Mecha Zeta

Project Title: Next-Generation Real Time Internet Game (Self-proposed)

Final Project Report

Compile Date : **15 May, 2003**

Supervisor(s): **Dr. C.L.Wang, Dr. W.Wang and Dr. A.T.C.Tam**

2nd Examiner: **Dr. K.S. Lui**

Project Members: (CE)

- **Cheung Hiu Yeung, Patrick** **2000090550** **hycheung@csis.hku.hk**
- **Ho King Hang, Tabris** **2000090598** **khho2@csis.hku.hk**
- **Sin Pak Fung, Lester** **2000227701** **pfsin@csis.hku.hk**
- **Wong Tin Chi, Ivan** **2000277287** **tcwong2@csis.hku.hk**
- **Yuen Man Long, Sam** **2000248937** **mlyuen@csis.hku.hk**

• TABLE OF CONTENT

1 INTRODUCTION....4
1.1 BACKGROUND....4
1.2 MOTIVATIONS....4
1.3 GOAL AND SCOPES OF INVESTIGATION....5
1.4 DIVISION OF LABOUR....6
1.5 STRUCTURE OF REPORT....6
1.6 CONVENTIONS....6
2 PEER-TO-PEER NETWORK ARCHITECTURE AND NETWORK PROTOCOL....8
2.1 SCOPE OF INVESTIGATION....8
2.2 ANALYSIS....10
2.2.1 Connection Performance of Internet....10
2.2.2 Real Time Response....11
2.2.3 Client Server Architecture....12
2.2.4 Communication Subsystem....13
2.3 THEORETICAL PRINCIPLES COMPARISON....16
2.3.1 TCP vs UDP....16
2.3.2 unicast VS multicast....19
2.3.3 Frame per second System VS Event-driven System....20
2.4 FEASIBILITY OF JAVA....22
2.5 DESIGN AND CONSTRUCTION....25
2.5.1 Game Engine....25
2.5.2 Network Architecture....25
2.5.3 Network Protocol....26
2.5.4 Reliable UDP design....30
2.5.5 Application Protocol....33
2.5.6 Game Server....34
2.6 METHOD OF INVESTIGATION....35
2.7 ANALYSIS OF APPROACH AND RESULTS....36
2.7.1 UDP reliable transfer performance....36
2.7.2 Updates of the communication system....38
2.7.3 Congestion Control Comparison....39
2.8 CONCLUSION....41
3 PARTITIONING SYSTEM AND SOUND ENGINE....42
3.1 INTRODUCTION....42
3.2 PARTITIONING SYSTEM....46
3.2.1 Overview....46
3.2.2 Theoretical principles....46
3.2.3 Partitioning Mechanism....48
3.2.4 Design and construction....50
3.2.5 Results....53
3.2.6 Discussion of the partitioning system....53
3.3 SOUND ENGINE....58
3.3.1 Introduction....58
3.3.2 Theoretical principles....58
3.3.3 Choice of sound engine....60
3.3.4 Design and construction....60
3.3.5 Result and Conclusion....61
3.4 EVALUATIONS AND CONCLUSION....62
4 SYNCHRONIZATION SYSTEM....64
4.1 INTRODUCTION....64
4.1.1 Scope of investigation....64

4.1.2 Current design and implementation trend ... *64*
4.1.3 Significance and Objectives ... *65*
4.2 ANALYSIS OF SYNCHRONIZATION MECHANISMS ... 66
4.2.1 Synchronization mechanism ... *66*
4.2.2 Consistency Protocol ... *67*
4.2.3 Synchronization Algorithms ... *68*
4.2.4 Conservative Algorithms ... *69*
4.2.5 Optimistic Algorithms ... *69*
4.2.6 SUMMARY OF SYNCHRONIZATION MECHANISMS ... 71
4.3 INVESTIGATIONS ... 73
4.3.1 Incentives of investigations ... *73*
4.3.2 The choice of Bucket Synchronization ... *73*
4.4 OVERALL ARCHITECTURE ... 76
4.4.1 Synchronization architecture ... *76*
4.4.2 Game World Clock ... *77*
4.4.3 Classification of commands ... *77*
4.5 BUCKET SYNCHRONIZATION ... 78
4.5.1 An Example of Bucket Synchronization ... *78*
4.5.2 Software Architecture of Bucket Synchronization ... *80*
4.5.3 Analysis of Bucket Synchronization ... *82*
4.6 MULTI-STATES SYNCHRONIZATION (MSS) ... 84
4.6.1 An Example of MSS ... *85*
4.6.2 Implementation of MSS ... *86*
4.6.3 Analysis of MSS ... *87*
4.7 EXPERIMENTAL RESULTS ... 89
4.7.1 Scope of experiment ... *89*
4.7.2 Number of Rollback vs Frequency of command ... *89*
4.7.3 Number of Rollback vs Synchronization delay ... *91*
4.8 EVALUATIONS AND CONCLUSION ... 93

5 GRAPHIC ENGINE AND COLLISION DETECTION SYSTEM ... 94

5.1 INTRODUCTION ... 94
5.1.1 Responsibilities of this module ... *95*
5.1.2 Scopes of investigation of this module ... *95*
5.1.3 Structure ... *95*
5.2 METHODS OF INVESTIGATIONS ... 96
5.2.1 Feasibility studies ... *96*
5.2.2 Prototypes ... *97*
5.2.3 Tests and Benchmarks ... *97*
5.2.4 Specifications of benchmarks and tests ... *97*
5.3 GRAPHIC ENGINE ... 99
5.3.1 Analysis of Problem ... *99*
5.3.2 Feasibility Studies ... *100*
5.3.3 Design and Construction ... *107*
5.3.4 Test Results and Analysis ... *116*
5.3.5 Discussions and Evaluations ... *120*
5.4 COLLISION DETECTION SYSTEM ... 124
5.4.1 Analysis of Problems ... *124*
5.4.2 Feasibility Studies ... *126*
5.4.3 Design and Construction ... *130*
5.4.4 Testing Results and Analysis ... *144*
5.4.5 Discussions and Evaluations ... *151*
5.5 OVERALL EVALUATION ... 156
5.5.1 Efficiency of research, design and construction ... *157*
5.5.2 Interactions with other modules ... *158*
5.6 CONCLUSION ... 159

6 3D MODELING, ANIMATIONS AND REAL TIME SHADOWING ... 161

6.1 THEORETICAL PRINCIPLES ... 161
6.1.1 Overview ... *161*

6.1.2 Determining whether a point is above or below a plane given a plane equation 161
6.1.3 Silhouette Determination 163
6.1.4 Shadow Volume 165
6.2 DESIGN AND CONSTRUCTION 168
6.2.1 Overview 168
6.2.2 3D Modeling 168
6.2.3 Animation 180
6.2.4 Character control mechanism 185
6.2.5 Game logic 188
6.2.6 Graphical effects 190
6.3 FEASIBILITY STUDY ON SHADOW ALGORITHMS 196
6.3.1 Fake shadow 196
6.3.2 Vertex Projection 196
6.3.3 Shadow Z-buffer 196
6.3.4 Static shadow 197
6.3.5 Shadowing volume casting with Zfail approach 198
6.4 PERFORMANCE ANALYSIS 199
6.5 DISCUSSIONS AND CONCLUSIONS 200
7 TESTING & CONCLUSION 201
8 REFERENCES 202

1 Introduction

1.1 Background

In the 1960s, who would have ever thought that the computer video games would be growing so rapidly and so dramatically impacting our popular culture and entertainment art around the world? It is amazing that in such a brief time the computer and video game industry has truly revolutionized entertainment. Games now influence films and books, and make use of a wide variety of popular music and licenses. The industry had also produced advanced technology that offers game players a rich, immersive interactive entertainment experience, which many find more compelling than passive art forms like movies and television. In 1975, an agreement between Sears and Atari ignited the growth of the retail video game industry, with the "Pong" machine.

Since then, more and more different consoles were born. The most famous examples are the Nintendo Entertainment System, Super Nintendo, Sega series, and the Play Station series. In recent years, the personal computer game industry has been developing even more rapidly. Games are no longer restricted to standalone consoles. Interactivity between players becomes important and feasible thanks to the Internet. This raises our interests to develop this next-generation Internet game, the "Mecha Zeta"

1.2 Motivations

As game evolves, it now comes to the generation that online action games that is capable a large amount of players with excellent visual quality and experience. However, our group notices that most of the online action games are still using the client-server architecture, such as the well-known Quake, which is a First-Person-Shooter game. The number of

players in a game is limited to a relatively small number say below 10. Apart from game capacity, we also notice that commercial games always use approximate collision detection to detect contact between objects in the world to gain performance, such as MDK2, which is also an action game. Moreover, pre-computed shadows are always used in commercial games instead of real time shadowing.

As a result, our group catches the insights and tries to design and implement the game MechaZeta which aims at solving the above problems, and evaluate the success of our implementations.

1.3 Goal and Scopes of investigation

Our goal is to test the feasibility of developing **interactive, real and large capacity real-time multiplayer games under unreliable Internet communication in Peer-to-Peer architecture**. Upon the development of computer graphics, our group dedicates to the **performance and quality of object collision detection**. The goal is to implement **an efficient and accurate collision detection system as well as a robust and extensible graphic engine, realistic models, motion and special effects**.

Below are the scopes we are going to investigate:

- Peer-to-Peer network architecture over the Internet
- Partitioning
- Peer-to-Peer synchronization
- Graphic Engine Design and Accurate and Robust collision-detection
- Real-time shadowing, motion simulations and modeling

1.4 Division of Labour

As we are group of five peoples, the duties are distributed into five main areas:

Members	Duties
Cheung Hiu Yeung, Patrick	Peer-to-Peer network architecture and protocol System flow and game logic
Wong Tin Chi, Ivan	Peer-to-Peer Synchronization System
Sin Pak Fung, Lester	Partitioning System Sound System
Ho King Hang, Tabris	Graphic Engine Design Collision Detection System
Yuen Man Long, Sam	3D Modeling and Animation Real Time Shadowing System

1.5 Structure of report

In this report, analysis, designs, implementations, testing and evaluations of individual areas will be carried out in order to maintain the readability of individual module. After all, an overall testing of the system we implemented will be carried out as well as the analysis.

1.6 Conventions

In this report, all the main text is with font Times New Roman, size 12 and normal weighting. The text with font Courier New, size 12 and normal weighting is either source code, pseudo code, or class name.

All reference points are numbered with square brackets. Complete references list can be found in the Section 8.

2 Peer-to-Peer Network Architecture and Network Protocol

2.1 Scope of Investigation

One of the most important components of a game is the backend system of the whole system. The meaning of backend system includes the application protocols, network protocols and the basic operation of all types of game data. All of these elements are now realized by player easily, but is very important for a player to play the game smoothly. Therefore, the design of this backend system is an important part of a game to start playing.

Every game should have its own application protocol in order to facilitate the basic function like joining the game or firing a target. The design of application protocol should be based on the needs of user and make use of low level functions to achieve the result. Next, since our game is an Internet game, it must involve the use of network protocol when players are playing with each other. There are many kinds of network protocol theory available nowadays. Since we are going to implement a game which must have some customized network functions available for playing, our focus on protocols will be mainly transport layer protocol, the basic network protocol essential for communicating on the Internet. The reason is that most high level network protocol is designed for some specific use and may not be suitable for our game. We need to make use of the basic transport layer protocol to make a customized network protocol suitable for our design of the game. Furthermore, the design of the network architecture is also one of the main issues in our game. Our main goal of network architecture design is to archive a scalable,

interactive and real-time Internet game. Currently, most Internet games do not involve high interactivity or high scalability of players. Therefore, we try to evolve our idea by implementing the improving network architecture – Peer-to-Peer architecture to adopt in our game.

2.2 Analysis

2.2.1 Connection Performance of Internet

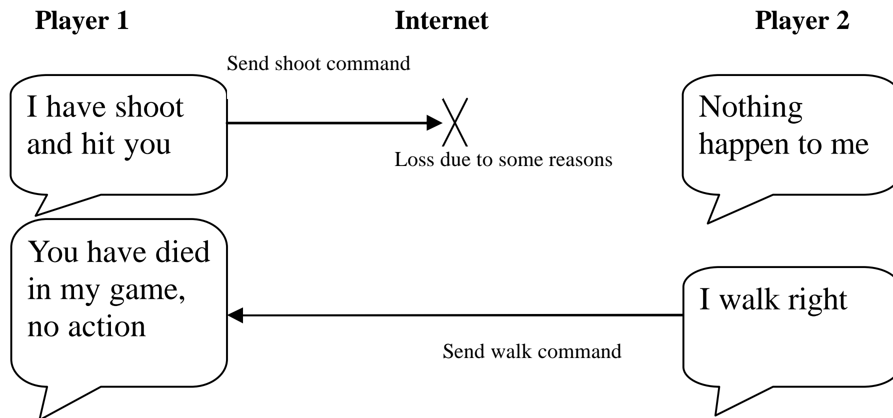
The first problem in designing the network protocol will be the poor connection over Internet. Internet is actually a large pool of networks. Each network is connected with routers, difference kinds of cables, firewalls or other network hardware. One can send a message to a person located in the other side of the world through this pool of networks. However, there is no guarantee that the message will arrive to the receiver correctly and on-time. In fact, when a person starts to send a message, the message first pass through the local routers and routed to the outside network. It then travel various channels, visit numerous routers and then to the destinations. During this trip, it may lose due to several reasons: router buffer overflow, physically damage to cables, out-dated route table, random queuing delay in router and many other reasons. It is tested with the 'ping' function provided by the OS. It is usually at least a few hundreds of round trip time from various locations. And sometimes even has lost packet even the location is actually reachable. There are some simple results of ping functions:

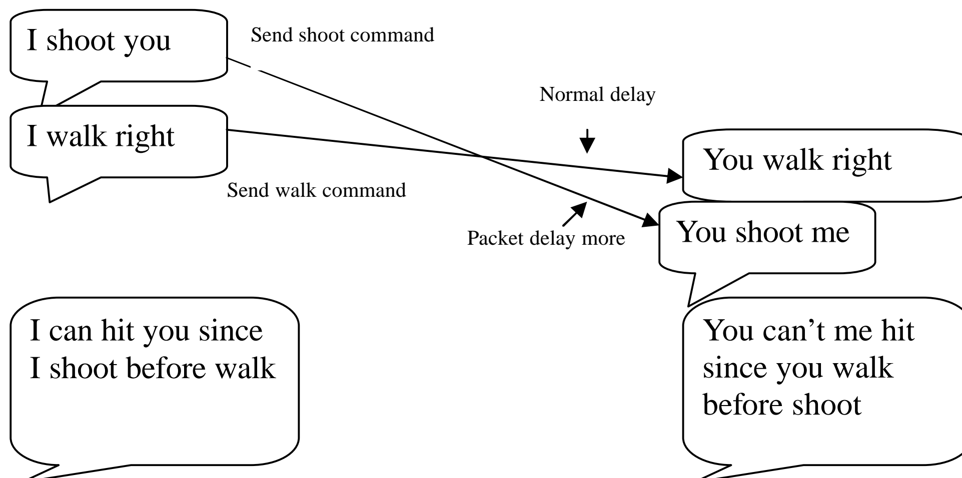
destination	**Max RTT(ms)**	**Min RTT(ms)**	**Avg RTT(ms)**	**Packet sent**	**Packet replied**	**Packet loss**
www.yahoo.com	908	966	932	4	4	0
hk.yahoo.com	15	29	20	4	3	1
www.google.com	494	521	507	4	3	1

Testing Environment: windows XP with home 10Mb broadband

This can be concluded that Internet has the capability to handle message flowing, but no proof for efficiency and correctness for the transmission. The above result is tested with ping function in which those are ICMP packets. If for some other packets like TCP or UDP ones, the result maybe even worse than the table list above.

For an interactive Internet game, losing packet or delay packet will result two main problems: inconsistency and slow or not real-time response. Let use a very simple example to explain the situation:





From the above two examples, it is found that maintaining game state and packet in-order is very important for the game. These two problems should be solved in order to survive in the poor connection Internet.

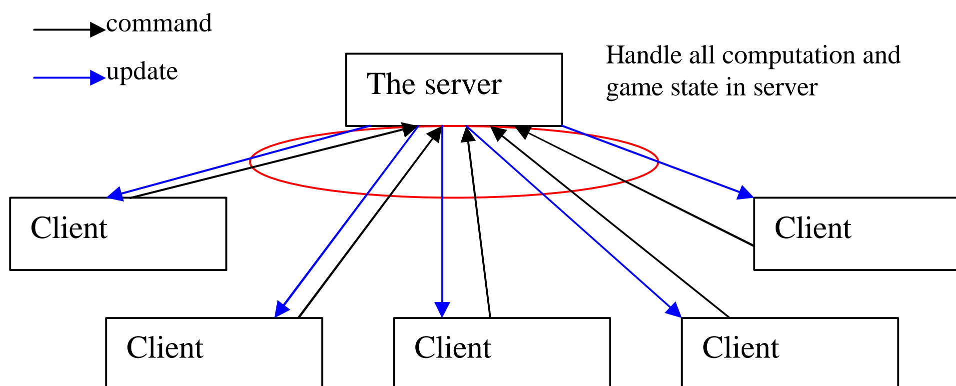
2.2.2 Real Time Response

For an interactive game, a real-time response is very important at the view of a player. As state above, the Internet itself has already produced many different kinds of delay on each

message delivery. This already greatly affects the real-time response of the game. Apart from the poor connection of Internet, the message size transmitted between also play an important role. If the size of the message is too large, the transmission delay will be larger for updating from one message. There should be solution that adopts a certain amount of delay for providing real-time response. In other words, when a player has issued a command, he may need to wait for a small amount of time in order to let the command being known by all other players. Then he performed an action as if everyone knows its action at the same time in order to simulate a real time response.

2.2.3 Client Server Architecture

Nowadays, most Internet games are using the Client-Server architecture as the backend system. In this architecture, the server is responsible for computing all game state and distributing updates about that game state to the client. The client acts as a view port into the server's state and passes primitive commands such as button presses and movement commands to the sever. Because this architecture allows most of the game state computation to be performed on the server, there is no consistency problem. The below graph show the whole design of this architecture:



Simplified Client-Server Architecture

However the gameplay is very sensitive to latencies between the client and the server, as

users must wait at least a round trip time for key presses to be reflected in reality.

And this architecture has another great problem which happens at the position of the red circle above. That is the connection is too centralized at the server. If the number of clients keeps increasing, the load at the server will also keep increasing. Although the computation power and bandwidth of a server is usually much higher than a single client, it cannot accept infinite or even large amount of clients at the same time. Then there will be a limit on the number of clients for a particular server. Also, if the server fails during the game, all clients will be disconnected and cannot play the game anymore. This is so-call the single point failure. In order to solve this problem, a different architecture should be adopted in our game.

2.2.4 Communication Subsystem

In a game, communication is one of the most important issues. Imagine that you are playing the game with a robot walking around. If you do not communicate with others with your movements or actions made during the game, others will not realize that what you are doing in the world. Then, they cannot act accordingly to perform some interactions. Then, you are just running the program with a robot walking around a terrain without any events happen. Therefore, it cannot be said that you are playing a game. So, designing a subsystem for all kinds of communication cannot be missed out during the development of the game. In our game, we have categorized the types of communication into three different kinds:

1. Getting data stored in the server when starting the game.
2. Each client needs to recognize the state of the game
3. Broadcasting the controls and position to other peers

The first one is the communication between a client and the server when initializing the

game. So, the steps of starting a game is defined as follow:

1. Try to establish a connection with the server before doing anything
2. Login the game by sending the user id and password to the server through the connection established.
3. If the login data is correct, the server will get the corresponding data from the database and send it to the client who is going to play the game.
4. Apart from sending those data, the game state of the peer group, which the client is going to join, is also sent in order to let the game start correctly.

So, it is very clear that this kind of communication needs a connection between the server and the client. And the transmission should be very reliable in order to let the client logging in the game and playing the game correctly without any errors. Also, the acceptable delay is quite high because when the client is going to start a game, it is usually very slow at the very beginning. Any users will feel acceptable even the start of the game is very slow.

The second kind of communication is the awareness of a game state for each client. Again, this is also very important, as all clients should have the same piece of game state for playing a game fairly. Also, clients should have a way to realize that who are playing with him at a certain moment. This kind of communication can involve both client and server, or clients only. We need to consider the different between these two kinds of awareness in the flow of the game.

The third kind is actually the most important kind of communication in the game. The controls made by one robot must tell other peers who are also playing within the same partition. Other peers should then update the player state according to the controls received.

The main requirement of this kind of communication is the little delay of delivery. If the delay is too high, inconsistency will arise similar to the case discussing above. Another concern is of course the accuracy of each delivery. Ideally, the command being sent and received should be the same without loss. However, to guarantee that the command being sent is in-order and arrive correctly will involve some mechanism like re-transmission and timeout. It will definitely make the delay of each command higher. Therefore, we need to make a deal with the delay of transfer as well as the accuracy.

2.3 Theoretical Principles Comparison

2.3.1 TCP vs UDP

As state above, the design of the game network protocol should make use of low level protocol in order to make a suitable one. In the five layer network protocol stack, we can make use of either transport layer or network layer protocol. However in Java, only transport layer is available, and therefore we try to research the two main protocols of transport layer: TCP and UDP.

Transmission Control Protocol, in short TCP, is a connection-oriented, reliable protocol. The meaning of connection-oriented is that it establishes an end-to-end link before any data moves. Both the sender and the receiver are aware of the connection. For reliable protocol, it safeguard against several forms of transmission mishaps. It will compare checksums included with a packet's data payload with a recalculation of the checksum algorithm at the destination to detect corrupted data. It will retransmit corrupted or lost data by providing methods for the destination to signal the source when retransmission is needed. If the data arrive out of sequence, this protocol will have a way to detect out-of-sequence packets, buffer them, and pass them to the Application layer in the correct order. During this process, it will detect and discard duplicate transmissions. The timeout of various acknowledgements is limited. The advantage of TCP is of course the reliability of delivery over any connection like Internet. However, the start of TCP is slow because of the three-way handshaking. Also, the congestion control

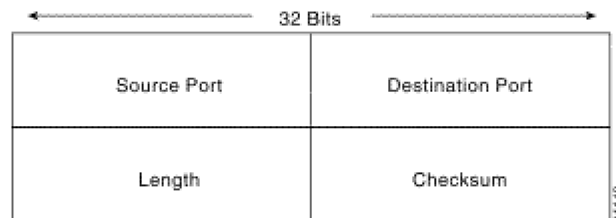
Source port			Destination port
Sequence number			
Acknowledgment number			
Data offset	Reserved	Flags	Window
Checksum			Urgent pointer
Options (+ padding)			
Data (variable)			

S1344a

TCP packet format

and the flow control of TCP also make the transmission slow when the network environment is not very good.

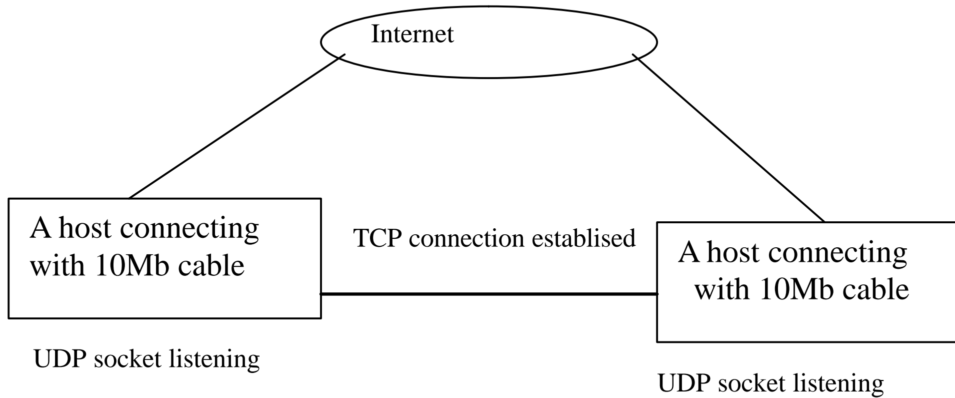
User Datagram Protocol, in short UDP, is a connectionless and unreliable protocol. It does not have an end-to-end link before any data delivery. All data are formatted into packets and then send out to the destination. There is no guarantee that the packets will arrive in-order at the receive side. Although it does not provide any reliability of delivery, it is relatively much faster at the start of delivery. It does not need any establishment and re-deliver for lost packet. UDP is the transport protocol for several well-known application-layer protocols, including Network File System (NFS), Simple Network Management Protocol (SNMP), Domain Name System (DNS), and Trivial File Transfer Protocol (TFTP).



UDP packet format

In order to decide which kind of protocol for the game, some tests were conducted in this aspect. We have implemented a ‘ping’ function for both the TCP connection and UDP connection. The idea of ping is rather simple. Each ping packet is formatted into a very small packet. The data contained in this packet is first an integer to indicate whether it is a ping message or ping reply message. The next component will be the timestamp. For a ping message, it contains the start time of the packet formation. For a ping reply message, it contains the timestamp of the ping message it received. The next step will be creating a thread to ping the peers periodically. It is set to ping every period of four seconds. The maximum timeout of one ping message is three seconds. If there is no message reply after this timeout limit, the message is treated to be lost and try ping again. Finally, establish a TCP connection with two hosts locating in a LAN environment and at the same time

running a datagram socket that accept UDP datagram for testing. The network environment is a simple Internet environment with two hosts. The set up is shown in the following figure:



The test is conducted by start the 'ping' function from host 1 to host 2. We first test with the TCP connection, ignore the start time of a TCP connection. After ten trials, we start the UDP 'ping' function with the same ping message being transferred in TCP. All values got from the program are generated from the method under Java class *System*, the method is called `System.currentTimeMillis()`. The result was as follow:

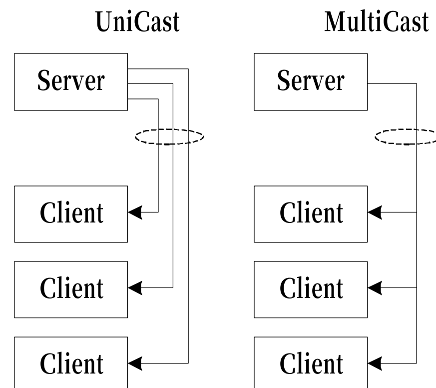
Trials	**TCP Connection RTT (ms)**	**UDP Datagram RTT (ms)**
1	261	40
2	311	100
3	300	120
4	891	60
5	300	timeout
6	241	30
7	321	141
8	261	110
9	651	timeout

10	901	80

Base on the result, it was found that TCP is relatively slow compare with fast response time of UDP. However, lost of packet of UDP does happen frequently when the network traffic condition is poor.

2.3.2 unicast VS multicast

Another issue will be the delivery of UDP packets. The most common and traditional way of using UDP was to first create a datagram packet containing the message that was going to send to others, then add the IP address and port number of the destination host and finally send out through the UDP socket. This method is called unicast of UDP. Therefore, if one wanted to send a piece of data to many others, he would have to prepare many copies of that piece of data and enveloped with the UDP datagram packet and then send them out one by one. Each



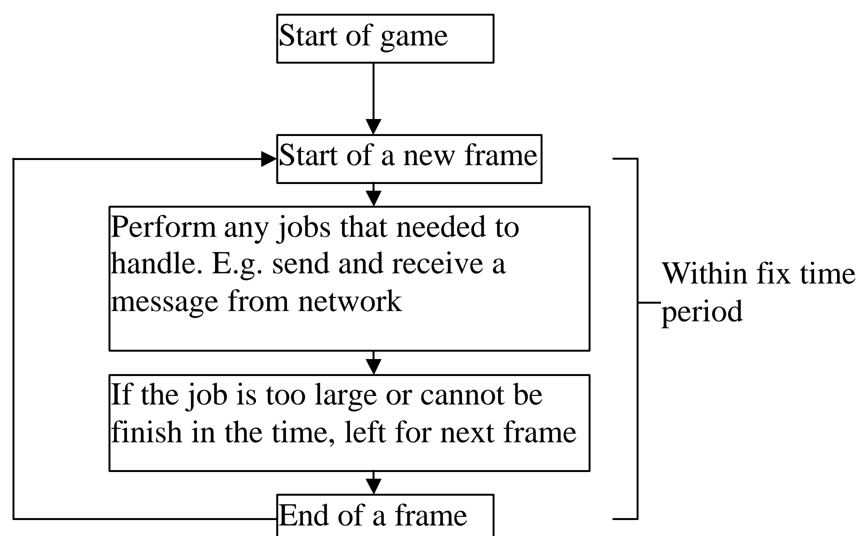
packet is then routed to various destinations by list of routers. It was actually a waste of time and bandwidth when both sender and receiver were in large amount, where exactly a multiplayer network game was. Therefore, there was another way of sending out the status of a client to other clients in the peer-to-peer network and that was the UDP multicast. The idea was that instead of duplicating the copy of data and the packet, there was only one copy for all hosts receiving that piece of data. The main principle of multicast is that some clients join to a class D IP address (224.0.0.0 to 239.255.255.255 inclusive) to form a 'group'. Any member who wants to broadcast the message to his group members just need to send a message to the group he has joined with correct port number. Then the router will

recognize that it is a multicast message and forward to all other members automatically. The diagram shows the main difference between unicast and multicast. It can be seen that with the use of multicast the bandwidth of the channel can be saved a lot when the number of clients grow larger. It is no doubt that the performance of the game will be higher with this faster delivery. Therefore, some tests have been conducted based on these two features as Java provides libraries ready for both unicast and multicast.

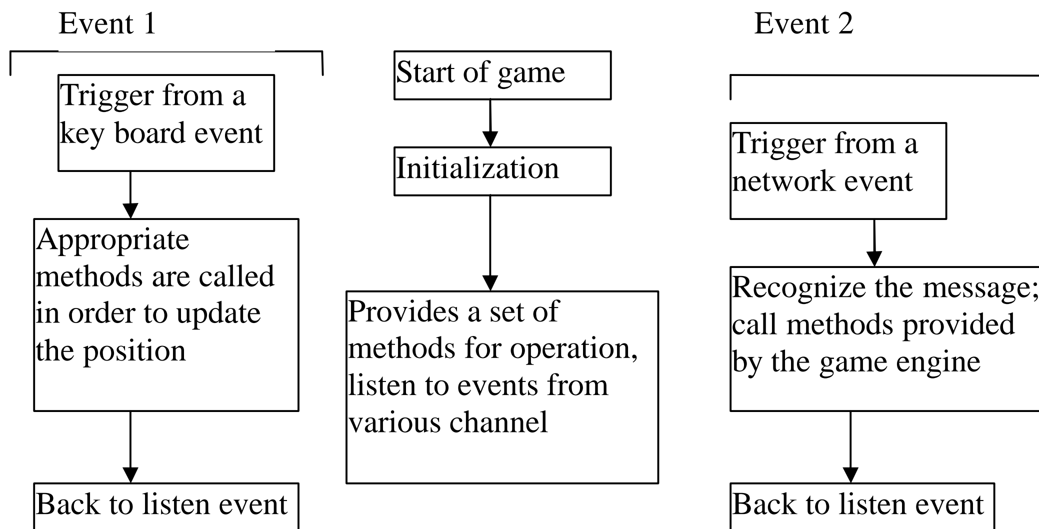
2.3.3 Frame per second System VS Event-driven System

Since a game is a very complicated system involving many different components such as network, graphics, sound … etc. Then every part of operation may not all done in a sequential manner. For example, the 3D engine needs to update the graphics in every frame to produce some animation. At the same time, there are some messages passing through the network to other hosts. If all the works are done sequentially, this may result late send message to the network or poor performance on smooth animation. Therefore, concurrent programming is needed for a game engine. There are two common ways for concurrency: FPS and Event-driven.

Here is the graph explaining what the flow of FPS design is:



Here is the graph explaining what event-driven design is:



The main different is that the FPS design only involve one main thread while Event-driven one has various threads for handling events. Both designs will be investigated in the area of game design as well as the feasibility of Java.

2.4 Feasibility of Java

Since our implementation will be based on programming language Java, it was first investigated whether Java can support for the network components implementation as well as different kinds of concurrent programming.

The basic network components provided by Java is under the `java.net` package. It has the libraries support for TCP and UDP implementation. For TCP, there are two kinds of socket, one is called `ServerSocket` and the other is called `Socket`. `ServerSocket` is the TCP server implementation in Java. It supports the method like `listen()` and `accept()` in order to let TCP client to establish a connection. `Socket` is for a TCP client to connect to the server. In Java, TCP is treated as a streaming protocol and all transmission is done through a stream. Therefore, both sockets have the methods `send()` to pass some data into that stream in order to send to the receiver. For UDP, then socket classes are `DatagramSocket`, the most basic one, and `MulticastSocket`, which support the basic operation of multicast transfer. Therefore, it is possible to test for the UDP unicast and multicast in Java easily and readily. Java also defines the data structure for flowing in and out these two sockets, which is `DatagramPacket`. However, Java does not support any lower level implementation on network stack. Therefore, the testing of network is limited to the transport layer only.

Another point of is the concurrent support of Java. In Java, the class Thread is the main idea of concurrency. A thread in the run-time interpreter calls the `main()` method of the class on the Java command line. Each object created can have one or more threads, all sharing access to the data fields of the object. The method for implementing a thread in Java is to extend the class *Thread* for a new object. Then child object will need to override

a method `run()` to start working as a thread. There is another method to have the same effect as extending the Thread class. An object can choose to implement the interface `Runnable` and then again override the method `run()` to produce the same result as before. However, the scheduling of multi-threading in Java is not well performed in the latest version Java virtual machine. Therefore, it will be a problem in multi threads scheduling. Despite of lack of scheduling, Java provides various methods and utility of thread handling. For example, there are methods like `Thread.sleep()` and `Thread.interrupt()` for suspending a thread or stop a thread respectively. Although there are no well-implemented libraries for timer in Java, it is possible to produce a FPS system based on the suspension and other operation on Threads.

All events handlings are provided by 2D graphics `java.awt` package. There are many examples like `KeyEvent` or `MouseEvent`. And it is possible in Java that make a customized event and so the event handler. The steps are as follow:

1. Build a new class that extends `EventObject` in Java. This is the main event class. All data that happens in an event are stored in this class
2. Build a new interface that extends `EventListener` in Java. This is going to tell others what event it is now happening and act as a bridge between the event and the object that perform certain actions based on the event.
3. Implement the listener interface at the desired object in order to listen to that event. Override the method provided by this listener and perform the job.

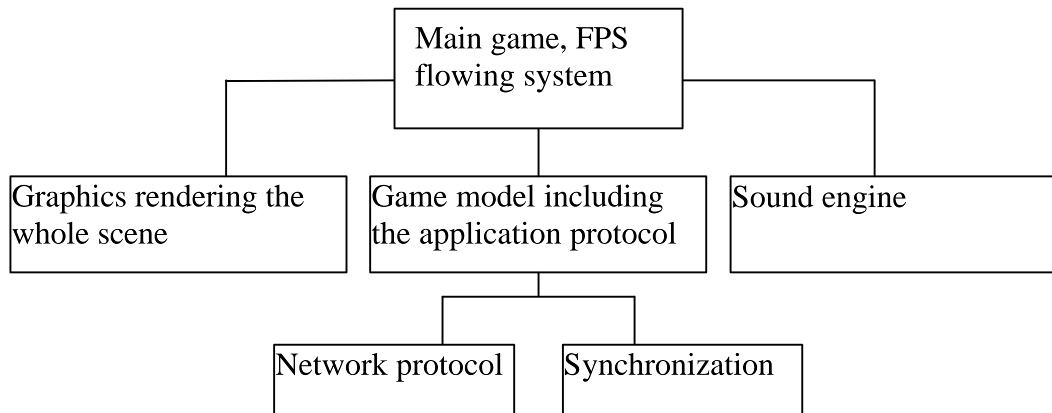
Therefore, it is possible for Java to write an event-based system based on the readily formed event system.

It is also found that Java is also suitable in designing an Object Oriented Game System with high productivity and suitable for 3D graphics and collision detection. For detailed tests and analysis, please refer to Section 5.

2.5 Design and construction

2.5.1 Game Engine

Below is the diagram showing the whole game engine design:



All components are connected and some of the components will be explained below.

2.5.2 Network Architecture

In order to start the implementation of the game, we must first decide for the main network architecture of this game. Since our main purpose is to enhance the scalability of the game, we decide not to use the traditional Client-Server architecture as it is already proved for the limit of players and the limit of server calculation. Therefore, we decide to use the peer-to-peer architecture. However, typical peer-to-peer architecture is not exactly what a game can apply on. A server should be present in order to provide the game details for each players of this game. On the other hand, this server should not handle the updates from clients and maintain game state. Instead, it is going to be a supervisor of each client for their details update. Therefore, there should be a channel between clients and server, provided that this channel will not be busy over the game running. Then, all clients should connect in a peer-to-peer way in order to communicate with each other. However, as state above, this will be a problem if the number of clients is growing. That is mainly due to the

limited bandwidth of each client. In order to solve this problem, all clients are partition into different groups in the network. There were no direct communication between each partition and thus limit the bandwidth workload to the number of only one partition. In order to provide management of a partition, one client is chosen to be the coordinator of one particular partition. He is responsible to communicate between clients and server. For more details of partitioning, please refer to the another report from one of my partners concerning partitioning. Below is the rough graph for the whole design of network architecture:

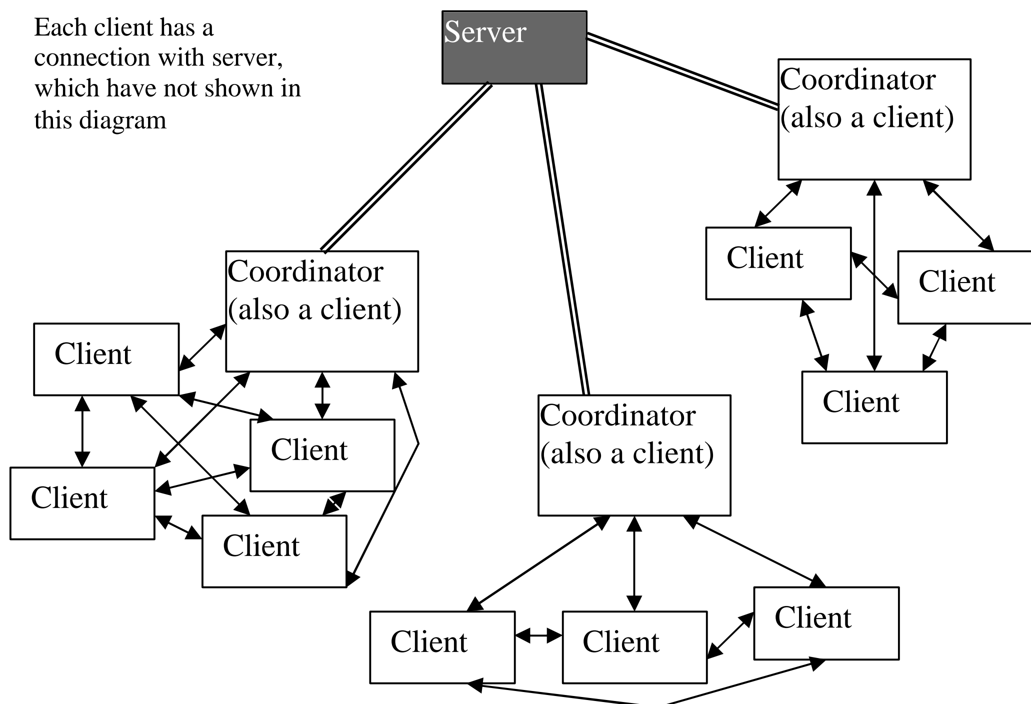


Diagram showing the whole network architecture

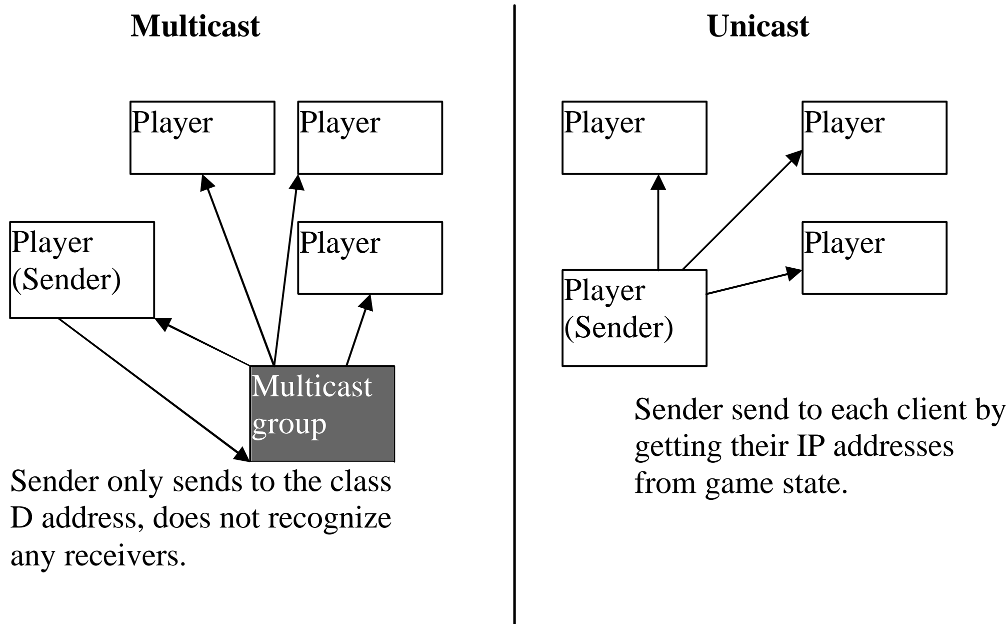
In order to cope with this design, the first step is to determine what kinds of protocol should be used in each different channel. And then set up suitable application protocol for the game playing in this architecture. The next paragraph will show the basic protocol built between different components.

2.5.3 Network Protocol

We have discussed before that there are three main types of communication needed. Each one is having different properties with different usages. Therefore, we have implemented a hybrid approach of these two protocols. The first one is the initialization of a game. At the start of the game, since all data must arrive to the client in order to start the game, we use TCP connection from one client to the main server. There were two kinds of socket, one was the server and the other was the client. The TCP server was running at the game server only. It could support multiple connections simultaneously from many clients. Whenever a client wanted to establish a connection to retrieve some data, a new thread would be produced for handling that particular client. Each of these threads would be recognized by the client's IP address. In order to preserve the uniqueness of each connection, we avoided the situation that hosts with the same IP run two copies of the game at the same time. Apart from getting game details at the start of the game, TCP is also useful for the channel between a coordinator and the server. The main usage of this channel is periodic updates of game state of each partition. Therefore, this channel requires a reliable transfer rather than a fast one. As each client must establish a connection with the server at the very beginning, it is convenient to use this TCP connection for the coordinator usage. Also, it will not result network congestion at the point of server. Another advantage of using TCP for this channel is that the server can change the coordinator very conveniently. Consider that one coordinator leaves the game suddenly. If a connectionless protocol is used, it needs more time to determine correctly that this coordinator has left the game due to the lost packet. Therefore, a connection-oriented protocol can tell the server at once when the coordinator leaves the game. Then the server can act quickly to choose a new client as the coordinator. For the bi-directional connections within peers, it is clear from the diagram that the density of connection is quite high. It is not feasible to use TCP for those connections, as this will make the workload on network part too heavy for every client. Another point is that clients

will keep on moving to any places in the game world. The partitions it belongs to change with time. Once it has joined a new partition, he will need to disconnect with the old partition and then connect to the new partition. If TCP is used in this situation, the slow start of TCP will be significant to client. We would like to maximize the efficiency of changing partition. If UDP is used, there will be no connection overhead no matter where the client has been to. Only one thread is needed for the UDP socket in order to listen to packets flowing in. Another reason is of course the faster round-trip time of UDP over TCP. To produce a real time response of a game, the delay of any messages should be minimized. Although the delay is still there for UDP, smaller value of delay will result a better synchronization within the network. For more details, it can refer to the synchronization part of the game.

After the use of UDP is confirmed, the next step is choosing whether unicast or multicast should be applied to the architecture. In the view of performance, multicast is better than unicast as it can save the bandwidth of the channel and so reduces the chance of congestion. Consider a case that there are eight clients within the partition. If they are all using multicast communication, assuming the packet size is 1KByte, then the send channel will have a rate only 1KByte/sec if the packet is sent within 1 seconds. But for unicast, the rate will become 1*8 KByte/sec. If there is a congestion rate bound set for the game, say 20Kbytes/sec, it is obvious that unicast will reach much faster then multicast. In order to investigate this proposition, we have implemented both the system of multicast and unicast. We try our test in a LAN environment with four players connected to the game. The implementation of these two methods is shown as follow:



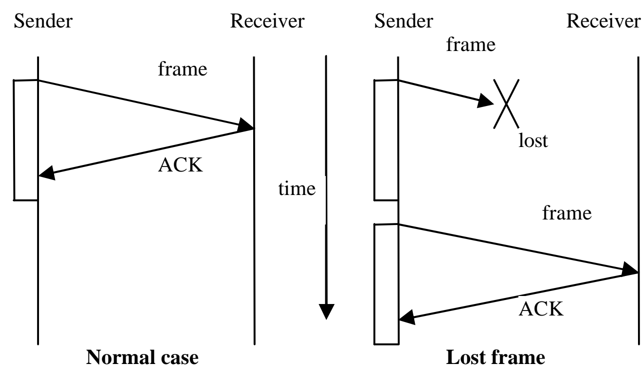
It was found that multicast has never reached the congestion rate limit while unicast has sometime above that rate. Therefore, in the view of performance, multicast is much preferable than unicast. However, there is a main disadvantage of multicast. It is not well supported by the Internet. Although there are some applications of on the Internet making use of multicast, there should be some special design routers located in the routes. Another problem is that it is difficult to avoid multiple receiving of one message. For example, consider two players are at the margin of two partitions. Then according to the partitioning algorithm, they both at the same time belong to two partitions. If one of them wants to tell others what action they have taken, they need to send a message to the multicast group. Since he belongs to two groups, it needs to send to two different multicast groups. However, at the same time another player who is also belonging to two groups will therefore receive this message twice. This is a problem for updating a state of a robot as action will be taken twice. If unicast is used, since it is to access the game state first in order to get the IP addresses, some precautions can be taken to avoid sending an identical message to the same player even they are belonging to more than one partition. Therefore, we concentrate on the application and improvement of unicast delivery in order to let the

game survive in the Internet. The two main classes that handle TCP and UDP respectively in a player are `TCP_Client` and `UDP_Client`. For the server, there two classes to handle the connection from clients by TCP, and they are `TCP_Server` and `TCP_serverThread`. The former class is going to handle any new connection from a new comer, while the latter class is the return of each connection. Once a new thread is formed, the client will use that thread to keep the connection with the server.

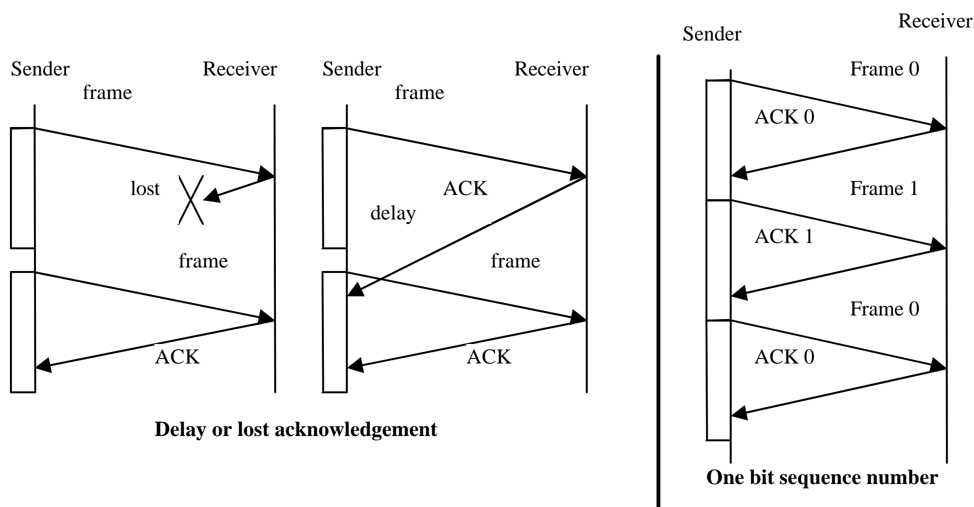
2.5.4 Reliable UDP design

Since the communication between peers in the network must be done by UDP, there is no way to communicate in a reliable manner. For example, a player wants to send a secret message to a particular user, then the message should not be lost if it is some mission command. Another case is that a client may need to update its game state from other client due to some reasons like out-of-sync or corrupted state. Then one client in the peers should try to send the game state to that particular client for updating. This transmission should be no error and fast delivery. Therefore, a reliable protocol is needed.

In order to provide a reliable one, either TCP or customized reliable UDP can be used. As tested before, the round trip time of a TCP packet is much larger than a UDP packet, and connection establishment also needs some time. Therefore, a reliable UDP protocol is needed to co-operate with the original unreliable UDP. The most basic reliable protocol will be the stop-and-wait protocol. The main idea of this protocol is that after transmitting a frame from the sender, it waits for the acknowledgement from the

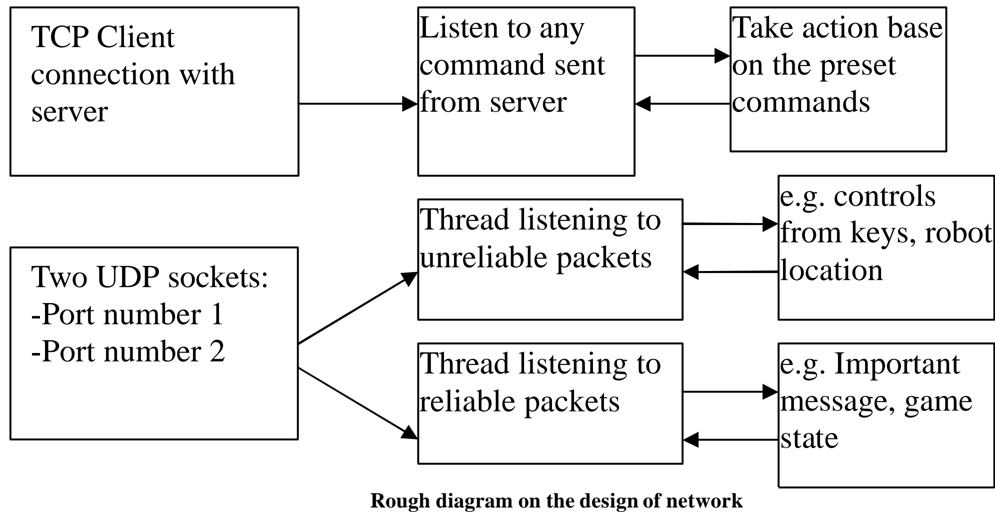


receiver to indicate that it has received the frame correctly. If the sender cannot receive this acknowledgement in a certain period, it will consider the transmission as lost and re-transmit the last frame. But this idea is not enough for a reliable transfer. If an ACK is lost on delivery, the sender will certainly consider its last transmission as a lost. However, the frame actually arrives at the receiver correctly. Then the receiver side will have duplicate copy of the frame. If this problem exists, the final re-assemble data will be corrupted. Therefore, a sequence number is needed to indicate the order of the frames. This sequence number can be simple as 0 or 1. If consecutive frames at the receiver have same sequence number, it must be a duplicate frame and therefore discarded. This protocol is so-called the one bit stop-and-wait protocol.



When implementing this reliable protocol, since it is also a UDP design, it should not mix up with the unreliable one, as there is no sequence number and acknowledgement. In order to separate these two different channels, the port numbers of them are difference. Therefore, if one packet is coming from the reliable port, it is treated as a reliable packet and performs data integration on that. The diagram below shows the operation of the network

components:



Rough diagram on the design of network

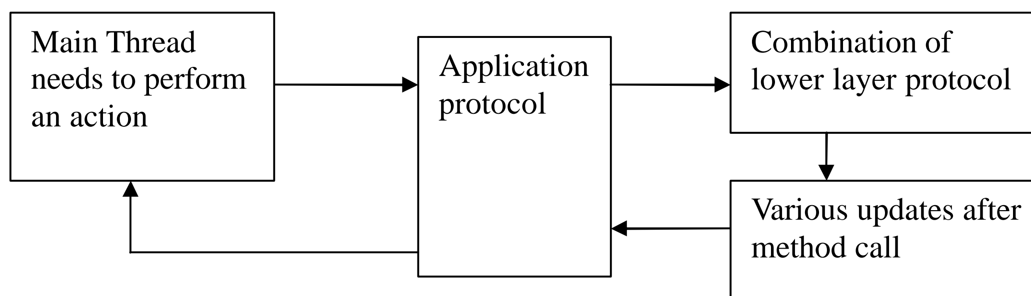
The implementation again is located in the `UDP_Client` class. There are two methods to handle reliable delivery. They are `sendReliableData()` and `receiveReliableData()`.

Another challenge is the outburst of controls being made by the player. If the player keeps on pressing the key continuously and very frequently, a lot of commands will be formed. From the view of a game, it is unable to determine whether the commands formed is needed or is just duplicate commands. Therefore, all those commands or most commands will need to be sent to other peers as the most basic communication. Then, the communication channel of this client will be full of packets containing those commands. It can be imagined that when the channel is full of packets, those packets do not have much work done on flowing, as there are too many packets. It will definitely make the delivery more slow. Therefore, the network protocol should have the function on congestion control. A simple congestion control called rate-based control is implemented in the game. First set a limit for the send channel rate, say 20Kbytes/sec. Then, calculate the rate of the send channel at a certain time interval. If the number of packets sending out at a moment exceeds this value, the number that is available for sending out in the next frame will be

less. The excess packets will be waiting at the buffer of the sender. If the sending rate is lower again, the number will be increased again. In our game, the value is adjusted to be 20Kbytes/sec because this value is acceptable for an Internet connection nowadays and at the same time maximize the channel for communication.

2.5.5 Application Protocol

The application protocol of this game includes various operations of the game. For example, when a player wants to start a new game, he must first send his login details to the server. Then the server will authenticate this player and retrieve the details that the user has got before. It sends all these information to the player in order to let him to start the game. Then the server should also send to the new comer that the details of existing members in the partition it is going to join. Next, it also needs to send the new comer's details to those existing members. All actions are treated as a method in either the game server or the client. The main principle of the application protocol is a bridge between player and all of the other components. Each method in this protocol consists of various kinds of lower layer methods like calling the network to send reliable data and then update the state of the game.



Main role of application protocol

One important function of the application protocol is to recognize whether a client is still playing the game.

2.5.6 Game Server

A game server is essential in many games, including ours. The responsibility of the server is not updating game state or calculating the game logic of our game. Instead, it is a supervisor on clients and also the database of all clients' details. As stated above, each client will establish a TCP connection with the server when they are going to start the game. Therefore, there is a thread that listening all the time to see if there is any new comer. The class `TCP_Server` will be responsible for accepting connections. It then checks if this IP address has already be used. If yes then do not start the connection with that player. The next thing the server will do is to authenticate the user id and password in order to find the appropriate data of that player. The database control is relatively simple and all data are now stored in a plain text file only. Server also keeps track of what ID has joined the game at a certain moment to prevent duplicate ID from happening in the game. The method of realizing whether the client is still alive in the game is determining the connection status of the TCP connection established at the beginning. If the connection is closed at the client side, the client is considered as leaving the game. Also, the server is acting as the backup of the network and the game. It will request the coordinator of each partition to upload the game state to it periodically so as to keep an update copy of game state. If there is any problem happens in one partition, server can load back the previous game state to restore the game accordingly.

2.6 Method of Investigation

The main investigation method is separated into two parts. The first part is to run the game in a LAN environment which a server is running in one of the machine in the LAN. Another part is testing in the Internet environment by connecting some clients in local to a server that located somewhere in the Internet.

In the LAN environment, first setup the server in one of the workstation. Then several clients start at the same time and connect to that server by knowing the IP address of that server. In order to check for the scalability of the game, we run a copy of the game in a computer in the LAN. Then we start playing the game with each other and monitor the message flow and the reaction of each player based on the command sent. Also, monitor the statistics produced during the game run on

1. round trip time from each client
2. send channel rate
3. status of each client and consistency

For Internet connection, all works are just same as the LAN environment setup. However, we concentrate more on the network status and also provide a method to compare statistics from different players. As it is an Internet environment, each player is not able to view others directly. All necessarily data will be shown on the screen with some debugging message. We record the data from the debug message as the same time observer the whole game flow of each player.

2.7 Analysis of approach and results

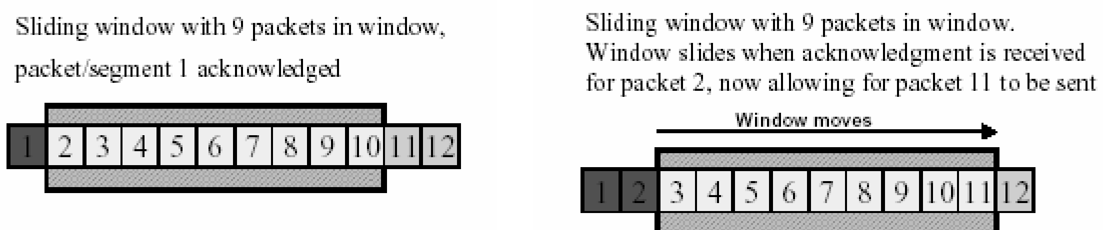
2.7.1 UDP reliable transfer performance

There is a comparison between the TCP file transferring and the UDP one bit stop-and-wait protocol file transferring. Two files are going to be sent out, which has two different sizes, one is 10K Bytes and the other is 100K Bytes. First we test with the LAN environment for comparison. The timeout value set for the UDP reliable protocol is 1 second and the packet size is set to be 1K Bytes. Then, at the second phase we try to send in an Internet environment. The result is as follow:

	LAN environment (ms)		Internet environment (ms)	
	TCP	UDP reliable	TCP	UDP reliable
File 1 (10K Bytes)	198	220	400	3412
File 2 (100K Bytes)	23724	3451	45564	43887
File 1 (10K Bytes)	341	202	3114	5089
File 2 (100K Bytes)	25286	1054	23293	46768
File 1 (10K Bytes)	187	190	3135	2744
File 2 (100K Bytes)	26057	2791	42880	37504
File 1 (10K Bytes)	241	201	3204	4576
File 2 (100K Bytes)	23553	5992	39456	38475

The time calculated for TCP is the time difference that starts from forming a connection to finishing the whole transfer. For UDP reliable, it is measured from start sending the whole data from finishing the whole transfer, includes cutting the data into packets as well as the timeout. It was found that when the size of the packet is not large, the efficiency of TCP is even better than UDP reliable. This is mainly due to the lost packet of UDP in Internet and the time needed for re-transmission and also the splitting of whole punch of data into

packets. However, when the data size is large, the performance of UDP is better than TCP as the re-transmission and flow control of TCP give more contribute on the delivery. During the test, it was also found that the performance of UDP reliable was not very constant comparing with TCP. That was mainly due to the re-transmission of packets. If that particular transfer consists of little lost packet or acknowledgement, the efficiency will be much higher than a TCP one. Therefore, we can get a conclusion from the result: If the network condition is excellent with very little packet loss of UDP, the performance of a customized reliable transfer is much better than a traditional TCP. However, if the network condition is not very good, the number of re-transmission of UDP will increase and thus the performance will be affected greatly. One reason why the efficiencies of UDP reliable and TCP have no much difference when tested in Internet environment is that the implementation of UDP is not optimized. The one bit stop-and-wait protocol has a drawback that each time only one packet can be sent out to the network. If this packet is lost, the whole system should wait for at least the timeout time in order to determine it has been lost and re-send the packet. The channel is clearly not fully utilized with the flow of packet. Another method that can solve this problem is called sliding window protocol. Actually this protocol is what a TCP is using nowadays. Both the sender and the receiver will have a group called window, which the size are the same. Below is the graph explaining the use of sliding window:



This method allows the sender to send packets up to the window size before an

acknowledgement is required. However, this method is efficient only if the channel involves a one-to-one file transferring. If one needs to send a file to several receivers, then the available channel of the sender will easily be congested and thus result a waiting at the sender. For a network game, a player always need to send a command to many others and so sliding window protocol is not very suitable for a game. Also, the size of a message is not very large at one time. Then it is no need to split a piece of data into many packets and thus cannot make the most use of sliding window. This is recommended to adjust the windows size so as to get a size that can have good performance at the same time not make the channel congest.

2.7.2 Updates of the communication system

Another important issue is the status of a game state in the peer-to-peer architecture. Since all players have their own copy of game state, it is hard to maintain the consistency of all game states. As this is related to the synchronization part, the result of synchronization can be referred to that part. Here we show the result that is without applying the synchronization mechanism. In a LAN environment, the result was satisfactory on the consistency even without synchronization. The delay of one command was very little and peers can show updates instantly. It was easy for any peer to recognize whether one of a peer is still alive in a partition. The round trip time show between each client was very small and the lost of packets seldom happen. Same work was performed under the Internet environment. The result was not as good as the LAN one in the area of consistency. Some control packets often lost and result a different action of a robot in the game. For example, the ‘start moving’ packet from a remote player is lost during transmission. However, the action has performed locally at the local machine and thus the state of these two players is different. Although the state is consistent again by means of the later updates from clients

that the position and orientation of the robots, the game did not run smoothly. In fact, the delay of UDP packets is quite small during the testing even in the Internet. The round trip time showing in the ping functions provided by the application protocol is relatively small. Therefore, the application of UDP is more suitable for a highly interactive game over TCP. One more point to point out is that the serialization of Java is quite fast comparing with the network delay of sending that packet.

Another aspect of this architecture is the awareness of connection of a player in the network. Although the ping implementation is available at the UDP clients, lost of ping message is possible to happen. In the first trials, a player treat a peer to be disconnected if cannot ping that peer only one time. It is tested with only two peers connected to each other through Internet. It was found that there was an occasion player one cannot ping player two but in fact player two was still running. Therefore, we should tune the value of re-try on ping mechanism. Here is the list of number of times needed for retry on ping to produce correct result:

No. of Times sending a ping message	Time for failure of a ping (min)
1	3
2	14
3	29
4	No failure in testing time

Testing time: 30 minutes, Timeout for Ping: 3sec, Time period for retry: 4sec

Therefore, sending four times for a ping message to determine the connection of a client is the most suitable one.

2.7.3 Congestion Control Comparison

In our game, the congestion control being applied is a simple sending rate detecting mechanism. Whenever the sending rate is too high in certain period, the available number of packets will decrease in the next frame. It is comparable with some other congestion control mechanism. One of them is called the AIMD congestion control. AIMD stands for Additive Increase Multiplicative Decrease, which is used by TCP. The idea is that whenever the channel is congested, it will decrease the sending rate by half. Afterwards, the rate will be increased gradually by a certain amount each time. Since the rate is reduced by half any time the channel is congested, it will make the delay of a command change rapidly, which will make the game suddenly move slowly because of this sudden change. We want to avoid this situation. Another one is the equation-based congestion control, which the optimal sending rate is calculated based on the sending channel situation. It is good at finding a suitable sending rate without a sharp change. However, this may not suit our game as it involves heavy calculation on find the rate each time. The game engine itself is actually very complicated. If the network protocol is also very complicated, the delay of a command may also higher. We would like to keep our congestion control mechanism to be a simple one and will not increase the load of the game engine much.

2.8 Conclusion

Network protocol plays an important role in an Internet game. Although a player cannot realize the situation from the screen, network has done a lot of communication with other players. The implementation of a peer-to-peer architecture is rather complex and need time for testing. However, this can really solve the problem that the workload of the server is very high. Many nowadays Internet game depends too much on the server computational power. Although computational power of a server is increasing by some techniques like clustering, there is still a limit on that. The potential of peer-to-peer architecture on this aspect is growing with the advance of network protocol. The idea of separating the computation among several computers is actually what a network is going to do. Problems in a peer-to-peer network are still here and are not easy to solve. Yet developing this is a must in the future trend. Therefore, we need to do and try more to reveal the true power of this architecture.

3 Partitioning System and Sound Engine

3.1 Introduction

In order to support the large scale of the game, the fundamental network architecture should be carefully designed. There are mainly two type of network architecture we may use.

Client-Server Architecture

This architecture is commonly used in current game industry. Under this type of infrastructure, every player is acted as a client and is connected to a server. The control of game logic, flow of messages, maintenance of a consistent game state are all done by the central server. After the player has login to the server, the server will issue the existing game state to the new players. When the player move or take an action, its command would first be transferred back to the server, the server would then calculate the new game state and then update every client upon the changes.

Since the server would keep a central game state and every client updates their own game state from it. . The only copy of precise game state is stored at the server. Every client only requires updating their own based on it. So the major benefit of this architecture includes the ease of game state synchronization. Other benefit includes the protection of the game from cheating, etc.

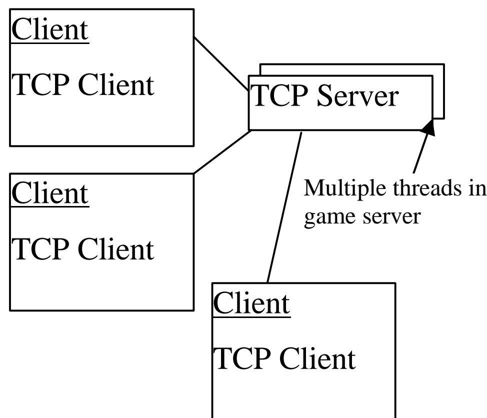
However, since every message transfer must be passed through a server, the client-server bottleneck is significant and therefore, it is not favored for a scalable internet game.

Moreover, it takes at least a round trip time between a player pressed a button and the screen was updated. Therefore, if the game requires instant response, such as our robot-fighting game, this is unacceptable.

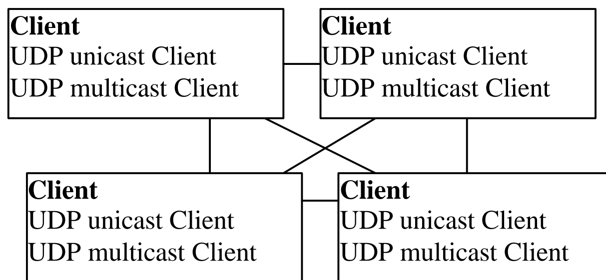
Peer to peer Architecture

This architecture does not require the existence of a server. Each client maintains its own copy of the game state based on messages from all the other clients. The clients will contribute its own computational power in calculating various kinds of data within the game. This can solve the problem that the messages need to first send to a potentially distant server and then propagate back to the clients for updates. Since they communicate with each other directly, this simplifies the transfer mechanism and could eliminate the client-server bottleneck. So it is favored for a scalable game.

However, the major drawbacks of it include the difficulties to maintain a user data for the internet game, and keep a synchronized game state.



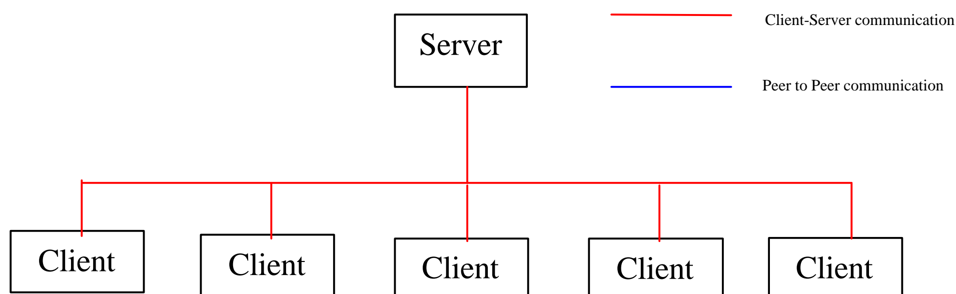
The client-server architecture



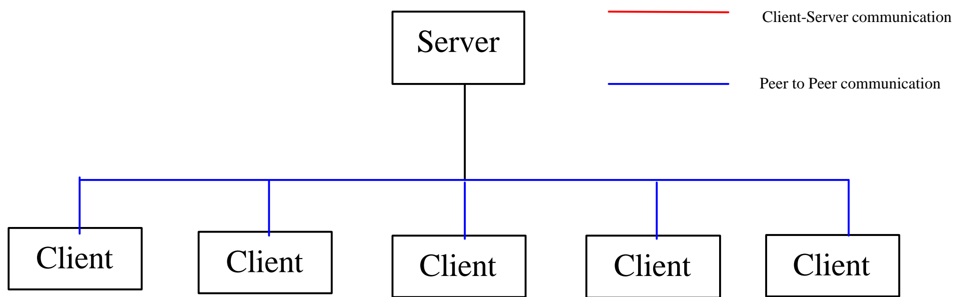
The peer to peer architecture

Hybrid Architecture

In our game, we decided to implement hybrid architecture. That is, we implemented both the client-server network architecture and the peer to peer architecture. When concerning messages which are critical and permanent to the game, such as registration/login, update of player's information, then client server architecture was used. However, when users interacting with each others, such as the update of position of robots, the message would be transferred using a peer to peer approach so as to reduce the time cost.



Transfer of critical data (Client-server architecture)



Transfer of client messages (Peer to peer architecture)

3.2 Partitioning System

3.2.1 Overview

As mentioned in the above chapter, since we were using peer-to-peer network architecture in the communication among players, most of the messages in the game were sent to other clients directly. However, this might be resulted in message flooding in the network. Let me use an example for illustration.

It is assumed that there are 100 players playing in the game now. In every second, every player is required to update it new position to others players. So in every second, there are 100 * (100 – 1) messages flowed in the network. As our game is stressed on the scalability, the number of players is expected to be large. It is essential for us to minimize the message transferred in the game.

Total massage in unit time = 100 * (100 – 1) = 9900 messages

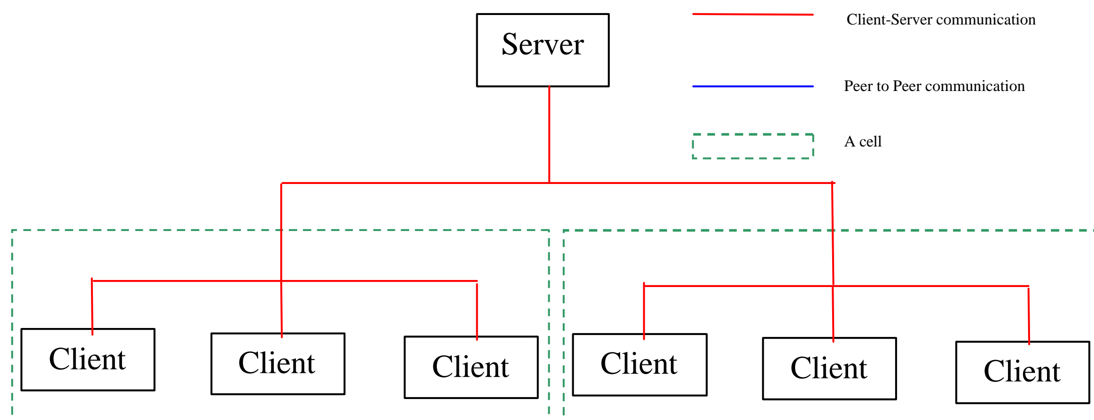
Fortunately, after investigation, it was found that we could minimize the network traffic by the game nature. Since the visible area of a robot should be limited, it is natural that they could not see robots far away. In other words, every robot in one place need not take notice of others that are far away from him.

Because of this reason, we had implemented a partitioning system. It aimed at grouping 'related players' together. When messages are transferred, they will only be transferred to those 'related players'. It is believed that this could minimize the network traffic by a huge amount.

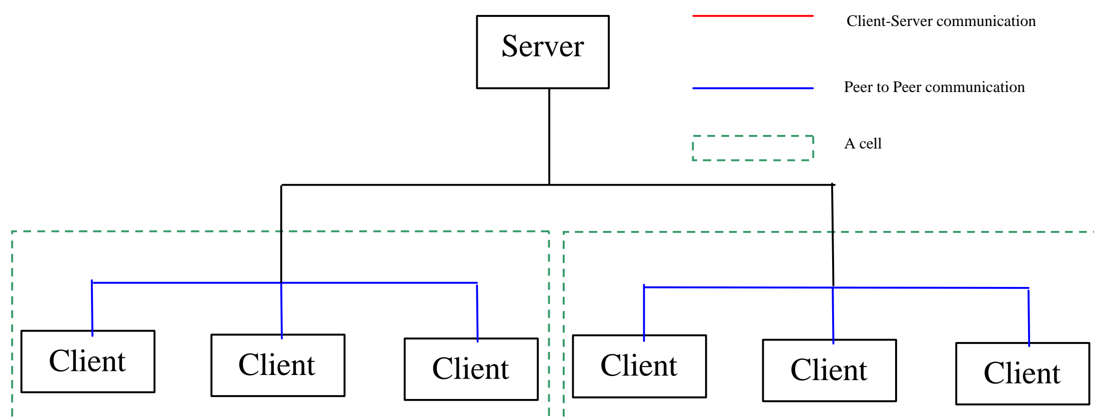
3.2.2 Theoretical principles

Partitioning system here refers to the grouping of players in such a way that essential network communication must be made between them. The game world is partitioned into different regions. Each region is named as a partitioned area, or a ***cell***.

In doing so, the network architecture could be further modified as follows.



Transfer of critical data (Client-server architecture, w/ partitioning system)



Transfer of client messages (Peer to peer architecture, w/ partitioning system)

Let us discuss our example back. Assume those 100 players are now separated in 10 cells. Each cell contains 10 players. At each time interval, every player need to send its message

to other players that belong to the same cell, that is, (10 – 1) players only. Therefore theoretically, the network traffic could be reduced as follows.

Total message in unit time = 100 * (10 – 1) = 900 messages

3.2.3 Partitioning Mechanism

There are two main types of mechanism to perform partitioning of the game world. The first one is the static partitioning mechanism. The other one is the dynamic partitioning mechanism.

Static partitioning mechanism

As the name implies, the partitioning structure is pre-defined. That is, the number of cells, the size of each cell is fixed and unchanged after the game has been started. The main advantage is its high efficiency as most of the setting could be pre-calculated and pre-loaded. The only thing concerning partitioning when playing the game is to load the related static data and perform corresponding actions.

Dynamic partitioning mechanism

This partitioning method initiates the game world with only one cell. When the number of player in this cell increases and reaches a maximum, the cell would be split into smaller. By performing this recursively, the number of cells increases as the number of player increases. It is obvious that dynamic partitioning mechanism allows control of maximum number of players in the cells. It is theoretically extensible as well.

Comparison

When comparing the two approaches, it is shown that the dynamic mechanism provides control on the maximum number of player in each cell, so the amount of traffic in every cell could be predicted and in control. However, the static mechanism generates faster response on sending message to adjacent cells, as no calculation is involved at run time.

In short, we could summarize their characteristics in the following table.

	static mechanism	**dynamic mechanism**
Determination of adjacent cells	Pre-calculated	**Calculated at runtime**
Control of maximum traffic in cell	**Not supported**	**supported**

Since we decided to use P2P architecture, every client should know whom he or she would need to communicate with. For a dynamic system, updating all players upon change of cells is required. That is, there could be frequent update of all clients by the server from time to time.

However, if a static system was used, the partitioned world could be pre-calculated and loaded to every client at compile time. Therefore, we need to to handle any more runtime calculation.

As a result, the time saved in a static partitioning mechanism outweighs the benefits of the dynamic partitioning mechanism. Therefore, a static partitioning mechanism was used. In other words, a static partitioning is preferred in P2P architecture, while dynamic partitioning is preferred in client-server architecture, as the clients need not to know whom he or she is required to communicate with. The only machine they communicate with is the

server.

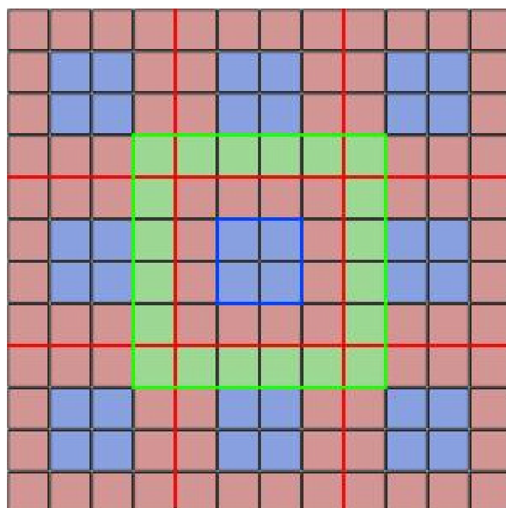
Justification

Since we had confirmed to use a static partitioning mechanism, we have to sacrifice the control of maximum number of players at a cell. We would like to minimize this weakness.

We adapted the partitioning system in such a way that when a new player enters the game, we would place it to a selected cell by a screening process. The screening process focuses on the number of player in the cells at that time. In so doing, we hope to minimize the number of players entering that heavily occupied cell.

3.2.4 Design and construction

We had implemented a static overlapping partitioning system. Each cell is overlapping with each other. When a robot is at private area, it sends its message to robots in its own cell. When a robot is at the overlapped area, it sends its message to all the cells associated. Our design could be summarized by the following diagram –



The Blue Area – the area without any overlapping. Robots here only send message to its cell.

The Red Area – the area that is overlapped by other cells. Robots here need to send message to its cell, and other associated cells.

The Green Area – the area that this cell overlaps its adjacent cells. Robots over there need to send their message not only to their cell, but also to this cell.

The detailed implementation was as follows.

The whole world was implemented as a 2D array of cells. The number and the size of the cells are predefined.

Each cell was implemented as an object. It stored the pointers of the all the robots belong to it.

Each robot would store a list of robot pointers that are visible. A robot also stores the associated cell information that it reserved. The information is in the form of an array of pointer of the game state of the cell. In every location of the world, there are at most four game states that the robot need to store:

- 1 game state when the robot is in the central area of the cell
- 2 game states when the robot is in the edge of the cell
- 4 game states when the robot is in the corner of the cell

In deciding the data structure for the whole world, there could be other choice such as a tree structure instead of the 2D array. However, the array structure was preferred. It is because,

(i) The tree structure could provide an efficient search time for a particular node. However, all the nodes in my game design are well defined. In other words, there is no need to search in the tree.

(ii) Since the viewable area for a robot is fixed, the number of cell that a robot should send its message to was also fixed at every point. Therefore, no matter which data structure was used, same number of cells should be retrieved. As a result, in our game design, the use of 2D array is preferred as the access time of the array is much shorter than the traversal in the tree structure.

Algorithm – Robot Joining

When a player joined a game, it would first connect to the server. The server would assign a cell to it and send the game state of that cell to the player. At the same time, the server would add this player's robot in its cell data structure, and then update other robots the existence of this new robot in the corresponding cells.

Algorithm – Robot Movement

When the robot moves, it needed not to notice the server so as to achieve the peer to peer design. Since the size and the number of cell were fixed at compile time, it was easy to find out which cell(s) the robot should send its messages to, according to their position at that time.

For me, I had implemented a mapping of position to the associate cell(s) in a class. A client could simply load it and retrieve the cell ID in every move. There is no need for any calculation, so time could be saved.

Algorithm – Cell Transition

As the robots moved, it is very often that a robot changes its cell from one to another. This is monitored by the server. There are frequent updates of the game state of each cell to the server by the coordinator. When the server receives an update of the game state, it checks if there is any robot that its belonging cell has been changed. If so, the server would notice that robot to change its cell and push the game state for the new cell to that robot. Then it would notice the robots at the original cell to remove that robot from their visible robot list. And finally, the server would notice the robots in the new cell to add this robot in their

visible robot list.

<u>Algorithm – Robot exiting</u>

This is rather simple. When a robot drops, the server need only to notice all the robots in its cell to remove it from their visible robot list.

Obviously, the cell transition part is comprised of pretty heavy calculation and in fact is the bottleneck in the whole partitioning system. However, this calculation was unavoidable.

3.2.5 Results

The implementation of a partitioning system definitely makes the game more efficient and reduces the network traffic. Its significance grows exponentially when the number of players increases. Therefore, the existence of such a partitioning system is crucial for a scalable online game.

3.2.6 Discussion of the partitioning system

As far as the performance of the partitioning system is concerned, it is found that the heavy calculation in cell transition should be put under discussion. Since robots are moving frequently, in order to keep precise determination of the cell that every robot belongs, a frequent update of the game state from the coordinator to the central server is necessary. Moreover, after the server receives each new game state, it has to keep checking all robots overthere to sort out those requiring a cell transition. So this arises two problems:

- The increase of network traffic between coordinator and server, so the client server bottleneck is made significant.

- The determination of cell transition involves heavy calculation of on the positions of every robot. This makes the central server heavily loaded and is not preferred.

Therefore, another mechanism is suggested. We can allow the checking of cell transition be done at the coordinator. Every coordinator only deal with the robots at its own cell. Since now the server is no longer responsible for the checking of cell transition, less frequent of game state update could be allowed. Also, the calculation work is now decentralized. Therefore, it seems that the 2 problems above could be solved at first glance.

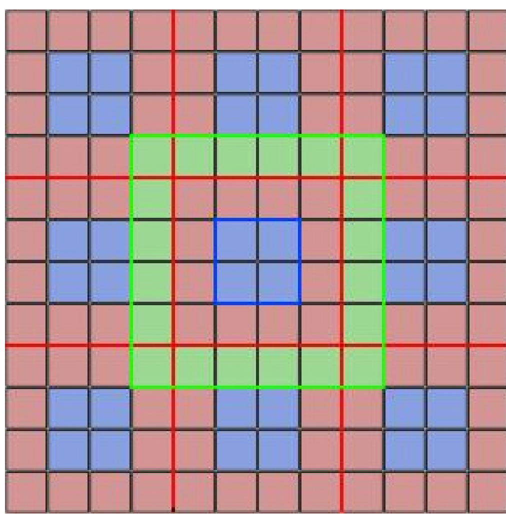
However, when we look at the suggested method more deeply, it contains its own drawback:

- After a robot is determined as 'transition required', the coordinator should report this case to the server side. It is because the update procedure consists of other cell and this cell does not get information of. As a result, the server role could not be avoided in the mechanism.
- Coordinators are actually normal player's computers. If the computation power of his/her computer is low, then the downgrading of its performance is considerable.
- It is also unfair for one particular player be selected to perform such a heavily loaded computation work, as the game performance could be adversely affected.

In conclusion, there is no better method to perform the checking of cell transition at this moment. As the computation power and the bandwidth of the server are controllable, we might increase the overall performance by using cluster server and larger bandwidth. Therefore, the first method is still in use in our game design.

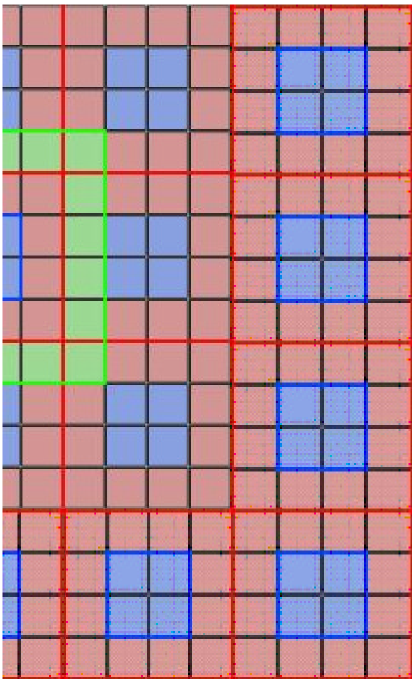
We had discussed and compared the usage of the dynamic partitioning mechanism and a static partitioning mechanism. It was concluded that since time saved in static partitioning mechanism outweighs the flexibility of the dynamic partitioning mechanism, we used the static partitioning mechanism in our game design. However, actually we might try to joint the two advantages together and consider a hybrid approach.

The idea is that at the compile time, the number and the size of cell are pre-calculated as usual. However, when the number of players grows and reaches certain upper limit, we may expand the game world by adding certain number of fixed size cell.



Original game world

Number of player reach upper limit.



New game world

(With more fixed-size cells)

In this way, the game world could be extensible. In other words, if the game engine is well designed and supported, the game could virtually support unlimited number of players.

However, it was foreseeable that the following problems could be resulted.

(i) In order to support the expansion of the number of cells, the game terrain should be expanded as well. Therefore, a mechanism to generate and load the terrain dynamically is required.

(ii) Since the cell map was now pre-loaded in the client size, once the game world expands dynamically, there should also be update on the cell map in the client side.

(iii) From the viewpoint of players, it would be strange if the game world expands suddenly when playing. Therefore, we may have to design a suitable story so as to suit the sudden expansion.

(iv) Other data structure and algorithm of the game may be required to modify in other to adapt it.

Unfortunately, due to the limited working time for this project, I could not implement this design and work out its feasibility. So, further study towards this direction is suggested to take.

3.3 Sound Engine

3.3.1 Introduction

In today's game industry, a remarkable sound entertainment is of equal importance as an outstanding graphic engine. A good sound engine is not necessarily representing the success of a game. However, a good game must include a quality sound engine. Therefore, for the sake of the players' interest, sound engine is indispensable in out project.

After investigation, it is found that there are many ways in implementing a sound engine in java.

(i) Using JDK 1.2 (making use of `java.applet.AudioClip`)

(ii) Using Java Sound API

(iii) Using Java Media Framework (JMF)

3.3.2 Theoretical principles

Java Sound provides a high-quality 64-channel audio rendering and MIDI sound synthesis engine that

- Enables consistent, reliable, high-quality audio on all Java platforms
- Minimizes the impact of audio-rich program on computing resources
- Reduces the need for high-cost sound cards by providing a software-only solution that requires only a digital-to-analog converter (DAC)
- Supports a wide range of audio formats including AIFF, AU, WAV, MIDI and RMF files

Java Sound can handle 8- and 16-bit audio data at virtually any sample rate. In JDK 1.2

audio files are rendered at a sample rate of 22 kHz in 16-bit stereo. If the hardware doesn't support 16-bit data or stereo playback, 8-bit or mono audio is output.

Java Sound also minimizes the use of a system's CPU to process sound files. For example, a 24-voice MIDI file uses only 20 percent of the CPU on a Pentium 90 MHz system.

The package of `java.applet.AudioClip` in JDK 1.2 is actually a built-in Java Sound Engine. It allows us to playback, loop and stop the corresponding audio clip.

JMF is a high-level API, designed mainly for easy playback of multimedia files, video as well as audio. Java Sound API, on the other hand, is a rather low-level API, designed for detailed control of the audio hardware.

The 'performance pack' versions of JMF use Java Sound to play and capture sound data. Because of this, they include the Java Sound implementation. This implementation is the same as in the jdk1.3.X. The 'all-java' version of JMF uses `sun.audio` classes to playback sound (capture is not possible here). So this version doesn't include a Java Sound implementation.

JMF has several features that Java Sound doesn't have:

- Much more codecs
- Support for synchronization between media streams, for instance syncing audio and video playback
- Support for streaming protocols like the Real-time protocol (RTP)

3.3.3 Choice of sound engine

In out project, the usage of `java.applet.AudioClip` in JDK 1.2 is finally chosen. It is because in our project, we need only to loop the background music and playback some sound effects. There is no need for us to handle the low-level control and manipulation of the audio clip.

JMF could be another choice. It is, however, a large package that supports also other media files like video. Since we would not use these features in our project, it is not chosen.

If there were further extension of the game such that some video clips will be playback, then JMF would be a better choice at that time.

3.3.4 Design and construction

First of all, we have to collect the media files for our game. We had downloaded many useful wav files from the free internet sources. The sound effect includes the walking of a robot, the shooting of a gun, the use of a sword, the explosion of a robot, etc.

For the sake of the playback performance, a little trick on the sound engine had been made. When the client program runs, all the sound files are pre-loaded. The pointers of the audio clips are stored in a hash map. Later, when the file are used, the pointer of that clip would be retrieved and start playing. This little trick could minimize the response time.

The `SoundEngine.java` provides public methods for the client program to call. When those methods are called, pointer of those audio clips is retrieved and corresponding

actions will be performed.

The `SoundList.java` contains a hash map which stores the pointer of the pre-loaded audio clips.

The `SoundLoader.java` load a media files to an audio clip and then add the pointer of the clip to the hash map.

There are three operations for an audio clip:

- *play* – Play the audio clip once. Use in most sound effect.
- *loop* – Loop the audio clip until a stop method is called. Use in background music
- *stop* – Stop the audio clip

The calling of these three method are event-driven, when the robot start to walk, then the sound engine is called to loop the walking sound for the robot. Once the robot stop moving, the sound engine is called to stop that sound.

Through these three functions, a simple sound engine for the game had been implemented.

3.3.5 Result and Conclusion

The performance of the sound engine is satisfactory. However, as we could not control the low level characters for an audio clip like volume, the sound given out is a bit soft.

3.4 Evaluations and Conclusion

Network architecture is 'invisible' but very important in an Internet game, even though a player cannot take notice of it from the screen. Owing to its importance, it is worthwhile for us to design a suitable and efficient infrastructure. The implementation of a peer-to-peer architecture is rather complex time is required for testing. However, this can really reduce the workload of the server and therefore, the client server bottleneck is minimized. Many nowadays Internet game depends too much on the server computational power. Although computational power of a server is increasing by some techniques like clustering, there is still a limit on that. The potential of peer-to-peer architecture on this aspect is growing with the advance of network protocol. The idea of separating the computation among several computers is actually what a network is going to do. Frankly, there is still room for improvement in a peer-to-peer network architecture. Yet developing this is a must in the future trend. Therefore, if we could try more on this area, we could become the pioneer of this future trend.

Besides the basic network infrastructure, other factors such as game protocol, partitioning in the game world and synchronization mechanism are also the main aspects of research to tackle the constraints of unreliable Internet communications.

In our project, we successfully implemented a static partitioning system to our game. However, as suggested in the previous chapter, this design was not perfect. It got its own weakness. There are two key points that we could further investigate so as to fine tune the system and make it even better.

The first one is the place where the determination of cell transition should be made. Currently it is done at the server side. Implementation at the coordinator could be another possible way, although it is not a good approach after analysis. However, as stressed before, since it is the bottleneck of the whole partitioning system. Further investigation is suggested to take. The second key point is the extensibility of the partitioned world. It is shown that an extensible world could be feasible which would result in unlimited number of players joining the game. This is obviously one of the goals in the present gaming industry. Therefore, further study towards this direction is recommended.

When it comes to the sound engine implemented in our project, I would regard it as 'just enough' for our game. If more sound operations are required to make in our game, then the low level Java Sound API would be a good choice among them. If apart from the audio, there are other media files that could enhance the competitiveness and the attractiveness of our game, then the JMF could provide more supports.

4 Synchronization System

4.1 Introduction

4.1.1 Scope of investigation

The present capacity of interactive Internet game is limited. The boundaries arise from the high consumption of client bandwidth and the unreliability of Internet. Transport and game protocol, partitioning in the game world and synchronization mechanism are the three main aspects of research to tackle the constraints of unreliable Internet communications to give response as close to reality.

As the nature of rapid responsiveness of real-time interactive game, the needs of eliminating network delay is indispensable to providing the human sense of "virtual reality". Adding to the quandary is the unreliable underlying Internet transportation. Synchronization evolves crucially to removing any discrepancy among clients and, simultaneously minimizing the response time.

The main and only investigation of this report pivots at the synchronization mechanism. S*ynchronization mechanism* defines a software architecture and implementation which detect and resolve inconsistencies revolving from the latency and loss of computer network communication. The *synchronization mechanism* deals with the details of maintaining consistent state with Internet communication underneath.

4.1.2 Current design and implementation trend

In turn-based or latency-insensitive games [4.1], conservative synchronization algorithms are adopted which perform poorly in fast-paced games where a constant rate of simulation is important. In fast-paced multiplayer game [4.1] like Mecha Zeta, therefore, optimistic

synchronization algorithms evolved to resolve the latency of conservative algorithms. Optimistic algorithms execute events or commands optimistically before they know for sure that no earlier events could arrive, and then repair inconsistencies. The speed at which game state changes causes the dead reckoning(rollback) to quickly multiply to resolve large divergences among clients.

The inadequacies of current optimistic algorithms are the anticipation of divergences and dead-reckoning when performing corrections of state.

4.1.3 Significance and Objectives

Since **Mecha Zeta** aims to accommodate an extensible capacity of client players under Internet, synchronization has to digest a large number of messages fired from rapid, frequent but delayed events. The project focuses on the "responsiveness" and "consistency" of the synchronization mechanism.

- Responsiveness vs Consistency
- Minimal disturbance to simulation
- Computation and storage overhead of error recovery

4.2 Analysis of Synchronization mechanisms

4.2.1 Synchronization mechanism

A shared view of a virtual world (game world) is often enhanced by replicating the information at each participant's site since replication allows local access which improves interactive performance [4.4, 4.5]. However, every update at each participant's site should be propagated to and its execution be synchronized with other participants to keep his view consistent with them.

As participants are geographically distributed over large networks like the Internet, the probability of view inconsistency among participants grows due to the increase of network delay [4.6].

Therefore synchronization of updates is a key to maintaining a consistent, shared view among participants in the game world.

Synchronization mechanisms are defined here 2 areas of study "*Consistency Protocol*" and "*Synchronization algorithm*". They are tailor-designed for specific applications upon the requirements of "responsiveness" and "consistency". The dilemma is due to the latency and loss of packets flowing in computer networks. "Responsiveness" defines the latency from the events generated by the system to the events be received and performed in the clients. The graphical rendering delay contributes to the latency but is not taken into account of "responsiveness" in synchronization.

Consistency protocol is roughly categorized as "state-based" and "command-based". "Stated-based" multicasts the client-generated game state from the client to all fellows. "Command-based" multicasts every command from the client to fellows. Presumption is the underlying Reliable Internet multicast protocol (no packet loss). The mechanisms to accomplish this should be as efficient as possible, both in bandwidth and computational

load.

Algorithms of synchronization are roughly being classified into 2 classes for virtual reality applications – conservative and optimistic [4.2]. "Conservative" algorithms perform poorly in fast-paced games where a constant rate of simulation is important, although they are still suitable for turn-based games. On the other hand, "Optimistic" algorithms execute events optimistically before they know for sure that no earlier events could arrive, and then repair inconsistencies when they are wrong.

4.2.2 Consistency Protocol

4.2.2.1 State-based Consistency Protocol

Game state is generated upon command execution. For a successful consistency protocol, each client should be able to generate the same game state from what to be popped from the protocol. It is, therefore, the direct incentive to multicast the individual game state generated at each client to fellows.

- ☑ *Easy to discover errors* - by comparing the game state received and generated
- ☑ *Correction is simple* – only to restore game state
- ☒ *Large consumption of bandwidth* – game state is normally larger than command
- ☒ *Large consumption of computation* – restoring a game state requires not only memory allocation but also repairing dynamic memory linkage.

4.2.2.2 Command-based Consistency Protocol

Each client timestamps commands, with reference to game world clock, reliably multicasts them on to fellows. Given the commands from each client and the time at which they were

issued, every client can compute the correct game state. No game state transmission is required in this protocol unless a global recovery occurs . The command-based consistency protocol is simple and provides low latency, reliable delivery of all commands. However, it does not distinguish between different classes of events and requires a many-to-many, low-latency, reliable multicast protocol.

☑ *Simplicity* in implementation

☑ *Without the need for a third-party trusted server* to generate global game state.

☑ *Efficient bandwidth usage* - small command packets.

☑ *Lowest possible latency* – commands packet flow around partition only

☒ Require a low-latency *Internet multicast protocol*

By using a reliable multicast protocol, we ensure that all commands eventually reach each fellow client. Each client can compute the correct game state by executing the commands in the order in which they were issued. In case there is discrepancy in execution of command packets, we must correct the game state when commands are received out of order. These corrections, or rollbacks, can cause unexpected jumps in the game state, severely degrading the user experience. Therefore, we must at all costs minimize the frequency and magnitude of rollbacks.

4.2.3 Synchronization Algorithms

Algorithms of synchronization are roughly being classified into 2 classes for virtual reality applications – conservative and optimistic. “Conservative" algorithms perform poorly in fast-paced games where a constant rate of simulation is important, although they are still

suitable for slower turn-based games.

4.2.4 Conservative Algorithms

“Conservative" algorithms perform poorly in fast-paced games where a constant rate of simulation is important, although they are still suitable for slower turn-based games.

4.2.4.1 Lockstep synchronization [4.2]

No member is allowed to advance its simulation clock until all other members have acknowledged that they are done with computation for the current time period.

4.2.4.2 Chandy-Misra synchronization [4.7]

Each member is allowed to advance as soon as it has heard from every other member it is interacting with. Additionally, it requires that messages from each client arrive in order. [Lemma: *Most updates come from interacting members*]

	Responsiveness	Consistency
Lockstep	**Extremely slow** – Fast client has to wait for slow client for every move.	**Absolutely high** – No rollback
Chandy-Misra	**Slow** – An advance version of lockstep but also depends on clients which are potentially slow.	**High** – Depends on the group size of members

Table 4.1 Comparisons of conservative algorithms

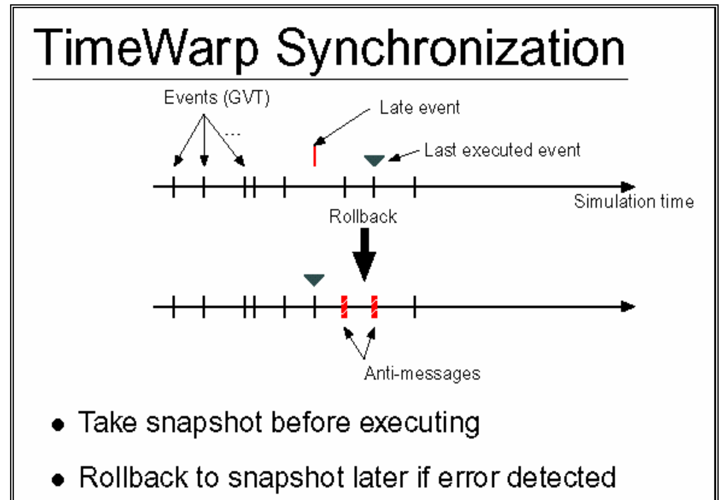
4.2.5 Optimistic Algorithms

“Optimistic” algorithms execute events optimistically before they know for sure that no

earlier events could arrive, and then repair inconsistencies when they are wrong.

4.2.5.1 Time-Warp Synchronization [4.8]

Fig 4.1 Time-Wrap Synchronization



Taking a snapshot of the state at each execution, and issuing a rollback to an earlier state if an event earlier than the last executed event is ever received.

☒ **Explosion of anti-messages**

As part of the rollback, anti-messages are sent out to cancel previously generated events that have become invalid

☒ **Great processing overhead**

Copying a context involves not just the memory copy but also repairing linked lists and other dynamic structures.

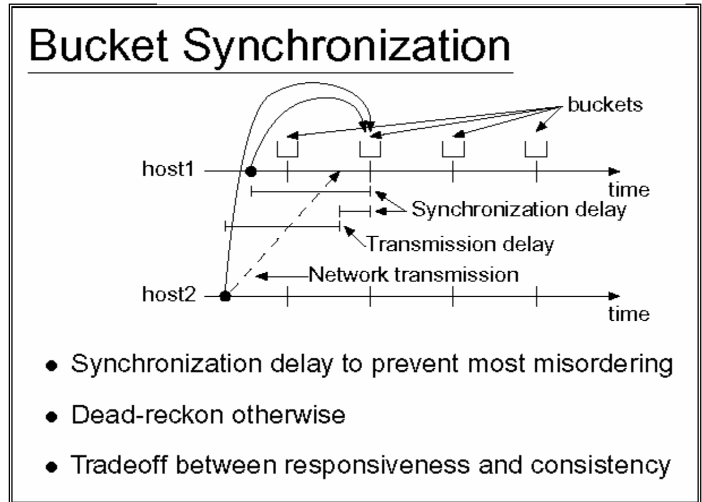
☒ **Less snapshots = More costly rollback**

4.2.5.2 Breathing algorithms [4.9]

Solve the problem of excessive rollbacks by setting an "event horizon" - Events beyond the horizon can not be guaranteed to be consistent, and are therefore not executed.

4.2.5.3 Bucket Synchronization [4.3]

Fig 4.2 Bucket Synchronization



Events are delayed for a time that should be long enough to prevent disordering before being executed. An event is held in a bucket at each participant's site if it is received before a given time limit called playout time and otherwise, the event is dropped. This allows an event to be played out at the same time at all participants.

	Responsiveness	Consistency
Time-Wrap	**Instant response** – events are executed without checking	**High** – rollback whenever error is detected
Breathing	**Instant response** – an extended time-warp.	Depends on the "event horizon"
Bucket	**Adjustable delay** – subject to the interval of buckets	**Low** – events may be dropped

Table 4.2 Comparisons of optimistic algorithms

4.2.6 Summary of synchronization mechanisms

4.2.6.1 Consistency Protocol

Since Mecha Zeta eyes surviving on the unreliable Internet, latency comes first in consideration. "Command-based" consistency protocol consumes the least bandwidth which is favorable to enhance the responsiveness of Mecha Zeta. On the other hand, Mecha Zeta eyes to entertain a certain large group of clients; the size and manipulation of game state is expensive. Therefore, higher consistency "State-based" protocol suits less.

The function of serving consistency is preserved by introducing classification of commands and events, which will be explained in the following chapter.

4.2.6.2 Synchronization Algorithm

Obviously, optimistic algorithms place emphasis on responsiveness while conservative algorithms aim to preserve consistency to responsiveness. For an interacting environment like Mecha Zeta, optimistic algorithms suit better to enhance the responsiveness.

However, in order to retain consistency among all clients, error correction and recovery must exist. The 3 different optimistic algorithms give almost instant response but the consistency and performance differ. Consistency of optimistic algorithms always contradicts the disruption of simulation.

It is the goal of optimistic synchronization to minimize the “flickering” effect caused by rollback or dead-reckoning when performing error corrections. To solve the problem, the performance of pre-processing and game state recovery should be investigated.

4.3 Investigations

4.3.1 Incentives of investigations

As Mecha Zeta is performing real-time interactions in the virtual game world, the direction of synchronization mechanisms is straight to optimistic algorithms. The investigations of these synchronization algorithms look into the following areas

- Responsiveness and Consistency
- Smoothness of the simulation
- Computation and storage overhead of error recovery

on the 3 commonly adopted optimistic algorithms - ***Bucket Synchronization***, ***Breathing Synchronization*** and ***TimeWrap Synchronization***.

4.3.2 The choice of Bucket Synchronization

4.3.2.1 Responsiveness and Consistency

With the study of the 3 algorithms in the previous chapter (Table 4.2), they all give satisfactory responsiveness because commands and events are not executed until absolute consistency is achieved. Most importantly, Mecha Zeta needs instant response which tolerates unnoticeable delay (informally, by experimental testing, 250ms). Here is the analysis of consistency. TimeWrap executes commands and events after taking snapshot of the game state. It does not guarantee the commands are in-order and performs rollback whenever faults are detected. The consistency is finally preserved since every fault is to be corrected. Breathing imposes a “horizon” to keep copies of events and commands where commands and events out-of-bound delayed cannot be detected. The consistency is subject

to the length of the horizon. Finally, Bucket bears the lowest consistency because events and commands arriving later than the bucket time, it will be dropped.

4.3.2.2 Smoothness of the simulation

The smoothness of simulation depends on 2 factors – synchronization delay and frequency of error corrections.

TimeWrap provides the least synchronization delay but no pre-ordering of commands and events. The frequency of error corrections is very high but the smoothness is fair because time-wrap points are unit. Breathing has no restriction on the synchronization delay but pre-processing of commands and events are not specified. Therefore, the frequency of error corrections is undefined. Bucket synchronization imposes a delay for each bucket but the frequency of error is reduced because pre-processing is taken place in each bucket. The flickering effect will be greater because intervals between buckets are larger than time-wrap.

4.3.2.3 Computation and storage overhead of error recovery

TimeWrap has to store up game states and commands of every execution which will eventually slump up the memory allocation. Breathing limits the boundary of commands and game states storage. Bucket slashes the storage of game states and commands because no backward restoring is performed.

For the computation issue, it depends on the size of game state when performing restore and re-execution. A high computational overhead may “halt” the “screen” of the client.

4.3.2.4 Conclusion

Within the 3 optimistic algorithms discussed above, none of it perfectly suits the needs of Mecha zeta – high responsiveness and consistency, smoothness and minimal overhead of error recovery. This project, therefore, focuses on other hybrid approaches.

4.4 Overall Architecture

4.4.1 Synchronization architecture

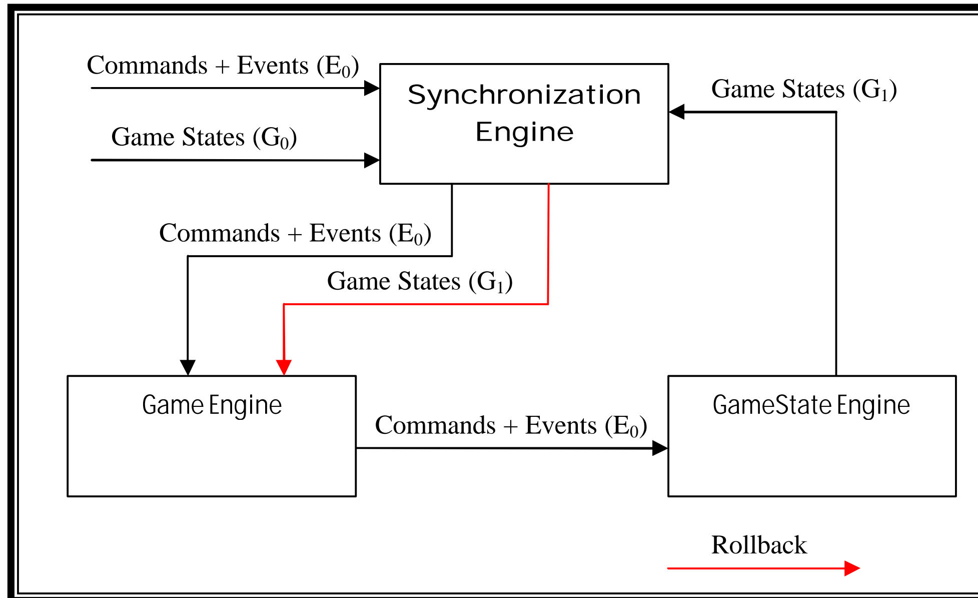


Fig 4.1 Synchronization architecture

For the ease of scalability and development, commands and all the driven events are first handled by the synchronization engine before being directed to the game engine for rendering. The events and commands are timestamped with reference to the same game world clock when they are issued. While executing the commands and events, the game engine throws the same copy to game-state engine to take "snapshot" of the game world. The corresponding game state is saved in the synchronization engine. In case there are inconsistencies occurred, rollback of game states is activated. The synchronization engine is responsible for re-ordering the incoming commands and events as well as recovering from inconsistency.

Synchronization algorithm, which is the main context of this project, is to be adopted in the synchronization engine. In this project, 2 different algorithms are being tested in the synchronization engine – **Bucket Synchronization** (Section 4.5) and **Multi-States Synchronization** (Section 4.6).

4.4.2 Game World Clock

The synchronization engine also has to synchronize the game world time of all clients so that commands and events from different clients can be absolutely ordered.

The main difficulty of synchronizing the game world time is the add-on network transmission delay. When the server tells a client the game world time **t** in initialization, the client receives the timestamp **t** at **t+d**, where **d** is the network delay. The client, however, can never be able to verdict if the world time received synchronized with the server.

Here the project adopts a common approach – NTP (Network Time Protocol) [4.10]. NTP provides algorithms to remove network jittering. The Network Time Protocol (NTP) is widely used in the Internet to synchronize computer clocks to national standard time.

4.4.3 Classification of commands

Mecha Zeta adopts the command-based consistency protocol to minimize the bandwidth consumption because the p2p traffic of Mecha Zeta is very large and frequent.

There are 2 categories of commands in Mecha Zeta defined. It is consistent command and real-time command. Consistent commands are those must be strictly synchronized among members and allow no loss. It includes FIRE and HIT event. For real-time command, it allows delay and loss but immediate delivery. MOVE is one of the examples.

Consistent command are sent over reliable protocol while real-time are sent over unreliable protocol. Synchronization engine only performs error correction with the consistent command.

4.5 Bucket Synchronization

It employs the bucket mechanisms to buffer the incoming events and commands. The executed bucket is then achieved to serve the function of future rollback like TimeWrap. The difference is that game state of every bucket is achieved instead of every command or event. A threshold of error recovery is imposed when there is an arriving command with timestamp older than the threshold limit borrowed from the Breathing algorithm.

There will always be a forward buffer bucket ahead of the current execution. The bucket serves as a buffer which stores up and sorts the commands or events inside the bucket. When the execution time of the bucket, with a synchronization delay added, expires, the bucket is achieved and the content of the bucket will be poured out for execution. Then, the synchronization engine will acquire the respective game state and achieve it for the next bucket.

Error is discovered from out-of-time commands or events arriving. The engine will still accept and store any arriving command or event but the execution is suspended until rollback is finished. Rollback is started by placing the late command or event into the respective bucket. The game state of that bucket will be restored and all the commands and events inside will be re-executed.

If a command exceeds the threshold limit, the synchronization engine will request the up-to-time game state from the coordinator of the partition, which is supposed to maintain the most consistent view of the partition.

4.5.1 An Example of Bucket Synchronization

Fig 5.1 demonstrates the normal execution and rollback of bucket synchronization. Three FIRE commands (with timestamp 260 (host 3), 430 (host 2) and 470 (local host)) are

high-lighted. The current game world clock time is 500. The synchronization delay (interval of buckets) is 200ms. Assume that the command 260 arrives after the execution of the grey bucket (forward buffer bucket) to illustrate rollback.

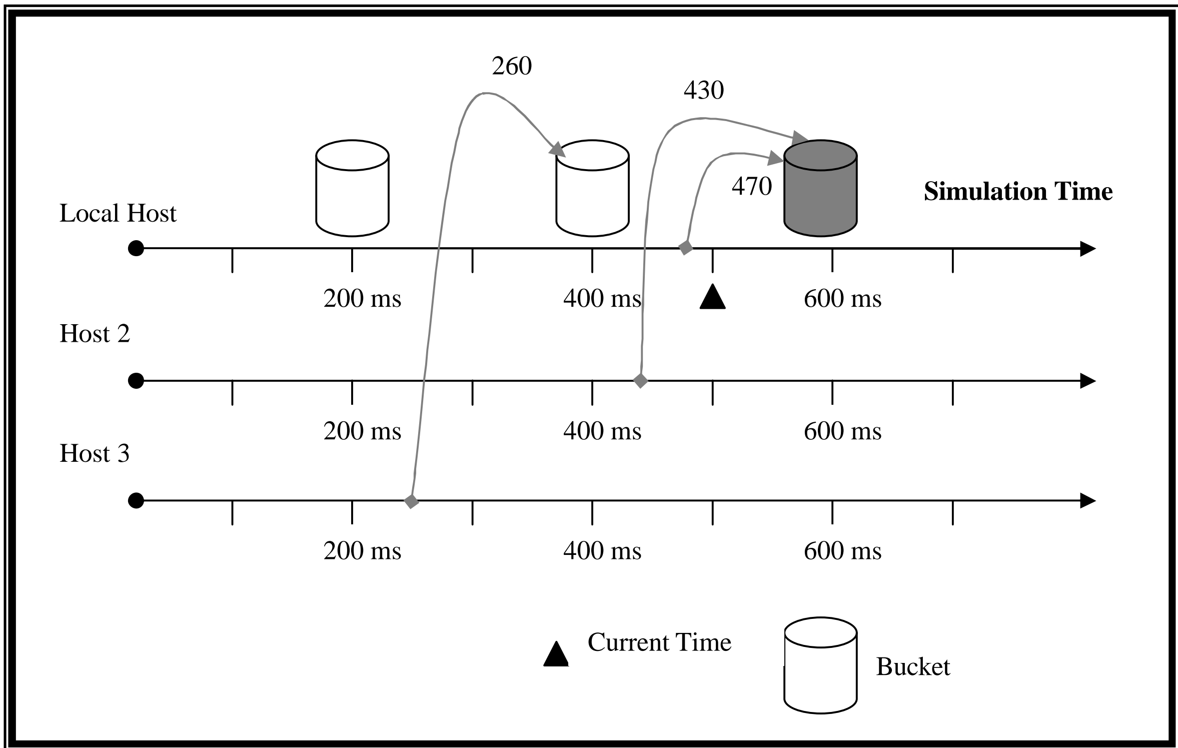


Fig 4.3 An example of Bucket synchronization

For the sake of providing buffer to reorder commands and events, each command and event will be re-timestamped a *playout time* in local host.

Playout time = World clock timestamp + Synchronization delay

Therefore, the 3 commands will be re-timestamped as 460, 630, 670 respectively. Suppose command 460 has not arrived yet. Current Time of local host is 500. Command 630 and command 670 are put into the forward buffer bucket at time 600 because they are on time. They are being sorted in the bucket. When the current time comes to 600, the forward buffer bucket expires and pours out command 630 and 670 for execution. The content of the forward buffer bucket is achieved and the forward buffer bucket is re-initialized with

timestamp 600+200=800. After the execution of 670, the synchronization engine acquires the game state from game state engine and achieve in the history bucket 600.

At time 700, while bucket 600 is executing, the command 260 arrives with playout timestamp 460. Rollback is called on action because current execution time is 700. Then, the execution of forward buffer bucket (600) is suspended until rollback is finished. The command 460 is placed into bucket 400 and is sorted there. The game state in bucket 400 is restored and all commands and events in bucket 400 and bucket 600 are re-executed in-orderly.

It is possible that inconsistencies still exist when performing rollback. For example, another late command arrives when rollback is taken place. The rollback will be interrupted immediately and re-ordering of is taken place.

4.5.2 Software Architecture of Bucket Synchronization

The full inheritance UML diagram is displayed in Appendix. Here shows an abstract inheritance and association UML class diagram.

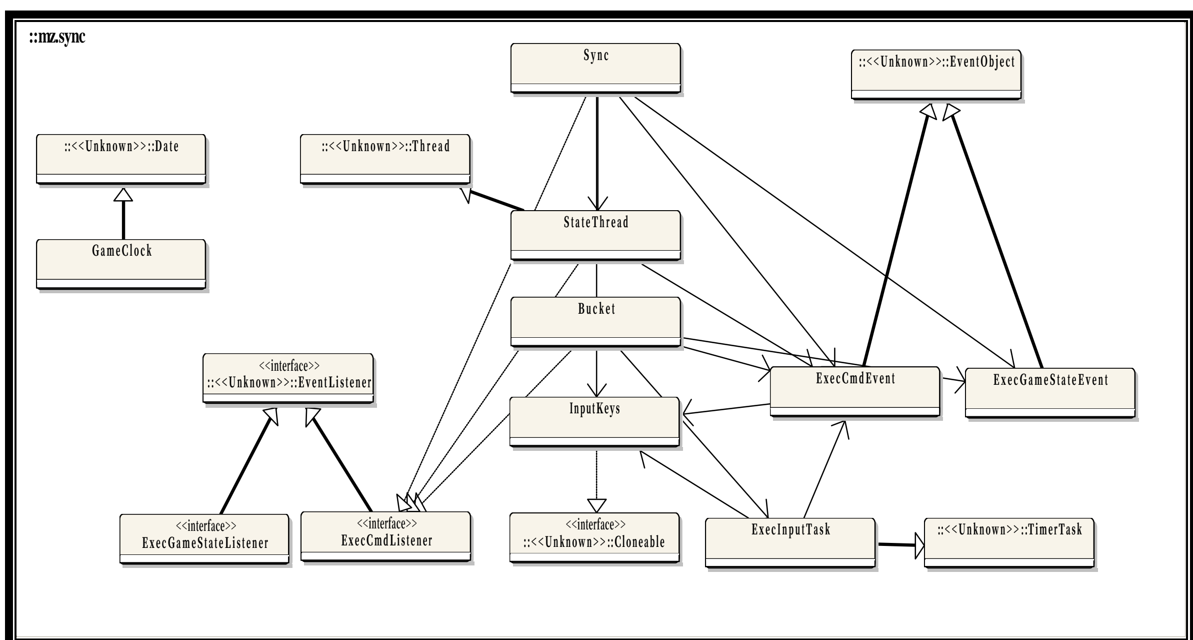


Fig 4.4 UML Inheritance and association diagram of Bucket synchronization

When commands (InputKeys) arrives in Sync, it is passed into StateThread where the Bucket Synchronization takes place. StateThread has an ArrayList to provide a series of forward buckets for buffering. The reason of using ArrayList is because the searching time is O(n) and the n is number of forward buckets. The number is extremely small, say 1 to 5.

The command is inserted to respective bucket after being re-timestamped the playout time. As StateThread is implemented as a thread, it will check if there is any bucket expired in every loop.

The Bucket bears a playout timestamp and a TreeMap for containing all commands. TreeMap has O(log n) seaching time and insertion time. In addition, the retrieval of commands from TreeMap is already sorted.

Every command is executed as an object of TimerTask which runs on a background thread Timer. It schedules the execution of each command according to its timestamp. When it executes, it is embed in ExecCmdEvent which is thrown out of the Sync. The game engine listens to this event and executes the command when Sync throws out the ExecCmdEvent.

Rollback is detected in StateThread when a command cannot be inserted into any forward buffer bucket. The achieved bucket is stored in another ArrayList because the searching algorithm starts at the "newest" achieved bucket until the respective bucket is found. The commands are handled as normal execution but restoring game state precedes. The game state in the selected bucket is thrown to game engine by

`ExecGameStateEvent`. After that, `ExecCmdEvent` carries the commands to the game engine for execution.

4.5.3 Analysis of Bucket Synchronization

Bucket Synchronization is clearly very different from any of the conservative algorithms, since its scheduling of execution is based on synchronization delay and not when it is safe. It is also clearly different from traditional bucket synchronization since it provides absolute synchronization for events delayed no later than the longest synchronization delay. Bucket synchronization is a little more optimistic than Time Warp in the sense that it does not keep a snapshot of the state before executing every command so that it can recover as soon as a late command arrives.

4.5.3.1 Responsiveness and Consistency

The responsiveness of Bucket Synchronization depends on the synchronization delay (playout time). Less synchronization delay means faster response and vice-versa. Bucket Synchronization improves the consistency by allowing a synchronization delay for pre-ordering and error-correction while being strict to consistency when error is discovered. The key to achieve a more satisfactory performance is to determine the synchronization delay dynamically with the network status.

4.5.3.2 Smoothness of the simulation

Recall that smoothness depends on 2 factors - synchronization delay and frequency of error corrections. Again, the synchronization delay plays an important role not only on

responsiveness and consistency but also the smoothness. On the frequency of error correction, it also subject to the synchronization delay to minimize the error probability.

4.5.3.3 Computation and storage overhead of error recovery

The storage overhead of buckets (commands, events and game state) in the achieved history is much less than TimeWrap because achieving game state is of each bucket instead of each command. The storage of game state is further reduced to the threshold binding where unlimited use of storage to preserve consistency is avoided. The error recovery procedure makes no difference as normal execution except that game state is to be restored.

The computational overhead of restoring game state is quite high. It is because Mecha Zeta aims to accommodate large group of clients, though it partially depends on the game engine's design of memory management. A high computational overhead may "halt" the "screen" of the client.

4.6 Multi-States Synchronization (MSS)

Although Bucket Synchronization has improved in the 3 aspects in problem, the computational and storage overhead is not improved. **Multi-States Synchronization** is the second approach to the problem by running the game concurrently in different delays. It preserves the responsiveness from "optimistic" algorithm with the consistency from "conservative" algorithm.

The main philosophy of MSS is to minimize the computational overhead of game state recovery so as to accelerate the recovery process. MSS is an optimistic algorithm, and must execute rollbacks when inconsistencies are detected. However, it does not suffer from the high memory and processor overheads of Time Warp. When rollbacks are required, instead of copying the state from a snapshot taken just prior to the offending command as TimeWarp does, MSS copies the state from a second copy of the same game which is running at a delay relative to the inconsistent state. This second copy of the game state, since it is following the 1st state in execution, has had more time to reorder commands and does not have the inconsistency that is to be repaired. Instead of keeping snapshots at every command, MSS keeps multiple copies of the same game state, each at a different simulation time. MSS is able to provide consistency because each trailing state will see fewer mis-ordered commands than the state preceding it by waiting longer for delayed commands to arrive before executing.

There is also a threshold of synchronization "horizon" where consistent commands arrive later than the horizon will request an absolute game state restoring from the coordinator of the partition.

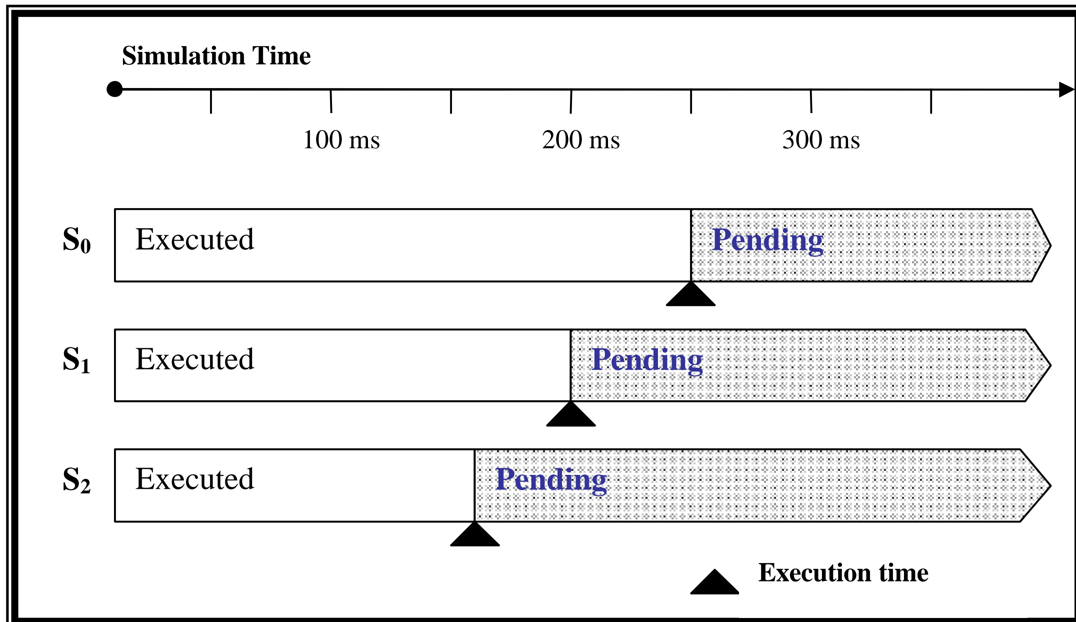


Fig 4.5 MSS Runtime Execution

4.6.1 An Example of MSS

For example, in Fig 6.2, there are 3 states with synchronization delays of 100ms, 200ms and 300ms each. Two commands are used for demonstration. Command A is a MOVE, a real-time command, issued locally at t=150ms and executed at 250ms in the leading state (the only game copy for rendering). At time t=350ms, the first trailing state reaches a simulation time of 150ms and executes command A. Since A was on time, its execution matches the leading state and no inconsistency occurs. Similarly, at t=450ms, the final trailing state reaches simulation time 150 and executes command A. It too finds no inconsistency, and no state is left to check it. In the other words, inconsistency could not be discovered for simulation time t<150ms.

Command B is a FIRE issued at time t=150ms by another member. By the time it arrives, the game world time is 275ms. The command is executed immediately in the leading state at t=275ms. At t=350 ms first trailing state executes B. When it compares it's results with the leading state's at t=250ms, it is unable to find FIRE from the same player at time t=250ms,

and signals the need for a rollback. The state of the trailing state becomes the leading state by placing B at time 250ms. The leading state then marks all commands after time 200 as unexecuted and re-executes them up to the current simulation time.

4.6.2 Implementation of MSS

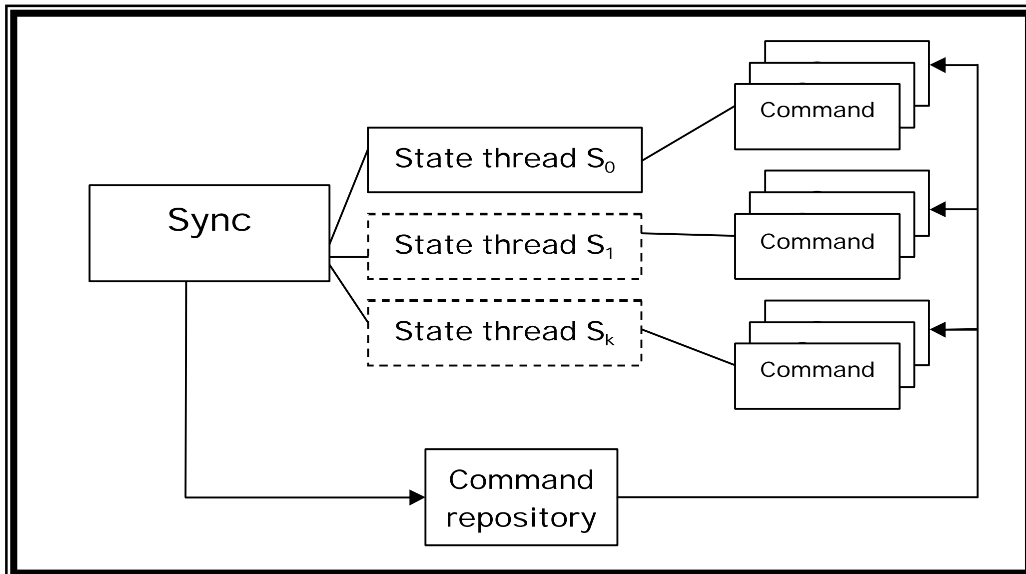


Fig 4.6 MSS software architecture

MSS is built on multi-thread. MSS allows flexibility on the implementation of the trailing states thread. All the threads, of individual synchronization delay, are implemented as "forward execution" – executing commands directly with the playout timestamp.

Sync provides the storage and sorting of all commands and events. It is implemented in a TreeMap where searching and insertion cost are O(log n). Each state thread is implemented using threading with a different synchronization delay. In each state thread, a TreeMap stores reference to respective commands in Sync.

The execution of each command and game state recover is the same as the implementation of Bucket Synchronization – using TimerTask and EventListener.

MSS requires an additional function on game state – comparison between game states. Moreover, since the trailing states compare the leading state's game state respective to each command, storage of game state of every command is needed.

In order to reduce the storage overhead, it is implemented that only the first state thread keeps all the game states. It is because if trailing state S_k executes a delayed command at time=t and gets the game state after this execution, S_k compares it with the game state in S_{k-1} at t. S_{k-1} must share the same game state as S_0 at t because inconsistency would have been discovered by S_{k-1} instead of $S_{k.}$.

4.6.3 Analysis of MSS

TSS performs best in comparison to other synchronization algorithms the situation is present in the game:

- game state is large and expensive to snapshot

Mecha Zeta is designed to be large in game state because of the large capacity of members. The time and storage of taking snapshot of game state is therefore expensive. MSS minimizes the computational side-effect of error recovery by switching the trailing state thread to be the leading state thread.

4.6.3.1 Responsiveness and Consistency

Same as Bucket Synchronization.

4.6.3.2 Computation and storage overhead of error recovery

Storage overhead depends on the number of states thread but computational overhead depends on the efficiency of game state comparison. Yet there is concern that running the game in multiples adds to the computational overhead.

4.6.3.3 Smoothness of the simulation

Computational overhead when error recovery is reduced. Hence, the performance of error recovery is enhanced.

4.7 Experimental Results

4.7.1 Scope of experiment

In order to test the performance regarding to the 3 objectives of this project, the 2 synchronization algorithms were tested under the same environmental parameters. The topics of experiments entail error recovery in the following aspects:

- Number of Rollback vs Frequency of command
- Number of Rollback vs Synchronization delay

4.7.2 Number of Rollback vs Frequency of command

The frequency of issuing commands tests the ability of handling commands in bursting manner. It happens very frequently in Mecha Zeta where several client robots may come into close and frequent attack. It is an aspect of testing consistency of the 2 algorithms.

System Settings	Network Settings	Program Setting
Intel Pentium 700 on Windows XP platform	• System link bandwidth: 100Mbps • Number of clients: 2 • Average ping RTT: 66 ms	• Synchronization delay: 100 ms

4.7.2.1 Testing result

	Rollback times	Total commands	Period of commands (ms)	Rollback costs(ms)
Run 1	11	500	150	26.01
Run 2	15	500	100	27.43
Run 3	14	500	70	31.25
Run 4	33	500	50	36.58
Run 5	53	500	25	49.46

Fig 4.7 Bucket Synchronization Results

	Rollback times	Total commands	Period of commands (ms)	Rollback costs(ms)
Run 1	13	500	150	11.93
Run 2	17	500	100	19.01
Run 3	15	500	70	18.32
Run 4	27	500	50	26.18
Run 5	41	500	25	28.45

Fig 4.8 MSS Results

4.7.2.2 Result Analysis

Bucket Synchronization - The result demonstrates an average trend of growth of errors from the reflection of frequency of command. The rollback cost also grows with the frequency of commands. It is because more commands to correct in each synchronization period. When the frequency of command is about the same as RTT, the number of rollback is about the same as lower frequency.

MSS – The result approximates the result of Bucket Synchronization except the rollback costs are greatly reduced. It can be deduced from the avoidance of restoring game state by switching the rendering thread.

4.7.3 Number of Rollback vs Synchronization delay

Synchronization delay in both algorithms plays crucial role in the performance. It is the balance between responsiveness and consistency. Here gives an experimental result. From experience, a max synchronization delay of 500ms gives the max tolerable reaction delay.

System Settings	Network Settings	Program Setting
Intel Pentium 700 on Windows XP platform	• System link bandwidth: 100Mbps • Number of clients: 2 • Average ping RTT: 102 ms	• Command period : 100 ms

4.7.3.1 Testing result

	Rollback times	Total commands	Synchronization delay (ms)
Run 1	18	500	100
Run 2	17	500	150
Run 3	14	500	250
Run 4	14	500	350
Run 5	7	500	500

Fig 4.9 Bucket Synchronization Results

	Rollback times	Total commands	Synchronization delay (ms)
Run 1	14	500	100
Run 2	15	500	150
Run 3	11	500	250
Run 4	9	500	350
Run 5	8	500	500

Fig 4.10 MSS Results

4.7.3.2 Result Analysis

The synchronization delay close to the RTT gives the least performance while larger delay gives better performance by removing jittering for a longer period. The result illustrates synchronization to be set statically gives no optimization.

4.8 Evaluations and Conclusion

In this report we have presented the synchronization mechanisms and put the focus on 2 optimistic synchronization algorithms – Bucket Synchronization and Multi-States Synchronization. These 2 are for multiplayer games with low latency but strong consistency requirements like Mecha Zeta. Low-latency consistent gameplay is guaranteed through the use of rollbacks while unnoticeable delayed response is achieved for the sake of buffering. The report illustrates that because of the use of command-based consistency protocol, these synchronizations can perform well in high-speed games where there is a large game state and many commands to be synchronized.

MSS performs smoother than Bucket Synchronization by switching between already-running game copies. However, Bucket synchronization saves the computational overhead by taking 1 snaphot in every bucket instead of every command in MSS. The storage overhead of both algorithms are fairly the same, which are both subject to designer's choice.

There are still a number of unexplored areas for future work. As discussed above, a more thorough examination of the parameter space for synchronization delays is

needed, as well as dynamically determining the number of, and delays for, needed states. In addition, the throughout and intensive study of command classification helps the performance of the synchronization algorithms. In this project, only 2 types of command are being classified.

5 Graphic Engine and Collision Detection System

5.1 Introduction

The aims of this project is to investigate the feasibility of multiple-player Internet action game with high scalability, intensive interactivities and realistic experiences such as accurate collision detection and model simulation, implement and evaluate them. The project is divided into five modules and this part belongs to the module "Graphic Engine and Collision Detection".

Analysis of the problems about graphic engine and collision detection has been carried out, and robust and high extensibility graphic engine and accurate collision detection system are main focuses in this module. Feasibility studies have been carried out and the game is implemented using Java and GL4Java[5.1] for OpenGL programming. The accurate collision detection is carried out by external C++ library ColDet[5.2] with the help of Java Native Interface (JNI).

After implementation and testing, the graphic engine is found to be extensible and object oriented, with ease of adding new kinds of objects to be rendered. It is also made more robust by several performance optimizations. Accurate collision detection is implemented in checking attack collision with accuracy 100% from test results and co-operative collision detection system is used to reduce the amount of duplicated detection by doing collision detection co-operatively among all players.

5.1.1 Responsibilities of this module

- Research on feasibility of graphics programming using Java
- Overall graphic engine design and implementation
- Design and implementation of collision detection system

5.1.2 Scopes of investigation of this module

The main scope of investigation of this module is to study the feasibility, design, and construction and evaluate:

- Graphic engine that is robust, extensible and able to produce realistic output
- Collision detection system that has high accuracy, robust and able to use in high scalability scenario, and most importantly, to be useful in real time.

5.1.3 Structure

In each of the area (Graphic Engine and Collision Detection), problem analysis, feasibility studies, challenges and solutions in the design and construction phases, evaluations are done and mentioned in detail.

5.2 Methods of investigations

To carry out the project, several methods of investigations have been carried out to investigate the various sections of the project, including feasibility studies before the project design and construction, as well as testing the functionalities, effectiveness and performance of various modules of the projects during the development. In this chapter, various methods of investigations used, for this module, are going to be introduced.

5.2.1 Feasibility studies

The methods used to carry out the feasibility studies include:

- **Researches on related topics in the Internet**

  The Internet provides us a large resource and information pool for various topics. For example, researches about the feasibility of using Java as game programming language, possibility of doing accurate collision detection in a real time game, as well as using OpenGL in Java have been conducted before the game design stage. Projects, demos and reports related to the issues above have been found and studied in detailed to estimate the feasibility and practicability of the issues.

- **Functionality tests**

  Simple test programs have been built to test the required functionality, feasibility of core purposes as well as performance, such as simple OpenGL rendering program using Java, accurate collision detection of simple polygons.

5.2.2 Prototypes

During the design and construction stages, prototypes have been built to test various functionalities achievements. By making prototypes at different phases of development, it can ensure that the functionalities have been achieved progressively as well as testing the effectiveness of the implementation. Moreover, it also helps to prepare the next phase implementation as well as testing the co-operation between other modules.

5.2.3 Tests and Benchmarks

At various stages, different tests and benchmarks have been conducted to test the performance and functionalities of various ways of implementations. Such tests help to determine the appropriate ways of implementations. In addition, various tests have also been conducted to test the functionalities and performance of the implementation, as well as comparing with different implementations to analysis the arguments of the project.

5.2.4 Specifications of benchmarks and tests

Unless otherwise specified, all the benchmarks and tests carried out in this report follow the specification below:

CPU	Intel Celeron 850MHz
RAM	192MB
Display adapter	NVIDIA GeForce2 MX 400 with 32MB RAM
Operating System	Microsoft Windows XP Professional
Display Driver	NVIDIA Detonator XP 43.45
OpenGL Version	1.4.0

Resolution	800 x 600
Color depth	32bit
Vertical Sync	Off
Java compiler	J2SE 1.4.1_02
C++ compiler	Microsoft Visual C++ .NET

Additional specifications are accompanied with the test specification when needed.

5.3 Graphic Engine

5.3.1 Analysis of Problem

Requirements

For graphics rendering in a real time game, high speed and fast response are critical factors to success, since the game is not joyful to the player if the graphic output cannot give the player immediate response as well as non-smooth graphic output. At the same time, realism is another critical factor to make the players feel more joyful when playing. As a result, the graphic engine requires:

- rendering the objects in the game world on to the screen
- high performance as well as produce realistic output

Analysis

The graphic engine is responsible to render the objects in the game world onto the screen. From the above requirements, it should have high performance as well as produce realistic environment. In addition, as the project is divided into several modules, the graphic engine should be able to use by other modules easily. Moreover, it should be easy to be extended to display other kind of objects for possible future development.

After fetching all requirements and analyzing them, the following are concluded to be focused for the design and construction phase:

- Make use of optimization to boost up the performance
- Employ various techniques to improve the realism of the environment
- Design and object-oriented structure for object representation in the graphic engine, so that new kinds of objects can be easily added and inherited
- The interface to other modules should be kept as simple as possible for ease of use by other modules

5.3.2 Feasibility Studies

5.3.2.1 The use of programming language

From above, it is believed that using an object-oriented programming language can facilitate our development as well as making our game having a much higher extensibility as well as flexibility, which should be an important aim of our graphic engine. Java and C++ are both of our choices. Before the decision, I have conducted a wide range of researches and small tests to facilitate the decision.

First of all, several criteria of the programming language used in our project are listed below:

- Able to offer great productivity and facilitate the coding phases
- Highly extensible and being Object-Oriented in nature
- Able to facilitate the graphics, network and logic programming
- Offer robust performance

A brief introduction of characteristics related to the programming of this project of Java and C++ have been listed below:

Language	**Java**	**C++**
API Coverage	Wide range of useful and organized API covered and available in term of Java classes	Large amount of system functions but rather unorganized. Standard Template Library (STL) provides a limited amount of useful containers

Object-Oriented Programming Feature	Java is actually designed for OOP, functions like inheritance, abstraction, polymorphism is built into the language	C++ is Object-Oriented version of its decedent C. Same as Java, all kind of OOP features are supported.
Portability	If the program is implemented in 100% Java Code, it is portable to any platform that has Java Virtual Machine port with any recompilation.	Several standard of C++ exists (GNU, ANSI, ISO, Win32, etc). To port the code to another platform that bind with different standards need modification before compilation
System Features	The Java API acts as a layer of abstraction of system level implementation. Calls to native system call is possible by Java Native Interface (JNI)	Extensive amount of system calls that can facilitate the development and have greatest performance
Graphics Programming	Several JNI implementation of Graphics Library mapper such as GL4Java which making use of JNI to call system OpenGL calls. Java3D, by Sun Microsystems,	Able to call system OpenGL System call to build OpenGL application

	is also available as 3D API to call system graphics library like OpenGL or DirectX	
Network Programming	Numerous network API is bundled with Java SDK to serve as various functionalities	Socket Programming and direct call to system network libraries

**Tests of JNI and GL4Java with counterpart C++ have been carried out and are going to analyzed and report in next section*

Analysis

From above, we can see that Java is providing a rich set of API which can facilitate the productivity of coding as well as offering a better object oriented structure than C++ since Java is decided to be used as object oriented design. It can be referenced from the article of "Java offers Increase Productivity" by Wells[5.3]. The author analyzed various reports investigating and comparing the productivity of Java with C++ and found that Java usually yields 10-20% productivity higher than using C++ and even 30%-40% coding phase is shorten. In addition, it is mentioned that the object oriented framework of Java can lead to a more bug-free development. As we need to finish a project within an academic year, using programming language likes Java that yields a higher productivity is definitely helpful.

On the other hand, C++ provides us direct implementation in both graphical and network programming since no middle abstraction is presented which can give more robust and effective performance, with the trade off of ease of programming and high productivity, as well as the portability. The performance of Java is slower than C++ for pure logical computation. In Section 8.8 of the report "Evaluating Java for Game

Development" by Jacob Marner [5.4], the author concluded that Java is averagely around 20% lower than C++ counterpart. It is not a surprising figure since the Java bytecode is run in a virtual machine that acts as an intermediate layer comparing with binary code of C++ that directly ran by the kernel of operating system.

So we see that Java has advantages on project productivity as well as object-oriented in nature while C++ is relatively better in term of performance and has direct implementation to graphics and network without intermediate layers. Since both of programming language satisfies around half of criteria stated above, as well as there is some difficulties for Java in programming graphical side and networking side, some demos and tests have been conducted to evaluate the use of Java for graphical programming.

5.3.2.2 The use of graphic API

In Java, there are two main streams to do graphical rendering. The first way is using external wrapper packages to call OpenGL system library directly to do the rendering via Java Native Interface. There are a number of such packages available but only GL4Java[5.1] by JauSoft is most comprehensive as well as complies with OpenGL 1.3 standard (others are complying older versions). The latest version of GL4Java is 2.8.2.0. The second way is to use Java3D[5.5] by Sun Microsystems.

Introduction to GL4Java

GL4Java is a set of Java API that maps the whole OpenGL library call to be callable in Java. It complies with OpenGL 1.3 standard. It makes use of Java Native Interface (JNI) to call to underlying system library. Programming OpenGL in Java using GL4Java is very similar to do so in C++ counterpart since most of the GL methods are

mapped. For detail of implementation please refer to GL4Java homepage[5.1]. It has several ports to different systems such as Linux, Windows, etc.

Introduction to Java3D

Java3D is a scene-graph API that is developed by Sun Microsystems. It aims at giving the functionality of 3D rendering using Java. It also uses Java Native Interface to call native system graphic libraries. It supports both OpenGL and DirectX (Windows only). However, it uses a unified layered architecture that all the call to that API is independent to the graphical API using underlying. That is, to use Java3D you need to use a set of API that is different from and abstracted above OpenGL or DirectX. For detail implementation and other information, please refer to Java3D Homepage[5.5]. It should be a good idea to make the things easier to be implemented. However, though, its performance is not that robust since an abstract layer is presented. It is found that, in Section 8.4 of report "Evaluating Java for Game Development" by Jacob Marner[5.4], using Java3D to implement an OpenGL program is roughly 2.5 times lower than that using GL4Java to render the same scene. Such a slow down maybe due to the complicated structure of the implementation of Java3D. As a result, our target has been focused to the comparison between Java with GL4Java and C++.

Comparisons between OpenGL in Java and C++

A simple test has been conducted to see the overhead of calling OpenGL commands through GL4Java in Java with direct implementation in C++. I would like to remind that all the tests and benchmarks are being untaken in the specification stated in Section 1.4.

Results and Analysis

In order to test the feasibility of using Java with GL4Java in our game, we need to find that whether it can support up to high amount of polygon as our game is expected to have a relatively high scalability. The 1000 polygons are represented as 250 pyramids which are drawn is the exact position to increase the complexity of drawing. The results are represented in frame per second (FPS) as the following table with the specifications of this test.

	Test Case 1	**Test Case 2**	**Test Case 3**	**Test Case 4**
Language and API	**Java with GL4Java**		**C++**	
Number of polygon	**40**	**1000**	**40**	**1000**
Transformation	Transform and Rotate			
Display List	Used			
Result in FPS	**250**	**25**	**280**	**29**

Comparing with Java with C++:

Performance slow down for small polygon counts: around 11%

Performance slow down for high polygon counts: around 13%

From the result above, it can be observed that programming in OpenGL in Java using GL4Java is around 11% – 13% slower than that of C++. And as polygon count increases, the slow down remains in similar percentage. However, as mentioned above, Java is on average 20% slower than the C++ in logic intensive program and such a slow down should also be accounted in the results above. By the way, a 10%

performance slowdown is acceptable given the high productivity it yields.

Based on the analysis in Section 5.3.1 as well as the test result above, we found that Java has a very good productivity and object-oriented in nature. It is very suitable to be used for implementing a game in several modules since Object-Oriented architecture facilitate the abstraction and such the underlying implementation of individual module can be abstracted and hidden to other modules. In addition, using GL4Java with Java to build OpenGL program has just a 10% overhead comparing with OpenGL program in C++. As a result, even though Java has a performance average lower than C++, it is chosen and it is believed that its advantage can overcome such difficulties as it provides a much higher productivity as well as the computational power nowadays is high and a slight performance lag is acceptable.

5.3.3 Design and Construction

5.3.3.1 Overall design

In our game Mecha Zeta, the graphic engine is designed to be capable to display the following:

- 3D Objects:
  - A Terrain Landscape (ground and skybox)
  - Static objects like trees, buildings, etc.
  - Dynamic objects like mechas (robots), missiles, laser, etc
- 2D Objects:
  - GUI components like icon, status bar, radar map, etc.
  - Texture fonts like various statistics (e.g, FPS, player's position)

Since one of the requirements of the graphic engine is high extensibility, that is, it is easy to add on new functionalities and easy to use in different situation, the graphic engine is designed such that it accepts a range of objects to be processed and rendered and it is easy to add other types of objects being rendered.

The following is the graphic engine design architecture:

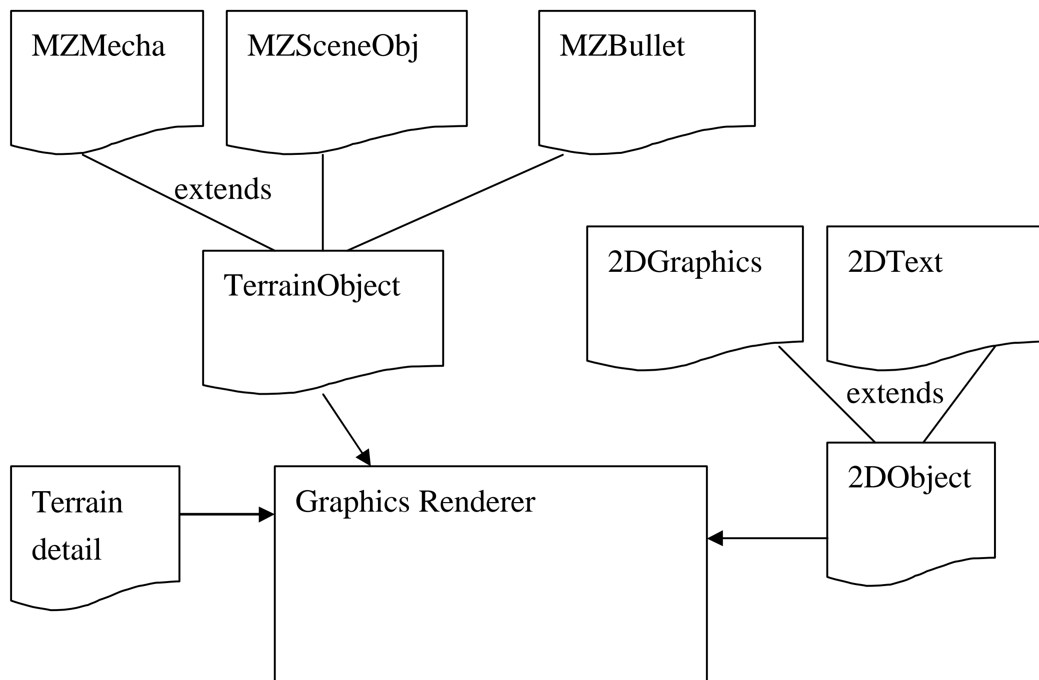


Fig. 5.1 Graphic engine design architecture

As expressed above, the renderer accepts different kinds of displayable objects, including 3D Terrain Object, 2D Object and the terrain. Actually the renderer itself maintains lists of displayable objects and renders the scene based on those objects. All the objects are having parameter like position, state, and other characteristics. There is no hard coding in the renderer and it just base on the objects to render, which offer great degree of flexibility.

5.3.3.2 Terrain generation and rendering

Initial implementation

Terrain rendering is the first functionality I implemented in the renderer. The terrain generation concept is referenced from a Landscape Demo in GL4Java website[5.1]. The terrain is generated using fractal midpoint displacement algorithm. Vertices are generated with different height levels and color levels according to the algorithm

above and forming lot of triangles to render. There is also a function to get the height level of any point on the terrain. Those non-vertex point will be interpolated and give out the height level. The height level is used to accurately place the static objects as well as dynamic objects like robot on the terrain.

However, in order to have lighting effect smoothly on the terrain, normal of each vertex is needed but it is lacked from the algorithm. So in the terrain loading process, I have implemented a two-pass normal calculation routine using vector cross product. The first pass calculate the normal of each triangle while the second pass calculate the normal of each vertex by taking the average of normal of all nearby triangles sharing that vertex. The normal calculated are normalized so that OpenGL can display the lighting effect properly.

Afterwards, the vertex and normal information are maintained in two arrays and they are rendered in each frame. However, to render the terrain effectively, it is impossible to render the whole terrain since it will slow down the rendering process very much. In order to speed up the process, frustum culling is used. Only the triangles of the terrain falling in to the viewable frustum will be drawn. For detail of implementation of frustum culling, please refer to Section 5.3.3.5 – Performance optimization.

Realism improvement

The terrain generated is of color of sand as well as terrain vegetation, which gives a quite realistic landscape environment. However, the realism is not high as the background of the environment is only expressed in single colour. In order to increase the realism, fog is used to make the distant objects not showing suddenly but progressively. In addition, a skybox is made with texture mapping to produce the

effect of sky and distant landscape, and the player can never reach the edge of the skybox to retain the realism. Below are some screenshots for the beginning stage of the implementation:

Fig. 5.2 Screenshot without skybox

Fig. 5.3 Screenshot with skybox

The skybox is made by cube with a texture mapping on each face. The texture is arranged in a way so that the player cannot notice the edge of the skybox. The skybox is made by flipping up the above texture to form a cube and surround the world. Actually if we dissemble the skybox we can see the texture is look like:

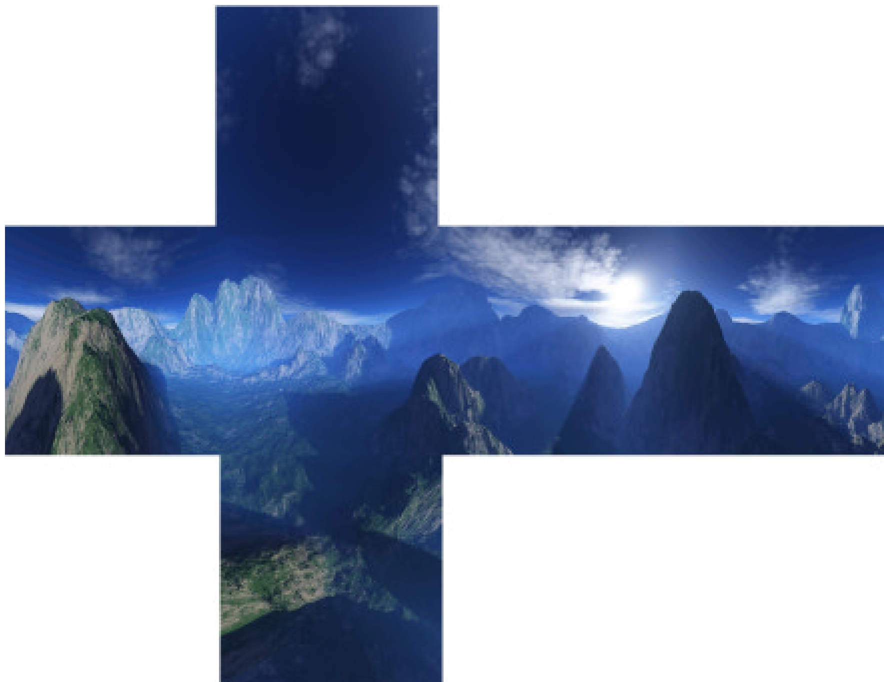


Fig. 5.4 Dissembled skybox

However, there is some display flaws in the edges of the skybox. It is because, normally, OpenGL renders polygon with texture mapped not covering the edges of the polygon, that is, we can see the pixel-edge of the texture-mapped polygon having the color of the polygon itself. This problem seriously lowers the realism of the skybox since edges are noticeable by the player. In order to solve the problem, OpenGL texture extension `GL_CLAMP_TO_EDGE` need to be used to make sure the texture also covers the edges of the polygon. Since this extension may not be available in certain hardware implementation but available in others (like NVIDIA GeForce), a checking is made before enabling this extension.

5.3.3.3 Object-oriented world container

In the world, apart from the terrain, all other 3D objects including the robots, bullets, trees, etc are displayable objects and maintained in a list in the renderer. The use of such object-oriented structure is to maximize the flexibility of the renderer such that it can display another type of objects easily and with fewer modifications.

All those objects are belonged to a parent class called `TerrainObject`. It has some common parameters that are essential to represent objects on the terrain such as position, forward vector, 3D model, etc. Additional object class `MZMecha` which extends `TerrainObject` is used to store robot, which has additional properties like animation, control, etc. Similarly, `MZSceneObj` is used to store scenery objects like trees and `MZBullet` is used to represent flying bullets in the environment.

Since all the entities in the renderer are objects, we can set individual parameters to them so that they can behave what they should. For example, after the game logic can

set the position, movement, animation of individual robots in the scene, the renderer can readily render the output and effect. For the detail of model animation and control, please refer to the report "Modeling, Animation and Special Effects".

Similar to the terrain, it is an extremely slow process to render all the objects on the terrain. Frustum culling is employed to render objects that fall inside player's viewable frustum, while other objects will not be rendered. For detail of implementation of frustum culling, please refer to Section 5.3.3.5 – Performance optimization.

The use of object-oriented world container also facilitates the collision detection mechanism, we can easily fetch two objects to test whether they are in contact or not. For detail implementation of collision detection, please refer to Section 5.4 of this report.

5.3.3.4 2D objects

There are 2 kinds of objects, text and graphics. Texture mapped font is used to render the font. At initialization stage, a font texture which having all the alphabetic characters, numbers and commonly used sign will be loaded and display list will be built for each of the character. For 2D graphics, they are quadrilaterals in shape and mapped with texture for specific use such as GUI. The texture is in TGA format such that an additional alpha channel is stored in the texture, so that some of the texture area can be transparent so that any shaped graphics can be produced by just mapping to a quadrilateral. In the class `2DObject`, position and size is stored to be drawn on the screen. Fig. 5.5 is a screenshot with 2D objects like text and graphics.

Fig. 5.5 Screenshot with 2D object rendering

5.3.3.5 Performance challenges and solutions

Analysis

Performance should be a major concern on the success of a graphic engine because performance is crucial to give realistic and smooth visual effects to the player. In order to boost the performance, the first aim is to reduce the number of objects to be rendered. Actually those objects that are not viewable to the players are not need to be rendered. Another way to boost the performance is to dynamically adjust the number of objects to be rendered in order to keep the frame rate high. As a result, several optimization techniques have been employed to boost the performance of it.

Frustum culling

For the rendering process above, if we exhaustively render the whole terrain and all the objects on the terrain in every frame, it is a redundant operation since not all of the objects and terrain are visible to the player. In addition, such operations will slower the speed of rendering by a large degree.

In order to solve the problem, frustum culling is employed. The idea of frustum culling is to make the renderer to display the polygons inside the viewable frustum of the player viewport. Any other polygons outside the frustum will not be rendered. Theoretically frustum culling should not bring any experience degrade to the player sin it only discard the polygons outside the frustum that originally invisible to the player.

The implement frustum culling, we need to find the specification of the viewable frustum of player's viewport before render anything of each frame. The frustum is extracted using the current projection and modelview matrix. The frustum is a 6-plane object that has near, far, left, right, top, and bottom plane, and represented as 6 plane equations. The algorithm of preparing viewable frustum from projection and modelview matrix is reference from the article "Frustum Culling in OpenGL" by Mark Morley[5.13].

After the frustum specification is prepared, the terrain polygons and 3D objects are tested whether falling inside the viewable frustum and rendered if so. To carry out the frustum test, the distances between the testing position and each of 6 frustum plane are calculated. If the testing point is in front of all frustum planes then the point is in the frustum.

Limit the amount of objects to be displayed

If we just use the frustum culling technique above, the performance of the renderer will still be quite slow if there are many objects in the viewable frustum. In order to maintain a smooth frame rate to the player in order to maintain the playability, the size of the frustum is **dynamically** adjusted by the previous history. As a result, if there are

too many objects to be rendered, those farther objects will not be rendered. However, if the objects in the frustum are few, all of them will be rendered.

Display lists

Since many objects are having identical model but have different positions and view such as trees, rendering them in duplication is a very slow process. As a result, OpenGL display lists are built in the initial stage. Take the trees rendering as an example, the tree rendering process is compiled into a display list when the game initializes, so that those OpenGL vertex and color call do not need to be called in each frame. Instead, `glCallLists()` is called to draw the precompiled display list.

5.3.4 <u>Test Results and Analysis</u>

In this section, it is going to list out a set of benchmarks that investigate different implementation of graphic engine as well as the effectiveness of implementation of performance tuning, as well as the analysis. In this section, various benchmarks are going to be conducted to investigate the effectiveness of using frustum culling and display list.

5.3.4.1 Frustum culling

The use of frustum culling is to make sure than the polygons that are visible in the player's viewport is rendered. Two performance tests are going be carried out to investigate the usage of frustum culling and its effectiveness:

- Cost of extracting frustum in each frame
- Speed up with frustum culling enabled

Cost of extracting frustum in each frame

To use frustum culling, a frustum specification needs to be prepared to test whether a point is fall inside the frustum. The frustum needs to be built in every frame from the projection matrix and modelview matrix. This test is to find the cost of frustum extraction. Below are the specifications of the test as well as the results in FPS:

	Test Case 1	**Test Case 2**
Terrain Rendering	Disabled	Disabled
Objects Rendering	Disabled	Disabled
Extract frustum in each frame	**Yes**	**No**
Frustum culling	Disabled	Disabled

Result (in FPS)	230	231

The only difference between two test cases is whether it extracts frustum in each frame or not. For the result, the high frame rate achieved in the above tests is due to nothing is rendered actually. The difference between 2 test cases is not significant and it can conclude that the cost frustum preparation process in each frame is unnoticeable.

Speed up with frustum culling

This test intends to investigate the effectiveness of frustum culling. Below are the specifications and results:

	Test Case 1	**Test Case 2**
Terrain Rendering	Enabled	
Objects Rendering	Trees and player robot	
Frustum culling	**Enabled**	**Disabled**
Display List	Used for trees	
Result in FPS	**57**	**34**

From the above result, the speed up of frustum culling reaches the percentage of 67%. The speed up is very significant. It is because those objects and terrain fall outside the viewable frustum is not rendered.

As we can see from above benchmarks, frustum culling really improves the performance very much without any visual quality degrades. It is because frustum culling only discards the objects that fall outside the frustum. Moreover, the

performance boost is more obvious when the number of objects increase. It is because the number of objects that can be culled also increases. As a result, frustum culling is a very effective way to boost the performance of the graphic engine, especially in high scalability scene that has many objects around the world, since more objects will be culled.

5.3.4.2 Display List

The use of display list is to pre-compile of static objects into list in OpenGL so that no vertex and color calls are needed when drawing them. The test is going to show the effect of using display list in drawing a small amount of trees and large amount of trees.

	Test Case 1	**Test Case 2**	**Test Case 3**	**Test Case 4**
Terrain Rendering	Enabled			
Objects Rendering	Enabled			
Amount of trees	**Few trees**		**Lots of trees like forest**	
Frustum culling	Enabled			
Display List	**Used for trees**	**Disabled**	**Used for trees**	**Disabled**
Result in FPS	**55**	**26**	**52**	**14**

From the results, the performance drop from few trees to forest is just 5% when using display list while the drop is around 46% when display list is not used. In addition, the display performance speed up the around 112% using display list in few trees area and the speed up in dense trees area is even 271%.

From the test above, the benefit of using display list is more obvious if the number of duplicated objects increased. In a high scalability scenario, duplicated objects in the game world are very common. The use of display list effectively optimizes the graphic engine, boost the performance and give the player a smoother graphics for better experience.

5.3.5 Discussions and Evaluations

5.3.5.1 Using Java as game graphics programming language

Java was chosen to be the programming language in this project. During the implementation phase, it offers many conveniences to us such as providing a rich set of useful API for us to use, use of object-oriented methodology. The project is divided into several modules which serving different functions, and making use of interface and abstraction, the connections between different modules have been simplified and minimize to keep the abstraction high.

During the feasibility studies, we assume that Java has performance, on average, 20% slower than the C++. However, after the game implementation, it is found that the performance is still very satisfactory, having around 40 – 50 fps.

Moreover, the game can be ported to various operating systems very easily. For the Java code itself, it can be left unmodified. What we need to do is to use other ports of GL4Java[5.1] as well as collision detection library, and that is enough.

However, Java does also impose several inconveniences to us during the implementation, in term of graphics rendering and collision detection. Firstly, it cannot make use of native OpenGL library directly but need to making use of external package (GL4Java in this project). Extra effort needs to be spent on investigating the feasibility of using it as well as making use of it during game development. However, using OpenGL in Java with GL4Java is very similar to that using in C++ and its performance is very satisfactory indeed.

However, in view of the module graphic engine and collision detection, Java does help me to implement them in a faster and effective manner. Even extra efforts need to be made to make Java feasible in doing OpenGL programming and accurate collision detection, those effort is not very great and having much difficulties and they are worthwhile. The most important thing is that those extra work has not lowered the performance of the game very much. Java is still considered as suitable after the implementation.

5.3.5.2 Functions achieved

The graphic engine is implemented with the following functionality:

- An object oriented renderer that maintains a list of displayable objects
- Terrain generation and rendering
- Interface for game logic to manipulate the states of objects in the world
- 2D components rendering
- Various optimizations to cope with high scalability

The achievement of graphic engine design is found to be sufficient to this game. Objects are arranged in the renderer and the game logic can manipulate different parameters of them to represent the world situation. Performance optimizations have been done to boost the performance.

5.3.5.3 Importance of optimization

During the implementation of the graphic engine, it is found that performance optimization always plays an important role to make the development to be succeeded. In the initial stage of development, without the uses of performance optimization such as frustum culling and display list, the speed of rendering is slow to be unacceptable. From the tests result above, it is found that those optimization techniques improve the

performance by a very large degree, especially when the world has many objects in it. So it is obvious that performance tuning is very important to graphic engine that needs to support high scalability.

Except the optimization techniques employed in this implementation, there are many others techniques that can boost the rendering performance. One example is Level Of Detail (LOD). The main purpose of LOD is to render an object with different level of detail depending on the position of that object to the user viewport. The farther away the object is, the less detail that object will be rendered. On the other hand, the object will be rendered in much detail when it is closer to the viewport. However, since time is limited for this project, it is not possible to implement other optimization methods as there are another important focuses of this module of the project, that is, the collision detection system.

5.3.5.4 Extensibility of graphic engine

One of the purposes of the graphic engine is to make it easy to make new kind of objects to be rendered. According to the overall design of graphic engine in Chapter 6.4, all the 3D objects are belonged to `TerrainObject` which are fed into a list of displayable object and render in each frame by the renderer. Any kind of 3D objects that need to be rendered can extend `TerrainObject` with their own characteristic, having different models, properties, path, etc. Such operation does not need the modification of graphics engine since it is designed for render object of class `TerrainObject`. As a result, it is easy to make the renderer to render other kinds of objects when need.

5.3.5.5 Use of GL4Java

GL4Java is used for calling OpenGL library in Java. During the implementation stage, there is no special problem using GL4Java and using OpenGL with GL4Java in Java is very similar to that of using C++. Further, according to the tests being carried out in Section 5.2, it is found that the performance overhead using GL4Java is not great comparing with calling OpenGL functions directly in C++. There are also no specific difficulties to code OpenGL using GL4Java Java, unless there is some difference in setting some initial parameters since GL4Java, likes other Java API, and is represented in an object-oriented style.

5.4 Collision Detection System

5.4.1 Analysis of Problems

Requirements

It is no doubt that realism is a critical factor for a game to success. As this is an action game accompanying with various close attacks and near attacks, high accuracy collision detection is required to give high realism experience to the player.

As result, the collision detection system requires:

- detecting collision to prevent objects from penetrating through others
- detecting collision to judge whether an attack is successfully hit the target
- high accuracy detection
- high performance, even in complex scene with many objects such as high scalability scenarios

Analysis

Collision detection is the process that detect whether two objects collide together geometrically. In above, high accuracy and maintaining high performance in high scalability are the main requirements of the collision detection system in the game. In order to achieve high accuracy, traditional bounding shape methods cannot serve the function. External geometric collision detection libraries are required to do so. For the requirement of maintaining high performance in high scalability scenario, a co-operative collision detection system can be investigated in which different players share the collision detection of whole world, such that no player needs to do all the detections in the whole world.

After fetching all requirements and analyzing them, the following are concluded to be focused:

- Use of external accurate collision detection library to perform the collision detection with high accuracy.
- Implement the collision detection system in a co-operated manner, so that no members in the game need to do collision detection for entire world, in order to achieve high scaliblity.
- Performance optimizations such as pruning out all unnecessary detections before accurate collision detection take place, and decide whether accurate collision is employed for all collision checking

5.4.2 Feasibility Studies

The term collision detection is referred to the detection of whether two objects are in contact or not. Collision detection plays an important role in computer games. For example, collision detection is used to detect whether a player can hit others in an action game, whether a player can shoot another in a shooting game. Many commercial games are making use of some sort of simplified collision detection such as bounding boxes, spheres or cylinders to do collision detection by checking their intersection. For example, the game MDK2 uses bounding cylinders to check the collision between the player and the environment mentioned in an article in gamasutra.com by Stan Melax[5.6]. Using simplified collision detection system can give the players a fair experience but can increase the performance. However, accurate collision detection can give the player a very high degree of realism since the players will never experience the situation like he is going to hit the enemy but the game judges cannot. This section is going to investigate the feasibility of using accurate collision detection in game development.

5.4.2.1 Availability of external libraries

After conducting an extensive research about accurate collision detection, several libraries of accurate collision detection do exist. For example, the gamma team of the University of North California[5.7] has published numerous collision detection libraries such as RAPID[5.8], SWIFT++[5.9], etc. Apart from them, other collision detection library such as ColDet[5.2] is also being available. After studying their usages and mechanisms, it is found that most of them are having similar interface for user to use while they are using different implementation inside.

Most of them are model-oriented. That is, before doing any detection, you need to build a collision model by feeding in the polygon organization of the model and some of them even accept triangular polygon meshes. They supplies functions to detect the collision between two collision models as well as to set transformation to the model. Examples of using such interfaces are RAPID and ColDet.

Take ColDet as an example, it accepts any model even polygon soups. It uses bounding box hierarchies for fast detection and additional triangle intersection test. It is claimed that ColDet has accuracy for 100%. It provides checking functions like model-model test, ray-model test, sphere-model test, etc.

5.4.2.2 Usage in game

Since our models are designed to be built using triangles and each robot is making up from several parts that having different transformations in motion, it is feasible that a robot can be made up of several collision models such as head, arms, body, legs, etc. For detail of modeling and robot organization, please refer to Section 6.

5.4.2.3 Programming languages restriction

From above, accurate collision detection using the libraries above should be feasible in our game. However, since nearly all of the collision detection libraries are implemented using either C or C++, it may cause problem as our game is going to implement in Java. To solve it, we should either:

- port our collision detection packges from C/C++ to Java
- using Java Native Interface (JNI)to let the game engine (Java) to call the collision detection library (C/C++)

It is obvious that the latter solution has a greater feasibility. The reasons are:

- As mentioned above, Java is around 20% lower than C++ in logical intensive computation program. Collision detection is a logical intensive task.
- Porting a library take a relative large amount of time, as well as the time for debug and optimization.
- By using JNI, different collision detection libraries can be tried without expensive porting work.

However, the efficiency of using JNI to do the collision detection is not yet investigated. A demo program is used to detect the collision between two pyramids using ColDet. The demo is implemented in both C++ and Java where JNI is used to call ColDet library from Java. Below is the specification of the test and results of the tests:

	Test Case 1	**Test Case 2**	**Test Case 3**	**Test Case 4**
Language	**Java / JNI**		**C++**	
Collision detection	Enabled	Disabled	Enabled	Disabled
Number of triangles	8 (2 pyramids)			
Results (FPS)	**376**	**382**	**414**	**418**

Collision models are built in the beginning of the models while the collision models

are being transformed in each frame with corresponding modelview matrix. In each frame, collision between 2 collision models is tested. Two pyramids are kept rotating and their positions are reset after collision is occur.

Analysis

Ratio slower after enable ColDet in Java / JNI: around 1.6%

Ration slower after enable ColDet in C++: around 0.9%

From the test results above, it can found that the percentage slowdown after enabling ColDet in Java is 0.7 % higher than that C++. However, the extra slow down is so small that it is not significant to the whole performance of rendering process. As a result the overhead of JNI in this scenario is very small and not significant and it is feasible to use JNI in accurate collision detection in this game.

As concluded from above, accurate collision detection should be feasible in the game development. However, due to the computational intensive nature of accurate collision detection, exhaustive pair-wise detection is not feasible in a real time game.

5.4.3 Design and Construction

Collision detection is the process that detect whether 2 objects are in contact. The uses of collision detection in this game include:

- Prevent robots to pass through other objects like terrain, trees, and other robots.
- Detect whether the player successfully attack others or not, by means of close attack and shooting

Accurate collision detection is used in implementation some of the functions above. Moreover, the implementation of collision detection system should support a high scalability without lower down the performance very much.

5.4.3.1 Overall design – Cooperative Detection System

One of the uses of collision detection is to prevent the robots from penetrating into other objects. Actually the position of each object in the renderer is updated in every frame depending on the game logic. For example, the robots move according to external control, animation, and other factors. Bullets are also moving a predefined path and their positions are updated. Below show an extremely simplified loop when the game is running:

```
while(1) {
    backup_position();
    update_position();
    for (all_object) {
        if (collision) {
            restore_poisiton(object);
        }
```

```
        }
        render_scene();
    }
```

Problem: Exhaustive pair-wise detection

The above loop is very much simplified, but it can demonstrate the use of collision detection in preventing the objects in the environment from penetrating each other. The collision checking is done for each object with all other objects on the environment. So the time complexity of the above checking method, which is a pair-wise method, is actually in a $O(n^2)$ time. The above method is only useful and possible for game that has a small and limited number of players so that the number of iterations in the above for-loop is limited**. However, for our game, it is supposed to have a large number of players playing together, it is a very expensive task to check the collision using the above exhaustive pair-wise method since n will be very large**. A different method should be used in order to make the collision system in this game feasible.

Solution: Co-operative collision detection system

By the nature of network game, there should be different computers running the game with different views since different players are playing. As a result, if we use the above method, each of the computers joining the game actually doing duplicated collision detection, not to mention the poor performance of this method. Using this way is just a waste of resources in doing collision detection that is not related to player itself as well as doing the same detection with others. As a result, making use of the nature of network game, the collision detection system of each player running the game only detects the potential collision involving the own player, while no

detection will be done with the objects that is unrelated to the own player. Once collision is happened, a message will be sent to other players that are nearby the self player in the game to notify the collision to take appropriate action.

By using this method, each player is only doing collision detection related to them while not doing other unrelated detection. All the computers that join into the game can cooperate together to do all the collision detection needed in the world. The time complexity **is shrink to O(n)** since the number of checking for each player is just depend on the number of objects needs to be checked with own robot.

However, such kind of implementation which depends on network transfer may suffer from problems such as network delay, congestion, etc. Detailed analysis of this method comparing with traditional method for high scalability network games will be done in the "Result and Analysis" and "Evaluation" section later.

5.4.3.2 Candidate Pruning System

Problems: The number of detection needed is still high

There are two main types of detections in this game, they are:

- Detection with environment objects
- Detection of attack (close and distant attack)

Even though the co-operative detection system can effectively lower the number of detection need by letting each clients to handle the detection on their own, there are **still many collision detections** need to be done for each client in high scalability scenario as number of players in the game increases. However, majority of the detection being done is actually checking for impossible collision. For example, it is

impossible for the player's robot to be collided with a distant tree, or it is also impossible for a robot to be hit by a bullet where the robot is not on the path of the bullet. Since our aim is to design a robust and accurate collision detection system, we need to minimize the number collision detection need, especially for the expensive accurate collision detection used in attack collision, which is going to be covered in detail later.

Solution: Candidate Pruning System

No matter for which type of detection, we just want to check the potential collisions but not the others. The use of candidate pruning system is to eliminate all impossible candidates for collision and make sure that only essential collision will be done in order to save the time for collision detection in each frame.

The candidate pruning system handles static and dynamic objects in different manners, as shown below.

Static object

For static objects like trees, their positions are predefined in the terrain detail. In the candidate pruning process, we can select out the static objects from the terrain detail nearby the player's robot. The range is a small radius since the robot cannot suddenly move from one place to another place faraway in a frame. After this pruning process, the player's robot will be only checked for collision with nearby trees.

Dynamic object

For dynamic objects, the candidate pruning process is a 2-level process.

1st Level – Selecting candidates from nearby partitions

For dynamic objects such as other robots on the terrain, their position are not fixed but varied. Since our game has partitioning system that allocates different players into different partitions depending on their position in the game world, the robot in the same and nearby partitions can be obtained and selected from the partition module. Thus nearby robots can be selected for detecting environment collision.

2nd Level, - Selecting potential candidates within the partitions

However, most of the candidates in the nearby partitions are not possible to have collision with the players actually. As a result, a 2nd level candidate pruning process is carried out to further select the potential candidate. There are two approaches to different type of collisions. For the environmental detection and close attack collision detection, only candidates who have a distance with the player's robot smaller than a threshold will be selected since it is impossible for the player's robot to be collided with distant objects in the same partition. However, for the distant attack collision detection, a different approach is employed since it's the detection between the bullet with other targets but not involving the player robot. For such collision, Ray-Model Collision Detection is used to select the candidates. It aims at fetching the candidate that fall on the path of the bullet so that no excessive collision detections will be done. Detail of such collision pruning technique will be mentioned in later part.

As a whole, the goal of the candidate pruning system is to minimize the number of collision detections need to be done in order to fulfill the real time detection requirement, as well as eliminating all unnecessary detections that waste the time, especially the accurate collision detection techniques used in the attack collision, which is expensive. For the test results and analysis of the success of this pruning system, please refer to Section 5.4.4 "Test Results and Analysis" .

5.4.3.3 Detection with environment objects

Overview

In the game environment, there are many objects in the game world that need to be checked whether collision is occurred and prevent the user's character to penetrate into them. As mentioned above, each player is only checking for collision for his own robot but not others. The flow of the process is listed as follow:

1) Backup the position of the robot
2) Update the position of all objects in the world
3) For each potential objects that may have collision with the player's robot, perform detection with it.
4) If there is any detected collision, restore the last position of robot and send the incidents to nearby players.

The process of collision detection with environment objects is intended to be rather simple. It is because collisions between the players and environment objects are very likely to be happened and it should be a very frequent operation. Keeping it simple is essential to keep the performance high.

By the use of Candidate Pruning Process above, such detection will only be done for nearby objects but not the others.

Detection implementation

As mentioned in overview, simplicity should be the first concern of this part. As a result, accurate collision detection is not employed in this level of detection in order to keep this part fast and simple, in order to get these frequent operations simpler.

In many commercial games, a bound is usually used to cover the player character to check environment collision with other objects. One of the examples is MDK2[5.6]. The examples are bounding box, bounding cylinders, bounding spheres, etc. In this case, bounding cylinder should be most appropriate since it can tightly cover a robot, as well as no change need to be made when the robot is rotating. Other objects say trees are also having a bounding cylinder. The cylinder is always upright.

The cylinder property is expressed as a radius and height and accompanied with each objects. To check the collision, we just need to check whether 2 cylinders collide or not. To check whether 2 cylinders collide or not, we need to check whether:

- *The distance between the top of the cylinder and the xz plane of the lower situated cylinder is greater than the distance between the bottom of cylinder and the xz plane of another cylinder; and*
- *The distance between the projections of center base of both cylinders* are less than their sum of radius

If the 2 selected cylinders are both satisfied the above conditions, they are collided together. By using such bounding cylinder method, the robots can prevent penetrating through other objects in the environment.

5.4.3.4 Collision Detection for Attack (Accurate Detection)

Overview

In this game, accurate collision detection is used to detect the collision for attack actions. For example, it is used to test whether the player's robot can successfully hit the enemy by the sword as well as whether my robot can shoot the distant enemy successfully.

With the Candidate Pruning process mentioned above, the number of detections needed to be carried out is greatly reduced to keep the performance high especially in environment with many players. In addition, the use of Ray-Model Detection in Candidate Pruning is going to be covered in detail.

5.4.3.4.1 Use of external library and JNI

Feasibility studies on accurate collision detection in this game is carried out and some external libraries that perform accurate collision detection which suitable to this game were chosen. They are ColDet[5.2] and RAPID[5.8].

For both ColDet and RAPID, they are both implemented in C++ in object-oriented way, as well as having very similar structure and functions. The basic structure is collision model, and triangles can be added to the model to supply the geometric structure of the model. After adding all triangles and finalization, they provide functions to transform the model, to test whether two collision models are collided, as well as many other checking functions.

Since they are C++ libraries and our game is implemented using Java, the game program cannot directly make use of those libraries to use their functions. In order to make them available in Java, Java Native Interface (JNI) is used. Actually what JNI does is to enable Java Program to call native system function (e.g. C/C++) as well as enable native system program to call Java methods. In this scenario, only the former function is used.

In order to use a C++ class in Java, interfaces in both Java and C++ sides are made.

On the Java side, a reflect class is made with method you want to call to the C++ library with `native` keyword. Those native methods do not have implementation because the Java Virtual Machine will forward the call of those functions to the corresponding C++ function, with the help of wrapper program. Java SDK has a built-in program to generate a C header file from the reflect class above. The C program implemented from this header file is called wrapper. It diverts the call from Java to C++. This program can access both variables in Java and C++. For example, when a collision model is created on Java side, native method `init()` is called, then `init()` in wrapper program is called. In the wrapper program, it creates a new C++ collision model and sends it back the pointer to Java side for storage (as an integer). Afterwards, Java side can call different native methods to call different functions of this C++ object and get required functionality.

As a result, Java reflect class of ColDet and RAPID can be built and use them just like in C++ with the help of the wrapper program implemented for the purposes. After testing, it is found that RAPID is around 2 – 3 times lower than ColDet when using in this game for collision detection. So ColDet is chosen for the accurate collision detection library for this game.

5.4.3.4.2 Collision model manipulation

In order to detect collision accurately between two robots, they need to have corresponding collision models. Collision model in ColDet is built by feeding a number of triangles for that model, and the detection can only be made after finalize the collision model. However, after finalizing the model, the triangle structure can not be changed anymore but only able to transform the model, by a 4x4 modelview matrix.

But robots in this game are usually undergoing continuous animation and movement. If only one collision model is built for one robot, the animation and movement of the robot definitely cannot be expressed as a modelview matrix that can be used to transform the collision model. It is because the whole model is not rigid, that is, the polygon structure is changing over time. However, it is not feasible to build the model in each frame to test for collision since collision model building of ColDet is not designed for run time as it is a relatively slow process.

However, the robot itself is formed by various smaller models, such as head, arms, body, legs, and weapons. In addition, those smaller models are all rigid and their polygon structure will never change once they are in the game. The animation and movement of the robot are the results of geometric transformation of all its smaller models. As a result, those smaller models can be used to build collision models as well as transforming them in each frame by modelview matrix, depending on the robot's animation.

So when the robot is loaded into the game in the initialization stage, several collision models are made and bound to the robot. In order to transform the collision models when the robot is moving, before drawing each small part of the robot, the current modelview matrix is fetched and used to transform the corresponding collision model. As a result, the robot's movement can readily reflected to the collision model without the needs of rebuilding of them. Such transforming manipulations are done in every frame.

5.4.3.4.3 Close Attack

From the candidate pruning system, only candidates closed to the player's robot will

be selected for detection. The weapon is a part of model of the robot. As a result, there is a collision model for the sword and bound to the robot. In order to determine whether the sword can hit the target, the detection is implemented as:

```
S = collision model of sword of player robot
for (each of potential robot from pruning process, Ri)
{
  for (each of collision model of that robot Cj) {
      if (S collides with Cj) {
          // hit and break the loop
      }
  }
}
```

The library ColDet can readily compare whether two collision models are collided. After the implementation, it is found that the game can accurately determine whether the player's sword can hit other robots.

5.4.3.4.4 Distant Attack

The distant attack in this game means the player robot shooting other robot. When the player shoots, a new object of class `MZBullet`, which also extends `TerrainObject`, is created to represent the bullet the player's gun fired. The bullet has its own straight-line path which is determined by its forward vector. Under the concept of co-operative collision detection system, each player only detects its own collision. For far attack, each player only detects the collisions between bullets fired by him and other enemy robots. Once there is a collision event will be sent to other players in the same partition (through the network module).

5.4.3.4.5 Ray-Model Collision Detection for Candidate Pruning

Since the bullet is going through a straight line, the potential robots that can be hit by it should be fall on the path of the robot, and it is the concept of Ray-Model collision detection used in the candidate pruning process for distant attack.

From the ColDet library, there is a function that can be used to check whether a ray passes through a collision model. The ray is represented as a starting point and a forward vector. As a result, all candidate robots are being tested for the ray-model collision and only robots which the bullet's path passes through are selected. Then the prune step is done. The pruning step is represented by the following diagram.

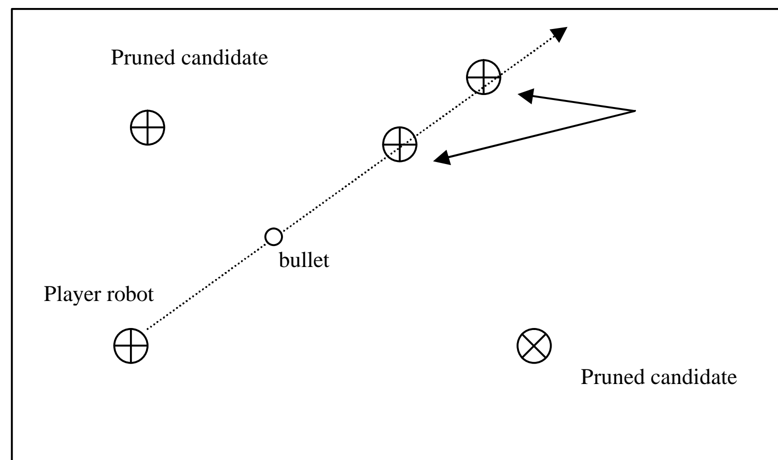


Fig. 5.6 Ray-model collision for candidates selection

To detect whether a bullet collides with a robot accurately, the follow routine is used:

```
B = collision model of the bullet;
for (each of potential robot, Ri) {
  for (each of collision model of that robot Cj) {
     if (B collides with Cj) {
```

```
                // hit and break the loop
            }
        }
    }
```

It is very similar to the detection routine of close attack with the collision model of sword being replaced by that of bullet.

Problem Encountered on Accuracy – Missing Frame Problem

However, during implementation, it is found that the system cannot detect collision occasionally. Even when the player robot and target robot are remained in same position and the player robot continue to shoot the target, sometime the hit can be detected while sometime cannot but it should already hit the target.

After investigation, it is found that there is a flaw in the above routine. Consider the following case:

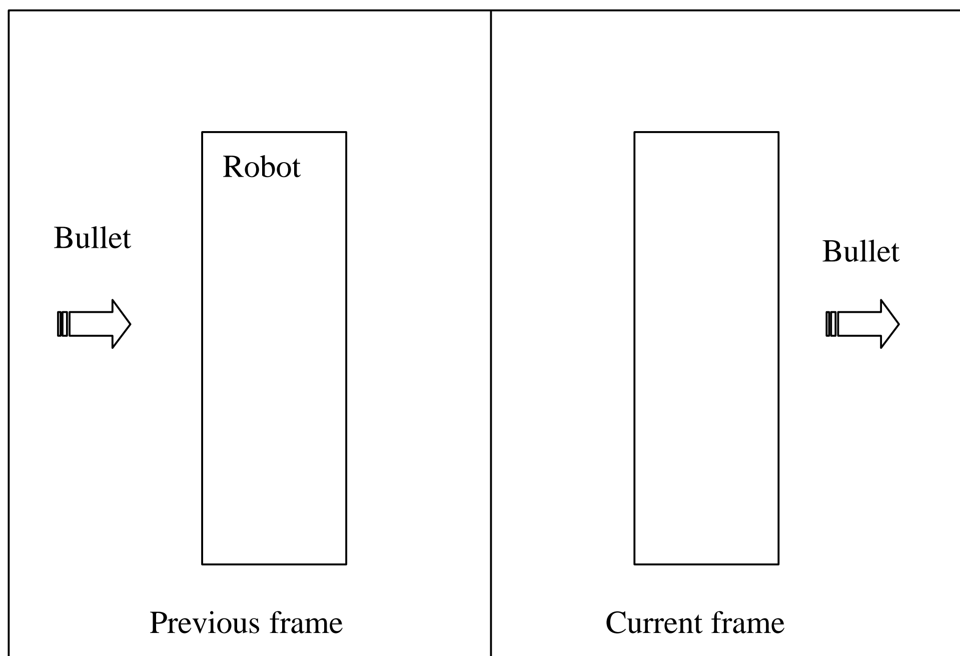


Fig. 5.7 Flaw in shooting attack detection

Although the bullet should hit the target, but the bullet really does not have any contact with the target in both of frames above. As a result, the above routine cannot

determine the collision. This problem happens because the bullet is moving in a very high speed comparing with that of robots, and this problem happens more frequent when the frame rate is slow.

Solution to Missing-Frame Problem

In order to solve this problem, an additional checking is performed if the collision detection between the bullet collision model and all collision models of other robot is failed, as shown below:

```
B = collision model of the bullet;
for (each of potential robot, Ri) {
  for (each of collision model of that robot Cj) {
      if (B collides with Cj) {
          // hit and break the loop
      }
  }
  if (bullet is approaching to Ri in last frame and
      bullet is leaving Ri in this frame) {
          // consider as hit and break the loop
  }
}
```

The additional test checks whether the bullet is approaching that robot in previous frame while leaving the robot in the current frame. If it is the case, then it should also hit the robot. Recalls that a prune step is carried out before the above routine and all potential robots in the list should be being penetrated by the bullet's forward vector. Given the bullet is just started leaving the robot in current frame while approaching in

last frame, it should have already hit and penetrated through the robot. After the above modification to the detection routine, the problem is successfully eliminated.

5.4.3.5 Collision response

Collision response is the action being taken after collision is detected. In this co-operative collision detection system, each player only focuses on his own collision and thus once collision is detected, an event with specific information of collision is sent to the network module to it will broadcast the message to all other robots that should get aware of this event.

For environment collision detection, the last position of the player robot is restored as well as other players in the same partition are informed of the event and restore the position of corresponding robot in their worlds.

For attack collision detection, the action needs to be undertaken are:

- Notify System Logic a hit is occurred to do corresponding logic computation such as increase scores, decrease live of the target robots, etc. It will further notify the network module to broadcast the event to other robots that should be aware of.
- Notify the animation module to carry out appropriate animation

Actually it can be seen that the network reliability is very important to the broadcast of collision event. It is because, for each player, any collision that does not involve the player robot will be done by others. Those information sending is very critical to maintain the realism of the game.

5.4.4 Testing Results and Analysis

The effectiveness of collision detection implemented in this game can be investigated for 2 directions, accuracy and performance. Accurate collision detection is one of the focuses in this game and such technique is employed in the detection of attack collision. Extensive accuracy test is carried out. In addition, performance analyses of different implementation of collision detection systems that differ from the current implementation are going to be conducted and analyzed with current implementation.

5.4.4.1 Accuracy test

As accurate collision detection is one of our focuses, the accuracy of collision detection is an important measure to the success of the project. One of the game features is target locking such that once a target is locked the bullet will be shot in a direction that going to the target. Given that the target is fixed in the position when the bullet is fired, the bullet must hit it. The test below setup 1 enemy robot in the world and let the player robot to shoot to it continuously, in order to measure the accuracy.

	Test Case 1	**Test Case 2**
Terrain Rendering	Enabled	Enabled
Object Rendering	2 Robots, one player and one target	2 Robots, one player and one target
Target Locking	**Enabled**	**Disabled, shoot next to the target (e.g. spaces between the legs)**
No. of shoots made	**100**	**100**
Result		
Hit detected	**100**	**0**

Misses	0	100

The result is yield an accuracy of 100% for shooting collision detection with target locking enabled. On the other hand, for the shoot that made not directly to the target robot but shooting next to it, no hit is detected throughout 100 shoots. Thus the collision system yields a very excellent accuracy in both determining hit as well as having no false detection.

5.4.4.2 Performance benchmarks

In the current implementation, co-operative collision detection system is used. That is, the player only detects the collision that is related to self robot. The following test investigates the performance of different detection libraries as well as the effect of such collision system with the traditional collision detection that detects all objects' detection.

5.4.4.2.1 Accurate collision detection libraries

During the implementation, ColDet is chosen over RAPID for the accurate collision detection functions. A test has been carried out between ColDet and RAPID for do environment collision detection with accurate collision. Below are specifications of the test:

	Test Case 1	**Test Case 2**
Library	**ColDet**	**RAPID**
Terrain Rendering	Enabled	
Object Rendering	Trees and player robot	

Result (in FPS)	55	18

It is obvious that ColDet offers a better performance then RAPID and the difference is very large. It may due to RAPID has not been updated for a number of years as well as ColDet is designed for game as quoted in its homepage[5.2]. As a result, ColDet is chosen for accurate detection library for development onward.

5.4.4.2.2 Co-operative collision detection system

The test is going to investigate the difference in performance of collision detection in using co-operative collision detection system and traditional exhaustive method. Bounding cylinder collision detection will be carried out for the collision test. A number of robots are placed on the terrain and checking for environment collision should be done. For co-operative collision detection system, only collision with the player robot is carried out. For traditional collision detection, all environment collisions in the world will be checked. Both few number of robots as well as large number of robots case will be tested and investigated. Below is the specification of this test and results:

	Test Case 1	**Test Case 2**	**Test Case 3**	**Test Case 4**
Detection System	**Co-operative**	**Traditional**	**Co-operati ve**	**Traditional**
Detection Method	**Bounding cylinder**			
Number of robots	**2**	**2**	**200**	**200**
Terrain Rendering	Enabled			
Objects Rendering	Disabled			

Frustum culling	Enabled			
Result (in FPS)	**120**	**120**	**114**	**20**

Please note that the robots will not be rendered and models are not loaded because we are only interested in the effect of collision system uses. Object rendering will lower the significance of the difference for different system, especially in the cases that have many robots, in which rendering of robots will take a large amount of resources.

Analysis

From the above result, we can see that the speed up bring by co-operative system is great comparing with traditional way. However, the speed up is unnoticeable when there are only few robots on the terrain. But the speed up is much greater when the number of robots in the worlds increase, which better simulate the environment of high scalability. It is obvious that if traditional way has a bottleneck in checking environment collision if the number of robots in the world keep growing. As a result, the current co-operative implementation is suitable for high scalability network game like the one in this project, and it brings satisfactory performance to this game.

5.4.4.2.3 Candidate Pruning System

Similar to above, during the testing, no robots will be rendered. There are 200 robots distributed nearby the player robot and it keeps continuously shooting so that accurate collision detection for distant attack is continuously taken place.

	Test Case 1	**Test Case 2**
Candidate Pruning	**On, using Ray-Model**	**Off**

	detection as pruning	
Detection System	**Co-operative**	
Detection Method	**Accurate Model-Model Detection**	
Number of robots	**200 around the players**	
Terrain Rendering	Enabled	
Objects Rendering	Disabled	
Result (in FPS)	**114**	**30**

Analysis

Without candidate pruning, the bullet's collision models needs to detect with ALL other robots in the scene while candidate pruning mechanism can eliminate the majority of them since only the robots fall on the path of the bullet (checking by Ray-Model Collision Detection) will be selected for accurate detection. As a result, there is a huge performance boost, around 4 times, for the candidate pruning system, which is also crucial to let the accurate collision detection to be feasible in real time.

Conclusion

From the above benchmarks, it shows that the current implementation is always using some methods to improve the performance of the overall collision detection system. For example, the co-operative system eliminates all redundant and duplicated detections and leaves them for their own players. In addition, although accurate collision detection yields a very good experience to user, extensively use of it lower the performance of whole game and finally lower the playability of the game. As a result, accurate collision detection is only used in detecting attack collision that the players find it most important, since the accuracy of it affects the game results and

playability heavily. More importantly, the Candidate Pruning process can greatly reduce the number of collisions, both bounding box detection for environment collision and accurate detection for attack collision, to cope with the real time requirement.

5.4.5 Discussions and Evaluations

5.4.5.1 Functions achieved

The collision detection system implemented in this game is divided to 2 main areas:

- Detection with environment objects such as robot-tree, robot-robot.
- Detection for the attack collision

With the environment collision detection, which makes use of bounding cylinder checking, dynamic objects like robots and static objects likes trees will never penetrate with others. For the attack collision, including close attack and far attack, accurate collision detection technique is employed and all attack collisions are being detected accurately to improve the realism of the game.

A co-operative collision detection system is implemented in this game and each player only focuses to detect collision involving their own robot. By taking this co-operative approach, the players co-operate together to do the detection for the whole virtual game world. Candidate pruning mechanism also helps to minimize the number of detections needed to be done in each frame.

5.4.5.2 Accuracy achieved

One of the purposes of the module of this project is to implement accurate collision detection system in the game. The test above completed with result of 100% accuracy if accurate collision detection library ColDet[5.2] is used to do the detection. The accuracy achieved is very satisfactory and there is no wrong detection resulted. As a result, for checking collision, the system yields an excellent accuracy of detection.

For the environment collision detection, accurate collision detection is not used in

order to increase the performance. An approximated bounding cylinder is used to cover the object. So, it is possible that a robot cannot walk further even it has not yet collided with the tree. However, implementing accurate collision detection in all of the collision including environment collision detection is not feasible in this project, since it drastically slows down the performance of the game. As a result, accurate collision detection is used only for attack collision detection while bounding cylinder is used for environment collision detection. The bounding cylinder's parameters are chosen according to the model of the object and keep the cylinder as tight to the model as possible.

In conclusion, the accuracy of collision detection in this game achieved is very high, especially for the attack collision, which having 100% accuracy.

5.4.5.3 Choice of accurate detection library

In the current implementation, ColDet is used for accurate collision detection. It provides a number of collision detection functions such as model-model detection, ray-model detection, and sphere-model detection. Model-model detection is used extensively in both close and far attack collision detection. In addition, the ray-model detection is very useful to select candidate when doing shoot attack collision, by selecting which robots have potential being hit by the bullet. After the game is implemented, it is found that ColDet provides enough functionality to the game as well as providing excellent accuracy. In addition, its performance is very satisfactory and having performance better than RAPID, another accurate collision detection library which provides similar functions and structures.

Actually there are many other libraries that provide the functionality of accurate

collision detection, some of them even make the user to be able to select pair of collided objects within a group of objects. However, the project only requires simple functions as above so they are not selected to maintain the simplicity.

5.4.5.4 Use of Java Native Interface (JNI)

Since the collision detection library ColDet is implemented in C++ while Java is used to code the project, the program cannot make use the library directly. Java Native Interface (JNI) is used to map the functions in libraries in C++ being callable in Java.

From feasibility studies, tests have been carried out to investigate the overhead of calling C++ library through JNI comparing with calling directly from C++. The overhead is very small and the result is satisfactory. Performance is not a problem in using JNI in this project.

However, after finished the JNI wrapper program for ColDet library, it is found that there is a small memory leakage problem when the program is running. After an extensive investigation over the code, it is found that the problem is originated from the wrapper program. In the wrapper program, variables in Java passed from the methods needs to be converted to C++ format before calling the C++ functions. In addition, special treatment is needed in converting array from Java form to C++ form. In the wrapper program, it needs to call special JNI function to let the Java VM to prepare the set of array to be available to be copied to the C++ array and pass to C++ functions. The memory leak problem happens because after copying the values of array to the C++ array, I forgot to release the Java array from the VM, while those variables prepared by JNI will not be garbage collected. Once the arrays resources in Java VM are released after calling, the memory leakage problem is solved.

In conclusion, using Java Native Interface is a very convenience method to make C/C++ libraries available in Java. However, the memory resources need to be carefully controlled and monitored to prevent the memory leakage problem.

5.4.5.5 Co-operative collision detection system

In this game, each player is responsible for the collision detection of their own. The collision event of other robots in the world is supplied and sent by other players. Each player is doing the minimal amount of collision detection. Comparing with traditional method that check for all collision detections in the world, the performance of this co-operative system is very high. From the tests above, it is shown that the performance boost is even more obvious for large amount of players in the world. It is because the time complexity of traditional method is $O(n^2)$ since each of the robot in the partition are being checked collision with each of the other candidate robots. However, this co-operative method is just having time complexity $O(n)$ since only the player's robot is check against other candidate robots. It is the reason why this system bring a huge performance boost especially when n is large. It is a key benefit of co-operative collision detection system that keeping performance high in high scalability scenario, that is, when n is large.

However, the effectiveness of this method is highly dependent on the network performance. It is because the collision event messages are sent to other players in the partition to take corresponding action. Reliable protocol and synchronization are other focuses in this project and they provide reliable and in order transfer to other players through the Internet, thus making the transfer of collision event message effective and robust.

Actually such collision event messages are analogous to the control events like forward, backward, left, and right, that also need to send to other computers when the player computer accepts the controls. Collision event messages are treated the same as the above control messages and send to other players through the network.

5.4.5.6 Degree of use of accurate collision detection

In the current implementation, accurate collision detection is used for attack collision while not the environment collision detection with other objects. It is because the performance of the game needs to be kept very high while environment collision detection is a frequent operation that cannot be expensive. In the player point of view, what he concerns in the collision detection in the game is that:

- The game judges the player robot successfully attacks another robot when the player's weapon geometrically collides with the target.
- The game will never judge the player robot being shot by others if the bullet is just passing through the player robot very near but not penetrating.

It is because the decision made in attack collision detection affects the players most and even decides them to win or lose. As a result, to produce high realism, high accuracy of collision detection in attack is essential. However, for environment detection, it just uses for prevent the robots from penetrating through the environment objects and the game players will not take much care about it. As a trade off of accuracy and performance, bounding cylinder is used to do collision detection for environment objects, with higher performance but lower accuracy.

5.5 Overall Evaluation

In the view of the whole module, both graphic engine and collision detection system have been investigated, researched and implemented for the game. Extensive researches and feasibility studies that being carried out before the construction phase are useful and valuable. It is because they do not just let me to be more familiar with the topics, as well as preparing a clear path for the stage of design and development, that is, the uncertainties during design and construction is minimized.

In this module, a graphic engine that is extensible and having satisfactory performance is implemented and tested. It has the basic requirements such as rendering the world, maintaining list of objects to be displayed, some tricks to improve realism as well as some performance tuning. It turns out that performance tuning is a very important step to design and implement graphic engine since it is found from the tests, that the performance boost after several optimizations in this game is already very significant. Generally speaking, the graphic engine implemented satisfies the requirements specified.

For the collision detection system, the use of accurate collision detection in game is well studied and researched, as well as successfully implemented in this game with very high accuracy for attack collision. Co-operative collision detection system is also implemented to lower then computational power need when the number of players is high. It is shown from tests that such co-operate system is having a greater performance than traditional one that do all the collision detections on their own. In addition, Candidate Pruning process successfully minimize the number of detections needed. Generally, for the collision detection system implemented, it satisfies the

requirements of high accuracy, high performance and adapts to high scalability.

5.5.1 Efficiency of research, design and construction

For this module, overall graphic engine design and collision detection, I have spent around 40% of time to do researches and feasibility studies of related topics such as OpenGL graphic programming, methods to carry out accurate collision detection, the use of Java Native Interface (JNI), etc. All of the researches and studies are carried out in the Internet. The Internet is really a rich set of resources pool that let me find the related papers, tutorials and reports. Feasibility studies are carried out after gathering enough information. The efficiency of this part is good and lot of information are gathered and studied. It makes me much more familiar with the topics and can think solutions to the project problems. In addition, information is shared among all the members and collaborations about OpenGL programming are carried out with my group-mate, Yuen Man Long, who is responsible for the model rendering and animation.

For the design and construction stage, since Java is used and it has a set of useful API, the development is easier than C++. Functions were implemented one by one and tests were conducted when each functions is finished. Tuning and debugging were also done if there were any problems. The efficiency is quite good except when developing the JNI wrapper program for the collision detection program. It is because inadequate studies about memory management of JNI have been conducted as well as the unfamiliar with this new style of programming. Recalls the JNI bug I encountered, quite a lot of time are spent to figure out where the problem was originated, checking for the errors and code amendment.

5.5.2 Interactions with other modules

The organizations of the graphic engine and collision detection system are in object-oriented style such that the interface letting other modules to use is keeping simple as possible, as well as to encapsulate the internal implementations to the outside interface.

For example, the system game logic can manipulate the objects in the graphic engine easily such as moving objects, calling the objects to animate, as well as other manipulations. However, such operations rely on a set of standard function of the renderer and the implementations of them are unknown to the system game logic.

On the other hand, when there are any events like collision is detected, the collision detection system also notifies the system game logic so that this message is broadcast to all players in the same and nearby partitions. With the help of system game logic, it reads and dispatches the event to network module and send to selected peers. All those operations are hided from this module.

Apart from that, this module is also closely related to the module of model animation and rendering. According to different commands set by the system game logic, they are processed and feed into this module and appropriate action and animation will be rendered by that module. During the implementation of this project, such kind of interfaces organization eases the development a lot. It is because the implementations of modules are changing throughout the construction process. By using this organization, as long as the interfaces of each module do not change, modules can still co-operate and interact together properly.

5.6 Conclusion

Throughout various stages of the project, the project matters are being researched and studied extensively, design and construction have been carried out following by a series of tests and benchmarks to analyze and evaluate the effectiveness of this project. In the module of Graphic Engine and Collision Detection, as evaluated extensively in previous sections, it is found that both of them satisfy the requirements. An extensible, object-oriented and effective graphic engine and an accurate, co-operative, and robust collision detection system are successfully implemented in this project and being used by the game Mecha Zeta. In addition, Java is also found to be an effective programming language to implement the game in a relatively short period of time as it yields a better productivity, despite of a small performance lag to C++. However, the performance of graphic engine and collision detection system is still very satisfactory according to the test results.

In case there will be any future development, for the graphic engine, it is recommended that more time should be put on the field of optimization. Actually there are many optimization techniques that can bring performance boost apart from those used in this module of the project. As the complexity of the scene increases, the need of optimization also increase since fast graphic engine can give the players a joyful and realistic game experience.

For the collision detection, since only two libraries are being investigated and tried in this project, more can be investigated and tested to see how they perform comparing with the one which used in this project. In addition, the co-operative collision detection system incorporated in this project lets the players of this game to do

collision detection in a co-operative manner by doing collision detection related to own player only and sharing the results to all the others. Actually some other co-operative methods can be investigated to give more robust performance to the collision detection system.

6 3D Modeling, Animations and Real Time Shadowing

6.1 Theoretical principles

6.1.1 Overview

In this section, we are going to look at three theoretical principles. The first one is to determine whether a point is above or below a plane given a plane equation. The second one tells us how to obtain the silhouette edges of a 3D object. The last one is about how to project or cast shadow in a 3D environment onto the drawing screen.

6.1.2 Determining whether a point is above or below a plane given a plane equation

To determine whether a point is above or below a plane, first of all we need an equation for the given plane. Given any 3 point: A (x1, y1, z1), B (x2, y2, z2) and C (x3, y3, z3), the plane equation is obtained as follows (the side with A, B, C in counter-clockwise order is regarded as the upward face):

Let a'x + b'y + c'z = 1 be the equation.

Then we have: a'x1 + b'y1 + c'z1 = 1

a'x2 + b'y2 + c'z2 = 1

a'x3 + b'y3 + c'z3 = 1

i.e. in matrix form:

$$\begin{pmatrix} a' & b' & c' \end{pmatrix} \begin{pmatrix} x1 & x2 & x3 \\ y1 & y2 & y3 \\ z1 & z2 & z3 \end{pmatrix} = 1$$

And the inverse of $\begin{pmatrix} x1 & x2 & x3 \\ y1 & y2 & y3 \\ z1 & z2 & z3 \end{pmatrix}$

$$= \frac{\begin{pmatrix} y2\times z3 - y3\times z2 & z2\times x3 - z3\times x2 & x2\times y3 - x3\times y2 \\ y3\times z1 - y1\times z3 & z3\times x1 - z1\times x3 & x3\times y1 - x1\times y3 \\ y1\times z2 - y2\times z1 & z1\times x2 - z2\times x1 & x1\times y2 - x2\times y1 \end{pmatrix}}{x1(y2\times z3 - y3\times z2) + x2(y3\times z1 - y1\times z3) + x3(y1\times z2 - y2\times z1)}$$

$$\therefore a' = \frac{y1(z2 - z3) + y2(z3 - z1) + y3(z1 - z2)}{x1(y2\times z3 - y3\times z2) + x2(y3\times z1 - y1\times z3) + x3(y1\times z2 - y2\times z1)}$$

$$b' = \frac{z1(x2 - x3) + z2(x3 - x1) + z3(x1 - x2)}{x1(y2\times z3 - y3\times z2) + x2(y3\times z1 - y1\times z3) + x3(y1\times z2 - y2\times z1)}$$

$$c' = \frac{x1(y2 - y3) + x2(y3 - y1) + x3(y1 - y2)}{x1(y2\times z3 - y3\times z2) + x2(y3\times z1 - y1\times z3) + x3(y1\times z2 - y2\times z1)}$$

Rewriting the equation into the following format: **ax** + **by** + **cz** +**d** = **0**,

setting $d = -(x1(y2\times z3 - y3\times z2) + x2(y3\times z1 - y1\times z3) + x3(y1\times z2 - y2\times z1))$,

we will have $a = y1(z2 - z3) + y2(z3 - z1) + y3(z1 - z2)$

$$b = z1(x2 - x3) + z2(x3 - x1) + z3(x1 - x2)$$

$$c = x1(y2 - y3) + x2(y3 - y1) + x3(y1 - y2)$$

Note that the normal of the plane with this plane equation is pointing to the direction obtained by the right-hand rule. i.e. the "counter-clockwise side" is the upward face. If we try to find the plane equation with the points in reversed order, we will get a plane equation facing an opposite direction. i.e. –(ax+by+cz+d) = 0.

Substituting the x, y, z co-ordinates of a point into ax + by + cz + d, we will get a value proportional to the distance of the point from the plane. If the obtained value is negative, the distance found will be negative and that means the point is under the plane w.r.t. the normal. On the other hand, if the obtained value is positive, the point

should be above the plane.

So, we can conclude that, given a point P (x0, y0, z0) with plane equation ax + by + cz +d = 0, if $a \times x0 + b \times y0 + c \times z0 + d > 0$, the point P is above the plane.

This theory is applied in determining whether a face is visible to a point light source in Silhouette Determination. A face is considered "visible" to the light source if the light source is "above" the plane. Details about the Silhouette Determination will be discussed in the next section.

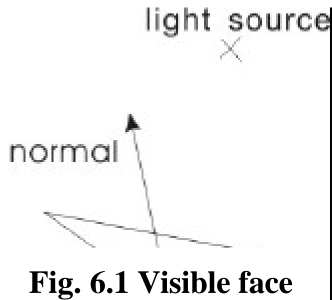


Fig. 6.1 Visible face

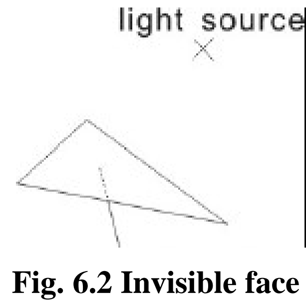


Fig. 6.2 Invisible face

6.1.3 Silhouette Determination

The Silhouette edges for a 3D object are actually the outlines of the object when you look at the object from a certain point (the reference point) in the 3D environment. These edges together bound and form the shape of the object when viewing from that reference point.

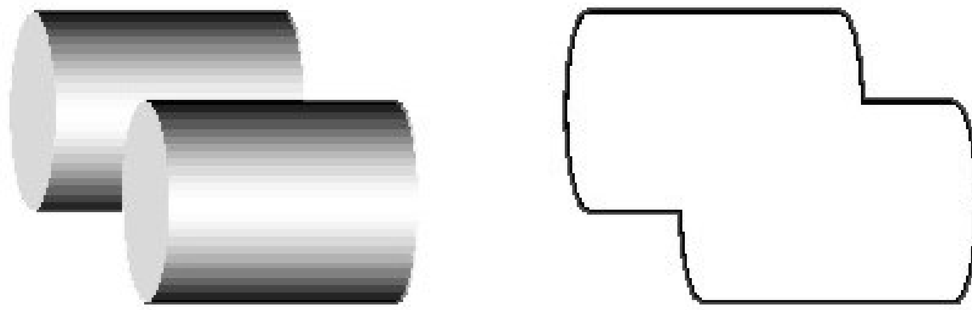


Fig. 6.3 Shaded objects and the Silhouette edges for the objects

To obtain the Silhouette edges for a 3D object from a certain view point, we need a special structure to hold the 3D object. As usual the 3D object should contain a list of face (say a triangle face). The face consists of three indices for the three vertices. An edge is identified by a pair of vertex indices. For a closed mesh, there should be a neighboring face for each of the three edges. In addition, we need the plane equation for each face in order to tell whether the face is facing the view point.

With the above information, we can start identity the Silhouette edge with the following steps:

1. Loop through all of the object's faces (triangles).
2. Determine whether the face is visible (facing) the "view point" given its plane equation and the co-ordinates of the view point using the theorem.
3. Loop through the faces again
4. If the face is visible, go through all the three edges
5. If the neighboring face is not visible, this is a Silhouette edge. (For non-closed mesh, if there is no neighboring face for this edge or the neighboring face is not visible, we will treat it as a Silhouette edge)

For example, in figure 6.4, the only two visible faces are the two triangles on the top. For the two triangles, the edge between these two triangles is not a Silhouette edge as both of the faces containing this edge are visible. The resulting Silhouette edges are the four edges bounding the top two faces.

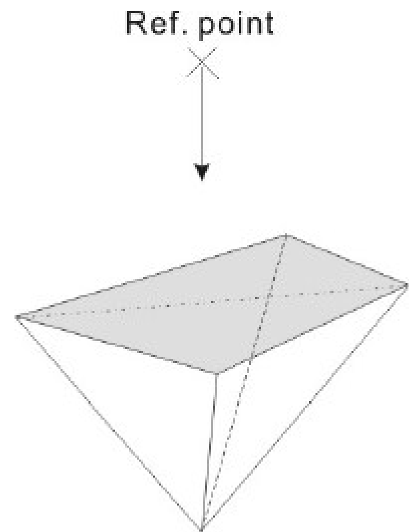


Fig. 6.4 Forming Silhouette edges

6.1.4 Shadow Volume

6.1.4.1 What is Shadow Volume

A shadow volume is simply the half-space defined by a light source and a shadowing object. Or you can think of it as a volume in which object will be shadowed.

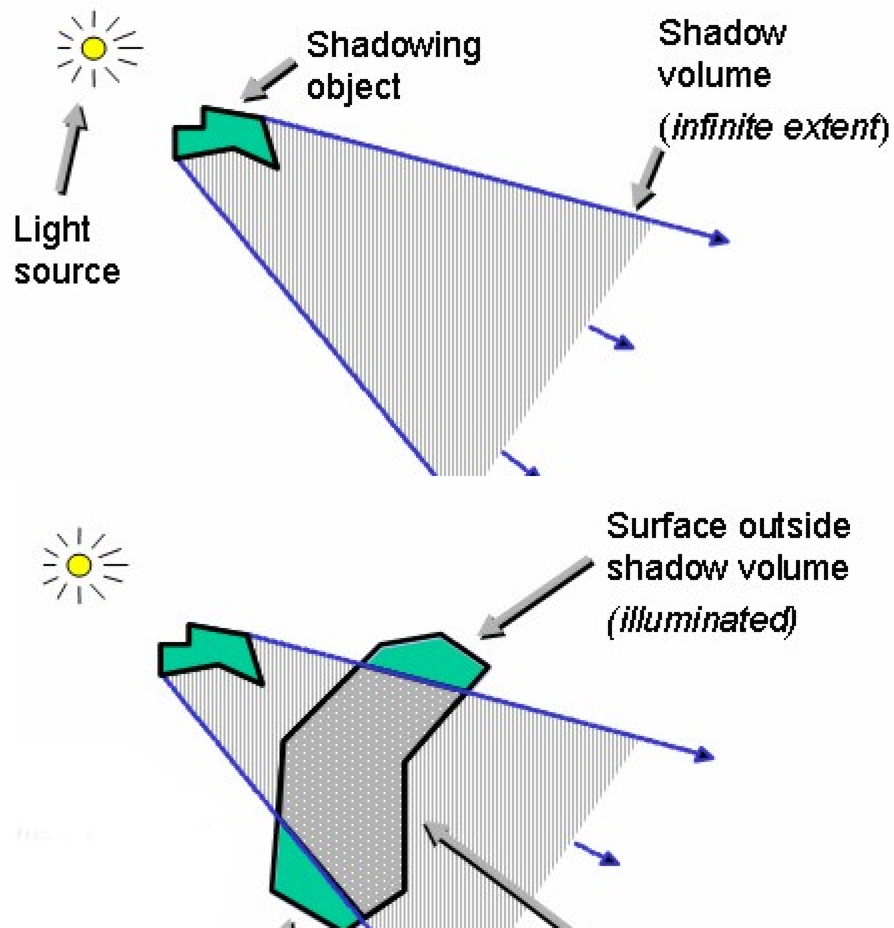


Fig. 6.5 Object partially inside shadow volume

6.1.4.2 How to determine whether a point on an object is inside the shadow volume w.r.t. a given eye position

One of the most common and efficient method is the **eye-to-object stencil count approach** in **Zpass** mode. Imagine you are look at a point on an object from a distance. A light source (e.g. the Sun) is casting a shadow volume from a shadowing object like in **figure 6.5** (Shadow casting will be discussed later). A count is used and initialized to zero. Traveling from the eye position to the point on the object, if you enter a new shadow volume, increase the count. If you are leaving one of the shadow volumes, decrease the count. Stop counting when you reach the pint on the object. Finally, if the count is greater than zero, that means the point on the object is under one or more shadow volumes (i.e. the number of shadow volumes covering the point is equal to the final count) (See **figure 6.7**). Otherwise, the point is not shadowed (See **figure 6.8**).

6.1.4.3 What does that mean by "Zpass"

Z-value represents the depth of the currently nearest point in certain viewing direction from the eye position. The above approach is a "Zpass" approach, because we start

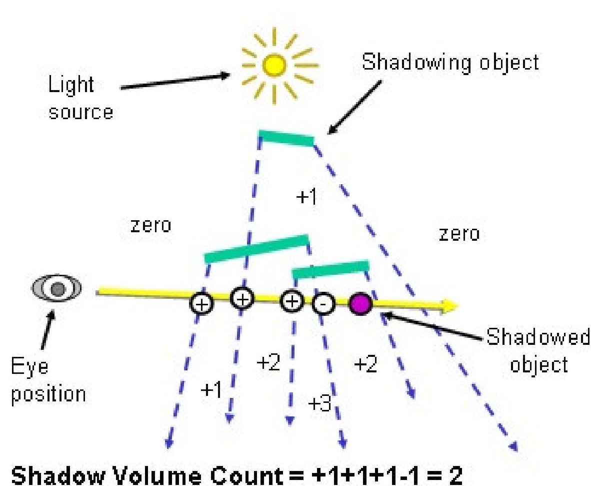


Fig. 6.7 Shadowed part on an object

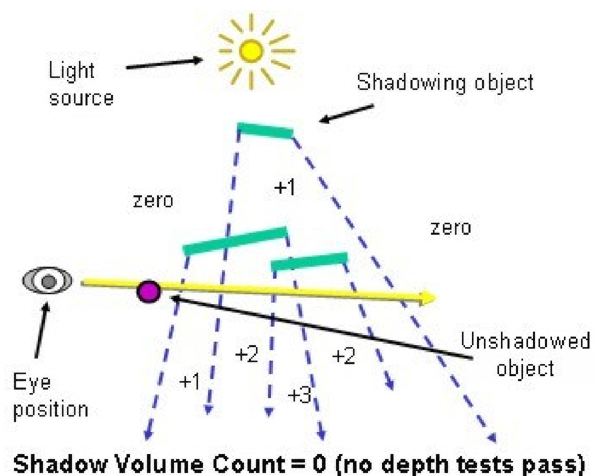


Fig. 6.8 Unshadowed part on an object

the counting from the eye position in certain viewing direction to the nearest object,

where depth-test can be passed or the z-value is less than the z-value of that nearest object.

6.1.4.4 Shadow volume casting

To construct the shadow volume, we have to obtain the Silhouette edges for the shadowing object using the techniques mentioned in last section with the light source being the reference point. Then for each edge in the set of Silhouette edges, project the two vertices of this edge from the direction of the light source to infinity (See **figure 6.5**). Implementation will be shown later.

6.2 Design and construction

6.2.1 Overview

The design and construction for which I am responsible can be roughly divided into five parts: 3D modeling, animation, character control mechanism, game logic and graphical effects. The role of these five parts can be summarized in the following diagram (represented in red blocks):

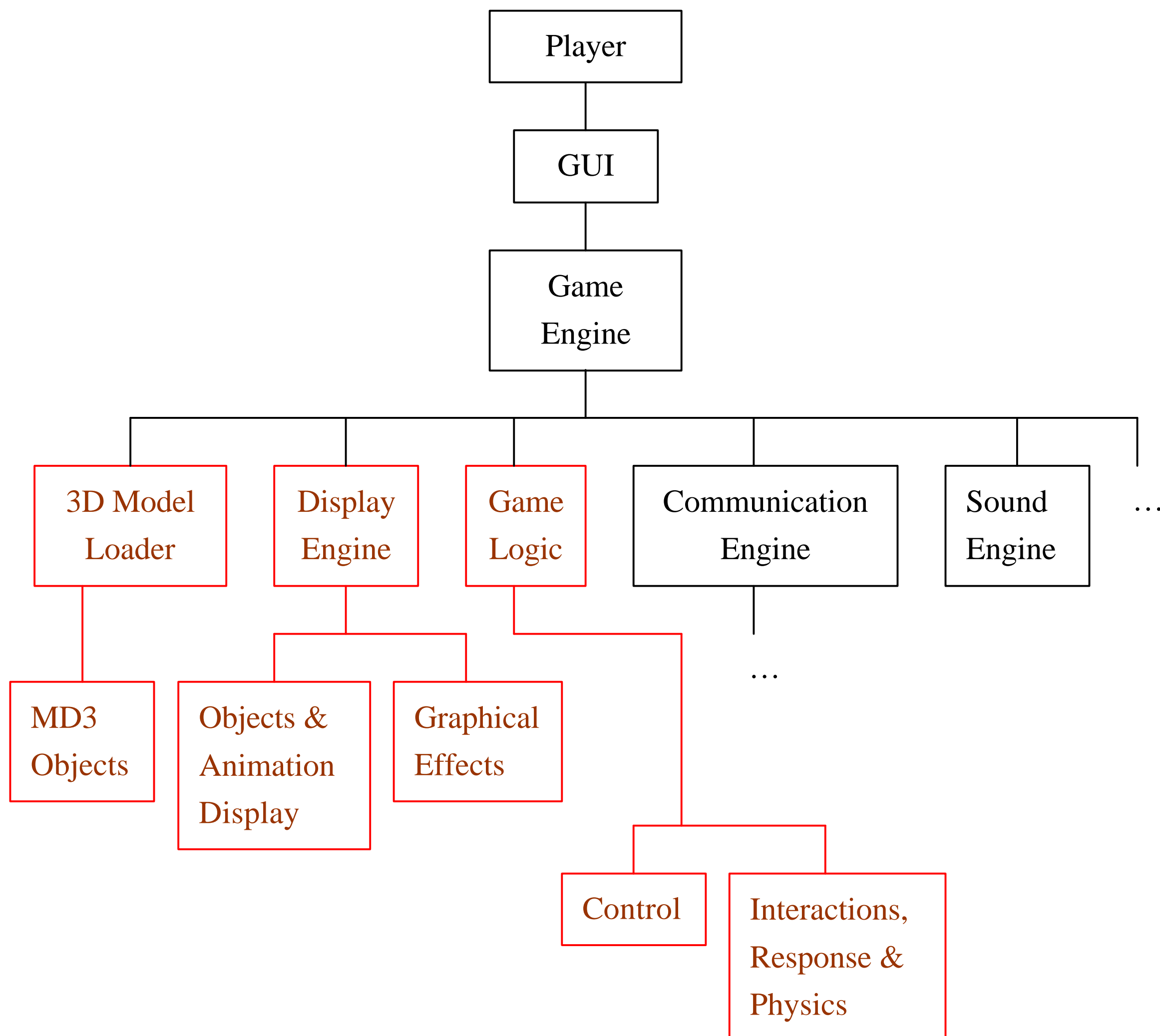


Fig. 6.9 Structural view of concerned parts

6.2.2 3D Modeling

6.2.2.1 Overview

The 3D modeling involves expressing the objects inside the game, such as robots and

scenery objects, in a computer format. The concerns in the modeling mechanisms include interpretation of the models in Java, mechanism for modeling the animation, hierarchical structure of the models, flexibility of the model and etc. We aim at designing robots which are composed of reusable parts (e.g. head, body, arms & legs). Each robot is actually a combination of these parts. To construct a robot, all we need is a set of IDs for each of the part.

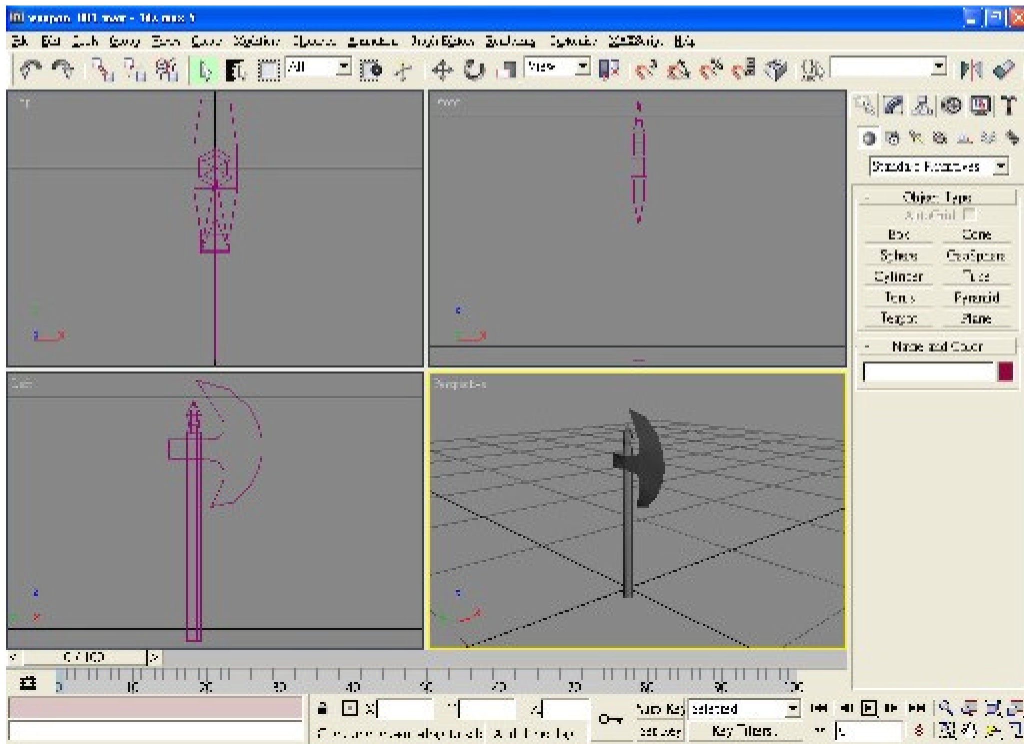
To build up a model, for example a tree, first of all we need to draw the meshes of the model using 3D model constructing software. After that we need to do texture mapping for the meshes. And if the model is a dynamic object, we have to build animation for it at this stage. And then we export the texture mapped model into an understandable format (so that we can parse the file in program). We parse the exported file and read the information into Java objects. Necessary initializations are done before the object can be displayed.

6.2.2.2 Choice of 3D model constructing software

3D Studio Max 5 is chosen to be the software to build 3D models in our project. Reasons behind are:

1. High flexibility and power for creating shapes and animations
2. Different plug-ins readily available, e.g. exporter and tools for texture mapping.
3. The .max file format contains high level information, such as animation in vector format (but not recording all the vertex co-ordinates for each frame).

4. The .max file can be export to the format we need easily with most of the information we need.



Feasibility for using other 3D model constructing software

The "**Maya**" is another powerful 3D software. It is not our choice because it is difficult to find some plug-ins or utility tools (such as "Texporter" which will be mentioned below) for it. And we cannot find a way to export the model to our desired format. On the contrary, the "**Milkshape**" is another good choice as it is easy to use and pick up. However, due to its limiting functions and plug-ins, it is not chosen.

6.2.2.3 Texture mapping (skinning) in 3D Studio

To map texture onto different parts of the model with exact position for complex shape, we need to do it in 3D Studio rather than in programming. First, when we get an object (See **figure 6.10**), we need to divide the mesh into several meshes to make it planar, by rotating the sub-mesh and adjusting the position of the vertices. Then generate a UVW map using a plug-in called "Texporter" (See **figure 6.11**).

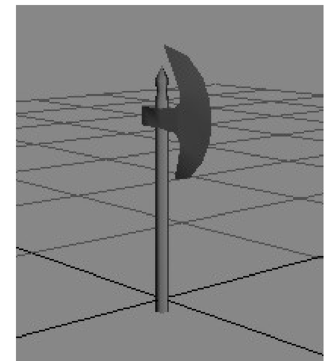


Fig. 6.10 An axe

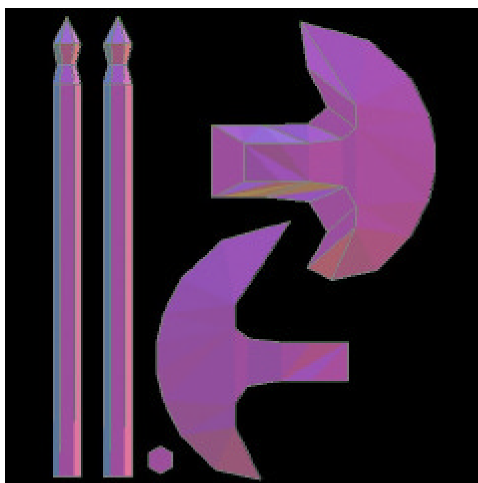


Fig. 6.11 Texture map

Fig. 6.12 Painted texture map

After painting on the texture map generated by Texporter (See **figure 6.12**), we can map it onto the object (See **figure 6.13**). This mapping can be exported and read into our data structure in Java. For more details, please refer to Reference.

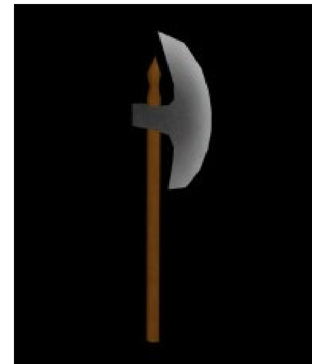


Fig. 6.13 Skinned axe

6.2.2.4 MD3 format

The .md3 format is chosen as the format for holding 3D models. It is the model format that is directly read into Java objects. The MD3 format is originally the model format for the well-know 3D online shooting game Quake III. We choose it because of the following reasons:

1. The bit-wise structure of MD3 format is opened to public so that we can write a complete loader for it.
2. It makes use of the "tag" concept which allows different parts to be connected conveniently. This makes it possible to draw the parts separately. Different combinations of parts can be done at run-time.
3. MD3 stores animation by storing all the faces and vertex positions for fixed time intervals (e.g. 30fps animation means there are 30 sets of frame in 1 sec. for each model). Comparing to the method of storing the initial positions and the changes of positions over time, MD3 has the advantage that extracting an intermediate animation does not require transforming the initial polygons. Only interpolation between two frames is needed. This results in better performance.

Limitation of MD3

1. The number of vertices in the model must be fixed through the animation.
2. Frame rate has to be determined at model building time. Real time display

requires interpolation of frames.

3. Scaling of the model can only be done through scaling the vertices in the model, but not through the tag.
4. The bounding box of the model is not included thus hindering collision detection. One of the measures to solve this problem is to roughly estimate a bounding cylinder for the model or to calculate the bounding box at startup time.

Feasibility for using 3DS or MAX format

1. There are still a lot of unknowns in the bit-wise file format, especially for the animation information. It is difficult to develop a loader for either of them. We have even less information on the MAX format.
2. They record animation in the form of initial position, translation, rotation and scaling of meshes and vertices. This requires more job to be done if we want to extract an intermediate frame as mentioned before.

6.2.2.5 Design and structure for model

Tag

Before explaining the design of the structure of our model, first we have to introduce a important concept called "Tag". It plays an important role in connecting different parts of the model. It is actually a vector with no volume. It is a flat right angled triangle with a normal pointing upward and perpendicular to the triangle (See **figure 6.14**). The position of the tag defines the translation of the part to be connected and the orientation of the tag defines the rotation. No scaling can be done through the tag. A tag somehow acts like a joint.

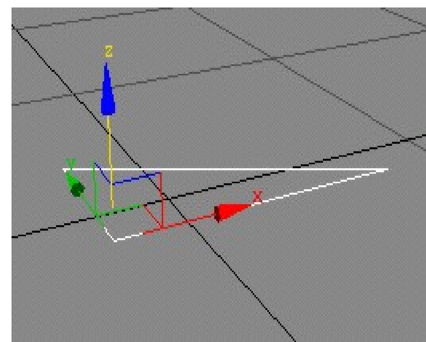


Fig. 6.14 A tag

Data structure for model

The basic part of a model is stored in a class called MZModel in our program. An instance of MZModel would contain a list of "objects", a list of links connected by tags, some information about the texture used and some information about the animation etc. Each "object" contains a list of faces, a list of non-repeating vertices, a list of normals and a list of texture co-ordinates. A face in turns contains the indices of the three vertices that form the triangle face. Moreover, each face will have its own plane equation, the indices of the three neighboring faces, for the sake of shadow-casting. All the data will be read by a loader from the MD3 file at startup time except the plane equation and indices of the three neighboring faces. These two elements will be computed after reading the basic information about the model. In the meantime, the texture (skin) will be loaded for each of the parts according to the **.skin** file.

MZModel
object list
link list
texture
animation

Obj
face list
vertex list
normal list
tex-coord list

Face
ndices
uation
neighboring faces
visibility

Fig. 6.15 Basic model structures

Hierarchical structure of robot

Basic on the tag-concept, a hierarchical structure of robot is built.

Original design

The parts of the robot are linked in a tree structure, with the leg as the root (the leg is in turn linked to the floor, i.e. the terrain). The body is connected to the leg. The head, left and right arms are connected to the body. The shield and weapon are connected to the left and right hands respectively (See **figure 6.16**). When the leg moves, the body

will also move accordingly. When the right hand swings a sword, the sword will follow the motion of the arm.

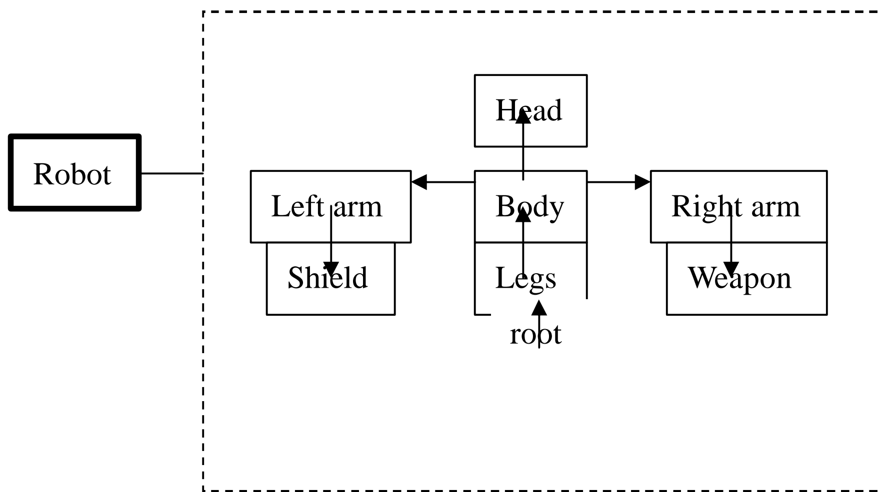


Fig. 6.16 Original hierarchical structure of

All these parts are connected by tags. Each part has its own animation

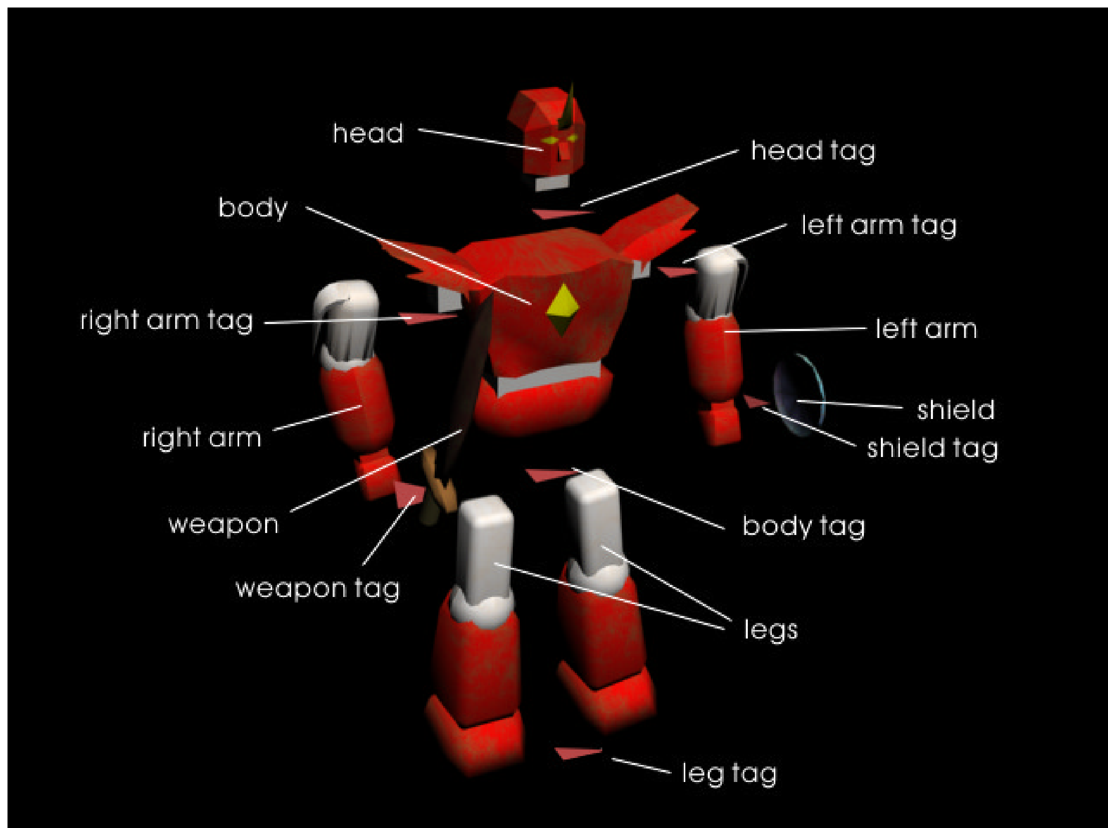


Fig. 6.17 Image for original hierarchical structure of robot

Critical problem

For any current approach of doing collision detection, the above implementation is infeasible. It is because in order to do real-time collision detection efficiently and accurately, each part of the model (including all the vertices) should remain fixed. We can only inform the collision detection engine that the part has been translated or rotated (but not transformed freely). With the above implementation, when you swing a sword, the position for connecting the right hand is basically fixed. The animation is achieved by having the vertices in different positions in different frames. This violates the restrictions of the collision detection engine. (For details, please refer to the part of the Section 5 "Graphic Engine and Collision Detection")

Solution – a new design

The robot is still connected through tags. The main different to the original design is that we will separate the animation and objects into different parts. What does it mean? In the current design, two new parts, namely "lower" and "upper" are created to hold the animation for the lower and upper body. They are parts that contain no meshes as all. They are only consisting of tags, translations and rotations of tags. The lower part contains the tags for the different sub-parts of the legs. The upper part contains the tags for the head, sub-parts of body, sub-parts of arms, weapon and shield. The lower part becomes the root of the robot. The upper part is connected to the lower part yet by another tag. Meanwhile, every individual part has only one frame, i.e. fixed mesh. This approach makes it possible to have good collision detection.

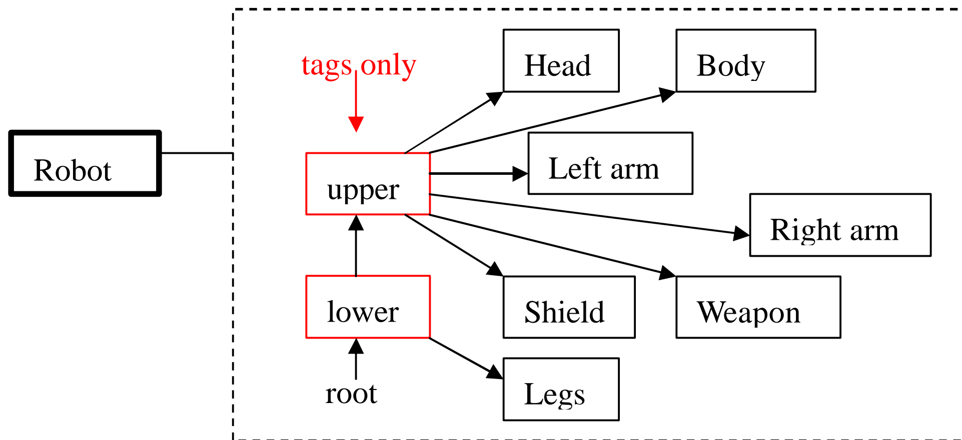


Fig. 6.18 New hierarchical structure of robot

(For other advantages of using the new design, please refer to the sections for animation and stencil shadow casting)

Displaying the model

To display a part of the model, we need the following steps:

1. Bind the texture used for this part (glBindTexture).
2. Go through all the faces.
3. Extract the indices for the vertices in each face.
4. Set up the normal and texture co-ordinates for each vertex (glNormal & glTexCoord).
5. Draw the vertices to form faces (glBegin(GL_TRIANGLES), glVertex & glEnd).

One of the features about our model structure is the recursively connected parts. i.e. the arms motion is relative to the upper body, and the upper body motion is in turns relative to the lower body, etc. So, when we display the model, we would draw the parts recursively. Before we draw a link, we would apply “glTranslate” and “glMultimatrix” according to the tag for that link, so that the linked part is drawn with appropriate translation and rotation. This is done recursively until everything is drawn. After drawing one link, the previous matrix will be restored, preparing for drawing

the next link.

Synchronization of the model

In order to prevent the model information (such as current frame) from being modified during display (which will cause flickering of the screen), update of such information is buffered and proceeded after each display.

Outlook of models

Current two sets of robot parts are built (See **figure 6.19** and **6.20** below).

Fig. 6.19 Robot set 1

Fig. 6.20 Robot set 2

The parts can actually be combined arbitrary like in

figure 6.21.

Other models such as weapons and shields are also

built. And they can be combined with the robot.

(**figure 6.22**)

Fig. 6.21 Different combinations of parts

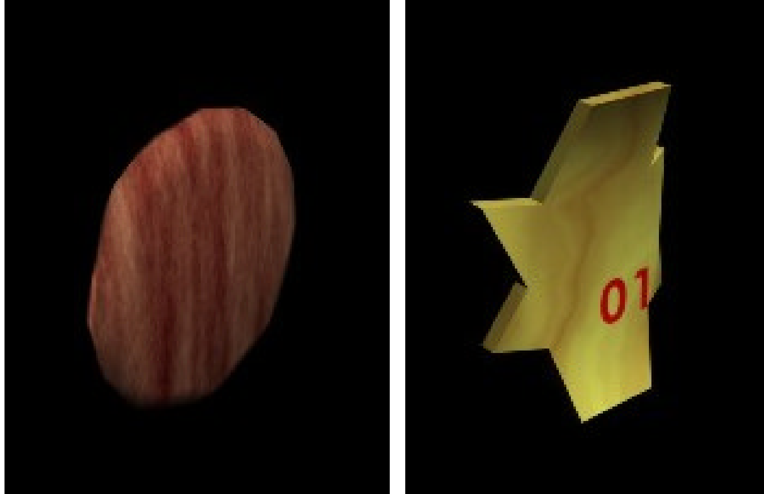


Fig. 6.22 Weapons, shields and equipped robot

Beside these dynamic objects, some static scenery objects are also there, for example, a tree model (**figure 6.23**).

Fig. 6.23A tree model

6.2.3 Animation

6.2.3.1 Overview

Animation plays an important role in a 3D action game. Transition between different animations and the continuity of frame within an animation is one of the emphases in

the graphics part of our project. We have also paid attention to the convenience of the data structure at the design stage.

6.2.3.2 Comparison of the two hierarchical structure of robot in animation

Original design

1. The robot animation is formed by coordinating the animations of individual parts. For example, swinging a sword requires the right hand to take action and the body to turn. This results in high flexibility and reusability of the individual animations. These individual animations can combine in many ways to form new animations.
2. Animation is fixed as create-time, i.e. the behavior of the animation is hard-coded in the model.
3. Display list in OpenGL cannot be adopted since the mesh for each part is not fixed.

New design

1. The robot animation is formed by coordinating only the animations of the lower and upper body. Swinging a sword only requires one animation in the "upper" part. Flexibility is still there as most of the actions require only coordination within the lower or upper part of the robot. And the animation combination is also simplified while it is still as expressive as before.
2. Animation can be adjusted more easily through modified translation and rotation base on the tags.
3. Display list can be adopted with fixed meshes, thus increase performance.
4. Size of the model file is smaller since the mesh for each part is fixed and the animations are only represented by tags which have no volume.

5. The “lower” and “upper” parts consist of pure tags forming a structure similar to skeleton or bone structure.

6.2.3.3 The .cfg file

It is a file for configuration of the animations. For each part containing more the one frame (i.e. having animations), there has to be a .cfg file. This file contains information about all the animations in the model. The animations for each part are stored as one long consecutive animation in the MD3 file. To extract an animation, we need to know at which frame it starts and at which frame it stops.

In the .cfg file, every line represents one animation. For example, for the line:

“**1 24 0 30 201 UPPER_PUNCH**”,

it is defining an animation called “UPPER_PUNCH” with ID 201 which starts from frame 1 and stop before frame 24. The fps for this animation is set at 30 and the zero here means that the animation is not going to be held at the last frame, and must be proceed to another animation.

6.2.3.4 Interpolation between frames and frame skipping

Interpolation

For each animation there is a preset frame rate (30 fps for the above example). However, it is impossible to have the display rate the same as this frame rate. And different animation may have different frame rate. In order to smoothen the animation, when the display rate is faster than the frame rate of certain animation, interpolation will be done on the nearest two frames. The interpolation is done on both the tag animations and mesh animation (no mesh animation in the new design). It is based on a variable “t” ranging from 0 to 1. When the animation proceeds to a new frame, “t”

will be set to 0. We use the frame rate for this animation to calculate how long each frame should last. "t" should increase gradually and reach 1 when it is time to display the next frame.

For example, let V1 (x1,y1,z1) and V2 (x2,y2,z2) be a vertex at frame 1 and 2 respectively. At time just before frame 2 should be displayed, we interpolate the vertex by V1 * (1 – t) + V2 * t. Every face will be drawn with interpolated vertices and the motion thus become smooth.

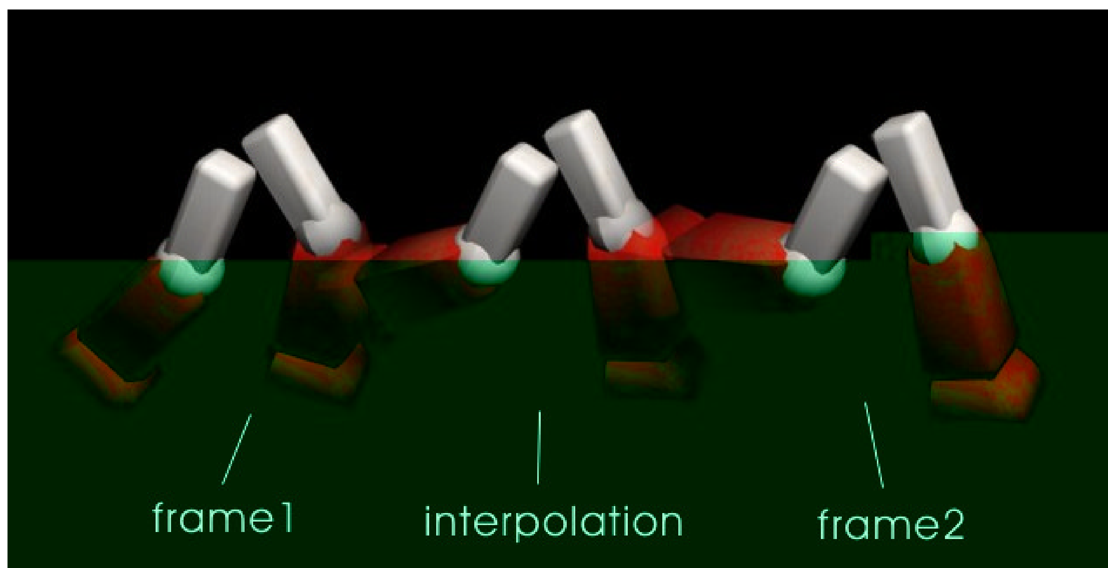


Fig. 6.24 Interpolation between frames

Frame skipping

When if the display rate is too low? In this case, we will skip some of the frames depending on how much time is elapsed from last changing of current frame. Let T be the time that each frame should last. If the elapsed time is greater than T, let n = (int)(elapsed time / T). And we will skip n – 1 frame and proceed to the nth frame. Also "t" will be updated to (elapsed time / T – n). This will compensate the inadequacy in display frequency. Although this will cause "jumping" on the screen, this measure makes slower computer to be able to catch up the game speed with other players over the internet.

6.2.3.5 Transition between different animations

When one animation ends and another animation starts, there should be a transition between the two animations. Since there are tens of animation for the lower and upper robot, it is quite impossible and not cost-effective to draw the transition frames for them. For example, for a running robot, you will never know at which frame of the animation the user will release the key to stop the robot. Sudden interrupting the running motion and making the robot stands still is not a visually favorable approach. In order to improve the situation, the following **new idea** is adopted. The stopping animation consists of only one frame. The special thing about this method is that this stopping animation is of much slower frame rate (say 5 fps). The stopping motion will be done by interpolating the frame at which the robot starts to stop and only frame in the stopping animation. This is a long interpolation interval as compared to normal interpolation. The effect with this approach turns out to be good. And a quite a large potion of animation transitions is making us of this approach.

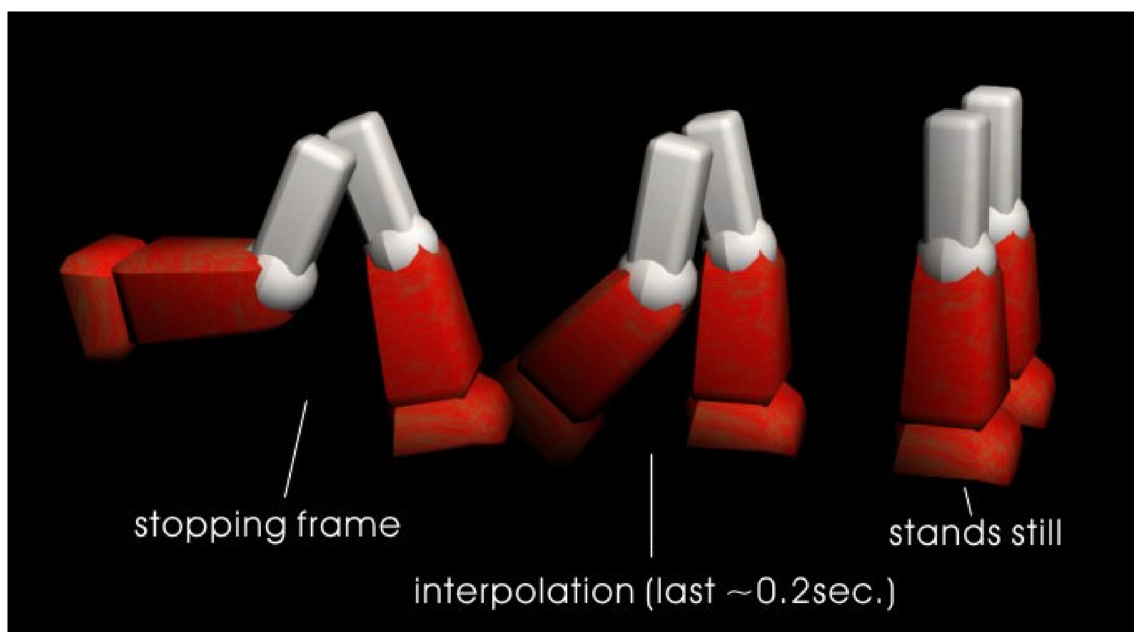


Fig. 6.25 Smooth transition

6.2.3.6 Animations in use

Currently there are more than 20 pieces of animations for each of the lower and upper robot. Several pieces of animation form a continual action, e.g. raise the shield, guarding, and letting the shield down. There are many other actions like swinging the weapon, jumping and getting up. (See **figure 6.26** and **6.27**)

Fig. 6.26 Attacking

Fig. 6.27 Guarding

6.2.4 Character control mechanism

6.2.4.1 Overview

There are mainly two attack modes in our game project, Mecha Zeta. They are the close attack and ranged attack. The players can control the robot to run, jump, attack, guard and etc. In this section, we are going to look at the control mechanism in Mecha Zeta.

6.2.4.2 Control keys

There are currently 7 control keys on the keyboard. They are "up", "down", "left", "right", "jump", "change weapon" and "reset camera". The mouse is used to control the shooting direction for the first shoot method. The left and right buttons of the mouse are for "attack" and "guard".

6.2.4.3 Control method

The four direction keys are for running in the four directions. For example, when pressing the left key, the robot will be running left relative to the screen, instead of just rotating the robot in an anti-clockwise direction (**figure 6.28**). Keep pressing the left button will cause the robot to rotate in an anti-clockwise direction gradually while running to the left.

Fig. 6.28 Running left

When pressing the "down" button, the robot will be running towards the screen (**figure 6.29**).

Fig. 6.29 Running down

6.2.4.4 Shooting mechanism

Here I would like to introduce one of the shooting methods in Mecha Zeta. To shoot a target, you have to change and use the ranged weapon, and move your mouse cursor to an appropriate place. On the top-right hand corner, there is a small window showing the current shooting directing for your gun. If you press the attack button, the gun will fire towards that direction. (**figure 6.30**)

Fig. 6.30 Shooting view

6.2.4.5 Reset camera

When you press the "reset camera" button, the camera will pan towards the view of the robot (direction in which the robot is facing) in a suitable way.

6.2.4.6 Combo attack

If you press the attack button just before the previous attack ends, a combo attack will follow. A maximum of 4-hits combo is possible. (**figure 6.31**)

Fig. 6.31 The 4 combos

6.2.5 Game logic

6.2.5.1 Overview

Game Logic is responsible for how the robot reacts with the environment and other robots, and what to do in response to the control signals. In other words it determines how the game is played. It will update the robot status upon external events. For example, if the collision detection engine detects that a robot is hit by a weapon, that engine can invoke the game logic to take action on the robot upon this event (e.g. reduce "health points" and let the robot fall down).

6.2.5.2 Update physics

Updating physics is one of the most important jobs of the Game Logic. A list of terrain objects that need to be updated is held by the MZGameLogic. At fixed interval of time, it will go into the list to help updating these objects. Types of update include updating position during running, updating height during jumping, panning the robot's view when start running, panning the camera view, updating the position for bullets and etc. It has the ability to control or limit the changing of the environment, e.g. how fast the bullet is flying and magnitude of gravity.

In the case the position of the running robot having to be updated, the game logic move the robot forwards by a certain amount depending on the running speed and the time elapsed between last and current updates. As a result, computer which can achieve different updating frequencies will move the robot in the same rate.

6.2.5.3 Handle the change of animations

When a player releases the “up” key, the robot should stop. This is very natural to all of us. However, in the world of game, the computer does not know which animation should be played after the robot stops running. “Which animation to play next” is actually handled by the game logic. When the game logic knows that the player is no longer pressing the key, it will change the animation of the lower part to “standing” animation.

When the robot finished the first hit, should it stop or continue the second hit? This is also decided by the game logic at the time the robot finishes the first hit (or the animation for the first hit ends).

6.2.5.4 API to handle control keys

The game logic also provide a set of API handling the control keys. When a control key is received by the game engine, it will notice the game logic about the key through the set of API (e.g. upPressed, downReleased and attackPressed). The game logic will response accordingly. It is actually the place where the control mechanism mentioned. It abstracts the control key handling job so that other parts in the game engine do not know and will not interfere what it is doing.

6.2.6 Graphical effects

6.2.6.1 Overview

Different graphical effects are employed in order to provide better the visual effect and make the scene more realistic. Basic lighting effect is enabled using functions provided by OpenGL. With the help of the stencil buffer, high quality shadow casting is also implemented. More lighting effects may be added in the near future if time is allowed.

6.2.6.2 Lighting

Lighting is an elementary technique which can improve the quality of the rendered scene significantly.

Before enabling lighting, some attributes about lighting have been set. They are the ambient light (the background light) and the diffuse light. Currently only GL_LIGHT1 is used. Every time before we drawing anything, we have the set the light position once, just after calling "gluLookAt". The two figures below (**figure 6.32** and **6.33**) show the difference in a scene with and without lighting.

Fig. 6.32 Lighting on

Fig. 6.33 Lighting off

The smooth shading is achieved based on the normal for each vertex. The color at each vertex is calculated accurately. And then the color pixels on the edge or inside the face are obtained by interpolation of the color at the three vertices, i.e. the Gouraud Shading.

6.2.6.3 Stencil shadow casting

To apply the shadow casting theory in OpenGL, we need the help of the stencil buffer. First of all we need to add a plane equation for each faces, so that we can determine whether a face is facing the light source. These plane equations are prepared at the startup time. Every time before we cast the shadow volume, the visibility of each face has to be re-determined as the real position or relative position of the light source may be moving. After drawing objects, the shadow casting starts. For each of the Silhouette edges w.r.t. the light source, project the edge to infinity virtually from the

direction of the light source, forming a quadrilateral. If this quadrilateral is counter-clockwise from the view of the eye position, increase the stencil buffer in the area covered by the quadrilateral. If is clockwise, decrease the stencil buffer in that area. This is actually an implementation of the techniques used. With OpenGL, the above procedure is done in two passes. The stencil buffer is cleared and set to zero at the beginning. The first pass is to increase the stencil buffer with those "counter-clockwise" areas. The second pass is to decrease the stencil buffer with those "clockwise" areas. The code is as follows:

```
glFrontFace(GL_CCW);
glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);
castShadowVolume();
glFrontFace(GL_CW);
glStencilOp(GL_KEEP, GL_KEEP, GL_DECR);
castShadowVolume();
```

Finally the shadow is drawn by placing a large blended black rectangle covering the whole screen, while only writing to pixels where the corresponding stencil bits is greater than zero:

```
glStencilFunc(GL_NOTEQUAL, 0, 0xffffffff);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
drawRectShadow();
```

The difference between doing two passes of shadow volume casting and one pass only are shown below. (**figure 6.34** and **6.35**)

Fig. 6.34 Two pass

Fig. 6.35 One pass

With the current implementation, it is possible to have a light source moving all the time, with the shadow being correctly displayed.

Near clip plane Problem

There exists a chance that the shadow volume is running out of the near clip plane. In this case the part of shadow volume will be missed, leading to mistaken count (as in **figure 6.36**). This problem is unavoidable for using the Zpass approach.

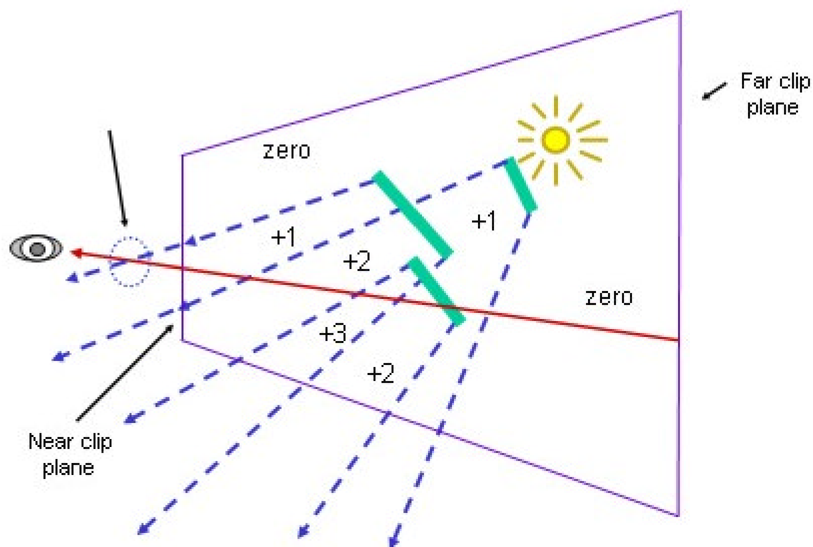


Fig. 6.36 Near clip plane problem for Zpass

Popping

Fig. 6.37 Popping

Popping is a side-effect generated from the above implementation. It will cause the faces that are not facing the light source to be darkened as if a shadow is on it. When the robot turns around, some of the faces will become visible or no longer visible to the light source suddenly. This results in sudden popping or disappearing of the unwanted shadow (illustrated in **figure 6.37**). At this stage two approaches have been employed to minimize the unwanted effect.

Insetting the shadow volume

Fig. 6.38 Insetting technique

When we cast the shadow volume in the two passes,, we inset the shadow volume by a little amount in the direction opposite to the normals of the vertices, shrinking the shadow volume apparently. i.e. use (V - k * N) for drawing the shadow volume instead of V, where V is the vector for a vertex on the Silhouette edge or the vector for the projected vertex, N is the normal vector at that vertex, and k is a small constant. This act will make the unwanted shadow to appear gradually, sliding over the faces, instead of popping out (see **figure 6.38**).

Drawing the real objects in two passes

The second measure to deal with popping is to render the real objects in two passes. In the first pass, only the visible faces, which are necessary for self-shadowing, are

drawn. The “not visible” faces are only drawn after casting the shadow volume. In this way, no unwanted shadow will be marked in the stencil buffer in the “not visible” area (see figure 6.39).

With this approach, the unwanted shadow in “not visible” area is eliminated. The performance is not lowered as compared to when there is popping. This is because the total number of faces drawn is not increased. They only difference is that they are drawn at two different time. However, there are still many other complex methods that can provide even better effects.

Fig. 6.39 No popping

6.3 Feasibility study on shadow algorithms

6.3.1 Fake shadow

It use projection of the center of the object to locate the center of the shadow. Then a approximated shape of shadow is placed around the center of the shadow.

Pros

- Simple and fast.

Cons

- Can be applied to flat ground only
- No self-shadowing
- Rotation of the shadowing object is limited to a certain axis.

6.3.2 Vertex Projection

It project the vertices of the object onto the ground using exact mathematics

Pros

- Simple but excat
- No limitations on rotation

Cons

- Only suitable to be applied to flat ground
- No self-shadowing

6.3.3 Shadow Z-buffer

First of all render the objects in the light source's point of view into the z-buffer. Prepare a transformed mapping of the Z-buffer (transform according to the camera's point of view). Shadow the area depending on the Z-value of the transformed Z-buffer. Render the objects again in the camera's point of view.

Pros

- Medium effort with medium accuracies

Cons

- Inaccuracies brought by transforming the pixels in the Z-buffer
- The objects are drawn twice
- No slef-shadowing

6.3.4 Static shadow

It is not a real-time Shadow Algorithm and the shadow will not be animating with the shadow casting object. The shadow is just a pre-calculated map and it is being rendered like normal polygons.

Pros

- Simple and fast

Cons

- Shape of shadow remains unchanged
- No self-shadowing

6.3.5 Shadowing volume casting with Zfail approach

It is very similar to the Zpass approach mentioned before. The only difference is that, the "count", used to determine whether a point is shaded, is increased or decreased only when a shadow volume face is cast to an area behind the point on the object, where the depth-test fails. And this time when a counter-clockwise quadrilateral shadow volume face is drawn, the counts in that area will be **decreased** instead, and vice versa. With this approach, there is still a "far clip plane problem". There will be missed shadow volume intersection due to far clip plane clipping, leading to mistaken count (see **figure 6.40**).

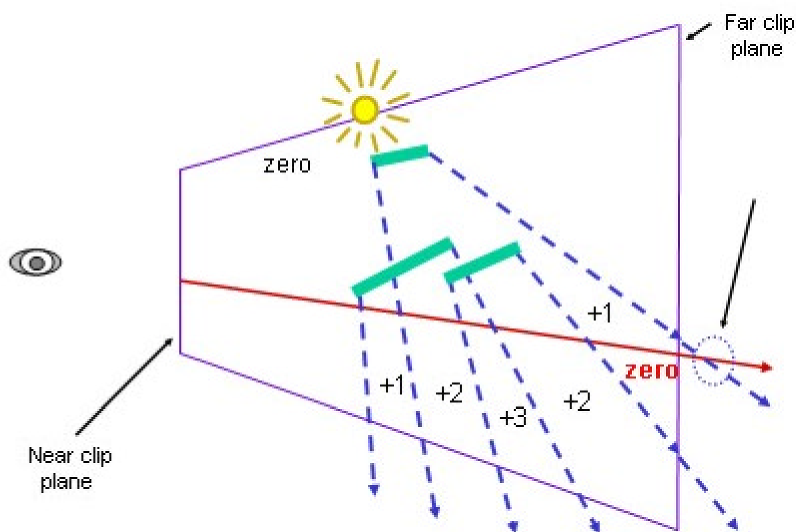


Fig. 6.40 Far clip plane problem

6.4 Performance Analysis

The performance of the program is significantly lowered after adding shadow volume casting. In view of this, I have noted down the frame rate in frame per second (fps) under different conditions. The testing is done with an AMD 700MHz computer with 256MB RAM, in Windows XP with window size 800 X 600. The robot contains around 200 triangles.

Condition	**fps**
No shadow casting	39.5
Shadow casting	14.2
Shadow casting (insetting volume)	14.2
Shadow casting (eliminate popping)	14.2
Shadow casting (one pass)	19.3
Shadow casting (two passes, but without drawing the big rectangle covering the whole screen for displaying the shadow)	16.2

From this figure, we can conclude that the bottlenecks of performance for drawing shadow come from:

1. Drawing the shadow volume's faces. (The number of faces is comparable to the faces of the object.)
2. Drawing with Stencil Test enabled. (It is time consuming to access, compare with or modify the stencil bits.)

6.5 Discussions and Conclusions

About the 3D modeling, I think there can be still a number of improvements. For example, it would be nice if the model loader can compute the bounding box for the object. This would compensate the inadequacies of the MD3 model. Besides, whether it is good to use other 3D file format with different features (say allowing the total number of vertices to change through out the animation) is still unknown.

At this stage, the number of elements in the game is certainly not enough, not to say to attract any players. In order to improve the playability of our game, more lighting effects such as laser and explosion should be added. Designing some attractive playing modes other than merely fighting or developing a story line are good ideas.

In addition, the two current shooting methods, shooting through cursor position and auto-lock mode, have not been tested properly. There should be some areas for improvement when we have experienced enough with these controlling methods.

Up till now, the problem on the performance of the Java language has become more significant, after the introduction of shadow into the rendering procedure. However, Java has eased the programming job and shortened our debugging time. Some sophisticated fine tuning on the bottleneck may be needed, so as to lower the computer requirement for this game.

On the whole, the time for this project is quite tight and there are still many unexplored areas in which we can develop our game.

7 Testing & Conclusion

After the integration of the different parts, we have conducted a trial for playing the game. It was found that eight players could play smoothly simultaneously after joining the server. However, there are quite a number of occasions that the server will fail when a client wants to join a game where many clients have already join the game, say about 7 clients. This is mainly due to the implementation error at the server side. It does not cast a critical problem, as the role of server in peer-to-peer architecture does not involve in the controls and positions communication between peers. Further implementation of server in respect to stability is left to future work.

The P2P architecture proves to provide adequate communications channel between the clients for Mecha Zeta. The ½ round-trip time communication of P2P is crucial to fast-responsive interactive game environment in order to minimize the transmission time overhead.

It is recommended that the implementation of the server should be refined in order to accept more connections at the same time for all peers. Mirror Server implementation can enlarge the capacity such that the communications between server and clients become more stable.

8 References

P2P Architecture and Protocol

[2.1] Jake Simpson. *Game Engine Anatomy*.

http://www.extremetech.com/article2/0,3973,594,00.asp

[2.2] J.F. Kurose and K.W. Ross. *Computer Networking: A top-down approach Featuring the Internet*. Addison Wesley, Preliminary edition, 2000

[2.3] IP Documentation. http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/

[2.4] Gary R.Wright, W. Richard Stevens. *TCP/IP Illustrated Volume 2: The implementation*

[2.5] Java sdk 1.4.1 API. http://java.sun.com/j2se/1.4.1/docs/api/index.html

[2.6] Java Gaming - http://www.javagaming.org

[2.7] Andrew D. Birrell. *An Introduction to Programming with Threads.* 1989

[2.8] Doug Lea. Presentation: *Concurrent Programming in Java*.

http://gee.cs.oswego.edu.

[2.9] *Concurrent Programming: The Java Programming Language.*, Oxford University Press. 1998

Partitioning and Sound system

[3.1] W. Richard Stevens. *Unix Network Programing*. Prentice Hall, 2nd edition, 1998

[3.2] James D. Foley et al., *Computer Graphics: Principles and Practice*. Pearson Education, 2nd edition, 1996

[3.3] Gaming tutorial http://www.flipcode.com/network/

[3.4] Gaming tutorial http://www.gamedev.net/

[3.5] Java gaming procedure http://www.njnet.edu.cn/info/ebook/java/javagame/

[3.6] Sound engine reference http://www.jsresources.org/

[3.7] Free sound sample http://www.3dcafe.com/asp/

[3.8] Free sound sample http://www.stonewashed.net/

[3.9] Free sound sample http://geekswithguns.com/sounds.html

Synchronization

[4.1] Michael Morrison, “Teach Yourself Internet Game Programming with Java in 21 Days”, ch.17,

[4.2] T.A. Funkhouser. “In Proc. of the SIGGRAPH Symposium on Interactive 3D Graphics”, pages 85–92. ACMSIGGRAPH, April 1995.

[4.3] L.Gautier and C.Diot, “A Distributed Architecture for Multiplayer Interactive Applications on the Internet,” *IEEE Network*, Vol.13, pp.6-15, 1999.

[4.4] M.Ahamad and R.Kordale, “Scalable Consistency Protocols for Distributed Services,” *IEEE Transactions on Parallel and Distributed Systems*, Vol.10, pp.888-903, 1999.

[4.5] C.Sun and D.Chen, “A Multi-version Approach to Conflict Resolution in Distributed Groupware Systems,” *InProceedings of the Twentieth IEEE International Conference on Distributed Computing Systems*, pp. 316-325, Taipei, 2000.

[4.6]. L.Vaghi, C.Greenhalgh, and S.Benford, “Coping with Inconsistency due to Network Delays in Collaborative Virtual Environments,” *In Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pp.42-49,London, 1999.

[4.7] CHANDY, K.M. AND MISRA, J., Asynchronous Distributed simulation via a Sequence of Parallel computations, CACM, April 1981, Vol. 24, No.11, pp.198-206.

[4.8] D. Jefferson, "Virtual Time", *ACM Transactions on Programming Languages and Systems,* Vol.7, No.3, 1985, pp.404-425.

[4.9] J. S. Steinman, R. Bagrodia, and D. Je_erson. *Breathing time warp.* In “*Proc. of the 1993 Workshop*

on Parallel and Distributed Simulation", pages 109{118, May 1993.

[4.10] http://www.eecis.udel.edu/~mills/ntp.html

Graphic Engine and Collision Detection System

[5.1] Jausoft, OpenGL for Java (GL4Java), 2002, Available online at http://www.jausoft.com/gl4java

[5.2] Photon, ColDet – Free 3D Collision Detection Library, 2002, Available online at http://www.photoneffect.com/coldet

[5.3] Wells R., Java offers Increased Productivity, 1999, Available online at http://www.wellscs.com/robert/java/productivity.htm

[5.4] Marler J., Evaluating Java for Game Development, 2002, Available online at http://www.rolemaker.dk/articles/evaljava/

[5.5] Sun Microsystems, Inc., Java3D API Homepage, 2003, Available online at http://java.sun.com/products/java-media/3D/

[5.6] Melax S., Gamasutra - Features – "BSP Collision Detection As Used In MDK2 and NeverWinter Nights", 2001, Available online at http://www.gamasutra.com/features/20010324/melax_01.htm

[5.7] UNC Gamma Research Group, Gamma: Geometric Algorithms for Modeling, Motion and Animation, 2002, Available online at http://www.cs.unc.edu/~geom/

[5.8] UNC Gamma Research Group, RAPID -- Robust and Accurate Polygon Interference Detection, 1997, Available online at http://www.cs.unc.edu/~geom/OBB/OBBT.html

[5.9] Ehmann S., SWIFT++ Speedy Walking via Improved Feature Testing for Non-Convex Objects, 2001, Available online at http://www.cs.unc.edu/~geom/SWIFT++/

[5.10] NeHe, NeHe Productions, 2003, Available online at http://nehe.gamedev.net

[5.11] GameTutorials, GameTutorials, 2003, Available online at

http://www.gametutorials.com

[5.12] Sullivan R., OpenGL for Java Tutorials and Demos, 2002,

http://web.hypersurf.com/~sully/OpenGL/DemoBox.html

Modeling, Animation and Real time shadowing

[6.1] Skinning techniques with 3D Studio

http://www.planetquake.com/q3empire/data/tutorials/modeling/skinning/skinning.htm

[6.2] Gouraud Shading

http://freespace.virgin.net/hugo.elias/graphics/x_polygo.htm

[6.3] Shadow volume
http://www.gamedev.net/columns/hardcore/shadowvolume/page2.asp