments of 500. Results are collected demonstrating the total time taken for a simulation step in each DVE being performed by a consistency group with only one member. The results from this experiment will demonstrate the overall performance trend of the collision detection algorithm used in distributed collision detection with respect to the number of objects inhabiting the DVE.

Num Objects	Time	Num Collisions
500	180	9
1000	360	16
1500	447	23
2000	604	30
2500	659	45
3000	825	56
3500	984	62
4000	1128	72
4500	1187	78
5000	1228	86

Table 5.1 Average Simulation Time

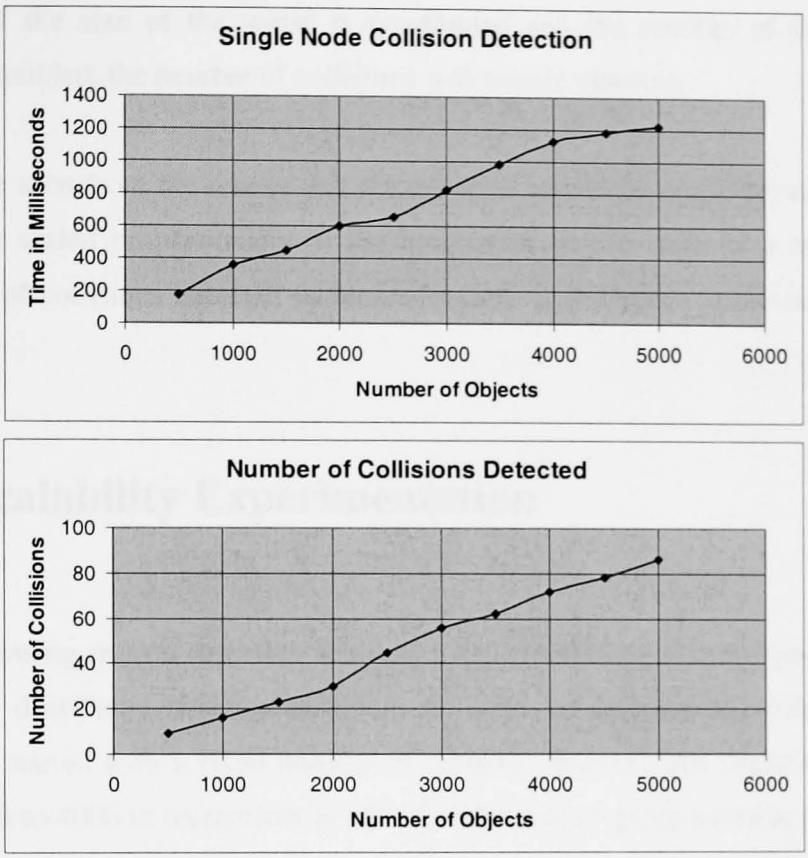

Figure 5.1 Single Node Collision Detection Performance as Number of Objects is Increased

The results in ***Fig*** 5.1 demonstrate the near-linear performance characteristics of the spatial partitioning approach adopted in the distributed collision detection approach presented in this thesis. In addition, the average number of collisions detected in the DVE as the number of objects is increased corresponds with a previously discussed topic: probability of collision. Previously, it was asserted that:

- If the size of the virtual world remains constant and the number of objects is doubled, the number of collisions will quadruple
- If the size of the world is doubled and the number of objects is doubled, the number of collisions will double

- If the size of the world is quadrupled and the number of objects is doubled, the number of collisions will remain constant.

It can be seen from the results that the previous assertions hold; the size of the world is scaled proportionally to the number of objects and, as a result, the number of collisions detected increases linearly with respect to the number of objects.

5.5 Scalability Experimentation

The following section describes scalability experiments conducted using a DVE utilising distributed collision detection. In order to demonstrate scalability, a DVE is started with a fixed number of dynamic deterministic objects ranging from 500 to 4000 in increments of 500. The number of group members within a consistency group is increased from 1 to 8 and the resulting time for each simulation loop is recorded. As each member collision detection node joins the DVE, its corresponding client introduces an additional avatar into the DVE which increases the number of objects inhabiting the DVE by one. For example, given 500 initial objects, with one collision detection node, there would be 501 objects in the DVE; with two collision detection nodes, there would be 502 objects, etc.

The results show two series of data:

- The average simulation time for all group members
- The average simulation time for the group leader

The times recorded show the total time required for a single simulation loop, including any message dissemination and message receipt. The average simulation time for all group members demonstrates the average scalability of the distributed collision detection algorithm; this average is taken over all collision detection nodes in the consistency group, including the node which is

appointed group leader. The average simulation time for the group leader demonstrates the performance overhead which is required by the group leader in comparison to the average performance required by all the group members.

Results

Num Objects	1 Node	2 Nodes	3 Nodes	4 Nodes	5 Nodes	6 Nodes	7 Nodes	8 Nodes
500	180	78	50	36	37	32	26	18
1000	360	123	98	83	71	67	50	49
1500	447	170	124	79	67	65	60	62
2000	604	264	152	149	133	112	81	77
2500	659	385	189	140	132	104	83	84
3000	825	520	230	185	155	137	104	92
3500	984	468	238	223	197	193	175	105
4000	1128	582	392	335	301	264	157	135

Table 5.2 Simulation Time on Average Collision Detection Nodes

Num Objects	1 Node	2 Nodes	3 Nodes	4 Nodes	5 Nodes	6 Nodes	7 Nodes	8 Nodes
500	1	2.30769	3.6	5	4.864865	5.625	6.92308	10
1000	1	2.92683	3.673469	4.33735	5.070423	5.37313	7.2	7.34694
1500	1	2.62941	3.604839	5.65823	6.671642	6.87692	7.45	7.20968
2000	1	2.28788	3.973684	4.05369	4.541353	5.39286	7.45679	7.84416
2500	1	1.71169	3.486772	4.70714	4.992424	6.33654	7.93976	7.84524
3000	1	1.58654	3.586957	4.45946	5.322581	6.0219	7.93269	8.96739
3500	1	2.10256	4.134454	4.41256	4.994924	5.09845	5.62286	9.37143
4000	1	1.93814	2.877551	3.36716	3.747508	4.27273	7.18471	8.35556
Average	1	2.18634	3.617216	4.49945	5.025715	5.62469	7.21374	8.36755

Table 5.3 Scale-up Factor

From the results shown in ***Fig*** 5.2, it can be seen that the as the number of collision detection nodes increases, the total time taken for a simulation loop decreases. This overall time taken for the simulation step is proportional to: $\frac{number\_of\_objects}{number\_of\_nodes}$. This can be seen to be the case as the simulation time taken for a DVE with 500 objects on one collision detection node is roughly the same as the time taken to simulate a DVE with 1000 objects on two collision detection nodes. This trend follows throughout all the obtained results, with

small degrees of variance due to processes competing for resources and variations in network load. This implies that the performance improvements offered by this approach are roughly linear. In order to analyse the performance improvements, the performance results recorded for a DVE with a given number of objects running on a consistency group with one collision detection node are used as a base for comparison. The performance improvement, or scale-up factor, offered by a consistency group with a given number of collision detection nodes, n, can be determined by: $\frac{performance(1)}{performance(n)}$, where the function $performance(k)$ returns the performance figure for a consistency group of size k. Table 5.3 shows the scale-up factor recorded. From these results, it can be seen that the performance is roughly linear as the average performance improvements for n nodes is roughly n. There are some minor deviations in which the performance is larger or smaller than n, most noticeably for 500 objects with 8 collision detection nodes, in which case the performance recorded is 10 times better than the performance recorded with 1 collision detection node. These variations are most likely caused by improved data cache coherence. This is caused by each collision detection node being responsible for collision detection on smaller portions of the virtual world and, therefore, fewer objects. Therefore, it is more probable that the data corresponding to the portions of the virtual world and objects each collision detection node is responsible for can be stored in cache memory for fast access. This avoids the need to fetch the data from system memory prior to performing collision detection. Conversely, as the portion of the virtual world which a collision detection node is responsible for becomes larger, and therefore the number of objects increases, it is less likely that the data can be stored in cache memory and, therefore, the probability of cache misses and the need to copy data from system memory to cache memory increases.

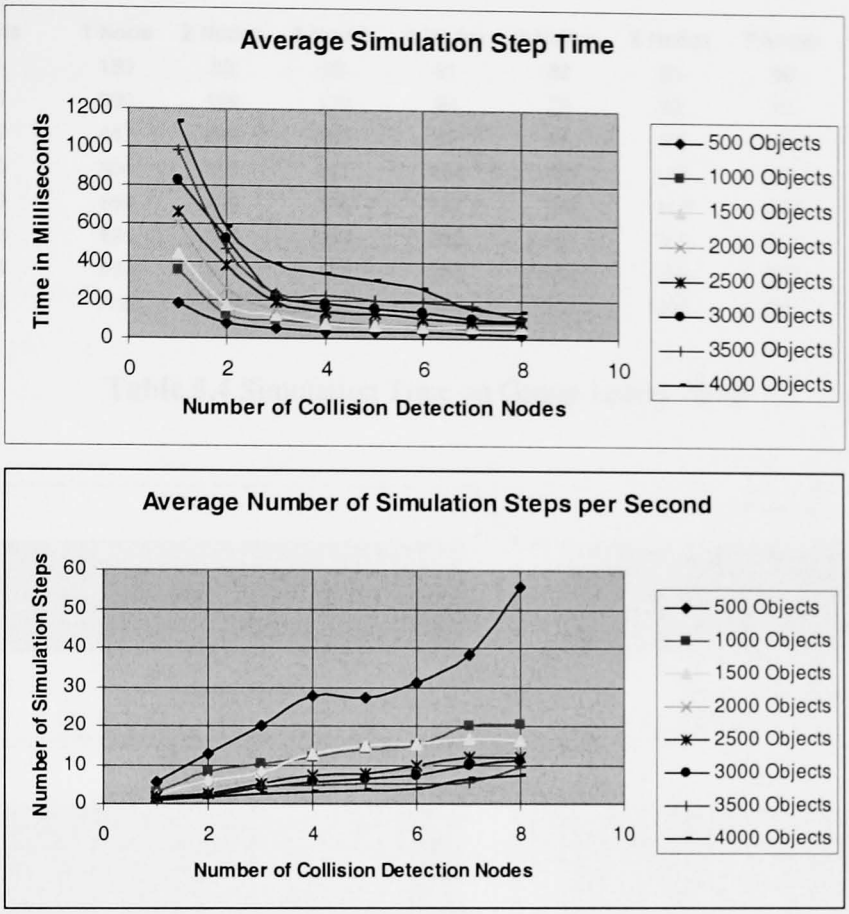

Figure 5.2 Average Time Taken for Collision Detection Iteration as Number of Collision Detection Nodes is Increased

In ***Fig*** 5.2, as the simulation step time becomes small as a result of increasing the number of collision detection nodes, the graph charting simulation time ceases to be able to effectively show the performance improvements offered by the distributed collision detection approach. In order to aid clarity, a graph showing the number of simulation steps which can be performed in a second is provided. This is a reciprocal measure of the simulation step time is used to offer a more effective mechanism of charting fine-grained performance variations. This mechanism is commonly used in charting the performance of commercial computer games engines; this measurement is often termed *Frames Per Second* (FPS).

Num Objects	1 Node	2 Nodes	3 Nodes	4 Nodes	5 Nodes	6 Nodes	7 Nodes	8 Nodes
500	180	82	92	91	82	83	86	53
1000	360	169	170	98	79	82	75	61
1500	447	208	188	87	84	97	100	68
2000	604	313	247	164	168	165	175	81
2500	659	410	385	187	189	168	163	87
3000	825	539	366	202	181	155	162	101
3500	984	471	475	257	264	258	262	143
4000	1128	620	627	374	372	380	364	178

Table 5.4 Simulation Time on Group Leader Node

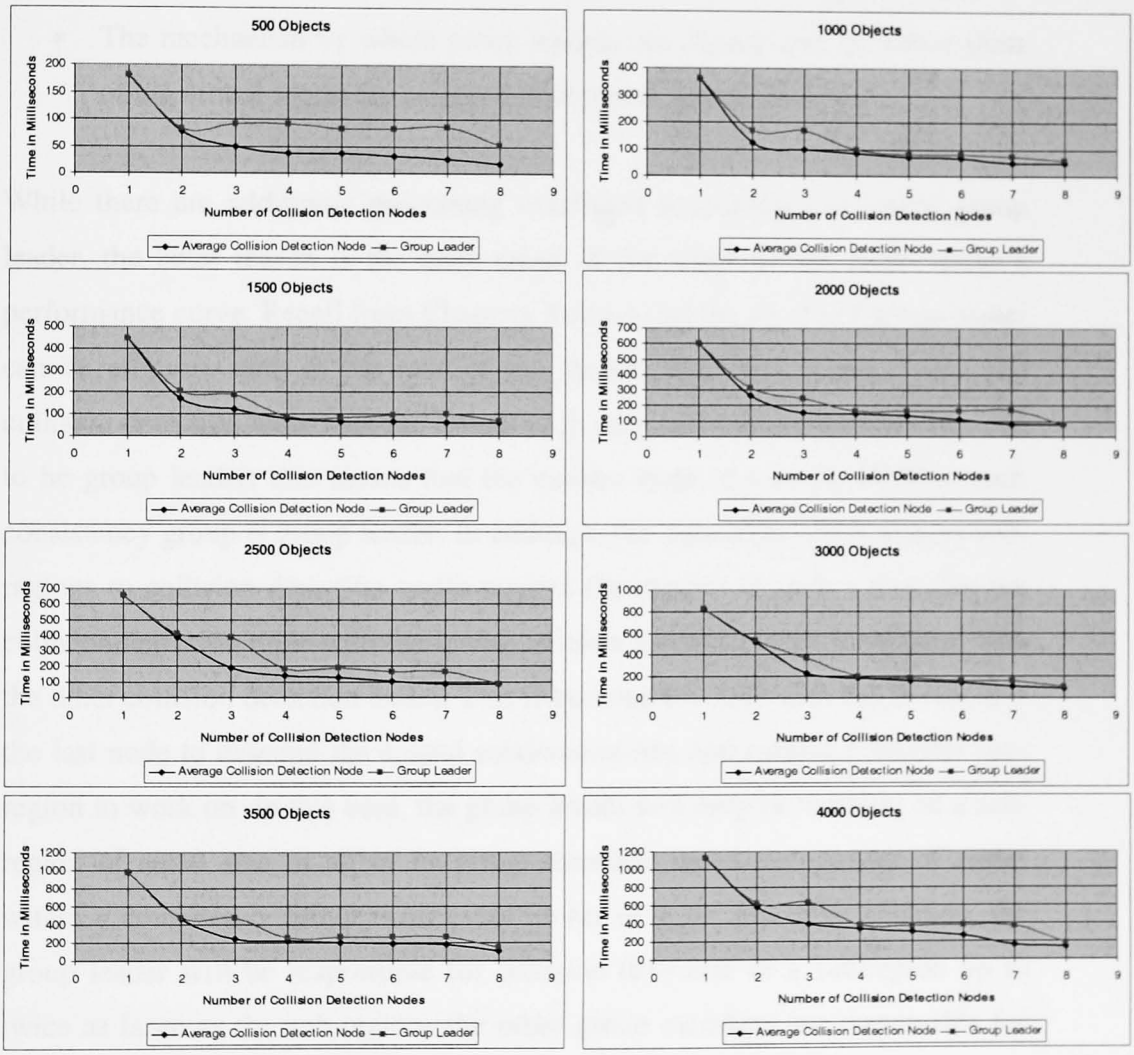

Figure 5.3 Distributed Collision Detection Performance

Fig 5.3 shows the overall performance of the collision detection nodes in DVEs ranging from 500 to 4000 objects. The results depicted in the graphs show the variations in performance between the group leader collision detection node and the other group member collision detection nodes. It can be seen that the all nodes in the consistency group observe performance improvements as the size of the consistency group increases. However, the group leader's performance is, in general, worse than the remaining collision detection nodes in its consistency group. This is due to:

- The additional processing overheads associated with being group leader
- The mechanism by which group leaders are chosen and the sub-regions of the virtual world are assigned to collision detection nodes.

While there are additional processing overheads associated with being group leader, the latter reason is the main cause of the shape of the group leader's performance curve. Recall from Chapters 3 and 4 that the choice of group leader can be arbitrary and, in the case of the system described in this thesis, the collision detection node with the lowest id in each consistency group is selected to be group leader; this means that the earliest node to join the DVE in each consistency group is group leader. In addition, the algorithm which assigns sub-regions to collision detection nodes assigns the regions in such a way that the collision detection node with the lowest id often has more work to perform than the other collision detection nodes. This is because the node with the lowest id is the last node to descend the spatial subdivision tree and receive a smaller sub-region to work on. In this case, the group leader will only be working on a sub-region of equal size to all of its group members when the number of nodes within a consistency group is an exact power of 2. In any other situation, the group leader will be responsible for collision detection in a sub-region up to twice as large as the sub-regions the other group members are responsible for respectively. As such, the performance overhead associated with the additional network communication responsibilities of the group leader can only be clearly seen when the group size is a power of 2; when the group size is a power of 2,

the additional overhead of being a group leader can be estimated as the group leader simulation time subtracted from the average collision detection simulation time.

Num Objects	1 Node	2 Nodes	3 Nodes	4 Nodes	5 Nodes	6 Nodes	7 Nodes	8 Nodes
500	0	4	42	55	45	51	60	35
1000	0	46	72	15	8	15	25	12
1500	0	38	64	8	17	32	40	6
2000	0	49	95	15	35	53	94	4
2500	0	25	196	47	57	64	80	3
3000	0	19	136	17	26	18	58	9
3500	0	3	237	34	67	65	87	38
4000	0	38	235	39	71	116	207	43

Table 5.5 Group Leader Overhead

Table 5.5 shows the perceived difference in simulation step time between the group leader and the average time recorded across all members of the consistency group. It can be seen that at all times, the group leader's performance is worse than the other group members. However, when the number of collision detection nodes is a power of 8, the difference between the performance of the group leader and the other group members is, in general, at its lowest.

Num Objects	1 Node	2 Nodes	3 Nodes	4 Nodes	5 Nodes	6 Nodes	7 Nodes	8 Nodes
500	0	4.87805	45.65217	60.4396	54.87805	61.4458	69.7674	66.0377
1000	0	27.2189	42.35294	15.3061	10.12658	18.2927	33.3333	19.6721
1500	0	18.2692	34.04255	9.1954	20.2381	32.9897	40	8.82353
2000	0	15.655	38.46154	9.14634	20.83333	32.1212	53.7143	4.93827
2500	0	6.09756	50.90909	25.1337	30.15873	38.0952	49.0798	3.44828
3000	0	3.52505	37.15847	8.41584	14.36464	11.6129	35.8025	8.91089
3500	0	0.63694	49.89474	13.2296	25.37879	25.1938	33.2061	26.5734
4000	0	6.12903	37.48006	10.4278	19.08602	30.5263	56.8681	24.1573
Average	***0***	***10.3012***	***41.99395***	***18.9118***	***24.38303***	***31.2847***	***46.4714***	***20.3202***

Table 5.6 Average Percentage Increase in Processing Overhead for Group Leader

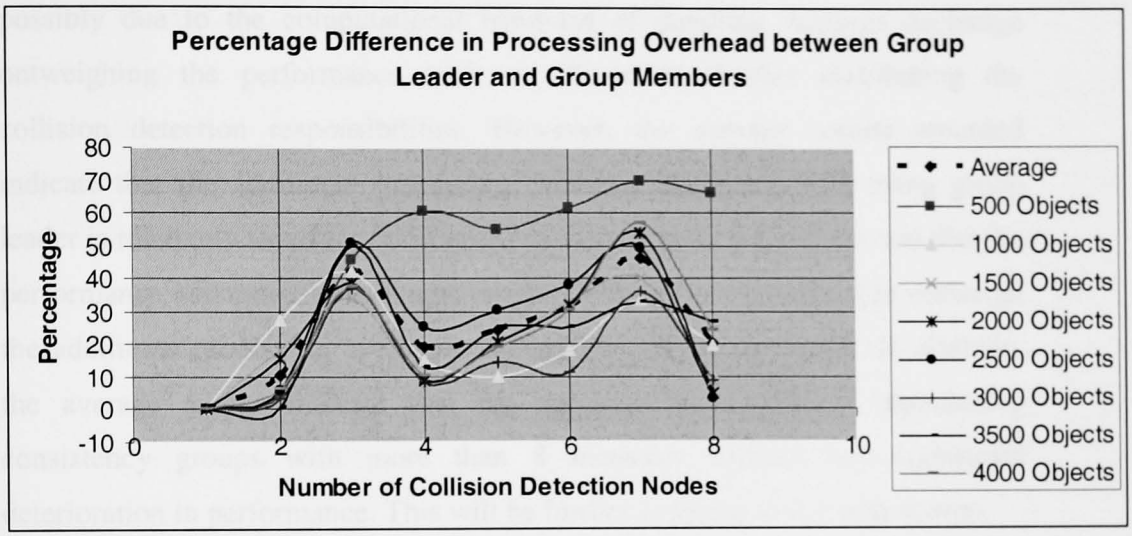

Figure 5.4 Group Leader vs Group Member

Table 5.6 and Figure 5.4 show the percentage difference between the processing overhead of the group leader and the other group members. Given the simulation step time for a group leader, *GL*, and the average time for the group members, *GM*, the data in Table 5.6 is generated by the following formula: $100 \times \frac{GL - GM}{GL}$. From this data, it can be seen that the average percentage increase in processing overhead for the group leader varies depending on the number of collision detection nodes in the consistency group. The perceived results were as follows:

- 1 collision detection node – No difference in performance
- 2 collision detection nodes – Average: 10% within the range [0.6%, 27.2%]
- 4 collision detection nodes – Average: 19% within the range [8.4%, 60.4%]
- 8 collision detection nodes – Average: 20% within the range [3.4%, 66%]

It should be noted that the largest percentage difference in performance between group leader and group members occurred in the DVE with 500 objects. This is possibly due to the computational overhead of handling message exchange outweighing the performance increase offered by further distributing the collision detection responsibilities. However, the average results recorded indicate that the additional processing overhead associated with being group leader is relatively small for consistency groups of up to 8 members and that the performance and consistency improvements offered by the technique outweigh the additional processing overheads incurred by the group leader. In addition, the average results indicate that the approach is capable of maintaining consistency groups with more than 8 members without any significant deterioration in performance. This will be further explored in the next section.

5.6 Maximum Consistency Group Size

The following experiments are designed to determine if there is an optimal consistency group size for a given DVE and what factors contribute to this size. It is expected that there will be a threshold value for consistency group size beyond which the collision detection performance will degrade. This will occur when the performance increase offered by distributing the collision detection overhead is overshadowed by the increased message dissemination required to manage additional group members. It is expected that this threshold value will depend on the number of objects inhabiting the DVE.

In these experiments, the number of objects inhabiting the DVE will be increased from 1000 to 3000 in increments of 1000. At each stage the number of collision detection nodes in a given consistency group will be incremented until either:

- The performance of the collision detection engine begins to noticeably degrade

- The maximum number of machines allotted for the experiment is exceeded: 32

Due to the nature of the spatial partitioning approach used in distributed collision detection, the number of collision detection nodes used in the experiments will be doubled at each increment. As such, these experiments will be performed with 1, 2, 4, 8, 16 and 32 collision detection nodes respectively. This is because the binary tree used to allocate sub-spaces to the collision detection nodes produces its most even distribution of workload when the number of collision detection nodes in a consistency group is a power of 2.

Results

Num Objects	1 Node	2 Nodes	4 Nodes	8 Nodes	16 Nodes	32 Nodes
1000	360	123	83	49	21	27
2000	604	264	149	77	38	26
3000	825	520	185	92	65	38

Table 5.7: Maximum Consistency Group Size Experiment Results

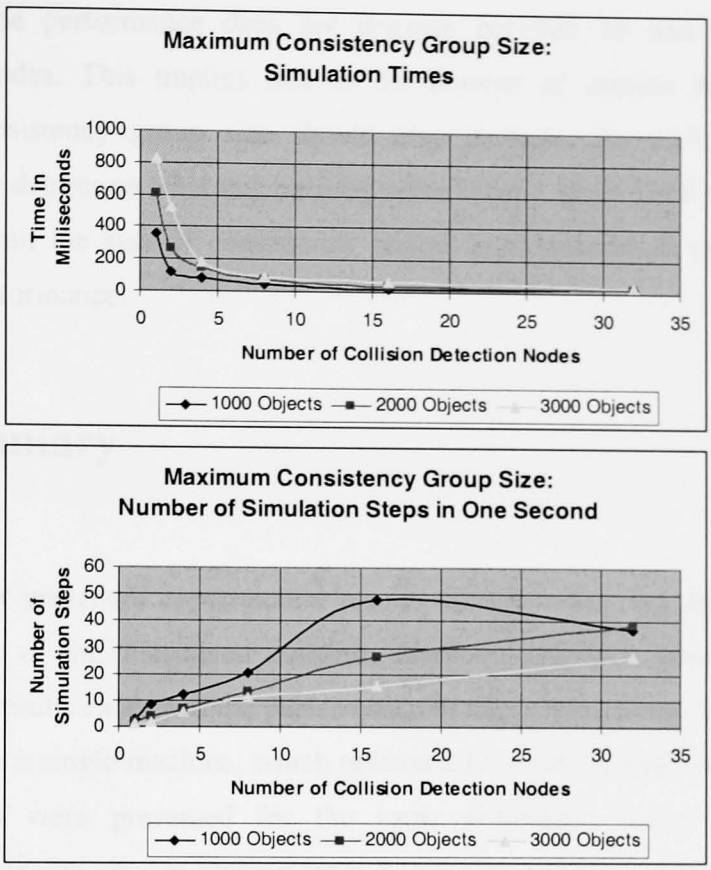

Figure 5.5: Determining the Maximum Consistency Group Size

From ***Fig*** 5.5, it can be seen that as the number of collision detection nodes increases, the performance also increases. However, the results do indicate that there is a point at which the inclusion of additional collision detection nodes causes a decrease in performance. This can be seen in the results for a DVE with 1000 objects, in which the average simulation step time for 16 objects is 21ms, whereas the average time for 32 collision detection nodes is 27ms. This indicates that the additional message exchange required in maintaining a consistency group of size 32 outweighs the reduction in collision detection each collision detection node is responsible for.

The point at which performance begins to degrade does not appear to be constant but, instead, it appears related to the number of objects inhabiting the DVE. This can be seen from the results for DVEs with 2000 and 3000 objects,

in which the performance does not degrade between 16 and 32 collision detection nodes. This implies that as the number of objects increases, the optimal consistency group size should also increase. As such, it may be necessary to determine what this optimal value is for a given DVE to enable the server to limit the size of consistency groups appropriately to yield the best possible performance.

5.7 Summary

This chapter presented experimental results demonstrating the scalability and performance of the distributed collision detection approach described in this thesis. The results examined the performance of the collision detection approach operating on a single machine, which offered a base for comparison. Following this, results were presented for the same simulation being executed in consistency groups of size ranging from 1 to 8. The results demonstrated linear performance improvements with respect to the number of collision detection nodes. While these results show the performance and scalability within a consistency group, the results do not demonstrate the consistency or scalability of message exchange between consistency groups, e.g. the volume of messages being transferred between consistency groups or the deviation between the perceived states of objects between consistency groups. The absence of these results is mainly due to the difficulty in recording run-time metrics between consistency groups. Such results could be approximated by simulating the distributed collision detection approach, although this is beyond the scope of this thesis.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The previous chapters have introduced Distributed Virtual Environments (DVEs) and their associated problems. After offering a brief overview of the topic of DVEs, Chapter 2 focussed its discussion on collision detection. This was initially discussed with respect to single-user virtual environments and a selection of collision detection approaches and algorithms were introduced and evaluated based on the following criteria:

- Performance characteristics
- Opportunities to exploit parallelism

The discussions on the possibility of exploiting parallelism in collision detection algorithms comes from the recent popularity of multiple processing cores in consumer PCs and games consoles. With the adoption of multi-processor environments, it is necessary to ensure that the algorithms adopted are able to exploit the additional processing power made available. From the analysis, it was found that the best opportunities for parallel execution were offered by spatial partitioning approaches, in which the virtual world is subdivided into

discrete regions; for more a more detailed explanation of this assertion, see Chapters 2 and 3.

Following the description of collision detection algorithms, the problems of scalability, responsiveness and consistency in DVEs were discussed with respect to collision detection. It was determined that the current approaches used for collision detection in DVEs lends to extreme levels of inconsistency which can significantly compromise user-immersion. Additionally, as collision detection is usually performed completely on all machines participating in the DVE, the performance of the DVE will degrade as the number of users increase. Therefore, it is necessary to develop a new approach to collision detection in DVEs which offers higher-levels of consistency, scalability and responsiveness in order to allow more users to interact with one-another within a DVE. The development of such an approach has historically been overlooked due to the consistency-throughput trade-off theory [Singhal99][Fischer83], which states that it is necessary to balance consistency with throughput. A completely consistent DVE can be achieved at the cost of throughput and interactivity and that it is possible to have DVEs whose throughput (interactivity) is at the maximum rate allowed by the host machines and network infrastructure at the detriment of consistency. This rule has been assumed to be rigid and, therefore, restrict the level of interactivity, responsiveness and consistency of DVEs.

Chapter 3 introduced the notion of distributed collision detection. It took an incremental approach in refining a collision detection approach which would offer improved scalability, responsiveness and consistency in DVEs running over heterogeneous networks and platforms. The discussion began by defining the choice of algorithms and data structures which are used in the collision detection approach; this included the choice of spatial subdivision as the broad-phase collision detection approach. While the distributed collision detection approach prescribes the use of a given broad phase collision detection algorithm, the approach allows the use of any narrow-phase algorithm. Following the

definition of the basic algorithms, the remainder of the chapter incrementally refined the collision detection approach by adding additional complexities, such as:

- Limited network bandwidth
- Message transmission latency
- Inconsistencies in object states between machines
- Unreliable machines and communication media

Chapter 3 introduced the components from which the distributed collision detection approach is composed:

- Server
- Collision Detection Node
- Client

The server is a central repository and directory service which is used to store the state of a DVE and to assist users in joining a given DVE. A collision detection node is a process hosted on a given machine which is responsible for collision detection on a portion of the virtual world. A client is a process hosted on a given machine through which a user interacts with the DVE. The discussion initially developed a collision detection approach which mirrored the client/server architecture commonly used in Internet applications. This approach offered linear performance increases as the number of collision detection nodes increased. However, this approach was only suitable for use on Local Area Networks (LANs) as it was not capable of dealing with variable network transmission delays. Therefore, the architecture of the collision detection approach was refined into a hybrid peer/server architecture to be able to deal with variable network transmission delays. This refinement introduced the concept of consistency groups. A consistency group is a group of machines which exhibit low-latency message transmission delays between one-another and, therefore, are capable of sharing the responsibility of collision detection for the DVE between one-another. The perceived state of the objects inhabiting the

DVE in each consistency group is completely consistent. However, there are no consistency guarantees between consistency groups as each consistency group operates completely separately of any other group. As with the client/server collision detection approach, the performance of each consistency group increases linearly as the number of collision detection nodes in the consistency group increases. Each consistency group is coordinated by a single collision detection node which is appointed group leader. The group leader acts as a surrogate server for its consistency group members and is a point of communication between consistency groups.

Chapter 4 described the implementation details of the distributed collision detection approach. This chapter gave a brief evaluation of the enabling technologies available to facilitate the development of a DVE. This discussion included the choice of programming languages, libraries, target platforms and any platform-specific issues of importance to DVE development. Following this, an introduction to some of the important maths involved in the distributed collision detection approach was provided.

Chapter 4 described the different components of the server, collision detection nodes and clients in the distributed collision detection approach. In addition, it described the auxiliary components which are shared between the server, collision detection nodes and clients. This discussion included in-depth descriptions of the messages exchanged between different components and the mechanisms by which the responsibility for collision detection is distributed between members of a consistency group.

Chapter 5 provided experimental results for the distributed collision detection approach. This chapter began by describing the DVEs in which experiments were performed. Following this, the expected results were described. The results which were presented demonstrated that the performance of a DVE adopting the distributed collision detection approach.

6.2 Future Work

The distributed collision detection approach has currently been implemented in Java and C++. The Java version of the approach is a prototypical implementation which was used to gather performance figures and determine the suitability of the approach. The C++ version has been integrated into a multiplayer game which is freely available for download [Storey06]. However, this implementation is currently hard-coded into the games engine rather than developing it as a middleware component. As such, it is possible that in the future the distributed collision detection approach will be incorporated into a commercial physics engine for use in computer games and physics simulations.

6.3 Summary

This thesis examined the applicability of a distributed collision detection approach in Distributed Virtual Environments. This approach provides improved scalability and responsiveness by sharing the processing overhead associated with collision detection between the machines participating the DVE. It adopts a hierarchical message dissemination approach which clusters machines which share low message transmission delays. This not only reduces the volume of messages which the main server must handle, but also alleviates the message throughput restrictions imposed on a client sharing a high-latency connection with the server. However, this approach may increase the message latency between consistency groups. In addition to improvements in scalability and responsiveness, the distributed collision detection approach provides improved consistency by enforcing that collisions between a pair of objects are detected on only one machine within a consistency group.

References

[Abrams98] Abrams, H., K. Watsen, et al. (1998). Three Tiered Interest Management for Large-Scale Virtual Environments. ACM Symposium on Virtual Reality Software and Technology, Taipei, Taiwan.

[ATI06] http://ati.amd.com/, ATI 2006, as viewed 3/12/2006.

[Basch99] Basch, J., L. J. Guibas, et al. (1999). Kinetic collision detection between two simple polygons. Tenth annual ACM-SIAM symposium on Discrete algorithms, Baltimore, Maryland, United States, Society for Industrial and Applied Mathematics Philadelphia, PA, USA.

[Ben-Ari06] Ben-Ari, M. Principles of Concurrent and Distributed Programming (Second Edition). Prentice-Hall, 2006.

[Bergen04] Bergen, G. V. D. (2004). Collision Detection in Interactive 3D Environments, Elsevier, 2004.

[Bharambe02] Bharambe, A. R., S. Rao, et al. (2002). Mercury: a scalable publish-subscribe system for internet games. 1st workshop on Network and system support for games, Bruanschweig, Germany, ACM Press New York, NY, USA.

[Bosser04] Bosser, A. Massively-Multiplayer Games: Matching Game

Design with Technical Details. Sixth Australasian Computing Education Conference, ACE2004.

[Bourg] Bourg, D. M. Physics for Games Developers, O'Reilly Press

[Burdeau03] Burdea, G., Coffet, P. (2003) Virtual Reality Technology, Second Edition, Wiley-IEEE Press, 2003.

[Carlssom93] Carlssom, C. and O. Hagsand (1993). DIVE – A platform for multi-user VE. Computer & Graphics 17(6).

[Chrysanthou95] Chrysanthou, Y. and M. Slater (1995). Shadow volume BSP trees for computation of shadows in dynamic scenes. symposium on Interactive 3D graphics, Monterey, California, United States, ACM Press New York, NY, USA.

[Coldet] http://sourceforge.net/projects/coldet as viewed 7/12/2006.

[C++] http://www.cplusplus.com/ as viewed 9/12/2006

[Epic06] http://www.unrealtechnology.com/html/technology/ue30.shtml, Epic Games, 2006, as viewed 3/12/2006.

[Ericcson05] Ericcson, Christopher. Real-time Collision Detection. Morgan-Kaufman Series on Interactive 3-D technology, Morgan-Kaufman, 2005.

[Erickson99] Erickson, J., L. J. Guibas, et al. (1999). Separation-sensitive

collision detection for convex objects. tenth annual ACM-SIAM symposium on Discrete algorithms, Baltimore, Maryland, United States, Society for Industrial and Applied Mathematics Philadelphia, PA, USA.

[Ezhilchelvan92] Ezhilchelvan, P., Shrivastava, S. K. A Distributed Systems Architecture Supporting High Availability and Reliability. 2nd International Working Conference on Dependable Computing for Critical Applications, Tucson, Arizona, US, February, Springer-Verlag 1992.

[Fischer83] Fischer, M. J., The Consensus Problem in Unreliable Distributed Systems (A Brief Survey), International Conference on Foundations of Computational Theory, Borgholm, Sweden, August 21-27, 1983, pp127-140, Springer-Verlag.

[Fuchs80] Fuchs, H., Z. M. Kedem, et al. (1980). On visible surface generation by a priori tree structures. 7th annual conference on Computer graphics and interactive techniques, Seattle, Washington, United States, ACM Press New York, NY, USA.

[Gottschalk96] Gottschalk, S., M. C. Lin, et al. (1996). OBBTree: a hierarchical structure for rapid interference detection. 23rd annual conference on Computer graphics and interactive techniques, ACM Press New York, NY, USA.

[Gottsman05] Gottsman, C. (2005). What's in a Mesh? A Survey of 3D Mesh Represenation Schemes. International Conference on Shape Modelling and Applications

[Greenhalgh] Greenhalgh, C. An Experimental Implementation of the Spatial Model.

[He99] He, T. (1999). Fast collision detection using QuOSPO trees. symposium on Interactive 3D graphics, Atlanta, Georgia, United States, ACM Press New York, NY, USA.

[Heckbert94] HeckBert, P. S. and M. Garland (1994). Multiresolution Modeling for Fast Rendering. Proceedings from Graphics Interface '94.

[Hubbard95] Hubbard, P. 1995. Collision detection for interactive graphics applications. IEEE Transactions on Visualization and Computer Graphics 1, 3, 218–230.

[Hubbard96] Hubbard, P. 1996. Approximating polyhedra with spheres for time-critical collision detection. ACM Transactions on Graphics 15, 3, 179–210.

[IBM07] http://www-03.ibm.com/industries/media/doc/content/news/pressrelease/359248111.html. IBM, as viewed 5/4/2007.

[ICollide] http://www.cs.unc.edu/~geom/I_COLLIDE/index.html as viewed 7/12/2006

[Id06] http://www.idsoftware.com/business/technology/, Id Software, 2006, as viewed 3/12/2006.

[Jaja92] Jaja, J. An Introduction to Parallel Algorithms. Addison-Wesley, 1992.

[Klein03] Klein, J. and G. Zachmann (2003). ADB-Trees: Controlling the Error of Time-Critical Collision Detection. 8th International Fall Workshop Vision, Modeling, and Visualization (VMV).

[Lawler02] Lawlor, O. S. and L. V. Kale (2002). A Voxel-Based Parallel Collision Detection Algorithm. International Conference in Supercomputing, New York, NY, USA, ACM Press.

[Lefebvre06] Levebre, S., Hugues, H. Perfect Spatial Hashing. Microsoft Research Labs 2006, http://research.microsoft.com/~hoppe/perfecthash.pdf.

[Lengyel03] Lengyel, E., Math for 3D Game Programming and Computer Graphics. Charles River Media, 2003.

[Li98] Li, T.-Y. and J.-S. Chen (1998). Incremental 3D Collision Detection with Hierarchical Data Structures. Symposium on Virtual Reality Software and Technology.

[Li01] Li, X., C. Meng, et al. (2001). Detecting Collision of Polytopes Using a Heuristic Search for Separating Vectors.

[Lin98] Lin, M. C. and S. Gottschalk (1998). Collision Detection between geometric models: a survey. IMA Conference on Mathematics of Surfaces.

[Lin] Lin, M. C., D. Manocha, et al. Collision Detection: Algorithms and Applications. Algorithms for Robotics Motion and Manipulation.

[Logan] Logan, B. and G. Theodoropoulos (2000). Dynamic Interest Management in the Distributed Simulation of Agent-Based Systems. 10th AI, Simulation and Planning Conference.

[McCoy04] McCoy, A., Delaney, D., McLoone, S., Ward, T. (2004) Towards Statistical Client Prediction – Analysis of User Behaviours in Distributed Interactive Media. CGAIDE 2004, International Conference on Computer Games: Artificial Intelligence, Design and Education, Microsoft Campus, Reading, UK.

[Microsoft06] http://www.xbox.com/ as viewed 9/12/2006.

[Miller95] Miller, D. C., Thorpe, J. A., SIMNET: the advent of simulator networking, Proceedings of the IEEE

[Moller97] Moller, T. (1997). "A fast triangle-triangle intersection test." Journal of Graphics Tools **Volume 2**(Issue 2): 25-30.

[Morgan05] Morgan, G., Lu, F., Storey, K. (2005). Interest Management Middleware for Networked Games. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. Washington USA, ACM SIGGRAPH.

[Morgan05 2] Morgan, G., Storey, K. (2005). Scalable Collision Detection for Massively Multiplayer Online Games. 19th International Conference on Advanced Information Networking and Applications (AINA '05). Taipei, Taiwan, IEEE.

[Morgan04] Morgan, G., K. Storey, et al. (2004). Expanding Spheres: A Collision Detection Algorithm for Interest Management in Networked Games. International Conference on Entertainment Computing, Eindhoven, Netherlands, ACM.

[Morgan03] Morgan, G. and F. Lu (2003). Predictive Interest Management: An Approach to Managing Message Dissemination for Distributed Virtual Environments. 1st International Workshop on Interactive Rich Media Content Production, Lausanne, Switzerland.

[Naylor90] Naylor, B., J. Amanatides, et al. (1990). Merging BSP trees yields polyhedral set operations. Computer graphics and interactive techniques, Dallas, TX, USA, ACM Press New York, NY, USA.

[NVidia06] http://www.nvidia.com/page/home.html, Nvidia 2006, as viewed 3/12/2006.

[OpCode] http://www.codercorner.com/Opcode.htm as viewed 7/12/2006

[Otaduy03] Otaduy, M. A. and M. C. Lin (2003). Sensation Preserving Simplification for Haptic Rendering. ACM SIGGRAPH 2003, ACM Press New York, NY, USA.

[Pajarola02] Pajarola, R. Overview of Quadtree-based Terrain Triangulation and Visualization.

[Palmer] Palmer, G. Physics for Game Programmers, Apress Publishers

[Pelenchano02] Pelechano, N., L. Bull, et al. (2002). "Fast Collision Detection Between Cloth and a Deformable Human Body."

[PQP] http://www.cs.unc.edu/~geom/SSV/ as viewed 7/12/2006.

[Rapid] http://www.cs.unc.edu/~geom/OBB/OBBT.html as viewed 7/12/2006.

[Redon02] Redon, S., Khaddar, A., Coquillart, S. Fast Continous Collision Detection between Rigid Bodies

[Redon04] Redon, S., Kim, Y. J., Lin, M. C., Manocha, D. Fast Continous Collision Detection for Articulated Bodies

[Samet84] Samet, H. (1984) The Quadtree and other Related Hierarchical Data Structures. ACM

[Schmalsteig] Schmalsteig, D. and R. F. Tobler Real-time Bounding Box Area Computation.

[Sedgewick96] Sedgewick, R., Flajolet, P. An Introduction to the Analysis of Algorithms, Addison-Wesley, 1996

[Singhal99] Singhal, S., Zyda, M., Networked Virtual Environments: Design and Implementation, Addison-Wesley, 1999.

[Solid] http://www.win.tue.nl/~gino/solid/ as viewed 7/12/2006.

[Sony06] http://www.playstation.com/ as viewed 9/12/2006.

[Storey04] Storey, K., F. Lu, et al. (2004). Determining Collisions between Moving Spheres for Distributed Virtual Environments. Computer Graphics International, Crete, Greece, IEEE.

[Storey06] http://homepages.cs.ncl.ac.uk/graham.morgan/dge/ as viewed 9/12/2006.

[Sun06] http://www.java.sun.com/ as viewed 9/12/2006

[Suri98] Suri, S., P. M. Hubbard, et al. (1998). Collision detection in aspect and scale bounded polyhedra. ninth annual ACM-SIAM symposium on Discrete algorithms, San Francisco, California, United States, Society for Industrial and Applied Mathematics Philadelphia, PA, USA.

[Sweeney99] http://unreal.epicgames.com/Network.htm. Epic Megagames, as viewed 5/4/2007.

[Swift] http://www.cs.unc.edu/~geom/SWIFT++/ as viewed 7/12/2006

[Taylor99] Taylor, S. J. E., J. Saville, et al. (1999). Developing interest management techniques in Distributed interactive simulation using Java. Winter simulation: Simulation---a bridge to the future, Phoenix, Arizona, United States, ACM Press New York, NY, USA.

[Tenenbaum96] Tenenbaum, A. S., Computer Networks Third Edition, Prentice-Hall, 1996.

[Tropp06] Tropp, O., Ayellet, T., Shimshoni, I. (2006). A fast triangle to triangle intersection test for collision detection. Computer Animations and Virtual Worlds.

[Vaghi99] Vaghi, I., C. Greenhalgh, et al. (1999). Coping with inconsistency due to network delays in collaborative virtual environments. ACM symposium on Virtual reality software and technology, London, United Kingdom, ACM Press New York, NY, USA.

[Valve06] http://developer.valvesoftware.com/wiki/Main_Page, Valve Software, 2006 as viewed 3/12/2006.

[VCollide] http://www.cs.unc.edu/~geom/V_COLLIDE/ as viewed 7/12/2006.

[Watt01] Watt, A. and F. Policarpo 3D Games: Real-Time Rendering and Software Technology, Volume 1, ACM, 2001.

[Wiley97] Wiley, C., I. A. T. Campbell, et al. (1997). Multiresolution BSP trees applied to terrain, transparency, and general objects. conference on Graphics interface, Kelowna, British Columbia, Canada, Canadian Information Processing Society Toronto, Ont., Canada, Canada.

[Wiki06] http://en.wikipedia.org/wiki/Frame_rate, Wikepedia, as viewed 5/12/2006.

[Wiki06 2] http://en.wikipedia.org/wiki/Programming_language. Wikepedia, as viewed 9/12/2006.

[Zhao01] Zhao, H. and N. D. Georganas Collaborative Virtual Environments: Managing Shared Spaces, 2001.